BERGISCHE
UNIVERSITÄT
WUPPERTAL

DOCTORAL DISSERTATION

On the Tight Security of the Transport Layer Security (TLS) Protocol Version 1.3

Denis Diemert, M.Sc.

January 18, 2023

Submitted to the
School of Electrical, Information and Media Engineering
University of Wuppertal

for the degree of
Doktor-Ingenieur (Dr.-Ing.)

Denis Diemert
Place of birth: Detmold, NRW, Germany
Author's contact information:
dediemert@gmail.com

Thesis Advisor:	**Prof. Dr.-Ing. Tibor Jager** Chair for IT Security and Cryptography University of Wuppertal, Wuppertal, Germany
Second Examiner:	**Prof. Dr. phil. nat. Marc Fischlin** Cryptography and Complexity Theory Group Technische Universität Darmstadt, Darmstadt, Germany
Thesis submitted:	January 18, 2023
Thesis defense:	April 19, 2023
Last revision:	June 5, 2023

LIST OF PUBLICATIONS

Publications in this thesis. The following publications are part of this thesis.

[DDGJ22] Hannah Davis, Denis Diemert, Felix Günther, and Tibor Jager. "On the Concrete Security of TLS 1.3 PSK Mode". In: *Advances in Cryptology – EUROCRYPT 2022, Part II.* Ed. by Orr Dunkelman and Stefan Dziembowski. Vol. 13276. Lecture Notes in Computer Science. Springer, Heidelberg, May 2022, pp. 876–906. DOI: 10.1007/978-3-031-07085-3_30.

[DGJL21] Denis Diemert, Kai Gellert, Tibor Jager, and Lin Lyu. "More Efficient Digital Signatures with Tight Multi-user Security". In: *PKC 2021: 24th International Conference on Theory and Practice of Public Key Cryptography, Part II.* Ed. by Juan Garay. Vol. 12711. Lecture Notes in Computer Science. Springer, Heidelberg, May 2021, pp. 1–31. DOI: 10.1007/978-3-030-75248-4_1.

[DJ21] Denis Diemert and Tibor Jager. "On the Tight Security of TLS 1.3: Theoretically Sound Cryptographic Parameters for Real-World Deployments". In: *Journal of Cryptology* 34.3 (July 2021), p. 30. DOI: 10.1007/s00145-021-09388-x.

Other publications.

[Bem+23] Pascal Bemmann, Sebastian Berndt, Denis Diemert, Thomas Eisenbarth, and Tibor Jager. "Subversion-Resilient Authenticated Encryption without Random Oracles". In: *ACNS 2023: 21st International Conference on Applied Cryptography and Network Security.* To appear in Lecture Notes in Computer Science. Springer, 2023.

[DGJL21] Denis Diemert, Kai Gellert, Tibor Jager, and Lin Lyu. "Digital Signatures with Memory-Tight Security in the Multi-challenge Setting". In: *Advances in Cryptology – ASIACRYPT 2021, Part IV.* Ed. by Mehdi Tibouchi and Huaxiong Wang. Vol. 13093. Lecture Notes in Computer Science. Springer, Heidelberg, December 2021, pp. 403–433. DOI: 10.1007/978-3-030-92068-5_14.

[BBDE19] Johannes Blömer, Jan Bobolz, Denis Diemert, and Fabian Eidens. "Updatable Anonymous Credentials and Applications to Incentive Systems". In: *ACM CCS 2019: 26th Conference on Computer and Communications Security*. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM Press, November 2019, pp. 1671–1685. DOI: 10.1145/3319535.3354223.

[Bem+18] Kai Bemmann, Johannes Blömer, Jan Bobolz, Henrik Bröcher, Denis Diemert, Fabian Eidens, Lukas Eilers, Jan Haltermann, Jakob Juhnke, Burhan Otour, Laurens Porzenheim, Simon Pukrop, Erik Schilling, Michael Schlichtig, and Marcel Stienemeier. "Fully-Featured Anonymous Credentials with Reputation System". In: *ARES 2018: 13th International Conference on Availability, Reliability and Security*. ACM, 2018, 42:1–42:10. DOI: 10.1145/3230833.3234517.

ACKNOWLEDGEMENTS

On the journey that led to this work, I was fortunate to be accompanied by a number of great people. Here, I would like to thank those who have been with me along this way.

First and foremost, I would like to thank my supervisor, Tibor Jager. Without Tibor this work would never have happened. He offered me the position in his group when I thought I had already decided not to stay at the university and convinced me that research was exactly what I was looking for to deepen my interest in IT security and cryptography. I am so thankful that he gave me this opportunity and made it possible to do research the way I wanted to. Tibor is a great leader who always stands up for his team and is always there to give useful advice, whether it is for our careers in general or for a certain research problem. Thank you so much, Tibor, for guiding and supporting me throughout!

Then, there are all of the great people I was lucky to call my colleagues over the years. Thank you, Pascal Bemmann, Peter Chvojka, Gareth T. Davies, Jan Drees, Amin Faez, Kai Gellert, Tobias Handirk, Raphael Heitjohann, Máté Horváth, Saqib Kakvi, Rafael Kurek, Lin Lyu, Jutta Maerten, David Niehues, Marloes Venema, and Jonas von der Heyden. I am grateful that I had the opportunity to be a part of such an awesome team, and all of you made this an enjoyable experience that I will always remember. I would especially like to thank Pascal for making the countless train rides less annoying and for always having an open ear, whether it was for personal matters or research.

The essence of research is collaboration and I was fortunate to work with a number of talented and inspiring people. I would like to thank my co-authors Pascal Bemmann, Sebastian Berndt, Johannes Blömer, Jan Bobolz, Hannah E. Davis, Fabian Eidens, Thomas Eisenbarth, Kai Gellert, Felix Günther, Tibor Jager, Lin Lyu, and the members of my project group "RE(AC)^t" at Paderborn University. I am not only happy to have met you, but also to have learned so much from you. I am especially grateful that Hannah and Felix handled our independent and current work so professionally, and opened the doors for our insightful collaboration.

I thank Marc Fischlin for agreeing on co-reviewing my thesis.

A very special thanks goes to the Codes and Cryptography group at Paderborn University, who always treated me as one of their own when I was a student. In particular, I would like to thank Johannes Blömer for teaching me the foundations of cryptography (and more) and sparking my interest in the field of cryptography. Moreover, I would like to thank Jan Bobolz and Fabian Eidens. Jan and Fabian gave me the opportunity to work with them as a student, which showed me how much I like scientific work. It was such a pleasure working with them such that they significantly influenced this part of my

career, and even though they would deny it, they teached me the majority of the skills I used on a day-to-day basis during my own research journey. Despite that I decided to pursue my research with a different focus, I am deeply grateful that the connection between the three of us is still unchanged after many years.

I also would like to thank my close friends. Caro, Eric, Meli, Pauli, and Timo, even though we are all scattered around Germany, I am happy that no matter how far we are apart, or how long we have not seen, nothing changes between us. I am grateful for our regular trips, which always feel like we were still at school and for your constant support, probably without even noticing. Aylin and Julian, I am thankful for our evenings with good food and games, which particularly helped me to forget everything for a bit and just enjoy the moment. Anna, Fabian, and Gregor, I am thankful for the days we have spent walking around, eating ice cream, playing games, or cooking Carbonara together. Especially, Fabian, who went through this process right before me, was able to help me with his experiences and advice through it all such that I kept going and made it to the end sanely. Thank you all so much for being part of my life!

Zu guter Letzt möchte ich meiner Familie danken, die all das hier möglich gemacht hat. Die Familie meiner Freundin, Ede, Kerstin, Nico, Kalli und Anni, hat mich von Beginn an als vollwertiges Mitglied der Familie aufgenommen und ich danke euch dafür, dass ihr mir ein zweites Zuhause gebt. Meine Schwester Jana und ihr Mann Basti, begleiten mich in allen Lebenslagen. Danke, dass ihr mich immer unterstützt, zu mir haltet und für mich da seid, wann immer ich es brauche. Auch wenn man sich Geschwister nicht aussuchen kann, könnte ich es mir nicht besser wünschen. Meine Freundin Lea hat vermutlich, ohne es zu wissen, den größten Teil an dieser Arbeit beigesteuert. Sie hat mich immer wieder motiviert weiterzumachen und mir vor allem das Gefühl gegeben, dass ich alles schaffen kann, wenn ich selbst nicht mehr daran geglaubt habe. In Zeiten in denen ich Ablenkung brauchte, hat sie für diese gesorgt, wenn ich Freiraum brauchte, hat sie mir diesen gegeben. Wenn es nötig war, hat sie alles von mir ferngehalten, damit ich mich auf mich konzentrieren konnte. Dankbarkeit drückt nicht im Geringsten aus, was ich für dich empfinde. Ich kann mir keine bessere Partnerin für den Rest meines Lebens vorstellen, denn ich weiß, dass wir zusammen alles meistern. Meine Eltern, Ralf und Ulrike, haben es mir durch ihre bedingungslose Unterstützung in jeglicher Hinsicht ermöglicht zu studieren, diesen unkonventionellen Weg zu gehen und somit schließlich diese Arbeit zu verfassen. Ohne euch wäre ich heute nicht an diesem Punkt, aber viel wichtiger noch, vor allem nicht der Mensch, der ich heute bin. Deshalb möchte ich euch diese Arbeit widmen.

ABSTRACT

Nowadays, a life without communicating over the Internet is unimaginable. This communication reaches from staying in contact with other people using instant messaging services or social media to delicate tasks like transferring money from home using online banking. The Internet itself needs to be considered as an insecure communication channel as potential attackers could read the communication or even make unwanted modifications to it. Presumably, the importance of the Internet would never have been as it is today without the means that enable *secure communication*. The de-facto standard for secure communication is the *Transport Layer Security (TLS)* protocol. It enables two parties to communicate *securely* over an insecure channel, such as the Internet, with the use of cryptography. In particular, it ensures that the communication data in transit cannot be read (*confidentiality*), or modified (without detection) by outsiders (*integrity*). Additionally, it ensures that the two parties communicating are certain about their communication partner's identity (*authentication*). In TLS, the two parties aiming for secure communication perform the following two steps. First, they run a *key exchange protocol* called the *TLS handshake* to establish a shared secret, agree on parameters, and prove their identities to each other. Then, the *TLS record protocol* uses the established secret and parameters to built-up the secure communication channel.

Since there are so many applications including delicate ones (e.g., online banking) that rely on secure communication over the Internet, a rigorous analysis of a security mechanism as important as TLS is crucial. Modern cryptography makes use of the tools of math to provide rigorous formal treatments of cryptographic constructions referred to as *security proofs*. It is even common that new constructions nowadays come with such a proof of security. A cryptographic security proof is always with respect to some well-defined *security model* that captures the cryptographic system and its security goals as precisely as possible to result in meaningful statements. These proofs are then based on computational problems that are widely assumed to be hard to solve with access to a reasonable amount of resources. The proof itself assumes that there is an algorithm that is able to break the security of the cryptographic system with respect to the well-defined security model. Then, it transforms this algorithm into an algorithm that solves the assumed-to-be-hard problem. This transformation is called *reduction*. Now, under the assumption that the problem is hard to solve, this reduction shows that breaking the system is also hard to achieve with access to a reasonable amount of resources.

A proof of security does not only provide an excellent assertion of the plausibility of a construction, but also can be used to select parameters, which determine the provided level of security of a cryptographic system, based on a theoretical foundation. Here,

we are particularly concerned about the *tightness* of the security proof. The tightness describes how close the relation between the algorithm that breaks the cryptographic system and the reduction that solves the assumed-to-be-hard problem is. Intuitively, the tighter the relation the more security guarantees from the hard problem are inherited to the cryptographic scheme. That is, if the relation is very loose, it is required to increase the security guarantees provided by the assumed-to-be-hard problem to compensate the loss in security induced by the security proof (reduction). This usually means increasing the parameters of the cryptographic scheme, which unfortunately, incurs a penalty when it comes to efficiency of the overall scheme. Hence, for cryptographic systems that are deployed in the real world achieving tight security proofs is of particular importance. Namely, tight security proofs allow it to deploy cryptographic systems with parameters that are theoretically-sound (i.e., backed up by the security proof) without the need to compensate any loss in security induced by the security proof. Thus, allowing for a deployment that does not need to make trade-offs between efficiency and security.

In this thesis, we give insights on the tight security of the most recent version of the TLS protocol, namely TLS 1.3 (RFC 8446). To this end, we present in one part of this thesis a tight security proof for all variants of the TLS 1.3 handshake protocol in an idealized model called the random oracle model (ROM). This improves on prior analyses of TLS 1.3 that suffer from a loss that is a quadratic function in the number of TLS sessions across all users. Thus, enabling a theoretically-sound deployment of TLS 1.3. Furthermore, we discuss the current state of the tightness of the TLS 1.3 record protocol. Finally, we present a new generic construction for a digital signature scheme that is secure in the very strong setting of existential unforgeability under an adaptive chosen-message attack in the multi-user setting with adaptive corruptions in the ROM. This scheme is the currently most efficient scheme achieving such security notion and is an ideal candidate to be used in tightly-secure key exchange protocols that use digital signatures for authentication. In particular, the signature schemes supported by TLS 1.3 at the moment do *not* satisfy this strong notion even though our tight analysis of the TLS 1.3 handshake requires it. Hence, our signature scheme demonstrates that the TLS 1.3 handshake protocol can be tightly instantiated with appropriate building blocks and we provide this scheme as a suggestion.

ZUSAMMENFASSUNG

Kommunikation über das Internet ist heutzutage kaum noch aus dem alltäglichen Leben wegzudenken. Die Anwendungen sind vielfältig: So nutzt man das Internet um mit anderen Menschen über Instant-Messenger oder Soziale Netzwerke in Kontakt zu bleiben bis hin zu heiklen Aufgaben, wie der Überweisung von Geld von zu Hause aus mithilfe von Online-Banking. Da potenzielle Angreifer die Kommunikation über das Internet mitlesen oder sogar unerwünschte Änderungen daran vornehmen könnten, muss das Internet als unsicherer Übertragungsweg betrachtet werden. Ohne Maßnahmen, die eine sichere Übertragung über das Internet gewährleisten, wäre dieses vermutlich nicht zu dem geworden, was es heute ist. Der de-facto Standard um eine sichere Übertragung zu gewährleisten, ist das Transport Layer Security (TLS) Protokoll. TLS ermöglicht es, dass zwei Parteien mithilfe von Kryptographie sicher über einen unsicheren Übertragungsweg, wie das Internet, kommunizieren können. Insbesondere sorgt TLS dafür, dass die gesendeten Daten während der Übertragung nicht von Außenstehenden gelesen (*Vertraulichkeit*) oder unbemerkt verändert werden können (*Integrität*). Außerdem stellt es sicher, dass die sich die beiden Parteien über die Identität ihres jeweiligen Kommunikationspartners sicher sein können (*Authentifizierung*). Zwei Parteien, die mithilfe des TLS-Protokolls sicher Kommunizieren wollen, führen die folgenden zwei Schritte aus. Zunächst führen sie ein Schlüsselaustauschprotokoll, den so genannten *TLS-Handshake*, durch, um ein gemeinsames Geheimnis festzulegen, sich auf Parameter zu einigen und sich gegenseitig ihre Identitäten zu beweisen. Dann verwendet das *TLS-Record-Protokoll* das ausgetauschte Geheimnis und die Parameter, um einen sicheren Kommunikationskanal aufzubauen.

Viele Anwendungen erfordern sichere Kommunikation über das Internet, darunter auch Anwendungen (wie Online-Banking), die hochsensible Daten verarbeiten. Hier ist von besonderer Bedeutung, dass sich ein so wichtiger Sicherheitsmechanismus wie TLS, strengen Sicherheitsanalysen unterzieht. Moderne Kryptographie nutzt die Werkzeuge der Mathematik, um genaue formale Analysen von kryptographischen Konstruktionen, so genannte *Sicherheitsbeweise*, zu erstellen. Heutzutage ist es sogar üblich, dass neue Konstruktionen zusammen mit einem solchen Sicherheitsbeweis entwickelt werden. Ein kryptographischer Sicherheitsbeweis bezieht sich immer auf ein wohldefiniertes *Sicherheitsmodell*, das das kryptographische System und seine Sicherheitsziele so genau wie möglich beschreibt, um möglichst aussagekräftige Ergebnisse zu erhalten. Diese Beweise basieren dann auf Berechnungsproblemen, von denen man annimmt, dass sie mit einer realistischen Menge an Ressourcen schwer zu lösen sind. Der Beweis nimmt an, dass es einen Algorithmus gibt, der in der Lage ist, die Sicherheit des kryptographischen

Systems im Hinblick auf das wohldefinierte Sicherheitsmodell zu brechen. Dieser Algorithmus wird dann in einen Algorithmus umgewandelt, der das Berechnungsproblem löst. Diese Umwandlung wird als *Reduktion* bezeichnet. Unter der Annahme, dass das Berechnungsproblem schwer zu lösen ist, zeigt diese Reduktion, dass es auch schwer sein muss, das System zu brechen, wenn man von Zugang zu einer realistischen Menge an Ressourcen ausgeht.

Ein Sicherheitsbeweis liefert nicht nur eine hervorragende Prüfung der Plausibilität einer Konstruktion, sondern kann auch bei der Auswahl von Parametern verwendet werden, die in direktem Zusammenhang mit der garantierten Sicherheit eines kryptographischen Systems stehen. Hier ist insbesondere die *Schärfe* (engl. *tightness*) des Sicherheitsbeweises von Bedeutung. Die Schärfe beschreibt, wie eng der Zusammenhang zwischen dem Algorithmus, der das kryptographische System bricht, und der Reduktion, die das angenommen schwierige Problem löst, ist. Je enger der Zusammenhang ist, desto mehr Sicherheitsgarantien werden von dem schwierigen Problem an das kryptographische System übertragen. Das heißt, wenn der Zusammenhang nicht scharf genug ist, müssen die Sicherheitsgarantien, die das schwierige Problem bietet, erhöht werden, um den durch den Sicherheitsbeweis (Reduktion) verursachten Sicherheitsverlust auszugleichen. Daher ist es für kryptographische Systeme, die in der realen Welt eingesetzt werden, von besonderer Bedeutung, scharfe Sicherheitsbeweise anzustreben. Diese ermöglichen es nämlich, kryptographische Systeme mit theoretisch begründeten Parametern (d. h., die durch den Sicherheitsbeweis gestützt werden) einzusetzen, ohne dass ein durch den Sicherheitsbeweis verursachter Sicherheitsverlust ausgeglichen werden muss. Dies bedeutet in der Regel, dass bei der Benutzung des Systems keine Kompromisse zwischen Effizienz und Sicherheit gemacht werden müssen.

Diese Arbeit gibt Einblicke in Bezug auf die scharfe Sicherheit der neuesten Version des TLS-Protokolls, TLS 1.3 (RFC 8446). Daher wird in einem Teil dieser Arbeit ein scharfer Sicherheitsbeweis für alle Varianten des TLS 1.3 Handshake-Protokolls in einem idealisierten Modell, dem sogenannten Random-Oracle-Modell (ROM), vorgestellt. Dies verbessert frühere Analysen von TLS 1.3, da diese einen Sicherheitsverlust quadratisch in der Anzahl der TLS-Sitzungen über alle Benutzer nach sich ziehen. Dadurch werden die Parameter von TLS 1.3 theoretisch begründet. Darüber hinaus wird der aktuellen Stand der scharfen Sicherheit des TLS 1.3 Record-Protokolls diskutiert. Schließlich wird eine neue generische Konstruktion für ein digitales Signaturverfahren vorgestellt, das die starke Sicherheitseigenschaft in Form von existentieller Unfälschbarkeit in einer Umgebung mit mehreren Benutzern und adaptiven Korruptionen im ROM erfüllt (engl. *existential unforgeability under an adaptive chosen-message attack in the multi-user setting with adaptive corruptions*). Diese Konstruktion ist das derzeit effizienteste Verfahren, das eine solche Sicherheitseigenschaft erreicht, und ist ein idealer Kandidat für den Einsatz in Schlüsselaustauschprotokollen, die scharfe Sicherheit anstreben und die digitale Signaturen zur Authentifizierung verwenden. Insbesondere erfüllen die derzeit von TLS 1.3 unterstützten Signaturverfahren diese starke Eigenschaft nicht, obwohl unserer scharfe Sicherheitsbeweis des TLS 1.3-Handshakes dies erfordert. Daher zeigt das vorgestellte Signaturverfahren, dass das TLS 1.3-Handshake-Protokoll mit geeigneten Bausteinen instanziiert werden kann, sodass es scharfe Sicherheit erreicht.

CONTENTS

ACRONYMS

AEAD	authenticated encryption with associated data
AKE	authenticated key exchange
CBC	cipher block chaining
CDH	computational Diffie–Hellman
DDH	decisional Diffie–Hellman
DH	Diffie–Hellman
DHE	Diffie–Hellman key exchange
DLOG	discrete logarithm
GDH	gap Diffie–Hellman
GGM	generic group model
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IETF	Internet Engineering Task Force
IMAP	Internet Message Access Protocol
KDF	key derivation function
LID	lossy identification scheme
MAC	message authentication code
MSKE	multi-stage key exchange
POP	Post Office Protocol
PRF	pseudorandom function
PSK	pre-shared key
ROM	random oracle model
RTT	round-trip time
SDH	strong Diffie–Hellman
SMTP	Simple Mail Transfer Protocol
SSL	Secure Socket Layer
TCP	Transport Control Protocol
TLS	Transport Layer Security

1

INTRODUCTION

Transport Layer Security. The de-facto standard for secure communication on the Internet (or, any computer network in general) is *Transport Layer Security (TLS)* specified in RFC 8446 [Res18]. As such it protects a wide range of application layer protocols, such as the Hypertext Transfer Protocol (HTTP) for web browsing, and the Internet Message Access Protocol (IMAP) [ML21], the Post Office Protocol (POP) [MR96], and the Simple Mail Transfer Protocol (SMTP) [Kle08] for email. These applications make TLS to one of the most important security mechanisms on the internet protecting billions of connections every day. Its most popular use clearly is its application to secure HTTP resulting in Hypertext Transfer Protocol Secure (HTTPS) ("HTTP over TLS"). This usually is indicated in the web browser by the padlock symbol next to the web address. In late 2022, roughly 80 % of web pages loaded by the Mozilla Firefox browser used HTTPS.[1] Similar numbers hold for Google's Chrome browser.[2] These numbers clearly demonstrate the importance of TLS on the Internet.

Classical communication over the Internet suffers from the limitation that most of the application layer protocols (e.g., HTTP, IMAP, POP, and SMTP) send their data in the plain. With the Internet having taken an increasingly important role in the everyday life, and new applications evolving day by day, the exchange of sensible data (e.g., personal bank information) increases such that sending data in plain is not sufficient anymore. TLS aims to close this gap by securing the data in transit and therefore enables to perform delicate tasks over the Internet, such as online banking, securely. Quoting the standard RFC 8446, the main goal of TLS is the following:

> *"TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery."*
>
> — Abstract of RFC 8446 [Res18]

To achieve this, TLS establishes a *secure channel* between two peers (i.e., client and server). Here, "secure", in particular, means that the channel established between the two endpoints satisfies:

[1] `https://letsencrypt.org/stats/` (visited Dec 31, 2022)

[2] `https://transparencyreport.google.com/https/overview` (visited Dec 31, 2022)

1. *Confidentiality*: Only the two communicating peers should be able to understand the data exchanged.
2. *Integrity*: It should not be possible to change the data in transit without the peers noticing that tampering occurred.
3. *Authenticity*: The server always authenticates towards the client. Intuitively speaking, a client should be sure that its communication partner is the intended one. Client authentication is optional and may be explicitly requested by the server.

TLS achieves these properties by only requiring a reliable (and in-order) transport of data. As its name already suggests, Transport Layer Security works directly on top of the transport layer and directly below the application layer in the Internet Protocol Suite (also known as the TCP/IP model). Usually, the reliable transport is implemented using the Transport Control Protocol (TCP) [Edd22]. Further, TLS is independent of the application it secures. Informally, this means that TLS takes the data of the application layer protocol that would be otherwise sent in the plain and protects this data as described above. To do so, TLS consists of two major components:

1. *Handshake protocol.* To establish the aforementioned secure channel, the two endpoints first perform a *handshake* in which they authenticate, negotiate cryptographic parameters, and establish shared keys.
2. *Record protocol.* The record protocol implements the secure channel to protect the data exchanged between the two communicating parties using the parameters negotiated in the previous handshake.

1.1 (Brief) History of TLS

SSL and TLS 1.0–1.2. Due to its relevance and the accompanying public attention, numerous attacks on TLS have been found since its introduction. TLS originally was published under the name *Secure Socket Layer (SSL).* The protocol was developed by Netscape Communications. Unfortunately, the history of SSL was accompanied by a number of serious security flaws. Its first version SSL version 1 (SSL 1.0) was never publicly available as it suffered from numerous shortcomings and flaws, e.g., the lack of integrity protection of data (cf. [Opp16]). Its first publicly available version SSL version 2 (SSL 2.0) [EH95] still suffered from serious flaws, e.g., the *cipher suite rollback* attack in which an active adversary is able to make changes to the list of cipher suites to enforce the communicating to use weaker cryptographic schemes than they actually support (cf. [WS96]). A number of issues of SSL 2.0 were addressed in Microsoft's concurrence protocol, which they called Private Communication Technology (PCT) [Ben+95]. SSL 3.0 [FKK11] adopted the improvements of PCT. The superordinated companies Microsoft and Netscape now faced the challenge that their products still needed to support each other's security protocol to ensure interoperability. Therefore, in 1996 the Internet Engineering Task Force (IETF) TLS working group was founded to find a unified

solution.[3] They published the first version of the new Transport Layer Security protocol (version 1.0) as RFC 2246 [DA99] in 1999. The changes compared to SSL were minimal, so TLS 1.0 can be seen as SSL 3.1 (cf. [Opp16]). TLS 1.0 was obsoleted by TLS 1.1 published as RFC 4346 [DR06] in 2006. This revision mostly addresses vulnerabilities[4] of the cipher block chaining (CBC) mode of operation for block ciphers used in the record protocol, which enable so-called *padding oracle attacks* first discovered by Vaudenay [Vau02]. Already two years after, TLS 1.1 was obsoleted by TLS 1.2, which was published in 2008 as RFC 5446 [KN09]. In the upcoming years, SSL/TLS has been subject to a number of attacks on various aspects of the protocol. To just name a few that raised public attention, there were the attacks BEAST [Duo11], CRIME [RD12], Heartbleed [Cod14], POODLE [Bod14], DROWN [Avi+16], and SLOTH [BL16b].

SSL 2.0 has been deprecated in March 2011 [TP11] due to its shortcomings, e.g., the use of hash function MD5 [Riv92], which is not considered collision resistant anymore [TC11] and the aforementioned lack of integrity protection when it comes to the list of cipher suites allowing active adversaries to enforce weak cryptographic schemes. In 2014, the POODLE attack [Bod14] on SSL 3.0 was published, which allows an adversary to decrypt messages encrypted with CBC without knowing the corresponding key material. This resulted in a deprecation of SSL 3.0 [BTPL15]. Versions 1.0 and 1.1 of TLS have been deprecated by the IETF in 2021 with RFC 8996 [MF21b] due to the use of legacy cryptography and some of the attacks listed above.

TLS 1.3. Motivated by the history of attacks, the IETF involved for the first time both academia and industry in the standardization process of the next TLS version 1.3. The standard was published in 2018 as RFC 8446 [Res18] and is the most recent version. The new version includes rather major changes compared to the previous standard TLS 1.2. For instance, it removes legacy cryptography from the selection of algorithms (e.g., only allowing authenticated encryption with associated data encryption algorithms), adds a low-latency (i.e., zero round-trip time) mode for early application data exchange, and removes RSA-encryption based and static Diffie–Hellman (DH) key exchange present in previous versions such that ephemeral DH key exchange is the only key exchange mode possible. Further, it disables the compression feature by default (to counteract CRIME-like attacks) and completely redesigns of the key derivation in the handshake protocol. Note that even though the changes to the TLS protocol from version 1.2 to version 1.3 seem quite incompatible, backwards compatibility with previous versions is an important feature of the new TLS 1.3 standard. For example, the standard says that "Implementations of TLS 1.3 which choose to support prior versions of TLS SHOULD support TLS 1.2." [Res18, Sect. 4.2.1] to ensure fallback to version 1.2. In fact, according to SSL Pulse[5] in December 2022 almost every TLS-enabled web server of the Alexa list of the most popular websites on the Internet supports TLS 1.2 (99,9 %) while only 58,9 % support the most recent version TLS 1.3 (four years after its introduction). This

[3] `https://datatracker.ietf.org/group/tls/about/`
[4] `https://www.openssl.org/~bodo/tls-cbc.txt`
[5] `https://www.ssllabs.com/ssl-pulse/`

especially demonstrates how important TLS 1.2 still is today even though it has been target of numerous attacks. Therefore, it also is non-trivial to securely configure TLS 1.2.

1.2 Provable Security

The history of TLS is an excellent example how important a rigorous analysis of a system, in particular a cryptographic one, is before deploying it in the wild. A common formal approach for such analyses in the area of cryptography is *provable security*, which first appeared in 1982 in the seminal work "Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information" by Goldwasser and Micali [GM82]. The provable security approach has proven to be a great tool, not only to check the plausibility of real-world cryptographic system, but also to make design decisions founded on a theoretically-sound ground. In the provable security approach one uses concepts stemming from computational complexity to give rigorous formal proofs of security for cryptographic systems. The central concept of the provable security approach is a *cryptographic reduction.*

1.2.1 Cryptographic Reductions

A fundamental concept of computational complexity theory is the *reduction.* The main idea is simple: Given an algorithm to solve some computational problem B, we construct an algorithm that solves some other problem A by giving an algorithm $\mathcal{R}$ (called reduction) that transforms an instance ϕ_A of problem A into an instance ϕ_B of problem B. Then, we can solve problem A by first transforming the problem instance ϕ_A into a problem instance ϕ_B for problem B using algorithm $\mathcal{R}$, and run the algorithm solving problem B on ϕ_B. Finally, we only have to find a way to map a solution of problem B to a solution of problem A. If all of this can be done *efficiently* (that is, e.g., in time and space polynomial in the input length), we say that problem A is *reducible* to problem B. This is sometimes written as $A \leq_p B$, which illustrates nicely what this actually means. Namely, that the problem A is at most as "hard to solve" as problem B. Put differently, if a solver for problem B can be used as a subroutine to solve problem A, problem B has to be at least as hard to solve as problem A. This technique classically is used to relate computational hardness of problems.

In modern cryptography, we make use of the exact same concept to formally prove security of a cryptographic system. Here, we base our security on computational problems that are assumed to be hard (e.g., discrete logarithms, factoring of composite integers, etc.). Now, equipped with the technique of a reduction, we can reduce the security of a cryptographic system to the hardness of a computational problem with the following steps.

1. Define a *security model* capturing the security requirements of the cryptographic system. This is used to precisely define the capabilities of an adversary and what it means for an adversary to be successful (i.e., breaking the cryptographic system).

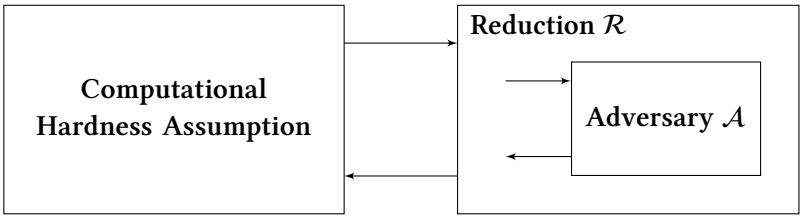

Figure 1.1: Block diagram illustrating a cryptographic reduction.

Usually, this model is formally captured in form of a *security experiment* (or *security game*).

2. Choose an appropriate computational problem that is assumed to be hard.
3. Assume there is an algorithm called *adversary* $\mathcal{A}$ that can break the security of the cryptographic system with respect to the security model defined in (1). Then, construct an efficient reduction $\mathcal{R}$ with the goal to solve the computational problem using adversary $\mathcal{A}$ as a subroutine. This step is illustrated in Figure 1.1.
4. Conclude that under the assumption that the computational problem is hard (i.e., no efficient algorithm can solve it), this efficient reduction $\mathcal{R}$ cannot exist. Hence, there cannot exist an efficient adversary $\mathcal{A}$ breaking the security of the cryptographic system.

As this technique arose from computational complexity theory, as discussed above, the results are mostly considered asymptotically. However, for real-world applications such asymptotic results do not give meaningful guarantees. This only yields results of the form "there exist sufficiently large parameters for our cryptographic system such that the cryptographic system is secure with respect to the defined security model". One desirable goal would be to use the full power of the security proof and use it, for example, to support the selection of parameters of the cryptographic system to actually have "theoretically-sound" parameters for deployment. Here, comes the *concrete security* (also called *exact security*) approach into play.

1.2.2 Concrete Security

In contrast to the previously outlined asymptotic approach to security, the concrete approach makes all bounds, e.g., on the running time and success probability of an adversary $\mathcal{A}$, *explicit*. In general, we deduce from a reduction $\mathcal{R}$ constructed based on some adversary $\mathcal{A}$ bounds of the form

$$\epsilon_{\mathcal{A}} \leq \ell(\mathcal{A}) \cdot \epsilon_{\mathcal{R}} \tag{1.1}$$

where $\epsilon_{\mathcal{A}}$ and $\epsilon_{\mathcal{R}}$ denotes the success probability of $\mathcal{A}$ and $\mathcal{R}$, respectively, and $\ell(\cdot)$ is a function of the adversary $\mathcal{A}$ called the *(security) loss* of reduction $\mathcal{R}$. In the asymptotic setting, it would be sufficient to show that ℓ is polynomially bounded and reduction $\mathcal{R}$ is efficient to conclude that the success probability is asymptotically small ("negligible")

and thus the scheme is considered "secure". The concrete approach focuses more on specific instances. Here, the relation between the algorithms (i.e., adversaries) is of higher importance. Also, one does not consider systems parameterized by the so-called *security parameter* determining the desired level of security. In the concrete approach, one rather considers an instance of the family of systems defined via the security parameter. For illustration, think of the security parameter to be, e.g., the length of the cryptographic key or the size of an algebraic group. In the asymptotic approach, everything is a function of this security parameter and thus asymptotically, there exists a choice of parameters (e.g., a key length) that is sufficiently large to "provide security". In contrast, concretely one rather focuses on the success probability of an adversary for a given (fixed) choice of parameters. This is much more practice-oriented as schemes obviously are deployed with fixed values for certain parameters in the real world. As such the concrete security approach also allows to deploy schemes with parameters that are actually supported by the formal security proof, as one can choose the parameters of the system such that a desired level of security is achieved. We demonstrate this with a toy example below.

Work Factors. To measure the efficiency of an adversary (resp. an algorithm in general), one can consider the *work factor* following Bellare and Ristenpart [BR09b, BR09a]. For an adversary $\mathcal{A}$ with success probability $\epsilon_{\mathcal{A}}$ and running time $t_{\mathcal{A}}$, we define the work factor of adversary $\mathcal{A}$ to be the term $\epsilon_{\mathcal{A}}/t_{\mathcal{A}}$.[6] Using the work factor, we can rewrite Equation (1.1) as follows:

$$\frac{\epsilon_{\mathcal{A}}}{t_{\mathcal{A}}} \leq \ell(\mathcal{A}) \cdot \frac{\epsilon_{\mathcal{R}}}{t_{\mathcal{R}}} \tag{1.2}$$

where $t_{\mathcal{A}}$ and $t_{\mathcal{R}}$ are the running time, and $\epsilon_{\mathcal{A}}/t_{\mathcal{A}}$ and $\epsilon_{\mathcal{R}}/t_{\mathcal{R}}$ the work factor of adversary $\mathcal{A}$ and reduction $\mathcal{R}$, respectively.

1.2.3 Tight Security and Theoretically-sound Parameters

The concrete security approach is not only interesting from a practical perspective, but also from a theoretical one as this also raises important new research questions. For example, now that all resources (e.g., running time and success probability) are made explicit, it would be desirable to get the relation of the adversary and the reduction as close as possible. Now, one may ask what the consequences of a loose reduction (or security proof) are (i.e., an overly estimated upper-bound on an adversary's success probability). Let us illustrate the impact of the security loss ℓ on the security and ultimately on the efficiency of a cryptographic scheme. Suppose a security bound as given in Equation (1.1) that relates the success of an adversary $\mathcal{A}$ against some cryptographic scheme and a reduction $\mathcal{R}$ against some conjectured-to-be-hard problem using $\mathcal{A}$ as a subroutine such that $\epsilon_{\mathcal{A}} \leq \ell(\mathcal{A}) \cdot \epsilon_{\mathcal{R}}$. Further, suppose that $\ell \leq 2^{-90}$ and we have a desired security level of 128 "bits of security" for the cryptographic scheme. That is, we want to protect the

[6] Opposed to Bellare and Ristenpart, we consider the inverse of their work factor just to avoid dividing by 0 in the somewhat artificial case in which $\epsilon = 0$. We may assume that $t > 0$ as the adversary at least needs to read its input.

cryptographic scheme against adversaries $\mathcal{A}$ such that

$$\frac{\epsilon_{\mathcal{A}}}{t_{\mathcal{A}}} \leq 2^{-128} \iff -\log_2\left(\frac{\epsilon_{\mathcal{A}}}{t_{\mathcal{A}}}\right) \geq 128.$$

Since the security proof "looses" 90 bits of security, we have to instantiate the cryptographic scheme (i.e., select its parameters) such that

$$128 \leq -\log_2\left(\ell(\mathcal{A}) \cdot \frac{\epsilon_{\mathcal{R}}}{t_{\mathcal{R}}}\right) \iff 128 \leq -90 - \log_2\left(\frac{\epsilon_{\mathcal{R}}}{t_{\mathcal{R}}}\right) \iff 218 \leq -\log_2\left(\frac{\epsilon_{\mathcal{R}}}{t_{\mathcal{R}}}\right).$$

Thus, we actually have to instantiate the underlying problem such that it provides 218 bits of security to yield a security level of 128 bits of security for the cryptographic scheme. Even though the choice of parameters providing 218 bits of security (e.g., an algebraic group of order roughly $2^{2\cdot 218}$) would yield a theoretically-sound deployment of the cryptographic scheme, it incurs a significant penalty when it comes to efficiency. In the above example we illustrated the impact of a constant (resp. concrete) loss. However, often this loss depends on parameters that might change over time, or at least are not known when deploying a scheme (e.g., the number of users using a scheme). Therefore, these values need to be estimated and this imposes new challenges. If the parameters chosen for deployment are underestimated, then there might be a point in time when the choice of parameters is not supported by the security proof anymore. If they are overestimated, then it might incur a significant loss in efficiency. Therefore, it is desirable to aim for a reduction, where the loss ℓ is a small constant (preferably independent of the parameters of the system or the adversary). Such reductions are called *tight*. Tight proofs are especially interesting in the context of real-world protocols such as authenticated key exchange (e.g., the TLS handshake) as straightforward proof techniques usually induce a loss that is a quadratic function in the number of key exchange protocol runs. In these key exchange protocols, this is particularly problematic as the number of key exchanges for such important protocols like TLS is huge. It is reasonable to assume that for TLS there are billions of users each running thousands of key exchanges (over the whole lifetime of the protocol) such that a number of users of around 2^{30} and 2^{15} key exchanges per user are realistic. This results in a number of 2^{45} many key exchanges. That is, for these numbers a loss of at least 90 bits of security needs to be compensated, which is huge.

To illustrate this practically, assume that the lossy reduction outlined in the previous paragraph is to some problem that is related to an (elliptic curve) group (in the case of TLS one can think of the Diffie–Hellman group used for the DH key exchange happening internally). According to the NIST recommendations[7], achieving 128 bits of security requires to use an (elliptic curve) group of order roughly $2^{2\cdot 128}$ (e.g., `secp256r1` [Nat13]). If one would instantiate TLS with this group and we have a security proof that looses 90 bits of security, then from a theoretical perspective our proof only guarantees $128 - 90 = 38$ bits of security, which is not even a third of the desired security level. That is, we actually would need to choose a group that is of order at least roughly $2^{2\cdot 218} = 2^{436}$ since

[7] `https://www.keylength.com/en/4/`

we computed that we need to achieve 218 bits of security of the underlying problem to compensate the loss. Of course, choosing larger groups, or increasing the system's parameters in general, comes with a significant performance penalty. Thus, it seems impractical for most real-world applications to choose parameters that are supported by the security proof. This, in particular, holds for the application of TLS as larger parameters might significantly increase the computation time, which would cause, for example, a delay of website delivery. This ultimately might increase the costs (resp. decrease the turnover) especially for major website providers. But not only "large-scale" applications might suffer from the performance penalty, also smaller Internet of Things (IoT) devices might simply not have the resources to use significantly larger parameters. Therefore, parameters in practice usually are not chosen such that they are sound with respect to a theoretical security bound. This however is undesirable in two ways. First, then there is no (provable) foundation for the selection of parameters. Secondly, the security proof is not meaningful for these kind of parameter choices and thus it merely serves as a "sanity-check" to verify the plausibility of a construction. Even though a formal security proof helps to understand the underlying problem and in fact is an excellent indicator for the plausibility of a construction's security, we believe that a security proof can be more than that and consider it desirable to enable a theoretically-sound selection of parameters in practice by giving tighter bounds of security.

1.3 Previous Analyses of TLS 1.3 and their Tightness

As already reported in Section 1.1, TLS 1.3 was the first version of SSL/TLS that was developed in close collaboration between industry and academia, which allowed for a thorough assessment of TLS 1.3 drafts before the final version was standardized. During this process a number of analyses using different kind of approaches have been published. In the following, we focus on the reduction-based (computational) analyses as we aim to assess the tightness of previous analyses. For an overview of works using different approaches, we refer to Section 1.6 below. The overall design of the TLS 1.3 handshake protocol is based on OPTLS proposed by Krawczyk and Wee [KW16]. For various drafts, security analyses for the TLS 1.3 handshake protocol [DFGS15, Koh+14, DFGS16, Li+16, FG17] and (aspects of) the record layer [Bad+15b, BT16, PS18, Del+17, GM17] have been conducted. For the final TLS 1.3 standard [Res18], Dowling, Fischlin, Günther, and Stebila [DFGS21] gave a complete analysis of all modes of the handshake protocol. For the record layer, Hoang, Tessaro, and Thiruvengadam [HTT18] revisited the analysis by Bellare and Tackmann [BT16] of the multi-user security of AES-GCM as used in the TLS record layer, and Degabriele, Govinden, Günther, and Paterson [DGGP21] extended this work to the other encryption algorithm used in the TLS 1.3 record layer, ChaCha20-Poly1305. For the handshake, none of the above analyses has a tight security proof as they all roughly have a loss that is quadratic in the total number of key exchange sessions. We will outline in Section 1.5 below where this loss comes from and why it is common in the context of key exchange protocols like the TLS handshake. For the record layer, we have tight bounds for the multi-user security of the buildings blocks

(AES-GCM and ChaCha20-Poly1305) (cf. [HTT18, BT16, DGGP21]), but there is a lack of a precise security model capturing all aspects of the secure channel implemented by the record layer protocol. (We discuss this in more detail in Chapter 11.)

The above considerations naturally raise the following research question:

How tightly-secure is TLS 1.3?

1.4 Contributions of this Work

In this thesis, we address the above research question and give major insights on the tight security of TLS 1.3. Here, we mainly focus on formally proving tight security for the TLS 1.3 handshake protocol. Concretely, we present the following results in this thesis.

Abstracting the TLS 1.3 Key Schedule. In a first step, we describe a new abstraction of the TLS 1.3 key derivation function (called the *key schedule* [Res18, Sect. 7.1]) used in the TLS handshake. During the TLS 1.3 handshake different intermediate values are derived to ultimately derive multiple keys for various purposes. We abstract the derivation of these intermediate values such that the key schedule only consists of a set of functions TKDF_x, where each of these functions only derives the key x. With this abstraction at hand, we formally prove using the indifferentiability framework [MRH04, BDG20] that each of these functions behaves like an independent (publicly available) ideal random function. This holds under the assumption that the hash function used in TLS 1.3 behaves like an ideal random function. This abstraction reflects the natural intuition of what one would expect from the keys derived in the key schedule, namely that they are "as good as" independently chosen random cryptographic keys. Furthermore, it reduces the complexity of the key derivation in the handshake protocol. Since the key schedule as defined in the standard [Res18, Sect. 7.1] is a complex procedure, which involves a number of interleaved computations. This ultimately also reduces the complexity of subsequent analyses of the handshake protocol.

Tight Security of the TLS 1.3 Handshake. The TLS 1.3 handshake comes in two variants. First, there is the "standard" full handshake that uses authentication based on public key certificates. At its core it uses the prominent ephemeral Diffie–Hellman key exchange (DHE) [DH76] based on a finite field or a elliptic-curve (EC) group. Therefore, this variant is called the *TLS 1.3 full (EC)DHE handshake*. Second, there are the pre-shared key handshakes. This variant uses that client and server can exchange an additional key called the *pre-shared key (PSK)* after the full handshake completed to perform an abbreviated handshake at a later point in time (*session resumption*). There is also the option that this PSK is established externally ("out-of-band"). In the PSK handshakes, authentication is essentially skipped as the knowledge of the PSK already provides a certain degree of (implicit mutual)

authentication. This variant comes in two modes: the *PSK-only* mode and the *PSK-(EC)DHE* mode. In the latter there is an additional DH key exchange performed to provide higher grade of security.

In this work, we give a tight security bound for all of the handshake modes reducing its security to the used building blocks. Here, the security of the full handshake reduces to the security of the digital signature scheme (unforgeability in the multi-user setting) used for authentication and the hardness of the strong Diffie–Hellman problem [ABR01] under the assumption that the TLS hash function is modeled as a publicly available ideal random function. The security of the PSK-only handshake holds under the assumption that the TLS hash function is a publicly available ideal random function. The security of the PSK-(EC)DHE handshake reduces to the hardness of the strong Diffie–Hellman problem [ABR01] under the assumption that the TLS hash function is modeled as a publicly available ideal random function. In our tight analysis, we leverage our abstraction of the key schedule to tame the complexity of the overall proof and to present it in a modular fashion. Following most of the previous computational analyses of TLS 1.3, we prove security using a variant of the multi-stage key exchange (MSKE) security model originally proposed by Fischlin and Günther [FG14] capturing complex properties of modern key exchange protocols.

In early work, the author of this thesis contributed to a tight analysis of the full handshake of TLS 1.3 [DJ21], which proved a tight bound under strong assumptions of the key derivation function of TLS 1.3. There also was independent and concurrent work by Davis and Günther [DG21a], which also gave a tight bound for the full handshake and also made similar assumptions about the key derivation function of TLS 1.3. Additionally, both neglected dependencies between different uses of the hash function in the key derivation function and the remaining TLS handshake protocol. To resolve this, the authors of [DJ21, DG21a] jointly developed a solution to these problems, which resulted in the abstraction of the key schedule and the tight analysis of the PSK handshake given in Chapters 7 and 10 of this work, respectively. These results were originally published in [DDGJ22b]. For details, we refer to Chapter 7. The analysis of the full handshake presented in this work is under weaker assumptions compared to the previous results [DJ21] and incorporates our new abstraction of the key schedule similar to the PSK handshake analyses in [DDGJ22b]. The results of [DJ21, DG21a] therefore are only implicitly part of this thesis.

Discussion of Tightness of the TLS 1.3 Record Protocol. In order to complete the treatment of TLS 1.3, we also (informally) discuss the tightness of record protocol. The MSKE security model in some versions [DFGS15, Gün18] contains a composition theorem for TLS, which proves generically security of the composition of TLS with an arbitrary, secure symmetric-key protocol (e.g., the record layer protocol keyed with the application traffic encryption key established in the handshake). Unfortunately, this composition theorem does not apply (directly) to the variant

of the MSKE model we are considering in this work. This is due to advanced properties of authentication the model captures. Furthermore, Dowling, Fischlin, Günther, and Stebila [DFGS21] using a similar variant of the MSKE model already pointed out that the above composition theorem does not necessary hold for these properties. Even though it is possible to prove the theorem given in [DFGS15, Gün18] tightly (cf. [DJ21]), a complete computational analysis of the record layer remains elusive mainly because there is no suitable security model. We discuss what we consider possible in that regard.

More Efficient Digital Signatures with Tight Multi-user Security. In the TLS 1.3 handshake, parties can choose among four different signature schemes: RSA-PSS [MKJR16, Cor02], RSA-PKCS#1 v1.5 [Kal98, MKJR16], ECDSA [JMV01], and EdDSA [JL17, Ber+11]. Due to the fact that RSA-based public keys are the most common in practice, the RSA-based schemes currently have the greatest practical relevance in the context of TLS 1.3. As usual in tightly-secure authenticated key exchange (AKE) [Bad+15a, GJ18, JKRS21, Han+21], we require in our tight security bound for the full handshake *existential unforgeability in the multi-user setting with adaptive corruptions* [Bad+15a] from the signature scheme used for authentication. To achieve a fully tight bound for the full handshake, we also need tightly-secure building blocks. Otherwise, even though the bound of the handshake seems to be tight, the overall bound will not be tight. For the signature scheme, two dimensions are relevant for tightness, (i) the number of signatures issued per user, and (ii) the number of users.

- RSA-PSS has a tight security proof in the number of signatures per user [Cor02, Kak19], but not in the number of users.
- RSA-PKCS #1 v1.5 also has a tight security proof [JKM18] in the number of signatures per user, but not in the number of users. However, we note that this proof requires to double the size of the modulus, and also that it requires a hash function with "long" output (about half of the size of the modulus), and therefore does not immediately apply to TLS 1.3.
- For ECDSA there exists a security proof [FKP17] that considers a weaker "one-signature-per-message" security experiment that is not tight.

All of these signatures have unique secret keys in the sense of [BJLS16], and thus according to Bader, Jager, Li, and Schäge [BJLS16] it is impossible for these schemes to be a tightly-secure in the number of users.

Since none of the standardized schemes is tightly-secure, we present an efficient signature scheme that is tightly (strong) existential unforgeability in the multi-user setting with adaptive corruptions. The construction is generic and it can be instantiated with any lossy identification scheme [AFLT12] satisfying certain properties.

1.5 Difficulty of Tightly-secure Authenticated Key Exchange and Signatures in the Multi-user Setting

Classical security models for authenticated key exchange (such as the widely-used Bellare–Rogaway [BR94] or the Canetti–Krawczyk [CK01] model and their countless derivatives, e.g., the MSKE model) try to reflect the real world as best as possible and thus are in the multi-user setting. This means they consider multiple users that can interact with each other and each of these users usually holds a (long-term) key, which classically corresponds to a key pair of a digital signature scheme for authentication. Moreover, these models consider very strong adversaries that have full control over the communication network and therefore, are able to modify, drop, or inject messages. In addition, the adversary can corrupt users by revealing their long-term secrets (e.g., the signature secret key) and also reveal established session keys. The overall objective in this setting is then for the adversary to not being able to distinguish real session keys established by the users running the protocol from uniformly random keys. Giving security proofs in such a model is already challenging due to the model's mere complexity, let alone achieving a tight security proof.

Guessing the adversary's behavior induces loss. Usually the security loss of a reduction is induced by guessing the adversary's behavior. For example, a common strategy in the multi-user setting is to guess a user that remains uncorrupted until the end of the security game to embed the reduction's challenge, so that the reduction can extract a valid solution for its challenge. This results in the reduction only winning with the same probability as the adversary if the guess was correct. For the example, this then would result in a success probability $\epsilon_{\mathcal{R}} \leq 1/\ell \cdot \epsilon_{\mathcal{A}}$, where ℓ is linear in the number of users. To avoid such guessing arguments, a reduction always needs to be prepared to extract a solution of its challenge for every possible behavior of the adversary. In the above example, this means a tight reduction would need to be able to extract a solution for *any* user and thus of course embed a (variant) of the challenge in any user. Next, let us have a look at why avoiding such guessing arguments is particularly important in the context of authenticated key exchange protocols.

Guessing in AKE proofs and the commitment problem. Nearly all classical security proofs for authenticated key exchange have a quadratic loss (in the number of key exchange sessions). The reason for this is a guessing argument to solve the "commitment problem" as it was called by Gjøsteen and Jager [GJ18]. In key exchange the challenge of the adversary is to distinguish exchanged keys from random keys in sessions where the adversary, informally, neither revealed the exchanged key nor corrupted any of the parties to exclude trivial attacks. These instances usually are referred to as *fresh*. A reduction to some computational problem now needs to embed its challenge into such a key exchange to be able to extract a solution if the adversary successfully distinguishes the key exchange in a fresh instance. However, let us illustrate the idea of the commitment problem using the Diffie–Hellman key exchange, which most of modern key exchange

protocols build upon. The protocol is a two-party and two-message protocol, where one party π sends a group element $A = g^a$ and the other party π' sends $B = g^b$ such that the established key is $B^a = g^{ab} = A^b$. The reduction now would get a (decisional) Diffie–Hellman challenge (A, B, C) and embeds A in a party π. Recall that the decisional Diffie–Hellman problem asks to decide whether $\mathrm{dlog}_g(C) = ab$ holds given $A = g^a$ and $B = g^b$. Now, the reduction "committed" to not knowing the discrete logarithm of A and thus the party π can only compute a key if it receives B. Even though there is another party π' that receives A and the reduction can embed B there, π' cannot compute its key as usual, but the reduction can embed the challenge C as the established key. If the adversary wants to be challenged on that key exchange, the reduction can output C and an adversary that distinguishes real from random keys will decide the problem for the reduction. However, the adversary still could tamper with the value sent to π (due to the strong key exchange model, in which the adversary has full control over the network) and send a group element B' to party π. If the adversary now asks π to reveal its key, the simulation embedding A would be discovered as the reduction cannot compute B'^a due to the lack of a (and the adversarially chosen b' such that $B' = g^{b'}$). To resolve this, one usually guesses the parties (π, π') exchanging the key that the adversary at the end wants to be challenge on and thus the reduction is guaranteed that embedding a challenge is safe, if the guess is correct, because the adversary cannot reveal the challenge key. Unfortunately, this results in a quadratic loss in the number of key exchanges, since the probability to guess the correct parties is $1/s^2$ where s is the number of key exchange instances run. To circumvent this a reduction needs to be able to embed a challenge in a user and at the same time still being able to reveal the corresponding keys.

Resolving the commitment problem. Recent approaches to tightly-secure authenticated key exchange such as [GJ18] resolve the commitment problem using a carefully chosen construction. We believe that the approach of "tight security by design" is the way going forward, however this is not an option for (existing) real-world protocols like TLS 1.3 as their construction is fixed. For these constructions, one needs to find clever proof techniques that leverage different aspects. In this work, we leverage the technique recently proposed by Cohn-Gordon et al. [Coh+19], which we discuss more in detail when we present our tight proofs in Chapters 9 and 10.

Handling adaptive corruptions of users. Another challenge is to handle adaptive corruption of users. As already mentioned above, a user in a key exchange protocol usually holds a long-term public-secret-key pair corresponding to a signature scheme. Bader et al. [Bad+15a] identified in the context of the first tightly-secure authenticated key exchange protocol that the security notion that reflects the requirements of a signature used in key exchange most realistically is *existential unforgeability in the multi-user setting with adaptive corruptions*. Namely, a key exchange protocol is also in a multi-user setting and a corruption in a key exchange protocol directly translates to a corruption of an instance the signature scheme. This notion tames the complexity of the actual key exchange proof, because the a reduction to signature security becomes straightforward.

However, this basically only moves a subset of the challenges of a tight security proof to the signature scheme. Nevertheless, this modular approach allows for less complex individual proofs of the key exchange protocol and the signature scheme. Unfortunately, single-user unforgeability implies multi-user unforgeability with adaptive corruptions only with a loss that is linear in the number involved in the system using a simple guessing argument, in which the reduction guesses the user for which the adversary will output a forgery attempt for. Following Bader, Jager, Li, and Schäge [BJLS16], this seems to be unavoidable for a large class of signature schemes (signatures with unique secret keys in the sense of [BJLS16]). To circumvent this impossibility, and prove tight multi-user security for a signature scheme, one has to avoid such a guessing argument while at the same time solving the following paradoxical situation. Namely, in such a reduction one needs to know all secret keys to be able to respond to any possible corruption to avoid guessing uncorrupted users. At the same time, this reduction also needs to be able to extract a solution to some computational problem from a valid forgery attempt, while knowing the secret key to such an instance of the problem.

Solving the paradox of tightly multi-user-secure signatures. To circumvent this paradox, we leverage a technique already leveraged in previous tightly secure signatures [Bad+15a, GJ18] in the multi-user setting, which one could refer to as the "double signature" approach. Here, a signature $\sigma = (\sigma_0, \sigma_1)$ consist of two components a "real" one σ_b and a "fake" one σ_{1-b}. The same holds for the public key $pk = (pk_0, pk_1)$ for the real public key pk_b, we know a secret key sk_b and for the fake one pk_{1-b}, we do not. The bit b is chosen uniformly at random for every new user individually. Therefore, the adversary does not know which of the components is real and which is fake. Upon a corruption, we always output the real secret key sk_b and embed our challenge always in the fake component $(1-b)$. Note that this only leaks the bit b of the corrupted user. Since, real and fake public keys should be indistinguishable, the adversary will forge with probability $1/2$ with respect to the fake public key pk_{1-b} allowing us to extract a solution with probability $1/2 \cdot \Pr[\text{adversary forges}]$. Unfortunately, this approach does not work for most signature schemes. As they usually lack a method to have "real" and "fake" public keys. Mostly, this feature is implied by the possibility to simulate signature in some way without the use of a secret key. We do implement this using lossy identification schemes [AFLT12], where a "real" key is a normal key and a "fake" key is a lossy key.

1.6 Further Related Work

In this section we present further related work that we did not mention before in this chapter.

Analyses of TLS prior to TLS 1.3. Even though there has not been any involvement of academia in the actual standardization process of previous version of TLS, the public attention of TLS attracted a number of analyses of the final standards, particu-

larly, TLS 1.2 [KN09]. There have been early works [MSW08, Gaj08, JK02] on (slightly) modified versions of the handshake protocol. Another early work [Gaj+08] discusses security of the unauthenticated handshake in the Universally Composable (UC) Framework [Can01]. In 2012, Jager, Kohlar, Schäge, and Schwenk [JKSS12] gave the first full analysis of the DH-based key exchange of TLS. To this end, they introduced a new model, which they called Authenticated and Confidential Channel Establishment (ACCE) tailored directly for the use of TLS-like protocols. ACCE merges a Bellare–Rogaway-like [BR94] authenticated key exchange model with the encryption notion of stateful length-hiding authenticated encryption to capture the record layer encryption introduced by Paterson, Ristenpart, and Shrimpton [PRS11]. Unfortunately, TLS prior to version 1.3 used the record layer encryption key already to encrypt the last messages of the handshake. Therefore, indistinguishability-based key exchange security is impossible to achieve for these versions as an adversary always can try to decrypt handshake messages with its challenge key to determine whether it receives a real or random key as challenge. Therefore, a modular proof of the TLS 1.2 handshake protocol as standardized is not possible in a standard key exchange model. The work by Jager, Kohlar, Schäge, and Schwenk was extended to further key exchange modes and mutual authentication by Krawczyk, Paterson, and Wee [KPW13] and Kohlar, Schäge, and Schwenk [KSS13]. Giesen, Kohlar, and Stebila [GKS13] analyzed the renegotiation of TLS, i.e., the interaction of multiple handshake protocol runs in the (extended) ACCE setting. Li et al. [Li+14] analyzed the pre-shared key cipher suites of TLS [ET05] in the ACCE setting. Dowling and Stebila [DS15] treated the often neglected negotiation of cipher suites and versions formally for TLS.

While most of the above analyses are "classical" reduction-based analyses, there also is the team around miTLS[8], which published the miTLS reference implementation of TLS [Bha+13]. They combined formal software verification with the tools of provable security to analyze TLS [Bha+14b].

Attacks on TLS prior to Version 1.3. Various aspects of the TLS protocol were target of attacks in the past years. For example, there have been attacks on the following aspects:

- RC4 [AlF+13, VP15, Man15]
- Hash functions, e.g., SLOTH [BL16b], [WY05, LW05, SLW07, Ste+09]
- DES/3DES, e.g., Sweet32 [BL16a]
- RSA PKCS#1 v1.5 [Ble98, Mey+14]
- CBC encryption and padding, e.g., BEAST [Duo11], POODLE [Bod14], Lucky13 [AP13]
- Support for older versions: [JSS15], DROWN [Avi+16]
- Parameter negotiation/downgrade to weaker cryptography: [WS96], Logjam [Adr+15], Freak [Beu+15]

[8] `https://www.mitls.org`

- Compression: CRIME [RD12], BREACH [PHG13], HEIST [VG16]
- Triple-handshake attack [Bha+14a]
- Flaws in implementations, e.g., Heartbleed [Cod14], SMACK [Beu+15], ROBOT [BSY18]

Other Analyses of TLS 1.3. Above we already named a number of reduction-based analyses of the TLS 1.3 protocol, but this classical provable security approach is not the only type of analysis. Cremers, Horvat, Scott, and van der Merwe [CHSv16] and Cremers et al. [Cre+17] conducted an automated analysis of drafts (10 and 21, respectively) of the TLS 1.3 standard using the Tamarin[9] prover. There also were analyzed combining formal verification and computational analysis [BBK17, Del+17]. Even though the other kind of approaches are out-of-scope of this work, they are not less important for understanding security of TLS 1.3.

In some works specific aspects of the TLS protocol were analyzed. For example, Bhargavan et al. [Bha+16] studied the downgrade resilience of TLS 1.3, Krawczyk [Kra16] studied client authentication, and Fischlin, Günther, Schmidt, and Warinschi [FGSW16] initiated a formal treatment of key confirmation in key exchange and its application to TLS 1.3. Further, Brendel, Fischlin, and Günther [BFG19] studied the breakdown resilience of different real-world protocol including TLS 1.3. Drucker and Gueron [DG21b] present a reflection attack on the TLS 1.3 PSK mode (called "Selfie"), which breaks mutual authentication in this mode. Arfaoui et al. [Arf+19] study the privacy of TLS 1.3, since improving the privacy was an important design goal of TLS 1.3. Aviram, Gellert, and Jager [AGJ19, AGJ21] proposed an improvement for the TLS 1.3 session resumption feature to achieve forward security efficiently for all messages including the zero round-trip time data that in the original version of TLS 1.3 are not forward secure.

Due to the rising interest in post-quantum cryptography, Schwabe, Stebila, and Wiggers [SSW20] developed a protocol called KEM-TLS, which tries to serve as a proposal for a future post-quantum-secure replacement for TLS using KEMs instead of signatures for authentication. In the series of follow-up works [SSW21, GRTW22, GRTW21, CHSW22, Cel+21] the KEM-TLS protocol has been further analyzed and extended by various aspects.

Tightly-secure Digital Signatures in the Multi-user Setting. The first tightly secure signature with adaptive corruptions in the multi-user setting, was proposed by Bader et al. [Bad+15a]. Unfortunately, their main signature scheme (i.e., the one with a tight reduction) is based on a tree-based construction from [HJ12] and thus results in rather large signatures (linear in the security parameter many group elements). There is a second construction in their paper with constant-size signatures, but the construction is only "almost tight", i.e., it has a security loss linear in the security parameter. Both constructions are pairing-based and in the standard model. Han et al. [Han+21] identified a subtle gap in the proof of the second, constant-sized signature variant by Bader et al.

[9] `https://github.com/tamarin-prover/tamarin-prover`

[Bad+15a, Sect 2.3]. As they were not able to close this gap, they provided a new variant, which they prove tightly secure in the multi-user setting with adaptive corruptions. The scheme still is pairing-based and in the standard model, and used the pairing-based, tightly-secure hierarchical identity-based encryption (HIBE) scheme of Langrehr and Pan [LP19] as a blueprint to correct the proof. Another scheme was proposed by Gjøsteen and Jager [GJ18] it has constant-size signatures and the security proof is tight in the random oracle model. Pan and Wagner [PW22] presented the first compact lattice-based signature scheme that is tightly secure signature with adaptive corruptions in the multi-user setting. Their construction builds upon the generic construction presented in this work, originally published in [DGJL21b].

1.7 Outline of this Thesis

The remainder of this thesis is divided into four parts, in which we discuss the following.

Part I: Preliminaries. In the first part, we introduce fundamental concepts and notions used throughout this thesis. To this end, we introduce basic notation in Chapter 2. Additionally, we define in Chapter 3 computational problems, and in Chapter 4 cryptographic building blocks relevant for this thesis.

Part II: On the Tightness of the TLS 1.3 Handshake Protocol. In the second part, we discuss the tightness of the TLS 1.3 handshake protocol. To this end, we first define in Chapter 5 the notion of multi-stage key exchange (MSKE) protocols. Then, we describe the TLS 1.3 handshake protocol in Chapter 6. With these two important prerequisites in place, we start the journey to prove a tight security bound for the TLS 1.3 handshake protocol in the MSKE model. Here, we first present our new abstraction of the TLS 1.3 key schedule in Chapter 7 and present in Chapter 8 an abstraction for the TLS 1.3 handshake encryption. Finally, we present in Chapter 9 our tight security analysis for the TLS 1.3 full handshake and in Chapter 10 our tight security analysis for the TLS 1.3 PSK handshakes.

Part III: On the Tightness of the TLS 1.3 Record Protocol. In the short third part, we discuss in Chapter 11 the current state of the tight security of the TLS 1.3 record layer protocol. Even though, we do not give a formal treatment, we point to interesting open questions for future work.

Part IV: More Efficient Digital Signatures with Tight Multi-User Security. In the fourth part, we present our efficient digital signature scheme with tight multi-user security with adaptive corruptions. Here, we present our generic construction based on lossy identification schemes in Chapter 13 and present two instantiations for this scheme in Chapter 14.

Finally, we summarize our results in Chapter 16.

Part I

Preliminaries

2

NOTATION

In this chapter, we introduce notation used throughout this thesis. This includes basic notation for numbers, strings, algorithms, cryptographic schemes and security experiments.

Numbers. Let $\mathbb{N}$ denote the set of natural numbers and let $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$. For an integer $n \in \mathbb{N}$, we denote the set of integers ranging from 1 to n by $[n] := \{1, \dots, n\}$. Let $\mathbb{Z}$ denote the set of integers and for $N \in \mathbb{N}$ let $\mathbb{Z}_N$ denote the ring of integers modulo N. For any set $X = \{x_1, x_2, \dots\}$, we use $(v_i)_{i \in X}$ as a shorthand for the tuple $(v_{x_1}, v_{x_2}, \dots)$.

Strings. Let Σ be an *alphabet* (i.e., a non-empty finite set of symbols) and let $s \in \Sigma^*$ be a *string* (i.e., a finite sequence of symbols from the alphabet). The *length of a string* s is denoted by $|s|$, and the string of length 0 is called the *empty string* denoted by ε. For symbols $a, b \in \Sigma$, we denote the concatenation of these symbols by $a \parallel b := ab$. Analogously, we denote the concatenation of two strings $s, t \in \Sigma^*$ by $s \parallel t$. In this thesis, if not stated otherwise, we focus on *bit strings*, i.e., $\Sigma = \{0, 1\}$.

Random Variables and Algorithms. We denote the operation of assigning a value y to a variable x by $x := y$. If S is a finite set, we denote by $x \xleftarrow{\$} S$ the operation of sampling a value uniformly at random from set S and assigning it to variable x. For a deterministic algorithm $\mathcal{A}$, we write $x := \mathcal{A}(y_1, y_2, \dots)$ to denote that $\mathcal{A}$ on inputs $y_1, y_2, \dots$ outputs x. In case $\mathcal{A}$ is probabilistic, we overload notation and write $x \xleftarrow{\$} \mathcal{A}(y_1, y_2, \dots)$ to denote the random variable x that takes on the value output by algorithm $\mathcal{A}$ ran on inputs $y_1, y_2, \dots$ with fresh random coins. Sometimes we also denote this random variable simply by $\mathcal{A}(y_1, y_2, \dots)$. We write $\mathcal{A}^{\mathcal{O}(\cdot)}$ if algorithm $\mathcal{A}$ has oracle access to some other algorithm $\mathcal{O}$.

Schemes. In cryptography, we often call combinations of algorithms for a specific task a *(cryptographic) scheme*. For example, a classical digital signature scheme, which we formally introduce in Section 4.4, consists of three algorithms: a key generation algorithm, a signing algorithm and a verification algorithm. For a scheme $\mathsf{Scheme} =$

(Algo1, Algo2, ...), we write Scheme.Algo1 to clarify that we refer to Algo1 of Scheme if the context might be ambiguous.

Lower-level cryptographic algorithms that are usually used a building block for cryptographic schemes are often called a *(cryptographic) primitives*. Examples for cryptographic primitives are (cryptographic) hash function and pseudorandom functions introduced in Chapter 4.

Security Experiments and Advantage. To formally define security for a cryptographic scheme (e.g., a digital signature scheme), we define a *security experiment*, or synonymously a *security game*, that reflects the security goal we want to define. In such an experiment, we challenge a (potentially malicious) algorithm called the *adversary* $\mathcal{A}$ to break the security of the cryptographic scheme by winning the experiment. Intuitively, the experiment is an algorithm that sets up an environment for the adversary $\mathcal{A}$, runs the adversary as a subroutine and checks whether the adversary satisfies some well-defined winning condition. For a security goal GOAL and a scheme Scheme, we denote the security experiment for GOAL with respect to Scheme and an adversary $\mathcal{A}$ by $\mathrm{Exp}^{\mathsf{GOAL}}_{\mathsf{Scheme}}(\mathcal{A})$. The experiment always produces an output that is either 0 or 1 indicating whether the adversary looses or wins the experiment. As an example, a classical security goal for a digital signature scheme is *existential unforgeability under an adaptive chosen-message attack* (EUF-CMA) formally defined in Figure 4.3 and Definition 4.6. Here, GOAL = EUF-CMA and in the experiment the adversary is challenged to output a digital signature for any (new) message without knowing the secret signing key even if the experiment provides digital signature for messages of the adversary's choice. The experiment handles the key generation, executes secret-key-operations for the adversary and checks whether the digital signature output in the end satisfies the winning condition.

To measure the success in breaking the security of a scheme, we define the *advantage* of an adversary, which we write as $\mathrm{Adv}^{\mathsf{GOAL}}_{\mathsf{Scheme}}(\mathcal{A})$ for the previous example of experiment $\mathrm{Exp}^{\mathsf{GOAL}}_{\mathsf{Scheme}}(\mathcal{A})$. The advantage is always defined with respect to some security experiment and depends on this security experiment. Common choices for the advantage are the following:

- The *bit-guessing advantage*: $\mathrm{Adv}^{\mathsf{GOAL}}_{\mathsf{Scheme}}(\mathcal{A}) := |\Pr[\mathrm{Exp}^{\mathsf{GOAL}}_{\mathsf{Scheme}}(\mathcal{A}) = 1] - \frac{1}{2}|$. This is a common choice if the security notion GOAL represents a decision problem (i.e., the solution to the problem is binary). Here, the challenge of the adversary is usually to guess a bit chosen at the beginning of the experiment. The advantage captures how much better the adversary performs compared to random guessing. A variant of this is $\mathrm{Adv}^{\mathsf{GOAL}}_{\mathsf{Scheme}}(\mathcal{A}) := 2\Pr[\mathrm{Exp}^{\mathsf{GOAL}}_{\mathsf{Scheme}}(\mathcal{A}) = 1] - 1$, which scales the range of the advantage to the interval $[0, 1]$ as opposed to the interval $[0, \frac{1}{2}]$ for the former notation.
- For search problems, i.e., problems where the adversary is challenged to output an arbitrary string, e.g., a digital signature as in the above example of EUF-CMA, it is common to set
$$\mathrm{Adv}^{\mathsf{GOAL}}_{\mathsf{Scheme}}(\mathcal{A}) := \Pr[\mathrm{Exp}^{\mathsf{GOAL}}_{\mathsf{Scheme}}(\mathcal{A}) = 1].$$
Here, the advantage directly measures the success probability of the adversary.

3

COMPUTATIONAL PROBLEMS

Contents

In this chapter, we introduce basic computational problems that we use for the security results in this thesis. In particular, we introduce problems based on the discrete logarithm problem and the related Diffie–Hellman problems. All of the problems presented in this chapter are asymptotically assumed to be hard. That is, no efficient algorithm (resp. adversary) can solve the problems with overwhelming probability. Since we only consider the advantages of algorithms concretely in this work, we only formally define the respective problem and the corresponding advantage. However, sometimes we might refer to a computational problem and the corresponding *(computational hardness) assumption* synonymously in an informal context.

3.1 Discrete Logarithm Problem

The majority of computational problems considered in this work are related to the problem of computing discrete logarithms (DLOGs). Thus, let us first formally introduce the notion of a DLOG and subsequently define the related problem.

Definition 3.1 (Discrete Logarithm). Let $\mathbb{G}$ be a cyclic group of prime order p, let g be a generator of $\mathbb{G}$ and let $h \in \mathbb{G}$. We call the unique element $a \in \mathbb{Z}_p$ with $h = g^a$ the *discrete logarithm (DLOG) of h with respect to base g* and write $a = \mathrm{dlog}_g(h)$.

Definition 3.2 (Discrete Logarithm Problem). Let $\mathbb{G}$ be a cyclic group of prime order p and let g be a generator of $\mathbb{G}$. We denote the *advantage of an adversary $\mathcal{A}$ in solving the DLOG problem for* $\mathbb{G}$ by

$$\mathrm{Adv}^{\mathsf{DLOG}}_{\mathbb{G},g}(\mathcal{A}) := \Pr[a \xleftarrow{\$} \mathbb{Z}_p \,:\, \mathcal{A}(g^a) = a].$$

3.2 Computational Diffie–Hellman Problem

In their seminal work, Diffie and Hellman [DH76] laid the foundation of public key cryptography. In this context, they proposed the famous Diffie–Hellman key exchange (DHE) protocol, which most of modern key exchange relies on.

Diffie–Hellman key exchange. The basic idea of the protocol is simple. Given a group $\mathbb{G}$ with generator g, two parties A and B that want to exchange a key, choose a number a and b, respectively, uniformly and independently at random. Then, A computes $K_A := g^a$ and B computes $K_B := g^b$, and both send their computed value to their partner. Upon receiving their partner's value, A computes K_B^a and B computes K_A^b. It is easy to see that (assuming no tampering occurred) that the two parties A and B will end up computing the same key $K_{AB} := K_B^a = K_A^b = g^{ab}$. We often refer to $K_{AB} = g^{ab}$ as the Diffie–Hellman (DH) key. The key K_{AB} is usually not directly used, rather the parties deterministically derive the actual key from K_{AB}, e.g., using a hash function (Section 4.1) or a pseudorandom function (Section 4.2)).

A passive adversary that tries to recover the exchanged key K_{AB}, faces the following problem referred to in the literature as the *computational Diffie–Hellman (CDH)* problem: For a fixed group $\mathbb{G}$ with generator g, given g^a and g^b, where $a, b \xleftarrow{\$} \mathbb{Z}_p$, the challenge is to compute g^{ab}.

Definition 3.3 (Computational Diffie–Hellman Problem). Let $\mathbb{G}$ be a cyclic group of prime order p and let g be a generator of $\mathbb{G}$. We denote the *advantage of an adversary $\mathcal{A}$ in solving the CDH problem for* $\mathbb{G}$ by

$$\mathrm{Adv}^{\mathsf{CDH}}_{\mathbb{G},g}(\mathcal{A}) := \Pr[a, b \xleftarrow{\$} \mathbb{Z}_p \,:\, \mathcal{A}(g^a, g^b) = g^{ab}].$$

Relation between CDH and DLOG. It is easy to see that the assumption that the CDH problem is hard is a stronger assumption than the one that the DLOG problem is hard. This follows from a straightforward reduction. Informally, an algorithm that computes discrete logarithms can be used to solve CDH by simply computing the discrete logarithms $a = \mathrm{dlog}_g(g^a)$ and $b = \mathrm{dlog}_g(g^b)$, and outputs g^{ab}. Whether the two problems are equally hard unfortunately remains unknown in general. However, for some special cases this is true (cf. [Mau94, MW96]).

3.3 Decisional Diffie–Hellman Problem

Even though, CDH might be the obvious choice to capture security of the DHE protocol, it does not reflect the actual security requirements we have for a key exchange protocol. Namely, in the real-world it might even be a problem if the adversary gets any information about the exchanged key. Hence, full recovery of the key might be too weak for real-world applications. To capture this better, one rather considers the decisional Diffie–Hellman (DDH) problem, which was first considered by Brands [Bra93]. The problem states that for a fixed group $\mathbb{G}$ with generator g, given (g^a, g^b) and either g^{ab} or g^c,

where $a, b, c \xleftarrow{\$} \mathbb{Z}_p$, the adversary has to tell whether it received g^{ab} or g^c. Informally, this captures that it should be hard for the adversary to tell a DH key g^{ab} from a random group element apart.

Definition 3.4 (Decisional Diffie–Hellman Problem). Let $\mathbb{G}$ be a cyclic group of prime order p and let g be a generator of $\mathbb{G}$. We denote the *advantage of an adversary $\mathcal{A}$ in solving the DDH problem for* $\mathbb{G}$ by

$$\mathrm{Adv}^{\mathsf{DDH}}_{\mathbb{G},g}(\mathcal{A}) \coloneqq \Big|\Pr[a, b \xleftarrow{\$} \mathbb{Z}_p : \mathcal{A}(g^a, g^b, g^{ab}) = 1] - \Pr[a, b, c \xleftarrow{\$} \mathbb{Z}_p : \mathcal{A}(g^a, g^b, g^c) = 1]\Big|.$$

Relation between CDH and DDH. As we have seen before, the assumption that the CDH problem is hard is a stronger assumption than the assumption that the DLOG problem is hard. The assumption that the DDH problem is hard, now in turn, is stronger than the assumption that the CDH problem is hard. This follows again from a straightforward reduction. Informally, given (g^a, g^b, h) and an algorithm solving the CDH problem, one can easily decide DDH by computing the DH key g^{ab} from g^a and g^b and comparing it to its input h.

3.4 Strong Diffie–Hellman Problem

Abdalla, Bellare, and Rogaway [ABR01] introduced a problem that is related to the DH problems, which they call the *strong Diffie–Hellman (SDH)* problem. The problem states that for a fixed group $\mathbb{G}$ with generator g, given (g^a, g^b) and oracle access to an algorithm DDH_a, the adversary has to compute g^{ab}. Here, DDH_a for a fixed value $a \in \mathbb{Z}_p$ is the function that on input (g^y, g^z) outputs 1 iff. $z = ay \bmod p$, where p is the prime order of $\mathbb{G}$. Formally, we define the SDH problem as follows.

Definition 3.5 (Strong Diffie–Hellman Problem). Let $\mathbb{G}$ be a cyclic group of prime order p and let g be a generator of $\mathbb{G}$. Further, let $\mathrm{DDH}_a(\cdot,\cdot)$ for $a \in \mathbb{Z}_p$ denote the oracle that on input $(g^y, g^z) \in \mathbb{G}^2$ outputs 1 iff. $z = ay \bmod p$. We denote the *advantage of an adversary $\mathcal{A}$ against the strong Diffie–Hellman (SDH) assumption* by

$$\mathrm{Adv}^{\mathsf{SDH}}_{\mathbb{G},g}(\mathcal{A}) \coloneqq \Pr[a, b \xleftarrow{\$} \mathbb{Z}_p : \mathcal{A}^{\mathrm{DDH}_a(\cdot,\cdot)}(g^a, g^b) = g^{ab}].$$

The SDH problem is a variant of the *gap Diffie–Hellman (GDH)* problem [OP01]. Okamoto and Pointcheval [OP01] introduced *gap problems*[1] as a new family of problems. Intuitively, a gap problem is to solve some computational (search) problem (e.g., CDH) with the help of an oracle for a related decision problem (e.g., DDH). In this context, they also defined the *GDH* problem. This problem is defined as the SDH problem except that the adversary here gets oracle access to a more general oracle DDH. Namely,

[1] The family of gap problems introduced by Okamoto and Pointcheval [OP01] should not be confused with the gap problems from computational complexity theory.

DDH is the function that on input (g^x, g^y, g^z) outputs 1 iff. $z = xy \bmod p$, where p is the prime order of $\mathbb{G}$, i.e., DDH_a is a special case, where the first component is fixed to g^a.

Intuitively, the GDH problem is hard in groups in which the DDH problem is easy, but the CDH problem still remains hard. Also, it is easy to see that the assumption that the GDH problem is hard is a stronger assumption than the assumption that the SDH problem is hard.

In this work, the SDH assumption will be the leverage to prove tight security for the TLS 1.3 handshake protocol in Chapters 9 and 10. The technique we use is due to Cohn-Gordon et al. [Coh+19] and it is in the random oracle model (introduced in Section 4.1).

4

CRYPTOGRAPHIC BUILDING BLOCKS

Contents

In this chapter, we define well-known cryptographic primitives used in the protocols and constructions presented in this work. Here, we only focus on the basic definitions of the syntax and security of these building blocks and we refer, e.g., to the foundational textbook by Katz and Lindell [KL21] for more details.

4.1 Hash Functions and the Random Oracle Model

In this section, we introduce cryptographic hash function, their security guarantees and their idealization in the form of random oracles.

4.1.1 Cryptographic Hash Functions

A *(cryptographic) hash function* is a deterministic algorithm that implements a function $H : \{0,1\}^* \to \{0,1\}^\lambda$, where λ is called the output length of the hash function. In general, a hash function is characterized by mapping inputs from a huge (possibly infinite) domain to a small range, thus *compressing* the inputs. Security-wise we require that the hash function is *collision-resistant*. That is, it should ideally be infeasible for an adversary to find inputs $x_1, x_2 \in \{0,1\}^*$ with $x_1 \neq x_2$ such that $H(x_1) = H(x_2)$. The tuple (x_1, x_2) then is called a *collision*. However, since the domain of H is larger then then its range by definition, it cannot be injective. Thus, a tuple (x_1, x_2) with $x_1 \neq x_2$ and $H(x_1) = H(x_2)$ has to exist, even though we might not be able to find it efficiently. However, even though we might not be able to write the collision down as humans, there is exists an adversary

that only outputs the collision (hardcoded in its definition). Therefore, a fixed, publicly known function H cannot be collision-resistant. In the asymptotic setting, one solves this by introducing a *hash key* because even though there can be an efficient algorithm that outputs a collision for a fixed key, there cannot be an *efficient* algorithm (using a reasonable amount of space) that outputs a collision for any key. But all of the hash functions ever used in practice, like MD5 [Riv92], SHA-1 [EJ01], SHA-2 [EH06, EH11], and SHA-3 [Dwo15], are *keyless*. Due to the real-world focus of this work, we also focus on keyless hash function only. For more information on this, we refer to [Rog06a, Rog06b].

To formalize the above intuition consider the following definition.

Definition 4.1. Let $H : \{0,1\}^* \to \{0,1\}^\lambda$ be a hash function with output length λ. We define the advantage of an adversary $\mathcal{A}$ against the collision-resistance of H as

$$\mathrm{Adv}_H^{\mathsf{CR}}(\mathcal{A}) := \Pr[(m_1, m_2) \xleftarrow{\$} \mathcal{A} : m_1 \neq m_2 \wedge H(m_1) = H(m_2)].$$

4.1.2 Random Oracle Model

A commonly used theoretical abstraction of a hash function is the *random oracle*. A random oracle captures an idealization of a hash function, which is considered to be flawless. The corresponding theoretical model is called the *random oracle model (ROM)*. Here, we assume that all parties involved in a cryptographic scheme (e.g., in a security experiment) have access to the *same* instance of a random oracle. Unfortunately, the assumption of the existence of a random oracle is too strong and cannot hold in a practical environment. The main reason is that its description is too large and thus it is impractical to even store a local copy of it. Nevertheless, from a theoretical point of view it is still interesting to analyze what is achievable in such a theoretical model as the ROM. Even for practical applications the *random oracle paradigm* [BR93] is a commonly accepted heuristic to construct practical cryptographic schemes that then can be proven secure in the ROM. Next, let us have a look at random oracles more formally.

Random oracles. A *random oracle* is a publicly available truly random function

$$\mathsf{RO} : \{0,1\}^* \to \{0,1\}^\lambda.$$

Every actor involved in a cryptographic scheme including the adversary has oracle access to the random oracle RO. The random oracle RO can be seen as a function $\{0,1\}^* \to \{0,1\}^\lambda$ that is chosen uniformly and independently at random at the beginning of the security experiment (resp. at initialization of the cryptographic scheme). One can also think about the random oracle as a *table* that is filled successively using *lazy sampling*. That is, initially this table is empty and whenever the the random oracle RO is queried on input $x \in \{0,1\}^*$ (that was not queried before), it samples a new image $y \xleftarrow{\$} \{0,1\}^\lambda$ as response. The pair (x, y) is then stored in the table so that the random oracle is able to answer consistently.

The random oracle methodology. Bellare and Rogaway [BR93] introduced the following paradigm to design practical (i.e., efficient) schemes:

1. Design a cryptographic scheme in the ROM.
2. Prove its security in the ROM.
3. Replace the oracle access to the random oracle by computations of a hash function H.

As already mentioned above this paradigm is also a heuristic. First of all, the hash function used cannot replace a random oracle one-to-one simply because its much smaller description. Then, the success of the above paradigm clearly depends on the quality of the hash function and its ability to "emulate" a random oracle. However, in practice this paradigm has "proven" well and still finds a lot of adoption. In particular, proving constructions in the ROM is a great tool to check the plausibility of a cryptographic scheme.

Programmable random oracles. Security proofs in the ROM often make use of a property of the random oracle called *programmability*. A reduction usually emulates the whole security experiment for an adversary. In the ROM, this also includes the random oracle RO. That is, the reduction has control over the random oracle table, including, for example, to implement the lazy sampling of the images. Now, this ability allows the reduction also to *program* the images of the random oracle. Here, it is important that the images remain uniformly distributed from the view of the adversary. This ability is an important proof technique in many proofs and we will also make use of it in this thesis.

4.2 Pseudorandom Functions

An important building block of numerous cryptographic constructions is a *pseudorandom function (PRF)*. They find applications mostly in the construction of encryption schemes, but also in the key derivation steps of key exchange protocols PRFs are vital. Informally, a PRF is a function that is indistinguishable from a truly random function. To be precise, consider the the space of all functions $\{0,1\}^* \to \{0,1\}^\mu$ denoted by $\mathcal{F}(*,\mu)$. A PRF is a keyed function (or family of functions) $\{F_k : \{0,1\}^* \to \{0,1\}^\mu\}_{k\in\{0,1\}^\kappa}$ such that the distributions

$$\{k \xleftarrow{\$} \{0,1\}^\kappa : F_k\} \quad \text{and} \quad \{F \xleftarrow{\$} \mathcal{F}(*,\mu) : F\}$$

are indistinguishable. To formalize this property, consider the following definition.

Definition 4.2. Let $\kappa, \mu \in \mathbb{N}$ and let F be a deterministic, keyed function F with key space $\{0,1\}^\kappa$, domain $\{0,1\}^*$, and range $\{0,1\}^\mu$. We define the *advantage of an adversary $\mathcal{A}$ against the pseudorandomness of F* as

$$\mathrm{Adv}_F^{\mathsf{PRF}}(\mathcal{A}) := \left|\Pr[\mathrm{Exp}_F^{\mathsf{PRF}}(\mathcal{A}) = 1] - \frac{1}{2}\right|$$

where $\mathrm{Exp}_F^{\mathsf{PRF}}(\mathcal{A})$ is defined on the left-hand side of Figure 4.1.

$\mathrm{Exp}_F^{\mathsf{PRF}}(\mathcal{A})$:

1 : $k \xleftarrow{\$} \{0,1\}^\kappa$
2 : $\mathcal{O}_0(\cdot) := F(k,\cdot)$
3 : $f \xleftarrow{\$} \mathcal{F}(*,\mu)$
4 : $\mathcal{O}_1(\cdot) := f(\cdot)$
5 : $b \xleftarrow{\$} \{0,1\}$
6 : $b' \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_b(\cdot)}$
7 : **return** 1 iff. $b = b'$

Figure 4.1: Security experiment for PRFs.

4.3 Message Authentication Codes

Message authentication codes (MACs) allow two parties sharing a (symmetric) key to authenticate messages using a *tag*. These tags (or sometimes simply referred to as MACs) then can also be verified using the same shared key. A MAC scheme is defined as follows.

Definition 4.3. A *message authentication code* $\mathsf{MAC} = (\mathsf{MAC.Gen}, \mathsf{MAC.Tag}, \mathsf{MAC.Vrfy})$ for key space $\mathcal{K}$, message space $\mathcal{M}$, and tag space $\mathcal{T}$ is a triple of algorithms such that

1. The randomized *key generation algorithm* $\mathsf{MAC.Gen}$ is given no input, and outputs a key $k \in \mathcal{K}$.
2. The randomized *tagging algorithm* $\mathsf{MAC.Tag}$ is given a key $k \in \mathcal{K}$ and a message $m \in \mathcal{M}$ as input, and outputs a tag $t \in \mathcal{T}$.
3. The deterministic *verification algorithm* $\mathsf{MAC.Vrfy}$ is given a key k, a message m, and a tag t as input, and outputs a bit b.

Correctness. We require that for a MAC scheme MAC for all $k \in \mathcal{K}$ and for all $m \in \mathcal{M}$, it holds that $\mathsf{MAC.Vrfy}(k, m, \mathsf{MAC.Tag}(k, m)) = 1$.

Security. Since MAC schemes are used to provide authenticity to messages, we require for security of a MAC scheme that, informally, it should be hard for an adversary to forge a MAC tag for any message *(existential forgery)*. This should even hold if the adversary gets oracle access to a tagging oracle providing MAC tag for messages of its choice (*adaptive chosen-message attack*). Clearly, to exclude trivial attacks the adversary has to forge for a new message, i.e., a message that has not been queried to the tagging oracle.

Definition 4.4. Let $\mathsf{MAC} = (\mathsf{MAC.Gen}, \mathsf{MAC.Tag}, \mathsf{MAC.Vrfy})$ be a MAC scheme. We define the *advantage of an adversary $\mathcal{A}$ against the existential unforgeability under an adaptive chosen-message attack (*$\mathsf{EUF\text{-}CMA}$*)* as

$$\mathrm{Adv}_{\mathsf{MAC}}^{\mathsf{EUF\text{-}CMA}}(\mathcal{A}) := \Pr[\mathrm{Exp}_{\mathsf{MAC}}^{\mathsf{EUF\text{-}CMA}}(\mathcal{A}) = 1]$$

$\mathrm{Exp}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{MAC}}(\mathcal{A})$:

1 : $\mathcal{Q} := \emptyset$
2 : $K \xleftarrow{\$} \mathsf{MAC.Gen}$
3 : $(m^*, t^*) \xleftarrow{\$} \mathcal{A}^{\mathsf{Tag}(\cdot)}$
4 : **return** $(m^*, \cdot) \notin \mathcal{Q}$
$\wedge\ \mathsf{MAC.Vrfy}(k, m^*, t^*) = 1$

$\mathsf{Tag}(m)$:

1 : $T \xleftarrow{\$} \mathsf{MAC.Tag}(k, m)$
2 : $\mathcal{Q} := \mathcal{Q} \cup \{(m, t)\}$
3 : **return** T

Figure 4.2: Security experiment for existential unforgeability under an adaptive chosen-message attack (EUF-CMA) for MACs.

where $\mathrm{Exp}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{MAC}}(\mathcal{A})$ is defined in Figure 4.2.

Remark 4.1. Definition 4.4 can be strengthened by changing the winning condition in Line 4 of $\mathrm{Exp}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{MAC}}(\mathcal{A})$ to

$$(m^*, t^*) \notin \mathcal{Q} \wedge \mathsf{MAC.Vrfy}(k, m^*, t^*) = 1.$$

That is, the adversary now wins if the forgery t^* was never output by the tagging oracle on input m^*. In contrast, to the condition that m^* was not allowed to be queried before. The resulting notion is in the literature called strong EUF-CMA-security, which we denote by sEUF-CMA. This notion is particularly interesting as this captures that the corresponding MAC scheme is "non-malleable". This means, informally, it is hard to compute a (different) tag t^* for a message m^*, even if a tag $t \xleftarrow{\$} \mathsf{Tag}(k, m^*)$ with $t \neq t^*$ is known.

4.4 Digital Signatures

In some applications, e.g., for authentication in the TLS handshake protocol, messages need to be authenticated such that they are *publicly verifiable*. A MAC scheme as introduced in Section 4.3, unfortunately, can only authenticate messages if users share a symmetric key. Since a shared symmetric key is not always present, an entity can authenticate messages in a publicly verifiable way using a *digital signature scheme*. The entity holds a secret key to compute signatures and publishes the corresponding public key, so that signature can be verified. For security, it shall be guaranteed that only the entity holding the secret key can compute valid signatures (under the public key). Thus, informally, it should feasible to a forge signature for some message. We recall the standard definition of a *digital signature scheme* by Goldwasser, Micali, and Rivest [GMR88].

Definition 4.5 (Digital Signature Scheme). A *digital signature scheme* for message space $\mathcal{M}$ is a triple of algorithms $\mathsf{Sig} = (\mathsf{Sig.Gen}, \mathsf{Sig.Sign}, \mathsf{Sig.Vrfy})$ such that

1. The randomized key generation algorithm Sig.Gen is given no input and generates a public (verification) key pk and a secret (signing) key sk.

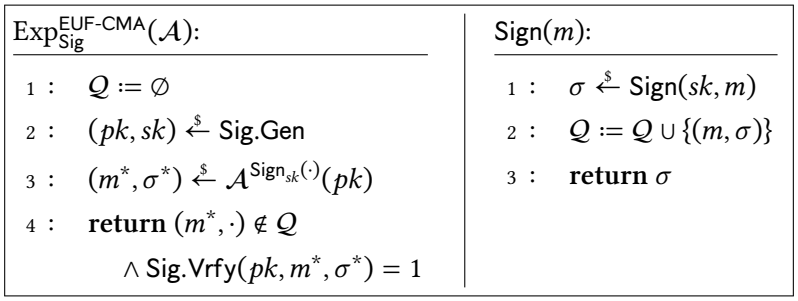

Figure 4.3: Security experiment for existential unforgeability under an adaptive chosen-message attack (EUF-CMA) for digital signature schemes.

2. The randomized signing algorithm Sig.Sign on input a signing key sk and a message $m \in \mathcal{M}$, it outputs a signature σ.
3. The deterministic verification algorithm Sig.Vrfy on input verification key pk, a message $m \in \mathcal{M}$ and a signature σ outputs either 0 or 1.

Correctness. We say that a digital signature scheme Sig is *correct* if for any $m \in \mathcal{M}$, and for any (pk, sk) that can be output by Sig.Gen, it holds

$$\mathsf{Sig.Vrfy}\left(pk, m, \mathsf{Sig.Sign}(sk, m)\right) = 1.$$

Existential unforgeability of signatures. The standard notion of security for digital signature schemes is *existential unforgeability under an adaptive chosen-message attack (EUF-CMA)*. As we already introduced this notion for MACs in Section 4.3 in a symmetric setting, we only briefly present the adapted definition here.

Definition 4.6. Let $\mathsf{Sig} = (\mathsf{Sig.Gen}, \mathsf{Sig.Sign}, \mathsf{Sig.Vrfy})$ be a digital signature scheme. We define the *advantage of an adversary $\mathcal{A}$ against the existential unforgeability under an adaptive chosen-message attack (*EUF-CMA*)* as

$$\mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Sig}}(\mathcal{A}) := \Pr[\mathrm{Exp}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Sig}}(\mathcal{A}) = 1],$$

where $\mathrm{Exp}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Sig}}(\mathcal{A})$ is defined in Figure 4.3.

Existential unforgeability in a multi-user setting. In a "real-world" scenario, the adversary is more likely faced a different challenge than described in Definition 4.6. Namely, a real-world adversary presumably plays against multiple users at the same time and might even be able to get the secret keys of a subset of these users. In this setting, its challenge is to forge a signature for any of the users that it has no control of (to exclude trivial attacks). Here, one can think of the TLS protocol where billions of users on the Internet exchange digital signatures for authentication in the handshake protocol. An adversary now operates on the Internet, in an environment with multiple users communicating, with the goal to impersonate (i.e., forging a signature) *any* of

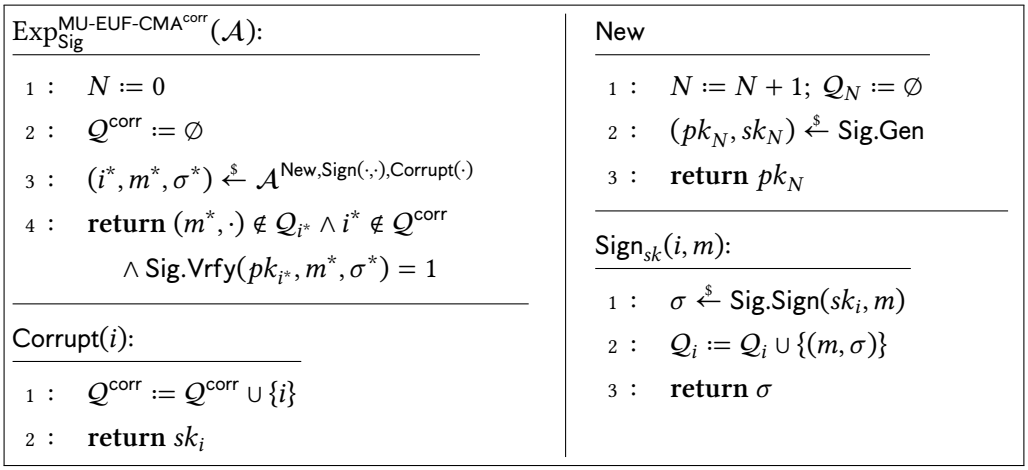

Figure 4.4: Security experiment for existential unforgeability under an adaptive chosen-message attack in the multi-user setting with adaptive corruptions ($\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathsf{corr}}$) for digital signature schemes.

these users. In Definition 4.6, the adversary rather operates in an "isolated" network in which only one user is connected. To capture this intuition we additionally consider the multi-user EUF-CMA notion with adaptive corruptions as proposed by Bader et al. [Bad+15a]. To this end, the single-user notion given in Definition 4.6 can naturally be upgraded to a multi-user notion with adaptive corruptions as follows.

Definition 4.7. Let $\mathsf{Sig} = (\mathsf{Sig.Gen}, \mathsf{Sig.Sign}, \mathsf{Sig.Vrfy})$ be a digital signature scheme. We define the *advantage of an adversary $\mathcal{A}$ against the existential unforgeability under an adaptive chosen-message attack in the multi-user setting with adaptive corruptions (*$\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathsf{corr}}$*)* as

$$\mathrm{Adv}^{\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathsf{corr}}}_{\mathsf{Sig}}(\mathcal{A}) := \Pr[\mathrm{Exp}^{\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathsf{corr}}}_{\mathsf{Sig}}(\mathcal{A}) = 1]$$

where $\mathrm{Exp}^{\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathsf{corr}}}_{\mathsf{Sig}}(\mathcal{A})$ is defined in Figure 4.4.

Remark 4.2. Analogously to MAC schemes, we can define $\mathsf{sEUF\text{-}CMA}$-security (resp. $\mathsf{MU\text{-}sEUF\text{-}CMA}^{\mathsf{corr}}$-security) for digital signature schemes in the same way.

Remark 4.3. This notion can also be weakened by excluding adaptive corruptions. The resulting experiment is analogous except that queries to the corruption oracle are forbidden. The corresponding notions are denoted by $\mathsf{MU\text{-}EUF\text{-}CMA}$ instead of $\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathsf{corr}}$.

4.5 Lossy Identification Schemes

Lossy identification schemes (LIDs) originally were introduced by Abdalla, Fouque, Lyubashevsky, and Tibouchi [AFLT12, AFLT16] in the context of a construction of a tightly-secure signature scheme. We recall the definition of a LID scheme presented in [DGJL21b]. This definition adapts the definitions of [AFLT12, AFLT16, KMP16]. Syntactically, a LID scheme is defined as follows.

Definition 4.8. A *lossy identification scheme* is a five-tuple $\mathsf{LID} = (\mathsf{LID.Gen}, \mathsf{LID.LossyGen}, \mathsf{LID.Prove}, \mathsf{LID.Vrfy}, \mathsf{LID.Sim})$ of algorithms with the following properties.

1. $\mathsf{LID.Gen}$ is the (normal) key generation algorithm. It outputs a public verification key pk and a secret key sk.
2. $\mathsf{LID.LossyGen}$ is the lossy key generation algorithm. It outputs a lossy verification key pk.
3. $\mathsf{LID.Prove}$ is the prover algorithm that is split into two algorithms:
   - $(\mathsf{cmt}, \mathsf{st}) \xleftarrow{\$} \mathsf{LID.Prove}_1(sk)$ is a probabilistic algorithm that takes as input the secret key and returns a commitment cmt and a state st.
   - $\mathsf{resp} := \mathsf{LID.Prove}_2(sk, \mathsf{cmt}, \mathsf{ch}, \mathsf{st})$ is a deterministic algorithm[1] that takes as input a secret key sk, a commitment cmt, a challenge ch, a state st, and returns a response resp.
4. $\mathsf{LID.Vrfy}(pk, \mathsf{cmt}, \mathsf{ch}, \mathsf{resp})$ is a deterministic verification algorithm that takes a public key, and a conversation transcript (i.e., a commitment, a challenge, and a response) as input and outputs a bit, where 1 indicates that the proof is "accepted" and 0 that it is "rejected".
5. $\mathsf{cmt} := \mathsf{LID.Sim}(pk, \mathsf{ch}, \mathsf{resp})$ is a deterministic algorithm that takes a public key pk, a challenge ch, and a response resp as inputs and outputs a commitment cmt.

We assume that a public key pk implicitly defines two sets, the set of challenges CSet and the set of responses RSet.

Further, we define the following properties of a LID scheme based on the definitions given in [AFLT12, AFLT16].

Definition 4.9. Let $\mathsf{LID} = (\mathsf{LID.Gen}, \mathsf{LID.LossyGen}, \mathsf{LID.Prove}, \mathsf{LID.Vrfy}, \mathsf{LID.Sim})$ be defined as above.

- *Completeness of normal keys.* We call LID ρ*-complete*, if

$$\Pr\left[\mathsf{LID.Vrfy}(pk, \mathsf{cmt}, \mathsf{ch}, \mathsf{resp}) = 1 \;:\; \begin{array}{l}(pk, sk) \xleftarrow{\$} \mathsf{LID.Gen};\\ (\mathsf{cmt}, \mathsf{st}) \xleftarrow{\$} \mathsf{LID.Prove}_1(sk);\\ \mathsf{ch} \xleftarrow{\$} \mathsf{CSet};\\ \mathsf{resp} := \mathsf{LID.Prove}_2(sk, \mathsf{cmt}, \mathsf{ch}, \mathsf{st})\end{array}\right] \geq \rho.$$

  We call LID *perfectly-complete*, if it is 1-complete.
- *Simulatability of transcripts.* We call LID ε_s*-simulatable* if for $(pk, sk) \xleftarrow{\$} \mathsf{LID.Gen}$, $(\mathsf{ch}, \mathsf{resp}) \xleftarrow{\$} \mathsf{CSet} \times \mathsf{RSet}$, the distribution of the transcript $(\mathsf{cmt}, \mathsf{ch}, \mathsf{resp})$ where

[1] All known instantiations of lossy identification schemes have a deterministic $\mathsf{LID.Prove}_2$ algorithm. However, if a new instantiation requires randomness, then it can be "forwarded" from $\mathsf{LID.Prove}_1$ in the state variable st. Therefore the requirement that $\mathsf{LID.Prove}_2$ is deterministic is without loss of generality, and only made to simplify our security analysis.

$\mathsf{cmt} \coloneqq \mathsf{LID.Sim}(pk, \mathsf{ch}, \mathsf{resp})$ is statistically indistinguishable from honestly generated transcript (with a statistical distance up to ε_s) and we have that $\mathsf{LID.Vrfy}(pk, \mathsf{cmt}, \mathsf{ch}, \mathsf{resp}) = 1$. That is, for any adversary $\mathcal{A}$ it holds that

$$|\Pr[\mathcal{A}(pk, X) = 1] - \Pr[\mathcal{A}(pk, Y) = 1]| \leq \varepsilon_s$$

where $(pk, sk) \xleftarrow{\$} \mathsf{LID.Gen}$, $X = (\mathsf{cmt}, \mathsf{ch}, \mathsf{resp})$ with $(\mathsf{cmt}, \mathsf{st}) \xleftarrow{\$} \mathsf{LID.Prove}_1(sk)$, $\mathsf{ch} \xleftarrow{\$} \mathsf{CSet}$, and $\mathsf{resp} \xleftarrow{\$} \mathsf{LID.Prove}_2(sk, \mathsf{cmt}, \mathsf{ch}, \mathsf{st})$, and $Y = (\mathsf{cmt}', \mathsf{ch}', \mathsf{resp}')$ with $(\mathsf{ch}', \mathsf{resp}') \xleftarrow{\$} \mathsf{CSet} \times \mathsf{RSet}$ and $\mathsf{cmt}' \coloneqq \mathsf{LID.Sim}(pk, \mathsf{ch}', \mathsf{resp}')$. If $\varepsilon_s = 0$, we call LID *perfectly simulatable.*

Note that this simulatability property is different from the original definition in [AFLT12] where the simulator simulates the whole transcript.

- *Indistinguishability of keys.* This definition is a generalization of the standard key indistinguishability definition of a lossy identification scheme extended to multiple instances. We define the advantage of an adversary $\mathcal{A}$ breaking the multi-key-indistinguishability of LID as

$$\mathrm{Adv}^{\mathsf{MU\text{-}IND\text{-}KEY}}_{\mathsf{LID}}(\mathcal{A}) \coloneqq \left|\Pr\left[\mathcal{A}^{\mathsf{New}} = 1\right] - \Pr\left[\mathcal{A}^{\mathsf{NewLoss}} = 1\right]\right|,$$

  where oracle New generates a key pair $(pk^{(i)}, sk^{(i)}) \xleftarrow{\$} \mathsf{LID.Gen}$ and returns $pk^{(i)}$ upon the i-th query and oracle NewLoss generates a key $pk'^{(i)} \xleftarrow{\$} \mathsf{LID.LossyGen}$ and returns it upon the i-th query.

- *Lossiness.* We call LID ε_ℓ-lossy if for any $pk \in \{pk : pk \xleftarrow{\$} \mathsf{LID.LossyGen}\}$, any commitment cmt, it holds that the ratio of challenges ch such that there exists a response resp with $\mathsf{LID.Vrfy}(pk, \mathsf{cmt}, \mathsf{ch}, \mathsf{resp}) = 1$ is at most ε_ℓ.[2]

Below we present a relaxation of the uniqueness property with respect to lossy keys. An information-theorectic variant is defined in [AFLT12, AFLT16].

Definition 4.10. We define the *advantage of an adversary $\mathcal{A}$ against the uniqueness of* LID *with respect to lossy keys* as

$$\mathrm{Adv}^{\mathsf{unq}}_{\mathsf{LID}}(\mathcal{A}) \coloneqq \Pr\left[\begin{array}{l}\mathsf{LID.Vrfy}(pk, \mathsf{cmt}, \mathsf{ch}, \mathsf{resp}) = 1 \\ \wedge\ \mathsf{LID.Vrfy}(pk, \mathsf{cmt}, \mathsf{ch}, \mathsf{resp}') = 1 \\ \wedge\ \mathsf{resp} \neq \mathsf{resp}'\end{array} : \begin{array}{l}pk \xleftarrow{\$} \mathsf{LID.LossyGen}; \\ (\mathsf{cmt}, \mathsf{ch}, \mathsf{resp}, \mathsf{resp}') \xleftarrow{\$} \mathcal{A}(pk)\end{array}\right]$$

Below we recall the property of min-entropy for LID schemes adapted from [AFLT12, AFLT16].

[2] This notion of lossiness is a stronger notion than the one originally used by Abdalla, Fouque, Lyubashevsky, and Tibouchi [AFLT12, AFLT16] and it adapts the notion of optimal soundness for three-message public-coin protocols (3PC) used by Fischlin, Harasser, and Janson [FHJ20] to lossy identification schemes.

Definition 4.11. Let $(pk, sk) \xleftarrow{\$} \mathsf{LID.Gen}$ be any honestly generated key pair and $C(sk) := \{\mathsf{cmt} : (\mathsf{cmt}, \cdot) \xleftarrow{\$} \mathsf{LID.Prove}_1(sk)\}$ be the set of commitments associated to sk. We define the *min-entropy with respect to* LID as

$$\alpha := -\log_2 \left(\max_{sk, \mathsf{cmt} \in C(sk)} \Pr\left[\mathsf{LID.Prove}_1(sk) = (\mathsf{cmt}, \cdot)\right] \right).$$

Further, we recall the definition of *commitment-recoverability* by Kiltz, Masny, and Pan [KMP16].

Definition 4.12. A lossy identification scheme LID is *commitment-recoverable* if the algorithm $\mathsf{LID.Vrfy}(pk, \mathsf{cmt}, \mathsf{ch}, \mathsf{resp})$ first recomputes a commitment $\mathsf{cmt}' = \mathsf{LID.Sim}(pk, \mathsf{ch}, \mathsf{resp})$ and then outputs 1 if and only if $\mathsf{cmt}' = \mathsf{cmt}$.

Below, we recall a new property for LID schemes introduced in [DGJL21b] which requires that the LID.Sim algorithm is injective with respect to the input challenge.

Definition 4.13. A lossy identification scheme LID has an *injective simulator* if for any $(pk, sk) \xleftarrow{\$} \mathsf{LID.Gen}$, any response $\mathsf{resp} \in \mathsf{RSet}$, any $\mathsf{ch} \neq \mathsf{ch}'$, it holds that $\mathsf{LID.Sim}(pk, \mathsf{ch}, \mathsf{resp}) \neq \mathsf{LID.Sim}(pk, \mathsf{ch}', \mathsf{resp})$.

Part II

On the Tightness of the TLS 1.3 Handshake Protocol

5

MULTI-STAGE KEY EXCHANGE PROTOCOLS

Author's contribution. The security model presented in this chapter is based on joint work with Hannah Davis, Felix Günther and Tibor Jager [DDGJ22b, DDGJ22a]. While we discussed all aspects of this paper together, Hannah Davis developed the code-based multi-stage key exchange (MSKE) model for the PSK modes of TLS 1.3 with the support of Felix Günther. The model as it is presented in this thesis is an extension of this model. The author of this thesis used the code-based PSK MSKE model as a foundation and extended it by a public-key version to give a unified code-based MSKE model for both MSKE variants following the example of the (textual) MSKE model by Dowling, Fischlin, Günther, and Stebila [DFGS21]. The most notable addition compared to the previous PSK MSKE model [DDGJ22b, DDGJ22a] is the integration of updatable authentication originally introduced in [DFGS21]. The new code-based public-key MSKE variant finds application in the tight analysis of the TLS 1.3 full handshake presented in Chapter 9.

Contents

5.1 Introduction

Classically, a key exchange protocol is a protocol in which two parties negotiate a *single* key, called the *session key*, that then, for example, is intended to be used to secure communication between the two parties using symmetric encryption or a more complex channel protocol. However, modern real-world protocols like Google's QUIC [IT21] and TLS 1.3 [Res18] deviate from this structure and negotiate *multiple* keys during their key

exchange phase. These keys also do not necessarily have to be intended to secure communication. In TLS 1.3, for example, there are also keys derived that allow for session resumption (i.e., a later abbreviated handshake between the same parties) or an exporter key that is intended to provide "good" key material for further use in the application layer. To capture these modern protocols more precisely, Fischlin and Günther [FG14] introduced the notion of *multi-stage key exchange (MSKE)* protocols. The model extends the classical Bellare–Rogaway (BR) model [BR94] for (single key) authenticated key exchange (AKE) to the needs of more complex key exchange protocols, like QUIC and TLS 1.3. The BR model is along with the model by Canetti and Krawczyk [CK02] the de-facto standard and the foundation for most key exchange models. In a series of works [DFGS15, DFGS16, FG17, Gün18, SSW20, DFGS21, DDGJ22b] the notion of MSKE protocols was extended. Here, the works [DFGS15, DFGS16, FG17] adapted the model initially introduced for the QUIC protocol to the needs of the TLS 1.3 handshake protocol (drafts), Schwabe, Stebila, and Wiggers [SSW20] in the context of a post-quantum-secure alternative for the TLS 1.3 handshake protocol (KEM-TLS) introduced multiple levels of forward secrecy and explicit authentication, and Dowling, Fischlin, Günther, and Stebila [DFGS21] refined the notion of authenticated in the form of *upgradable authentication* for an analysis of the final TLS 1.3 standard. While most of the above variants of MSKE were presented in textual form, Davis, Diemert, Günther, and Jager [DDGJ22b] presented the first code-based variant in the sense of code-based games by Bellare and Rogaway [BR06, BR04] of the pre-shared key (PSK) variant of MSKE protocols. Here, we refer to the PSK variant as the variant of MSKE protocols in which each party taking part in the protocol holds a pre-shared symmetric (long-)term key. This is opposed to the public-key variant in which each party holds a public-secret-key pair (e.g., a digital signature key pair in combination with some public key infrastructure).

Chapter outline. In this chapter, we give a unified, code-based version of the MSKE notion for both PSKs and public keys. To this end, we extend the model presented in [DDGJ22b] by a public-key variant mostly following Dowling, Fischlin, Günther, and Stebila [DFGS21], but including explicit authentication and different levels of forward secrecy, as in [SSW20], in this work similar to the PSK variant from [DDGJ22b]. We first give a informal description of MSKE protocols to define the setting in Section 5.2 and then define the notion including the security model formally in Sections 5.3 and 5.4. In Section 5.5, we briefly recall the notion of multi-stage session matching as defined by Günther [Gün18], which is required for the formal results of Chapter 8.

5.2 Setting

Since the MSKE model is a complex security model, we first informally introduce the setting of (multi-stage) key exchange protocols to aid the understanding of the formal model. To this end, we describe the actors involved in the protocol, and various aspects necessary to reflect the real world as precisely as possible such as the derivation of multiple keys, authentication, the distinction between internal and external keys, different

types of long-term keys, the replayability of certain messages, a paradigm of secret reveals, and the notion of forward secrecy. The informal description given in this section is then formalized in the following Sections 5.3 and 5.4.

Actors. A *key exchange protocol* KE is run between multiple parties, called *users*, connected via a network. We assume users to have some kind of *long-term* key. This can either be a *public-secret-key pair* (usually a digital signature key pair) or a symmetric key shared between pairs of users before-hand (*pre-shared key (PSK)*). Each user can run multiple instances of the key exchange protocol, either sequentially or in parallel; such an instance is called a *session*. A user can take two *roles* in a session determined by whether it sends or receives the first protocol message. Here, we call a session that sends the first message (i.e., it initiates a protocol run) the *initiator* and a session that receives the first message the *responder*. In a client-server setting, like we have it, for example, in the case of TLS, the client would usually be the initiator and the responder would be the server. We use these terms interchangeably in this work.

A key exchange protocol offers the following interface. Sessions can be *activated* (i.e., created), which initializes a new session and in case of an imitator session generates the first message. Once sessions are created, they can send and receive messages. To do so, they take messages as input, process them, update their *internal state* and generate a response. Sending messages to a session can also affect its *status*, which is *running* when it is waiting for the next message to arrive, *accepted* when a session key has been established, and *rejected* if the execution of the protocol failed.

Goals. Key exchange is a complex task and a model capturing modern key exchange protocols needs to consider a number of aspects to reflect the real world and its security goals as good as possible. Therefore, the following aspects are of importance for modern key exchange protocols.

Multiple Stages/Keys. The main goal of each instance of the protocol is to negotiate a *session key*. In a multi-stage key exchange, the sessions aim to negotiate *multiple* session keys in *stages*. The main security goal for these keys is indistinguishability from random. Informally, this means that the session keys established in an uncompromised session, usually referred to as a *fresh* session, should be as good as uniformly random keys. Thus, providing *secrecy* to the derived keys.

Authentication. Even though secrecy of the session keys is crucial for the security of a key exchange protocol, this alone is not enough. Namely, we not only have to achieve that "outsiders" of a protocol execution cannot learn the derived session key, we also have to achieve that only the "right" parties know the derived key. Here, comes the concept of *authentication* into play, which provides parties with guarantees on the identity of their intended communication partner. In a two-party protocol, authentication can be one-way (*unilateral*), i.e., only one party proves its identity, or *mutual*, i.e., both parties prove their identity. Furthermore, authentication can be *implicit* or *explicit*. Implicit authentication for a session π

informally means that any session π' (if existent) that derives the same key as π must be owned by the intended communication partner of session π. Note that this notion does not guarantee the existence of session π', only that if it existed it would derive the same key. The existence of such a partner session is guaranteed by the notion of explicit authentication. Namely, explicit authentication extends implicit authentication by the requirement that if π's intended communication partner is uncompromised, then there exists a session π' owned by π's partner that derived the same key. That is, session π can be sure that there exists an (honest) session π' deriving the same key and that is owned by the intended partner. Explicit authentication is a stronger notion than implicit authentication. In fact, de Saint Guilhem, Fischlin, and Warinschi [dFW20] show that implicit (key) authentication in combination with key confirmation (i.e., a session getting a guarantee that another session derived the same key; cf. [FGSW16] for a formal treatment of key confirmation) yields explicit (key) authentication.

Next, let us discuss how we reflect authentication in the MSKE model. We distinguish between three *levels of authentication* of each stage. A stage can be *unauthenticated*, *unilaterally* authenticated (only the responder authenticates), and *mutually* authenticated (both initiator and responder authenticate). The level of authentication for each stage can be defined individually and we allow every session to have a different *type of authentication*. Here we refer to a type of authentication as the vector that defines a level of authentication for every stage. For instance, we allow that there are two sessions of the same user, where one session in some stage aims for unilateral authentication and the other one for mutual authentication in the same stage. This captures exactly what we have in the real world, for example, in TLS. By default, most sessions on the Internet aim for unilateral authentication (the server showing its certificate to prove its identity to the client). However, some applications also require mutual authentication, for example, IoT devices reporting back to a server. As introduced by Dowling, Fischlin, Günther, and Stebila [DFGS21], we also allow the level of authentication of a stage to be upgraded upon acceptance of a later stage. To this end, we assume every stage to be initially unauthenticated and define at which stage the authentication level is upgraded to unilateral or mutual authentication. Here, we also allow that a stage can be authenticated right away or never establishes unilateral or mutual authentication.

In the MSKE model of Dowling, Fischlin, Günther, and Stebila [DFGS21], authentication is mainly captured implicitly, but as an extension we also include in our model the option of upgrading authentication to explicit authentication (e.g., if the key is later confirmed). This was already introduced by Davis, Diemert, Günther, and Jager [DDGJ22b] in the context of their code-based pre-shared key MSKE model, but there is the special case of all stages being already initially mutually authenticated via the PSK. Schwabe, Stebila, and Wiggers [SSW20] also extended the MSKE model by the notion of explicit authentication before. We incorporate explicit authentication, in particular for the public-key variant, similar to the (code-based) AKE model by [DG21a]. We also allow only unilateral authentication, for

example, resulting in only the initiator being able to receive explicit authentication of the responder.

Internal and External Keys. Classical key exchange protocols, like the (signed) Diffie–Hellman key exchange, usually capture the negotiation of session keys that are intended to be only used outside of the (key exchange) protocol, e.g., a secure channel protocol. However, modern protocols, such as TLS 1.3, also establish keys that might be used inside the protocol. For instance, consider the TLS 1.3 handshake described in Chapter 6. TLS 1.3 introduced the encryption of the handshake messages, particularly, to strengthen privacy. Thus, the TLS 1.3 handshake negotiates a key called the handshake traffic key intended to protect the data that is sent *during* the handshake protocol. This requires the distinction of *internal* and *external* session keys. Internal keys are allowed to be used during the execution of the protocol and external keys are *only* allowed to be outside of the protocol. An external session key could, for example, be the classical application traffic key negotiated in the TLS 1.3 handshake that is intended to be used to protect application data (outside of the handshake protocol). The distinction is crucial to provide reasonable security to the session keys as internal keys can only be "secret" (i.e., indistinguishable from random) if it has not been used during the protocol. This requires the model to basically pause, allowing the adversary to be challenged against an internal key. Namely, if an internal key would be treated exactly as an external key (not used during the protocol), this might provide a trivial attack to distinguish a challenge from random. In the above handshake example, the adversary could just decrypt an intercepted handshake message with the challenge key and if there is a reasonable output it knows that it has received the real (internal) key with certainty.

Long-term Keys. MSKE protocols come in two different flavors depending on the type of long-term key that the users hold. On the one hand, there is the *public-key variant*, which we refer to as pMSKE following [DFGS21], where the long-term key is a public-secret key pair. An example for such a MSKE is the full 1-RTT TLS 1.3 handshake protocol presented in Section 6.4. Here, the public-secret-key pair is a key pair of a digital signature scheme. We do not consider a public-key infrastructure in this work, but rather the model holds a list of all public keys (denoted by pkeys) that grants sessions access to all public keys (including its own). For a treatment of certification of public keys in AKE protocols, we refer to [Boy+13].

The other variant is the *pre-shared key* variant, which we refer to as sMSKE following [DFGS21], where long-term keys are symmetric keys, called pre-shared keys (PSKs), shared among pairs of users. An example here is the abbreviated TLS 1.3 PSK handshake protocol presented in Section 6.5. To keep the model simple, we assume PSKs to be chosen uniformly and independently at random from the PSK space $\mathsf{KE.PSKS}$ defined by the protocol. This can be generalized and strengthened by, e.g., considering any distribution on $\mathsf{KE.PSKS}$ or to allow for the registration

of adversarially chosen PSKs. However, we expect that most PSKs, particularly for TLS 1.3, were established in a prior protocol run and thus it is reasonable to assume them to be random. Pairs of users can usually share multiple PSKs, but we only allow a key to be used in a fixed role (i.e., either as an initiator *or* a responder) to avoid the Selfie attack [DG21b]. Intuitively, the Selfie attack is a Man in the Middle (MITM) attack abusing if a PSK can be used in different roles to break mutual authentication that is assumed to be provided by the PSK. To mitigate the attack a PSK must not be shared between more than one client and one server (cf. [DG21b, Sect. 6]). Our model also does not cover the negotiation of a PSK.

In general, we assume in our model that every session in either variant of MSKE knows at the start of the execution which key it uses.

Replayability. Some protocols, e.g., 0-RTT key exchange protocols, include protection against replays. That is, exchanged keys shall still be secure (i.e., indistinguishable from random) even if messages were replayed, which might result in one session negotiating the same key with multiple partners. To capture this feature as a property of the protocol rather than an attack, the MSKE model allows for protocol stages to be replayable.

Secret Reveals. The model captures the revelation of long-term keys and sessions keys negotiated during the execution of the protocol. This follows the classical model by Bellare and Rogaway [BR94]. We do not consider the compromise of ephemeral keys or the internal state of a session. Note that TLS 1.3, which is the main focus of this work, is not designed to be secure in a setting that includes the compromise of ephemeral keys or the internal state. The compromise of long-term keys is even necessary to capture forward secrecy as described below.

Forward Secrecy. Another important security property of modern key exchange protocols is the notion of *forward secrecy*. Forward secrecy (with respect to long-term keys), informally, captures that even if a long-term key of a user (e.g., the signature key pair or the PSK) gets compromised after a protocol run has been terminated, this compromise should not harm the secrecy of session keys established in this protocol run. This of course can only hold if the session keys have been erased after their usage. For TLS 1.3, the full handshake and the PSK-(EC)DHE handshake described in Sections 6.4 and 6.5, respectively, provide forward secrecy. The TLS 1.3 PSK-only handshake does not provide forward secrecy. This is due to the fact that the full handshake and the PSK-(EC)DHE handshake are based on the ephemeral Diffie–Hellman key exchange, which intuitively ensures that session keys always depend on local (ephemeral) randomness such that a long-term key reveal is not sufficient to recover keys established in a terminated session. Again, this only holds under the assumption that the randomness has been erased. However, the long-term key reveal implies that future protocol runs are inherently compromised, since an adversary in possession of a long-term can perfectly impersonate the owner of the long-term key. For a more detailed discussion on forward secrecy, we refer to the work by [BG20].

We follow Schwabe, Stebila, and Wiggers [SSW20] and distinguish between different levels of forward secrecy in our model. The first level is *no forward secrecy*, which means that a stage key is only guaranteed to be indistinguishable from random if the adversary never compromised the long-term key of the communication partner of the session that derived the key. The second level is *weak forward secrecy 2 (*wfs2*)*, which means that a stage key is only guaranteed to be indistinguishable from random either if the adversary was passive or if the adversary never compromised the long-term key of the communication partner of the session that derived the key. The third level is *(full) forward secrecy (*fs*)*, which means that a stage key is only guaranteed to be indistinguishable from random either if the adversary was passive or if the adversary did compromised the long-term key of the communication partner after *acceptance* of the key.

Similar to authentication we define the level of forward secrecy of each stage individually and allow the level to be upgraded upon the acceptance of a later stage. This extension was also already introduced in [DDGJ22b].

Adversary model. For key exchange protocols, we consider a powerful adversary $\mathcal{A}$, which has full control over the communication network. In particular, it can eavesdrop the communication, inject new messages, or drop messages.

5.3 Syntax

In the following, we formalize the intuitions given in Section 5.2 into a formal security model. We start with the syntax of MSKE protocols, which consist of MSKE-specific notation, properties that are specific to a MSKE protocol, the internal state of a session and finally the formal security experiment.

MSKE-specific notation. The set of users is denoted by $\mathcal{U}$ and every user in $\mathcal{U}$ is identified by a *user identifier* $u \in \mathcal{U}$. The sessions owned by a user $u \in \mathcal{U}$ are identified by π_u^i, where i is a *unique* index for that session.

In pMSKE, each user $u \in \mathcal{U}$ is associated with a long-term key pair (pk_u, sk_u). Here, we assume that pk_u is certified. In sMSKE, we assume that pairs of users share long-term symmetric keys (PSKs) chosen uniformly at random from set KE.PSKS called the PSK space. The assumption of uniformly random PSKs could also be relaxed to any distribution, but for simplicity, we chose the uniform distribution. Each PSK is identified by a (unique) *PSK identifier pskid* and is stored in a list pskeys indexed by $(u, v, pskid)$ where (u, v) are the users sharing the PSK identified by *pskid*. We require that for every index $(u, v, pskid)$ of pskeys, u only uses the respective PSK in the initiator role and v only in the responder role.

Protocol-specific properties. The definition of a MSKE protocol requires the definition of the following properties.

- $\mathsf{STAGES} \in \mathbb{N}$ denotes the numbers of stages (i.e., the number of keys derived);
- $\mathsf{AUTH} \subseteq \{(u, m, ea_{\mathsf{in}}, ea_{\mathsf{res}}) : u, m, ea_{\mathsf{in}}, ea_{\mathsf{res}} \in [\mathsf{STAGES}] \cup \{\infty\}\}^{\mathsf{STAGES}}$ denotes the set of supported authentication types of the MSKE protocol. Each vector $auth = (u^{(i)}, m^{(i)}, ea_{\mathsf{in}}^{(i)}, ea_{\mathsf{res}}^{(i)})_{i \in [\mathsf{STAGES}]} \in \mathsf{AUTH}$ represents an authentication type a session could be aiming for. It contains an entry for every stage and defines the stage that establishes a certain level of authentication for the corresponding stage key. For illustration, consider the i-th entry $auth_i = (u^{(i)}, m^{(i)}, ea_{\mathsf{in}}^{(i)}, ea_{\mathsf{res}}^{(i)})$. Initially all keys are *unauthenticated*. Now, $auth_i$ defines when stage i becomes *unilaterally*, *mutually*, and *explicitly authenticated*. That is, stage i is unilaterally authenticated upon acceptance of stage $u^{(i)}$, mutually authenticated upon acceptance of stage $m^{(i)}$, explicitly authenticated for the initiator upon acceptance of stage $ea_{\mathsf{in}}^{(i)}$, and explicitly authenticated for the responder upon acceptance of stage $ea_{\mathsf{res}}^{(i)}$. If any entry of the tuple is set to ∞ this means that stage i never reaches this authentication level.

  Note that there are constraints on how these tuples are formed, since they are dependent on each other. The entries $(u^{(i)}, m^{(i)})$ define when the stage-i key becomes *implicitly authenticated*. This means if $u^{(i)} = \infty$, then $ea_{\mathsf{in}}^{(i)} = \infty$, and similarly if $m^{(i)} = \infty$ then $ea_{\mathsf{res}}^{(i)} = \infty$, since implicit authentication (resp. key secrecy) is a prerequisite of explicit authentication. Moreover, it has to hold $u^{(i)} \leq m^{(i)}$, $u^{(i)} \leq ea_{\mathsf{i}}^{(i)}$, and $m^{(i)} \leq ea_{\mathsf{r}}^{(i)}$. We highlight that this does not exclude to set, for example, $u^{(i)} = m^{(i)} = ea_{\mathsf{in}}^{(i)} = ea_{\mathsf{res}}^{(i)} = i$ resulting in stage i being explicitly authenticated for both roles upon acceptance.

  We write $\mathsf{EAUTH}[r, s] = t$ if and only if stage s becomes explicitly authenticated for role r upon acceptance of stage t.
- $\mathsf{FS} = (\mathsf{FS}_{\mathsf{initiator}}, \mathsf{FS}_{\mathsf{responder}})$ with $\mathsf{FS}_{role} \in \{-, \mathsf{wfs2}, \mathsf{fs}\}^{\mathsf{STAGES} \times \mathsf{STAGES}}$ denotes the (triangular) matrices indicating which level of forward secrecy is established for a stage upon acceptance of this (or later) stage(s). To allow for a more fine grained distinction, we allow forward secrecy for each role to be established at different stages. Note that the level might change retroactively upon acceptance of a later stage. To simplify notation, we write for $lvl \in \{\mathsf{wfs2}, \mathsf{fs}\}$, $\mathsf{FS}[r, s, lvl] = t$ if and only if $(\mathsf{FS}_r)_{s,t} = lvl$, i.e., stage s of a session with role r establishes lvl in stage t. If $lvl \in \{\mathsf{wfs2}, \mathsf{fs}\}$ for role r in stage s is never established, we write $\mathsf{FS}[r, s, lvl] = \infty$. If $\mathsf{FS}_{\mathsf{initiator}} = \mathsf{FS}_{\mathsf{responder}}$, we shorten notation and only write $\mathsf{FS}[s, lvl]$.
- $\mathsf{INT} \in \{\mathsf{true}, \mathsf{false}\}^{\mathsf{STAGES}}$ denotes the vector indicating whether a key of a stage is internal. Here, $\mathsf{INT}[s] = \mathsf{true}$ if and only if the key accepted in stage s is internal.
- $\mathsf{REPLAY} \in \{\mathsf{true}, \mathsf{false}\}^{\mathsf{STAGES}}$ denotes the vector indicating whether a stage is replayable. Here, $\mathsf{REPLAY}[s] = \mathsf{true}$ if and only if stage s is replayable.

Internal session state. Every session π_u^i maintains an internal state, which includes the following information:

- $status \in \{\mathsf{running}_s, \mathsf{accepted}_s, \mathsf{rejected}_s \mid s \in [1, \ldots, \mathsf{STAGES}]\} \cup \{\mathsf{running}_0\}$: The variable *status* represents the state of execution, which upon activation of a ses-

sion always is set to $\mathsf{running}_0$. When a session accepts the key of stage s it sets $status \coloneqq \mathsf{accepted}_s$; when a session rejects a key it sets $status \coloneqq \mathsf{rejected}_s$[1]; and when it continues the execution after the stage-$(s-1)$ key was accepted it sets $status \coloneqq \mathsf{running}_s$.

- $role \in \{\mathsf{initiator}, \mathsf{responder}\}$: The variable *role* holds the session owner's role in the protocol execution. Note that there is no default value and the role has to be defined during activation of a session.
- $auth \in \mathsf{AUTH}$. The variable *auth* holds the intended authentication type for this session. There is no default value and the type has to be defined during activation of the session.

  Note that in the sMSKE variant, this field can be omitted as the PSK provides already implicit mutual authentication. That is, whether explicit authentication is achieved becomes basically a protocol-wide property.
- $pid \in \mathcal{U} \cup \{*\}$: The variable *pid* holds the identity of the intended communication partner of the session owner. The value is defined during activation of the session, but we allow the identity in the pMSKE variant to be set during the execution of the protocol indicated by "$*$".
- $pskid \in \{0,1\}^*$. The variable *pskid* holds the identifier of the PSK that is used in this session. This field is only available in the sMSKE variant. For simplicity, we require in the sMSKE variant that *pskid* and *pid* are set upon activation and there actually is a PSK $\mathsf{pskeys}[u, pid, pskid]$ (if $role = \mathsf{initiator}$) or $\mathsf{pskeys}[pid, u, pskid]$ (otherwise) set. This follows the assumption that we do not cover PSK negotiation, but the model could be adapted to support this resulting in a more complex model.
- $stage \in [\mathsf{STAGES}]$: The variable *stage* holds the current stage of the session. Initially, this is set to 0 indicating that the execution just started. It is set to i if the state of execution changes to $\mathsf{accepted}_i$ or $\mathsf{rejected}_i$.
- $sid[s] \in \{0,1\}^*$: The variable $sid[s]$ holds the session identifier for stage s. Initially, all session identifiers are set to $\perp$ indicating that they are not defined. The session identifier $sid[s]$ is set only if stage s is accepted and only set once.
- $cid_{\mathsf{initiator}}[s], cid_{\mathsf{responder}}[s] \in \{0,1\}^*$: The variables $cid_{\mathsf{initiator}}[s]$ and $cid_{\mathsf{responder}}[s]$ hold the contributive identifier for stage s, i.e., the communication that a session and its communication partner (depending on its role) must have honestly received. This is allowed to be set multiple times and initially both are set to $\perp$ indicating that they are undefined.

  The introduction of a contributive identifier for each role was introduced by Davis, Diemert, Günther, and Jager [DDGJ22b] to allow for a more fine-grained testing. In prior MSKE variants, a session only keeps one contributive identifier and not one for itself and one for its partner.

[1] Note that we allow a session to continue its execution if some stage key was rejected.

- $skey[s] \in \mathcal{K}_s$: The variable $skey[s]$ holds the key established in stage s. Initially, it is set to $\perp$ to indicate that it is undefined and it (usually) is set once upon acceptance of stage s. However, there are situations in which is might be updated (e.g., if it is an internal key). For every stage s, the MSKE protocol defines a key space $\mathcal{K}_s$.

Interface of a MSKE protocol. Syntactically, a key exchange protocol KE is a pair of algorithms (Activate, Run). The input and output of the algorithms differs depending on the MSKE variant, i.e., pMSKE or sMSKE.

- Algorithm Activate creates a new session. In pMSKE, algorithm Activate receives as input the list of all user public keys pkeys, the user identifier of the session owner $u \in \mathcal{U}$, the session owner's secret key sk_u, the user identifier of the intended communication partner $pid \in \mathcal{U} \cup \{*\}$, a role $role \in \{\text{initiator}, \text{responder}\}$, and an authentication type $auth \in \text{AUTH}$. In sMSKE, the algorithm receives as input the user ID of the session owner $u \in \mathcal{U}$, the user identifier of the intended communication partner $pid \in \mathcal{U} \cup \{*\}$, a PSK psk and a role $role \in \{\text{initiator}, \text{responder}\}$.

  In both variants, algorithm Activate outputs a new session π_u^i and (potentially) a first message $m' \in \{0,1\}^* \cup \{\perp\}$, and initializes the internal state of π_u^i with the given inputs.
- Algorithm Run implements the processing of messages. In pMSKE, the algorithm Run receives as input the list of all user public keys pkeys, the session owner identifier u, the session owner's secret key sk_u, the session to be ran π_u^i and a message m to be processed. In sMSKE, the algorithm receives as input the session owner identifier u, a PSK psk, the session to be ran π_u^i and a message m to be processed.

  In both variants, it outputs the session π_u^i (with potentially updated state) and a response m'.

5.4 Security Game

One of the most important notions to formally define security for MSKE protocols is the notion of *partnering*. Informally, we say that two sessions are partners if they have "matching conversations" [BR94], i.e., the messages sent and received by the sessions match. Formally, we define this via the session identifiers maintained by every session as follows.

Definition 5.1 (Partnering). Let $\pi \neq \pi'$ be two distinct sessions of the key exchange protocol KE and let $s \in \text{STAGES}$ be a stage of KE. We say that session π is *partnered* to session π' *in stage* s if and only if $\pi.sid[s] = \pi'.sid[s] \neq \perp$.

Note that the session identifier is only set upon acceptance of a stage. That is, if the session identifier of a session is different from $\perp$ the session already accepted. The notion of partnering also dictates the correctness requirement we have for a MSKE protocol. Namely, we require that all pairs of sessions that executed the protocol honestly (i.e., the adversary only is passive) are partnered in all stages upon acceptance/termination.

Game variables. The security game keeps track of the time certain events occur during the execution of the game to resemble the order of events. To that end, it maintains a counter time, which initially is set to 0 and is incremented whenever an oracle query is issued. For each session π_u^i, the game holds the following values. In the following, we refer to oracle queries defined in detail below.

- accepted$[s]$: The variable π_u^i.accepted$[s]$ denotes the point in time in which session π_u^i accepted stage s.
- revealed$[s]$: The variable π_u^i.revealed$[s]$ denotes the point in time in which the adversary issued the query RevSessionKey(u, i, s) disclosing the stage s key. If π_u^i.revealed$[s] < \infty$, we say that stage s of π_u^i is *revealed*.
- tested$[s]$: The variable π_u^i.tested$[s]$ denotes the point in time in which the adversary issued the query Test(u, i, s) to be challenged on the stage-s key. If π_u^i.tested$[s] < \infty$, we say π_u^i is *tested* on stage s.

Additionally, the game also keeps track of the points in time when a long-term key corruption occurred. Here, we need to distinguish between the two MSKE variants. In the public-key variant pMSKE, the game holds the time when RevLongTermKey(u) was issued, which we denote by revsk$[u]$ for user u. If revsk$[u] < \infty$, we say user u or any of its sessions π_u^i is *corrupted*. In the pre-shared key variant sMSKE, we track the time when a PSK has been corrupted. To this end, we maintain a list similar to pskeys called revpsk. An entry indexed by $(u, v, pskid)$ represents the time at which the PSK pskeys$[u, v, pskid]$ has been revealed. Initially, all entries are initialized to ∞. As a shorthand, we write in Boolean expressions just revpsk$[u, v, pskid]$ instead of revpsk$[u, v, pskid] \neq \infty$. If revpsk$[u, v, pskid] < \infty$, we say $(u, v, pskid)$ (or the PSK shared between u and v) is *corrupted*.

Oracles. During the execution of the game the adversary interacts with the protocol via the following oracles. The code of the oracles defined below can be found in Figures 5.1 to 5.5.

- NewUser: The NewUser oracle is only present in the public-key variant pMSKE and the code is shown in Figure 5.1.

  Querying this oracle results in creating a new user with user identifier users + 1, where users is initially 0. For simplicity, we give users consecutive user identifiers, however one can adapt this to adversarially chosen user identifiers by introducing the user identifier as input to the oracle. For each new user a public-secret key pair is generated and the public key is returned to the adversary. A variant of this model would be to parameterize the model by the number of users involved in the protocol.
- NewSecret$(u, v, pskid)$: The NewSecret oracle is only present in the pre-shared key variant sMSKE and the code is shown in Figure 5.1.

  A query $(u, v, pskid)$ establishes a fresh (and honest) PSK between user u and its partner v with (adversarially chosen) PSK identifier $pskid$ if no PSK has been established with this triple before. In this work, we assume for simplicity that PSKs

NewUser

1 : time := time + 1
2 : users := users + 1
3 : $(pk_{\text{users}}, sk_{\text{users}}) \xleftarrow{\$} \text{Gen}$
4 : $\text{pkeys}[\text{users}] := pk_{\text{users}}$
5 : $\text{revsk}[\text{users}] := \infty$
6 : **return** pk_{users}

NewSecret$(u, v, pskid)$

1 : time := time + 1
2 : **if** $\text{pskeys}[u, v, pskid] \neq \bot$ **then**
3 : **return** $\bot$
4 : $\text{pskeys}[u, v, pskid] \xleftarrow{\$} \text{KE.PSKS}$
5 : $\text{revpsk}[u, v, pskid] := \infty$
6 : **return** $pskid$

Figure 5.1: Code of the oracles NewUser and NewSecret.

are honest and chosen uniformly at random from space KE.PSKS. However, we highlight that the model might be generalized to any distribution on KE.PSKS.

- Send(u, i, m): The code of the Send oracle is shown in Figure 5.2. Note that the two variants only differ in the inputs of Activate and Run.

  The Send oracle allows the adversary to interact with sessions. In general, a query (u, i, m) sends message m to the i-th session of user u denoted by π_u^i. If session π_u^i does not exist yet, we assume for simplicity that the adversary always sends the required information to activate a new session. The oracle then runs Activate to create a new session and compute the initial message m' of the session. If session π_u^i was already activated before, the oracle runs the algorithm Run for session π_u^i to process message m and compute a response m' (Line 10). If the session π_u^i changed its status to $\text{accepted}_{\pi_u^i.stage}$ meaning that it accepted in its current stage, the oracle additionally sets the time session π_u^i accepted to the current time. The final step now is to check whether there was a session before that accepted π_u^i's current stage with the same session identifier (i.e., π_u^i is partnered to). If this is the case, we set π_u^i's session key consistently. Note that this only applies to internal stages, as int_repr is only set in Test for internal keys (see Test below). In either case, the oracle returns m' to the adversary.

- RevSessionKey(u, i, s). The code of the RevSessionKey oracle is shown in Figure 5.3.

  Querying this oracle on (u, i, s) allows the adversary to get access to the session key of π_u^i established in stage s provided that π_u^i exists and accepted in stage s. If RevSessionKey(u, i, s) was issued we call π_u^i revealed in stage s.

- RevLongTermKey(u) / RevLongTermKey$(u, v, pskid)$: The former query is only used in the public-key variant pMSKE of the model and the latter only in the pre-shared key variant sMSKE. The code is shown in Figure 5.4.

  The RevLongTermKey oracle allows the adversary to get hold of a long-term key. In the public-key variant pMSKE, this means it receives the secret key sk_u, and in the pre-shared key variant sMSKE it receives the PSK $\text{pskeys}[u, v, pskid]$. If RevLongTermKey was queried for either a user or a PSK, we say that the user or the PSK is corrupted.

```
Send(u, i, m)
1 : time := time + 1
2 : if π_u^i = ⊥ then
3 :     (pid, role, auth) := m | (pid, pskid, role) := m
4 :     if role = initiator then
5 :         psk := pskeys[u, pid, pskid]
6 :     else
7 :         psk := pskeys[pid, u, pskid]
8 :     (π_u^i, m') ←$ Activate(pkeys, u, sk_u, pid, role, auth | u, pid, psk, role)
9 : else
10 :    (π_u^i, m') ←$ Run(pkeys, u, sk_u, π_u^i, m | u, π_u^i.psk, π_u^i, m)
11 : if π_u^i.status = accepted_{π_u^i.stage} then
12 :    s := π_u^i.stage
13 :    π_u^i.accepted[s] := time
14 :    //  Reprogram internal key
15 :    if int_repr[π_u^i.sid[s]] ≠ ⊥ then
16 :        π_u^i.skey[s] := int_repr[π_u^i.sid[s]]
17 : return m'
```

Figure 5.2: Code of the oracle Send. Code marked with $x \mid y$ indicates that x is only present in the pMSKE variant and y is only present in the sMSKE variant, while code marked only is only present in sMSKE and omitted in pMSKE.

```
RevSessionKey(u, i, s)
1 : time := time + 1
2 : if π_u^i = ⊥ ∨ π_u^i.accepted[s] = ∞ then
3 :     return ⊥
4 : π_u^i.revealed[s] := time
5 : return π_u^i.skey[s]
```

Figure 5.3: Code of the oracle RevSessionKey.

RevLongTermKey($\boxed{u \mid u, v, pskid}$)

1 : time := time + 1

2 : $\boxed{\mathsf{revsk}[u] \mid \mathsf{revpsk}[u, v, pskid]}$:= time

3 : **return** $\boxed{sk_u \mid \mathsf{pskeys}[u, v, pskid]}$

Figure 5.4: Code of the oracle RevLongTermKey. Code marked with $\boxed{x \mid y}$ indicates that x is executed in the pMSKE variant and y in the sMSKE variant.

Note that we follow [DDGJ22b] and capture forward secrecy in the Fresh predicate defined below. Previous MSKE models, e.g., [DFGS21], capture this in the corrupt oracle. Using the predicate allows us to keep the oracles simpler and capture misbehaviour of the adversary afterwards.

- Test(u, i, s): The code of the Test oracle is shown in Figure 5.5.

  The Test oracle allows the adversary to be challenged on a session and a stage. On a query (u, i, s), the adversary receives the real session key if the challenge bit $b = 0$ and a uniformly random key otherwise. There are a couple of cases in which we prohibit the adversary to issue a certain query (u, i, s). First of all, we exclude trivial Test queries (u, i, s), which are that session π_u^i is undefined, has not accepted stage s yet, or has already been tested before. Then, we allow for internal keys that they can only be tested before they have been used in the protocol to exclude trivial distinguishability of the challenge. As already mentioned in Section 5.2, allowing the adversary to test internal keys after these keys have been used opens a side channel for the adversary. For example, think of the internal key as a key that is used to encrypt key exchange messages. If the internal key now is used after it has been tested, the adversary could try to decrypt the corresponding communication using its "challenge key" and would potentially be able to distinguish the key from random when, e.g.,, the decrypted message follows the expected format of a protocol message.

  Lastly, we require in the pMSKE model that unauthenticated stages and responder in unilaterally authenticated stages can only be tested if they have a contributive partner. We capture this using the rectified authentication level introduced by Dowling, Fischlin, Günther, and Stebila [DFGS21]. To that end, we define $\pi_u^i.rect\_auth_s$ for a stage s and a session π_u^i as follows:

$$\pi_u^i.rect\_auth_s := \begin{cases} \mathsf{mutual} & \text{if } stage \geq auth_{s,2} \wedge \mathsf{corrupted} \geq \mathsf{accepted}[auth_{s,2}] \\ \mathsf{unilateral} & \text{if } stage \geq auth_{s,1} \wedge \mathsf{corrupted} \geq \mathsf{accepted}[auth_{s,1}] \\ \mathsf{unauth} & \text{otherwise} \end{cases}$$

  where $stage = \pi_u^i.stage$, $auth_{s,1}$ and $auth_{s,2}$ denote the first and second entry of $auth_s$ with $auth_s$ being the s-th entry of π_u^i intended authentication type $\pi_u^i.auth$,

Test(u, i, s)

```
1 :  time := time + 1
2 :  if π_u^i = ⊥ ∨ π_u^i.accepted[s] = ∞ ∨ π_u^i.tested[s] ≠ ∞ then
3 :     return ⊥
4 :  //  Internal keys can only be tested before use
5 :  if INT[s] ∧ ∃π_v^j : π_v^j.sid[s] = π_u^i.sid[s] ∧ π_v^j.accepted[s] < ∞
6 :          ∧ π_v^j.status ≠ accepted_s then
7 :     return ⊥
8 :  //  Unauth. stages and responder in unilaterally auth. stages can only
9 :  //  be tested with contributive partner
10 :  if (π_u^i.rect_auth_s = unauth ∨ π_u^i.rect_auth_s = unilateral
11 :              ∧ π_u^i.role = responder)
12 :          ∧ ∄π_v^j ≠ π_u^i : π_u^i.cid_{π_u^i.role}[s] = π_v^j.cid_{π_u^i.role}[s] then
13 :     return ⊥
14 :  π_u^i.tested[s] := time
15 :  T := T ∪ {(u, i, s)}
16 :  k_0 := π_u^i.skey[s]
17 :  k_1 ←$ K_s
18 :  //  For internal keys, replace skey[s] of π_u^i and all its partners by k_b
19 :  if INT[s] = true then
20 :     foreach π_v^j : π_v^j.sid[s] = π_u^i.sid[s] ∧ π_v^j.status = accepted_s do
21 :        π_v^j.skey[s] := k_b
22 :  //  Store k_b under sid to ensure consistency of later accepting sessions
23 :     int_repr[π_u^i.sid[s]] := k_b
24 :  return k_b
```

Figure 5.5: Code of the oracle Test.

and corrupted and accepted$[auth_{s,i}]$ denotes the time the owner of π_u^i was corrupted and the time $auth_{s,i}$ for $i = 1, 2$ was accepted, respectively. Using this definition ensures that the authentication level of stage s is only upgraded to unilateral when stage $auth_{s,1}$ is reached and only upgraded to mutual when stage $auth_{s,2}$ is reached if no corruption occurred prior to these stages. Note that this only applies to the pMSKE variant, as in the sMSKE variant, we already have mutual (implicit) authentication via the PSK throughout.

If the query (u, i, s) is eligible for testing, the oracle sets the time of π_u^i being tested to the current time, and adds (u, i, s) to the set $\mathcal{T}$ to register the tuple as "tested". Next, the oracle prepares two keys k_0 and k_1, where k_0 is the real session key derived by π_u^i in stage s and k_1 is a uniformly random key. For consistency, the oracle replaces the session key of all partners of π_u^i by the challenge key k_b and stores the challenge key indexed with π_u^i's session identifier for possible later reprogramming. See oracle Send for further detail. Finally, Test outputs the challenge key k_b.

Note that the code in the pMSKE and sMSKE variant for Test is identical.

Predicates. Using the Test oracle we captures the security property of key secrecy (i.e., indistinguishability from random) by a allowing the adversary to adaptively query a "real-or-random challenge" for various pairs of session and stage. However, the security of the protocol can not only be broken by distinguishing a session key from a random key, but also by violating other aspects of the protocol. To this end, we define predicates Sound, ExplAuth, and Fresh that are used in the security games for pMSKE and sMSKE to determine the final output of the game. Here, the adversary breaks the security of the protocol and thus winning the game if it is able to violate either the Sound or ExplAuth predicate, and the final guess b' of the challenge bit b output by the adversary is only considered valid if the Fresh predicate is satisfied. These predicates are defined as follows.

- Sound: The predicate Sound shown in Figure 5.6 captures that session identifiers identify partnered session as indented. In detail, this includes the following properties:
  1. Partnered session share the same key.
  2. Partnered sessions have different roles in non-replayable stages.
  3. Partnered sessions agree on the authentication type.
  4. Partnered sessions agree on their contributive identifiers.
  5. Partnered sessions agree on their authenticated partner.
  6. The session identifier does not match across different stages.
  7. For all non-replayable stages, there are at most two sessions partnered.

  Note that the code of the predicate is almost identical for the pMSKE and sMSKE variant. The only difference is that in the sMSKE we additionally require in Property 5 that partners in mutually authenticated stages (which holds for every stage in sMSKE because of the PSK) agree on the PSK identifier.

Sound

1 : // Partnering $\Longrightarrow$ same key
2 : **if** $\exists \pi_u^i \neq \pi_v^j, s : \pi_u^i.\mathsf{accepted}[s] < \infty \wedge \pi_v^j.\mathsf{accepted}[s] < \infty$
3 : $\quad \wedge\, \pi_u^i.sid[s] = \pi_v^j.sid[s] \neq \bot \wedge \pi_u^i.skey[s] \neq \pi_v^j.skey[s]$ **then**
4 : $\quad$ **return** false
5 : // Partnering $\Longrightarrow$ different roles (in non-replayable stages)
6 : $\exists \pi_u^i \neq \pi_v^j, s$ with $\pi_u^i.sid[s] = \pi_v^j.sid[s] \neq \bot \wedge \pi_u^i.role = \pi_v^j.role$
7 : $\quad \wedge\, (\mathsf{REPLAY}[s] = \mathsf{false} \vee \pi_u^i.role = \mathsf{initiator})$ **then**
8 : $\quad$ **return** false
9 : // Partnering $\Longrightarrow$ same authentication type
10 : **if** $\exists \pi_u^i \neq \pi_v^j, s : \pi_u^i.sid[s] = \pi_v^j.sid[s] \neq \bot \wedge \pi_u^i.auth_s \neq \pi_v^j.auth_s$ **then**
11 : $\quad$ **return** false
12 : // Partnering $\Longrightarrow$ agreement on contributive ids
13 : **if** $\exists \pi_u^i \neq \pi_v^j, s : \pi_u^i.sid[s] = \pi_v^j.sid[s] \neq \bot \wedge (\pi_u^i.cid_{\mathsf{initiator}}[s] \neq \pi_v^j.cid_{\mathsf{initiator}}[s]$
14 : $\quad \vee\, \pi_u^i.cid_{\mathsf{responder}}[s] \neq \pi_v^j.cid_{\mathsf{responder}}[s])$ **then**
15 : $\quad$ **return** false
16 : // Partnering $\Longrightarrow$ agreement on authenticated partner
17 : **if** $\exists \pi_u^i \neq \pi_v^j, s \leq t : \pi_u^i.role = \mathsf{initiator} \wedge \pi_v^j.role = \mathsf{responder}$
18 : $\quad \wedge\, \pi_u^i.sid[s] = \pi_v^j.sid[s] \neq \bot \wedge \pi_u^i.sid[t] = \pi_v^j.sid[t] \neq \bot$
19 : $\quad \wedge\, ([\pi_u^i.auth_{s,1} \leq t \wedge \pi_u^i.pid \neq v]$
20 : $\quad\quad \vee\, [\pi_u^i.auth_{s,2} \leq t \wedge (\pi_v^j.pid \neq u \vee \pi_u^i.pskid \neq \pi_v^j.pskid)])$ **then**
21 : $\quad$ **return** false
22 : // Different stages do not share a session id
23 : **if** $\exists \pi_u^i, \pi_v^j, s \neq t : \pi_u^i.sid[s] = \pi_v^j.sid[t] \neq \bot$ **then**
24 : $\quad$ **return** false
25 : // In non-replayable stages, there are at most two sessions partnered
26 : **if** $\exists s,$ pairwise distinct $\pi_u^i, \pi_v^j, \pi_w^k$ with $\pi_u^i.sid[s] = \pi_v^j.sid[s] = \pi_w^k.sid[s] \neq \bot$
27 : $\quad \wedge\, \mathsf{REPLAY}[s] = \mathsf{false}$ **then**
28 : $\quad$ **return** false
29 : **return** true

Figure 5.6: Code of predicate Sound. Code marked with is only present in the sMSKE variant. The rest is identical in both the pMSKE and sMSKE variant.

- Fresh: The predicate Fresh shown in Figure 5.7 captures whether the adversary behaved correctly during the execution of the security experiment. In particular, it captures whether the adversary tested any session for which the challenge is trivially distinguishable. This occurs if the adversary tests a session on a stage and reveals the session key of that stage at any point in time.

  Note that the predicate in the pMSKE and sMSKE only differs in the syntax of how long-term key reveal is measured. Namely, in pMSKE, we need to check whether the session owner's secret key has been revealed. Whereas, in sMSKE, we need to check whether the session's PSK has been revealed.

- ExplAuth: The predicate ExplAuth shown in Figure 5.8 captures whether the adversary breaks the explicit authentication during the execution of the security experiment. Explicit authentication is broken if there is a session that *maliciously accepts* in a stage and a role at which explicit authentication is supposed to be established. Here, malicious acceptance means that a session accepts a stage that establishes explicit authentication with an uncorrupted peer, but there is no honest partner session. For corrupted peers we expect the adversary to trivially make a session accept without an honest partner.

  Note that the code of predicate ExplAuth in pMSKE and in sMSKE only differs in how long term key reveal is measured and that in sMSKE the partnered session also needs to share the same PSK captured by agreement on the PSK identifier (Line 13).

Games. Now that we have defined all oracles and all predicates, we are prepared to define the security games. We define MSKE in two variants: the first variant is pMSKE, which refers to the public key variant, and the second one is sMSKE, which refers to the pre-shared key variant, following [DFGS21]. The public key variant, adapts the code-based pre-shared key variant from [DDGJ22b] to the public key setting. The two security games only differ in the oracles the adversary is provided with and the exact implementation of the oracle depending of the MSKE variant as discussed in the definition of the oracles above. The final pMSKE model can also be seen as an adaptation of the code-based game of [DG21a] into the multi-stage setting.

Definition 5.2 (Public-key MSKE). Let KE be a pMSKE protocol with properties (STAGES, AUTH, FS, INT, REPLAY). We define the *advantage of an adversary $\mathcal{A}$ in breaking* pMSKE*-security of* KE as

$$\mathrm{Adv}^{\mathsf{pMSKE}}_{\mathsf{KE}}(\mathcal{A}) := 2 \cdot \Pr[\mathrm{Exp}^{\mathsf{pMSKE}}_{\mathsf{KE}}(\mathcal{A}) = 1] - 1$$

where $\mathrm{Exp}^{\mathsf{pMSKE}}_{\mathsf{KE}}(\mathcal{A})$ is defined in Figure 5.9.

Definition 5.3 (Pre-Shared Key MSKE). Let KE be a MSKE protocol with properties (STAGES, AUTH, FS, INT, REPLAY). We define the *advantage of an adversary $\mathcal{A}$ in breaking* sMSKE*-security of* KE as

$$\mathrm{Adv}^{\mathsf{sMSKE}}_{\mathsf{KE}}(\mathcal{A}) := 2 \cdot \Pr[\mathrm{Exp}^{\mathsf{sMSKE}}_{\mathsf{KE}}(\mathcal{A}) = 1] - 1$$

where $\mathrm{Exp}^{\mathsf{sMSKE}}_{\mathsf{KE}}(\mathcal{A})$ is defined in Figure 5.10.

Fresh

1 : **foreach** $(u, i, s) \in \mathcal{T}$ **do**
2 : $\quad t_{\text{test}} := \pi_u^i.\mathsf{tested}[s]$
3 : $\quad$ // Tested sessions not revealed
4 : $\quad$ **if** $\pi_u^i.\mathsf{revealed}[s] < \infty$ **then**
5 : $\quad\quad$ **return** false
6 : $\quad$ // Partners of tested session may neither be tested nor revealed
7 : $\quad$ **if** $\exists \pi_v^j \neq \pi_u^i \;:\; \pi_v^j.sid[s] = \pi_u^i.sid[s] \wedge (\pi_v^j.\mathsf{tested}[s] \vee \pi_v^j.\mathsf{revealed}[s])$ **then**
8 : $\quad\quad$ **return** false
9 : $\quad$ // Sessions tested on forward secret stages are fresh if corrupted
10 : $\quad$ // after fs was achieved or if they have a contributive partner
11 : $\quad$ **if** $\pi_u^i.\mathsf{accepted}[\mathsf{FS}[\pi_u^i.role, s, \mathsf{fs}]] < t_{\text{test}}$ **then**
12 : $\quad\quad$ **if** $\boxed{\mathsf{revsk}[\pi_u^i.pid] \mid \mathsf{revpsk}[u, \pi_u^i.pid, \pi_u^i.pskid]}$
13 : $\quad\quad\quad < \pi_u^i.\mathsf{accepted}[\mathsf{FS}[\pi_u^i.role, s, \mathsf{fs}]]$
14 : $\quad\quad\quad\quad \wedge \nexists \pi_v^j \neq \pi_u^i \;:\; \pi_u^i.cid_{\pi_u^i.role}[\mathsf{FS}[\pi_u^i.role, s, \mathsf{fs}]]$
15 : $\quad\quad\quad\quad\quad = \pi_v^j.cid_{\pi_u^i.role}[\mathsf{FS}[\pi_u^i.role, s, \mathsf{fs}]]$ **then**
16 : $\quad\quad\quad$ **return** false
17 : $\quad$ // Sessions tested on wfs2 stages are fresh if never corrupted
18 : $\quad$ // or if they have a contributive partner
19 : $\quad$ **elseif** $\pi_u^i.\mathsf{accepted}[\mathsf{FS}[\pi_u^i.role, s, \mathsf{wfs2}]] < t_{\text{test}}$ **then**
20 : $\quad\quad$ **if** $\boxed{\mathsf{revsk}[\pi_u^i.pid] \mid \mathsf{revpsk}[u, \pi_u^i.pid, \pi_u^i.pskid]} < \infty$
21 : $\quad\quad\quad \wedge \nexists \pi_v^j \neq \pi_u^i \;:\; \pi_u^i.cid_{\pi_u^i.role}[\mathsf{FS}[\pi_u^i.role, s, \mathsf{wfs2}]]$
22 : $\quad\quad\quad\quad = \pi_v^j.cid_{\pi_u^i.role}[\mathsf{FS}[\pi_u^i.role, s, \mathsf{wfs2}]]$ **then**
23 : $\quad\quad\quad$ **return** false
24 : $\quad$ // Session on non-fs stages are fresh if never corrupted.
25 : $\quad$ **elseif** $\boxed{\mathsf{revsk}[\pi_u^i.pid] \mid \mathsf{revpsk}[u, \pi_u^i.pid, \pi_u^i.pskid]} < \infty$ **then**
26 : $\quad\quad$ **return** false
27 : **return** true

Figure 5.7: Code of predicate Fresh. Code marked with $\boxed{x \mid y}$ indicates that x is only present in the pMSKE variant and y is only present in the sMSKE variant.

ExplAuth

1 : **return** $\forall(\pi_u^i, s)$:
2 : $s' \leftarrow \mathsf{EAUTH}[\pi_u^i.role, s]$
3 : // Sessions accepting in explicitly authenticated stages
4 : // that were not corrupted before acceptance of the stage
5 : // at which explicit authentication was (perhaps
6 : // retroactively) established...
7 : $\pi_u^i.\mathsf{accepted}[s] < \infty \wedge \pi_u^i.\mathsf{accepted}[s'] < \infty$
8 : $\wedge\ \pi_u^i.\mathsf{accepted}[s'] < \boxed{\mathsf{revsk}[\pi_u^i.pid] \mid \mathsf{revpsk}[u, \pi_u^i.pid, \pi_u^i.pskid]}$
9 : // are partnered
10 : $\implies \exists \pi_v^j : \pi_u^i.sid[s'] = \pi_v^j.sid[s']$
11 : // with the intended peer
12 : $\wedge\ \pi_u^i.pid = v$
13 : $\wedge\ \pi_u^i.pskid = \pi_v^j.pskid$
14 : // and are also partnered (upon acceptance) in the stage that
15 : // (perhaps retroactively) got expl. auth.
16 : $\wedge\ (\pi_v^j.\mathsf{accepted}[s] < \infty \implies \pi_v^j.sid[s] = \pi_u^i.sid[s])$
17 : **return** true

Figure 5.8: Code of predicate ExplAuth. Code marked with $\boxed{x \mid y}$ indicates that x is only present in the pMSKE variant and y is only present in the sMSKE variant, while code marked only is only present in sMSKE and omitted in pMSKE.

$\mathrm{Exp}_{\mathsf{KE}}^{\mathsf{pMSKE}}(\mathcal{A})$

1 : time := 0; users := 0; $b \xleftarrow{\$} \{0,1\}$
2 : $b' \xleftarrow{\$} \mathcal{A}^{\mathsf{NewUser},\mathsf{Send}(\cdot,\cdot,\cdot),\mathsf{RevSessionKey}(\cdot,\cdot,\cdot),\mathsf{RevLongTermKey}(\cdot),\mathsf{Test}(\cdot,\cdot,\cdot)}$
3 : **if** $\neg$Sound $\vee$ $\neg$ExplAuth **then**
4 : **return** 1
5 : **if** $\neg$Fresh **then**
6 : $b' := 0$
7 : **return** $(b = b')$

Figure 5.9: Code of the pMSKE game.

$\mathrm{Exp}^{\mathsf{sMSKE}}_{\mathsf{KE}}(\mathcal{A})$

1 : $\mathsf{time} := 0;\ b \xleftarrow{\$} \{0,1\}$
2 : $b' \xleftarrow{\$} \mathcal{A}^{\mathsf{NewSecret}(\cdot,\cdot,\cdot),\mathsf{Send}(\cdot,\cdot,\cdot),\mathsf{RevSessionKey}(\cdot,\cdot,\cdot),\mathsf{RevLongTermKey}(\cdot,\cdot,\cdot),\mathsf{Test}(\cdot,\cdot,\cdot)}$
3 : **if** $\neg\mathsf{Sound} \vee \neg\mathsf{ExplAuth}$ **then**
4 : **return** 1
5 : **if** $\neg\mathsf{Fresh}$ **then**
6 : $b' := 0$
7 : **return** $(b = b')$

Figure 5.10: Code of the sMSKE game.

5.5 Multi-stage Session Matching

The notion of (public) session matching was considered by numerous works when it comes to the composition of a key exchange protocol and an symmetric-key protocol (e.g., a channel protocol). Brzuska, Fischlin, Warinschi, and Williams [BFWW11] used this notion for the composition of "single-stage" key exchange protocols secure in the Bellare–Rogaway [BR94] model. This was extended for the multi-stage setting by Dowling, Fischlin, Günther, and Stebila [DFGS15] and Günther [Gün18]. We recall the definition of a *multi-stage session matching algorithm* stated by Günther [Gün18].

Definition 5.4. Let $\Pi \in \{\mathsf{pMSKE}, \mathsf{sMSKE}\}$. Let $\mathcal{A}$ be any adversary interacting in the experiment $\mathrm{Exp}^{\Pi}_{\mathsf{KE}}(\mathcal{A})$. We say an algorithm $\mathcal{M}$ is a *multi-stage session matching algorithm* if the following holds. On input a stage i, the public information of the experiment, an ordered list of all queries made by $\mathcal{A}$ and responses from $\mathrm{Exp}^{\Pi}_{\mathsf{KE}}(\mathcal{A})$ at any point of the experiment execution, as well as a list of all stage-j keys with $j < i$ for all session accepted at this point, algorithm $\mathcal{M}$ outputs two lists of pairs of all session in stage i. Here, the first list contains exactly those pairs of sessions that are partnered (i.e., they share the same session identifier sid_i). The second list contains exactly those pairs of sessions that are contributively partnered (i.e., they share the same contributive identifier cid_i).

Note that the session matching algorithms can be run at any point of the execution of the key exchange protocol.

6

TRANSPORT LAYER SECURITY HANDSHAKE PROTOCOL

Contents

In this chapter, we describe the cryptographic core of the TLS 1.3 handshake protocol standardized as RFC 8446 [Res18]. The handshake protocol comes in two major modes:

1. the full one round-trip time (1-RTT) (EC)DHE handshake, and
2. the (abbreviated) pre-shared key (PSK) handshake with optional zero round-trip time (0-RTT) data.

In Section 6.4, we focus on the description of the full handshake and in Section 6.5 we focus on the description of the PSK mode. Before we describe the two handshakes, we first introduce in Section 6.1 the two important building blocks of the handshake, HMAC and HKDF, clarify our view on the handshake, particularly, which features of TLS we are not considering in Section 6.2, and introduce specific notation for the presentation of the handshakes in Section 6.3.

6.1 HMAC and HKDF

In this section, we recap the definitions of the functions HMAC and HKDF. These two functions are an important building block of the TLS 1.3 handshake protocol. HKDF is the main building block of the TLS 1.3 key schedule [Res18, Sect. 7.1], which is the

key derivation procedure of TLS 1.3, and it is build from HMAC. Additionally, HMAC is used as a MAC scheme (Section 4.3) in TLS 1.3. For further detail, we refer to the reader description of the TLS 1.3 handshake protocol presented in Sections 6.4 and 6.5.

6.1.1 HMAC

A prominent example of a deterministic MAC is HMAC [BCK96, KBC97]. It is based on a cryptographic hash function H (Section 4.1) and it is defined as follows: Let H be a cryptographic hash function with output length $\lambda \in \mathbb{N}$ and let be $\kappa \in \mathbb{N}$ (called the key length).

Key generation. Algorithm HMAC.Gen chooses a key $k \xleftarrow{\$} \{0,1\}^\kappa$ and returns k.

Tagging. Algorithm HMAC.Tag on input a key k and a message m returns

$$t := H\big((k \oplus \mathsf{opad}) \,\|\, H((k \oplus \mathsf{ipad}) \,\|\, m)\big),$$

where opad and ipad are according to RFC 2104 [KBC97] the bytes 0x5c and 0x36 repeated B-times, respectively, for block size B (in bytes) of the hash function H.

Verification. Algorithm HMAC.Vrfy on input a key k, a message m and a tag t returns 1 if and only if $t = \mathsf{HMAC.Tag}(k, m)$.

Remark 6.1. The key k is padded with zeroes to match the block size B. If key k should be larger than the block size B, then it is compressed using the hash function and then padded to the right length as before. In this work, we only consider keys smaller than the block size, so that an additional hashing operation is not required.

Remark 6.2. In this thesis, we often refer to the HMAC function. As HMAC is not only a MAC scheme, but also can be viewed as a PRF (cf. [BCK96, Bel06, Bel15]). Therefore, we also define

$$\mathsf{HMAC}(k, m) := H\big((k \oplus \mathsf{opad}) \,\|\, H((k \oplus \mathsf{ipad}) \,\|\, m)\big).$$

6.1.2 HMAC-based Extract-and-Expand Key Derivation Function (HKDF)

The core of the TLS 1.3 key derivation [Res18, Sect. 7.1] is the *key derivation function (KDF)* HKDF proposed by Krawczyk [Kra10b, Kra10a] and standardized in RFC 5869 [KE10]. It follows the *extract-and-expand* [Kra10a] paradigm and is based on HMAC. The algorithm consists of two subroutines HKDF.Extract and HKDF.Expand. The function HKDF.Extract is a *randomness extractor* [NT99, NZ96] that on input a (non-secret and possibly fixed) extractor salt *xts* and a (not necessarily uniformly distributed) source key material *skm* outputs a pseudorandom key *prk*. The function HKDF.Expand is a *variable output length PRF*[1] that on input *prk*, (potentially empty) context information *ctx* and length parameter L outputs a pseudorandom key *km* of length L. That is, HKDF.Expand intuitively *expands* the input key *prk* to length L.

[1] A variable output length PRF is a variant of the PRFs introduced in Section 4.2. Informally, this primitive takes the desired output length as input and outputs a pseudorandom string of the desired length.

Construction. Intuitively, HKDF derives a pseudorandom key (i.e., indistinguishable from a uniformly sampled key) from some source key material and then expands this pseudorandom key to the desired length. Formally, we have the following construction, which is based on the HMAC function (cf. Section 6.1.1 and Remark 6.2).

1. $prk := \mathsf{HKDF.Extract}(xts, skm) = \mathsf{HMAC}(xts, skm)$
2. $km = K(1) \parallel \cdots \parallel K(\omega) := \mathsf{HKDF.Expand}(prk, ctx, L)$, where $\omega := \lceil L/\lambda \rceil$, λ is the output length of the underlying hash function used in HMAC and $K(i)$ is inductively defined by
   - $K(1) := \mathsf{HMAC}(prk, ctx \parallel \mathtt{0x01})$, and
   - $K(i) := \mathsf{HMAC}(prk, K(i-1) \parallel ctx \parallel \langle i \rangle)$ for $2 \leq i \leq \omega$ and $\langle i \rangle$ denotes the encoding of the integer $i+1$ in a single byte[2].

   $K(\omega)$ is simply truncated to the first ($L \bmod \lambda$) bits to get the desired length L.

We overload notation to denote by $\mathsf{HKDF.Expand}(prk, ctx)$ the function described above for a fixed length parameter L that is clear from the context.

Even though we never use it in that way, we denote by the function HKDF the execution of the functions HKDF.Extract and HKDF.Expand in sequence. That is, on input (xts, skm, ctx, L) it computes $prk := \mathsf{HKDF.Extract}(xts, skm)$ and outputs km with $km := \mathsf{HKDF.Expand}(prk, ctx, L)$.

6.2 Omitted Features of TLS

The cryptographic core of the TLS 1.3 handshake protocol we are considering in this thesis is a similar abstraction of the TLS handshake as in previous computational analyses of TLS 1.3 [DFGS21] and its predecessor TLS 1.2 (e.g., [JKSS12, KPW13]). In detail, we omit the following features.

Version negotiation. The standard allows clients to provide a list of supported TLS versions (via the `supported_versions` extension [Res18, Sect. 4.2.1]) so that prior TLS versions might be negotiated in case the server does not support TLS 1.3. This feature is out of scope of this work. We only consider negotiation of TLS 1.3. In particular, this means that we omit backward compatability features supported by the TLS 1.3 standard. Version negotiation and backward compatability was considered in [Bha+16, DS15].

Algorithm negotiation. TLS allows the client to present a selection of different cipher suites (i.e., pairs of authenticated encryption with associated data (AEAD) algorithm and hash algorithm; cf. [Res18, App. B.4]) from which the server selects its preferred one. The same holds for the DH group and signature algorithm used in some modes. However, the *negotiation* of these parameters is out of scope in this

[2] Note that this implies $L \leq 255 \cdot \lambda$.

work. We consider a selection of algorithms fixed once and for all before the protocol starts. This means our view captures *all possible* combinations of algorithms in every handshake mode, however we only consider each configuration in isolation.

`*Hello` extensions. The initial messages of the client and the server are called the `ClientHello` and `ServerHello` message, respectively, and allow for various *extensions* to be appended. These are, for example, used to negotiate supported versions, supported signature algorithms, etc. For details, we refer the reader to the specification [Res18, Sect. 4.2]. As most of the features are omitted in this work, we also omit the corresponding extensions. We only include the extensions that are mandatory for the negotiation of the cryptographic keys and explicitly mention the extensions appended in the protocol descriptions below.

Post-handshake messages. TLS allows for some messages to be sent after the main handshake (as we present it below). These message are defined in [Res18, Sect. 4.6] and include messages to issue a session ticket to define a new PSK, for post-handshake authentication and to notify the communication partner that a key update occurs. We cover neither of these messages in this work. Nevertheless, we briefly discuss the post-handshake session ticket in Section 6.5 to illustrate how PSKs are exchanged, but do not include the exchange of the respective message in our analyses. Post-handshake authentication allows the server to request client authentication (via certificate) at any point in time after the handshake has been completed. This requires that the client has sent the `post_handshake_auth` extension. We do not cover post-handshake client-authentication in this work and only cover either unilateral or mutual authentication *during* the (full) handshake via certificates.

Record layer protocol. In this section, we solely focus on the TLS 1.3 handshake protocol and do not cover the secure channel of the TLS 1.3 record layer protocol [Res18, Sect. 5]. We briefly discuss the record layer protocol in Chapter 11.

Alert protocol. The TLS 1.3 alert protocol [Res18, Sect. 6] handles closure and error information by defining a specific `Alert` content type of messages. In this work, we exclude the alert protocol from our view entirely.

6.3 Notation

Next, we introduce notation that we use in the description of two handshake variants in the following Sections 6.4 and 6.5. In the sequel, we denote the used AEAD scheme by AEAD, the hash algorithm by $\mathbf{H}$, the DH group by $\mathbb{G}$ and the signature scheme by Sig. The output length of the hash function $\mathbf{H}$ is denoted by $\lambda \in \mathbb{N}$, i.e., $\mathbf{H} : \{0,1\}^* \to \{0,1\}^\lambda$, and the prime order of the group $\mathbb{G}$ is denoted by p. TLS 1.3 makes use of its hash function in various ways. It is used to condense transcripts, as a subroutine to derive keys, and as a subroutine of the message authentication code computed during the handshake. We define the following three subroutines used in the TLS 1.3 handshake:

- The message authentication code **MAC** with

$$\mathbf{MAC}: \{0,1\}^{\lambda} \times \{0,1\}^{*} \to \{0,1\}^{\lambda}, \quad (k,m) \mapsto \mathsf{HMAC}[\mathbf{H}](k,m)$$

  using the hash function **H** in HMAC (Section 6.1.1).

- The functions **Extract** and **Expand** used in the key schedule with

$$\mathbf{Extract}: \{0,1\}^{\lambda} \times \{0,1\}^{*} \to \{0,1\}^{\lambda}, \quad (k,m) \mapsto \mathsf{HKDF.Extract}(k,m),$$

  and

$$\mathbf{Expand}: \{0,1\}^{\lambda} \times \{0,1\}^{*} \times \mathbb{N} \to \{0,1\}^{\lambda}, \quad (k,m,L) \mapsto \mathsf{HKDF.Expand}(k,m,L).$$

  Here, HKDF.Extract and HKDF.Expand (Section 6.1.2) are instantiated with **MAC** (resp. HMAC[**H**]). For almost every application of **Expand** (except the traffic key derivations) the output length parameter L is equal to the output length λ of the hash function. If not explicitly given, we omit L and set it implicitly to $L = \lambda$. In this case, it holds that $\mathbf{Expand}(k,m) = \mathbf{MAC}(k, m \,\|\, \texttt{0x01})$.

  The secrets derived (cf. Table 6.1) during the handshake are in the standard [Res18] defined by calls of the form

$$\texttt{Value} = \mathbf{Expand}(\texttt{Secret}, \texttt{HkdfLabel}, \texttt{Length}),$$

  where `Secret` is the base secret `Value` is derived from, `HkdfLabel` = `Length` ‖ `"tls13 "` ‖ `Label` ‖ `Context`, and `Length` is the output length of the cipher suite hash algorithm λ for all calls of **Expand** in the TLS handshakes. For simplicity, we cut the constant overhead of `Length`‖`"tls13 "` and only consider `Label`‖`Context` as input to **Expand**. Table 6.1 shows the values of base secret `Secret`, label `Label`, and context (hash) `Context` used during the handshake to derive secret `Value`.

Note that the definition of **H**, **MAC**, **Extract** and **Expand** closely reflects the TLS 1.3 standard [Res18] despite the different naming. This abstraction is only conceptual and aims to highlight which parts of the handshake make use of the hash function. In our analysis, we model the hash function as a random oracle and this abstraction helps to separate each use of the hash function precisely, and this is discussed in more detail in Chapter 7.

6.4 TLS 1.3 Full (EC)DHE Handshake

The full TLS 1.3 (EC)DHE handshake protocol is depicted in Figure 6.1. An overview of labels and transcript hashes used to derive the secrets during the handshake is given in Table 6.1 and an overview of the information used to derive the authentication messages (i.e., signatures and finished tags) is given in Table 6.2.

In the following, we describe the messages exchanged during the handshake in detail. We use the terminology used in the specification RFC8446 [Res18]. For further detail, we also refer to this specification.

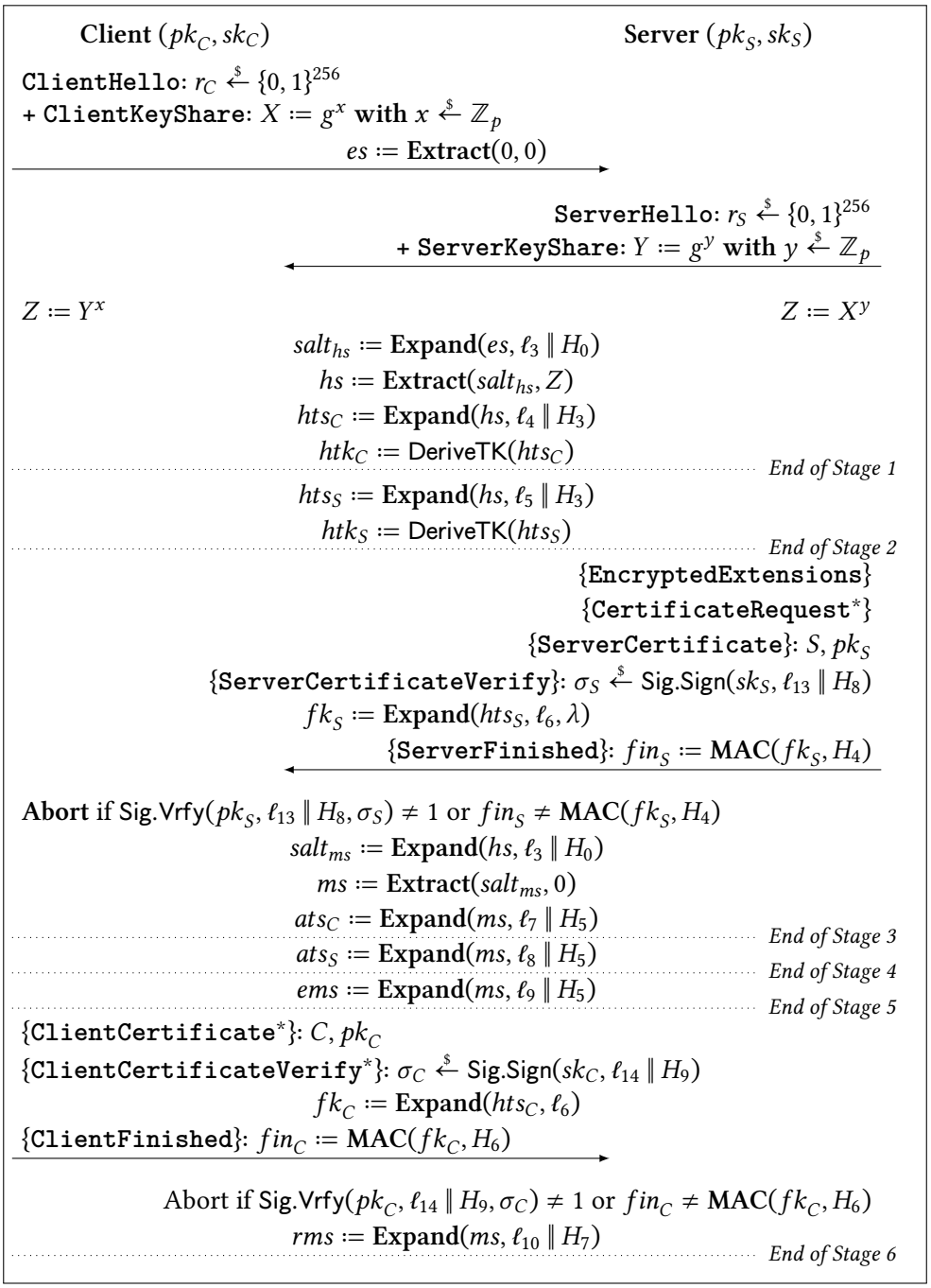

Figure 6.1: TLS 1.3 full (EC)DHE handshake. Every TLS handshake message is denoted as "`MSG` : C", where C denotes the message's content. Similarly, an extension is denoted by "`+ MSG` : C". Further, we denote by "{`MSG`} : C" messages containing C and being AEAD-encrypted under the handshake traffic key *htk*. A message "`MSG`*" is an optional, resp. context-dependent message. Centered computations are executed by both client and server with their respective messages received, and possibly at different points in time.

Table 6.1: Definition of the labels and transcript hashes used to derive the secrets using **Expand**. The messages marked with [·] are only present in the full and PSK-(EC)DHE handshake. A hash $\mathbf{H}(m \| \cdots \| m')$ is always over all message from m to m' inclusive, i.e., a hash is not always computed over the same number of messages as in some modes/configurations some intermediate message are omitted.

`Value`	`Secret`	`Label`	`Context` (Transcript Hash)
bk	es	ℓ_0 = `"ext binder"/"res binder"`	$H_0 = H(\varepsilon) = \mathbf{H}(\texttt{""})$
fk_{B}	bk	ℓ_6 = `"finished"`	ε = `""`
ets	es	ℓ_1 = `"c e traffic"`	$H_2 = \mathbf{H}(\texttt{CH} \| [\texttt{CKS}] \| \texttt{CPSK})$
$eems$	es	ℓ_2 = `"e exp master"`	$H_2 = \mathbf{H}(\texttt{CH} \| [\texttt{CKS}] \| \texttt{CPSK})$
$salt_{hs}$	es	ℓ_3 = `"derived"`	$H_0 = H(\varepsilon) = \mathbf{H}(\texttt{""})$
hts_C	hs	ℓ_4 = `"c hs traffic"`	$H_3 = \mathbf{H}(\texttt{CH} \| [\texttt{CKS}] \| \texttt{SH} \| [\texttt{SKS}])$
hts_S	hs	ℓ_5 = `"s hs traffic"`	$H_3 = \mathbf{H}(\texttt{CH} \| [\texttt{CKS}] \| \texttt{SH} \| [\texttt{SKS}])$
fk_S	hts_S	ℓ_6 = `"finished"`	ε = `""`
$salt_{ms}$	hs	ℓ_3 = `"derived"`	$H_0 = H(\varepsilon) = \mathbf{H}(\texttt{""})$
ats_C	ms	ℓ_7 = `"c ap traffic"`	$H_5 = \mathbf{H}(\texttt{CH} \| \cdots \| \texttt{SF})$
ats_S	ms	ℓ_8 = `"s ap traffic"`	$H_5 = \mathbf{H}(\texttt{CH} \| \cdots \| \texttt{SF})$
ems	ms	ℓ_9 = `"exp master"`	$H_5 = \mathbf{H}(\texttt{CH} \| \cdots \| \texttt{SF})$
fk_C	hts_C	ℓ_6 = `"finished"`	ε = `""`
rms	ms	ℓ_{10} = `"res master"`	$H_7 = \mathbf{H}(\texttt{CH} \| \cdots \| \texttt{SF} \| \cdots \| \texttt{CF})$

Table 6.2: Definition of the context string and content of the authentication messages (signatures and finished messages). The signatures are computed with the considered digital signature algorithm under the party's secret key and the message signed is `Context` ‖ `Content`. The finished messages are computed as the **MAC** of `Content` under the party's finished key. Note that messages for the transcript hashes are given for the full handshake below. In the PSK mode, the hash values for the finished values are computed over the same message, but all signature-related messages are omitted. The messages marked with [·] are only present in the full and PSK-(EC)DHE handshake.

Auth. Msg.	`Context` (Sig. only)	`Content`
$binder$	—	$H_1 = \mathbf{H}(\texttt{CH} \| [\texttt{CKS}] \| \texttt{CPSK}^-)$
σ_S	ℓ_{13} = `"TLS 1.3, server CertificateVerify"`	$H_8 = \mathbf{H}(\texttt{CH} \| \cdots \| \texttt{SKS} \| \texttt{EE} \| \texttt{CR}^* \| \texttt{SCRT})$
fin_S	—	$H_4 = \mathbf{H}(\texttt{CH} \| \cdots \| \texttt{SCRT})$
σ_C	ℓ_{14} = `"TLS 1.3, client CertificateVerify"`	$H_9 = \mathbf{H}(\texttt{CH} \| \cdots \| \texttt{SCV} \| \texttt{SF} \| \texttt{CCRT}^*)$
fin_C	—	$H_6 = \mathbf{H}(\texttt{CH} \| \cdots \| \texttt{CCRT}^* \| \texttt{CCV}^*)$

`ClientHello` (`CH`): The `ClientHello` message is the first message of the TLS 1.3 handshake and is used by a client to initiate the protocol with a server. The message itself contains information for the negotiation of the used version and the algorithms to be used in the protocol run. Most of them are not relevant to our analysis, because we do not consider negotiation of parameters.[3] In particular, we only consider the negotiation of TLS 1.3 (i.e., no negotiation of versions prior to TLS 1.3 or backwards compatibility), and we consider the algorithms (i.e., the cipher suite) to be fixed once and for all. As described earlier, our perspective covers all possible combinations of groups and cipher suites, but only in isolation. Moreover, the `ClientHello` message contains a field called `random`, which is the random nonce chosen by the client consisting of a 256-bit value r_C.

There are various extensions added to this message in the `extensions` field. For our view only the `key_share` extension [Res18, Sect. 4.2.8] is important. We denote this as a separate message called `ClientKeyShare` described next.

`ClientKeyShare` (`CKS`): The `key_share` extension of the `ClientHello` message consists of the public DHE value X chosen by the client. It is defined as $X := g^x$, where $x \xleftarrow{\$} \mathbb{Z}_p$ is the client's private DHE exponent and g the generator of the considered group $\mathbb{G}$. Note that if the negotiation of the group would be considered the standard allows for the client to present multiple groups it supports and thus also presents a key share for each of these groups. However, here it only contains a single key share as we only consider a single group, which is fixed once and for all before the execution of the protocol.

`ServerHello` (`SH`): In response to the `ClientHello` the server sends the `ServerHello`. This message is structured similarly to the `ClientHello` message. If version and algorithm negotiation would be considered, the server would now pick parameters from the values presented by the client. However, in our view again only the `random` field is of importance. Here, we denote the 256-bit random value chosen by the server by r_S.

As defined in the standard, the server would now pick a group among the the supported groups of the client, and responds with a `key_share` extension containing its key share for the selected group. Since we do not consider negotiation, the server does not need to pick a group, but responds with a key share for the fixed group $\mathbb{G}$. Similar to `ClientKeyShare`, we denote the server's key share extension by `ServerKeyShare` as a separate message.

`ServerKeyShare` (`SKS`): This message consists of the server's public DHE value Y. It is defined as $Y := g^y$, where $y \xleftarrow{\$} \mathbb{Z}_p$ is the server's private DHE exponent and g the generator of $\mathbb{G}$.

After this message is computed the server is ready to compute the *handshake traffic keys* used to encrypt the communication of the handshake from this point on. During the

[3] In Section 7.5, we discuss the structure of the message much more in detail.

handshake two handshake traffic keys are derived, namely one for each direction of communication. To this end, the server first computes the exchanged DHE key $Z := X^y$, where X is the client's public DHE value sent in the `ClientKeyShare` message. Using Z, it computes the *handshake secret* $hs := \mathbf{Extract}(salt_{hs}, Z)$ with $salt_{hs} := \mathbf{Expand}(es, \ell_3 \parallel H_0, \lambda)$ and $es := \mathbf{Extract}(0, 0)$. The handshake secret is then used to derive the *client handshake traffic secret* hts_C and the *server handshake traffic secret* hts_S defined as

$$hts_C := \mathbf{Expand}(hs, \ell_4 \parallel H_3) \quad \text{and} \quad hts_S := \mathbf{Expand}(hs, \ell_5 \parallel H_3)$$

with $H_3 = H(\texttt{CH} \parallel \texttt{CKS} \parallel \texttt{SH} \parallel \texttt{SKS})$. Based on the handshake traffic secrets hts_C and htk_S the server derives the *client handshake traffic key* htk_C and the *server handshake traffic key* htk_S as

$$htk_C := \mathsf{DeriveTK}(hts_C) \quad \text{and} \quad htk_S := \mathsf{DeriveTK}(hts_S).$$

Here, $\mathsf{DeriveTK}(s)$ outputs (k, iv) with $k := \mathbf{Expand}(s, \ell_{11}, l)$ and $iv := \mathbf{Expand}(s, \ell_{12}, d)$, where $l, d \in \mathbb{N}$ with l being the encryption key length and d being the IV length of AEAD[4], respectively, and $\ell_{11} = \texttt{"key"}$ and $\ell_{12} = \texttt{"iv"}$. In essence, it combines the traffic key derivation in the way that encryption key and initialization vector (IV) are now abstracted into a single key. The function $\mathsf{DeriveTK}$ is not described in the specification [Res18]. We introduce this function to tame the complexity.

Upon receiving (`SH`, `SKS`), the client performs the same computations to derive htk_C and htk_S (and all the intermediate values) except that it computes the DHE key as $Z := Y^x$.

All messages sent from here on are encrypted under the handshake traffic key using AEAD. For the direction 'server → client', we use the *server handshake traffic key* htk_S and for the opposite direction, we use the *client handshake traffic key* htk_C. This concludes the *key exchange phase* of the handshake. The next phase is the *server parameter phase*.

`EncryptedExtensions` (`EE`): This message contains all extensions that are not required to determine the cryptographic parameters. In previous versions, these extensions were sent as plaintext values. In TLS 1.3, these extensions are encrypted under the server handshake traffic key htk_S.

`CertificateRequest` (`CR`): The `CertificateRequest` message is a context-dependent message that may be sent by the server if it desires client authentication via a certificate. This concludes the negotiation of server parameters.

`ServerCertificate` (`SCRT`): This message consists of the actual server certificate used for authentication to the client initiating the *authentication phase* of the handshake. Since we do not consider any PKI, we view this message as some certificate[5] that contains some server identity S and a public key pk_S that is appropriate for the signature scheme.

[4] The key length is at most 256 bits and the IV length is always 96 bits (for references, see [McG08, Sect. 5], [NL18, Sect. 2.8], and [MB12, Sect. 6.1]). That is, for all $\lambda \in \{256, 384\}$ it holds $l, d \leq \lambda$.

[5] The certificate might be self-signed.

`ServerCertificateVerify` (`SCV`): To provide a "proof" that the server sending the `ServerCertificate` message really is in possession of the private key sk_S corresponding to the announced public key pk_S, it sends a signature

$$\sigma_S \xleftarrow{\$} \mathsf{Sig.Sign}(sk_S, \ell_{13} \parallel H_8)$$

over the hash H_8 of the messages sent and received so far, i.e.,

$$H_8 = H(\texttt{CH} \parallel \texttt{CKS} \parallel \texttt{SH} \parallel \texttt{SKS} \parallel \texttt{EE} \parallel \texttt{CR}^* \parallel \texttt{SCRT}).$$

Recall that every message marked with * is an optional or context-dependent message.

In the standard, the message signed is defined as $(\texttt{0x20})^{64} \parallel \texttt{Context} \parallel \texttt{0x00} \parallel \texttt{Content}$ (here: $\texttt{Context} = \ell_{13}$ and $\texttt{Context} = H_8$) to prevent an attack on previous versions of TLS. For further detail, we refer to the standard [Res18, Sect. 4.4.3]. We expect that this constant overhead does not change our analysis and thus drop it for simplicity.

`ServerFinished` (`SF`): This message contains the HMAC (Section 6.1.1) value over a hash of all handshake messages computed and received by the server. To that end, the server derives the *server finished key* fk_S from hts_S as $fk_S \coloneqq \mathbf{Expand}(hts_S, \ell_6)$. Then, it computes the MAC

$$fin_S \coloneqq \mathsf{HMAC}(fk_S, H_4)$$

with $H_4 = H(\texttt{CH} \parallel \texttt{CKS} \parallel \texttt{SH} \parallel \texttt{SKS} \parallel \texttt{EE} \parallel \texttt{CR}^* \parallel \texttt{SCRT} \parallel \texttt{SCV})$.

Now, the server can derive the *application traffic keys*. It first derives the *master secret* $ms \coloneqq \mathbf{Extract}(salt_{ms}, 0)$ with $salt_{ms} \coloneqq \mathbf{Expand}(hs, \ell_3 \parallel H_0)$ from the handshake secret hs derived earlier. Based on ms and the handshake transcript up to the `ServerFinished` message, the *client application traffic secret* ats_C and *server application traffic secret* ats_S, respectively, are defined as

$$ats_C \coloneqq \mathbf{Expand}(ms, \ell_7 \parallel H_5) \quad \text{and} \quad ats_S \coloneqq \mathbf{Expand}(ms, \ell_8 \parallel H_5)$$

where $H_5 = H(\texttt{CH} \parallel \texttt{CKS} \parallel \texttt{SH} \parallel \texttt{SKS} \parallel \texttt{EE} \parallel \texttt{CR}^* \parallel \texttt{SCRT} \parallel \texttt{SCV} \parallel \texttt{SF})$. Using ats_C and ats_S, the *client application traffic key* atk_C and *server application traffic key* atk_S is derived as

$$atk_C \coloneqq \mathsf{DeriveTK}(ats_C) \quad \text{and} \quad atk_S \coloneqq \mathsf{DeriveTK}(ats_S)$$

where DeriveTK is the same function used in the derivation of htk_C and htk_S.

After having derived atk_C and atk_S, the *exporter master secret* ems can also be derived from the master secret and the handshake transcript up to the `ServerFinished` message:

$$ems \coloneqq \mathbf{Expand}(ms, \ell_9 \parallel H_5)$$

where H_5 is defined above. Using atk_S the server can optionally already send application data over the record layer to the client.

After receiving and decrypting of (EE, CR*, SCRT, SCV, SF), the client first checks if the signature and the MAC contained in the SCV message and ServerFinished message, respectively, are valid. To that end, it retrieves the server's public key from the SCRT message, derives the server finished key fk_S based on hts_S, and recomputes the hashes H_8 and H_4 with the messages it has computed and received. The client aborts the protocol if either verification check fails. Provided the client does not abort, it derives the client and server application traffic key atk_C and atk_S just as described above. If application data was already sent, the client can now decrypt it using atk_S. Then, it prepares the following messages.

ClientCertificate (CCRT): This message is context dependent and is only sent by the client in response to a CertificateRequest message, i.e., if the server demands client authentication. The message is structured analogously to the ServerCertificate message except that it contains a client identity C and an appropriate public key pk_C.

ClientCertificateVerify (CCV): This message also is context dependent and only sent in conjunction with the ClientCertificate message. Similar to message ServerCertificateVerify, this message contains a signature σ_C computed over the hash H_9 of all messages computed and received by the client so far, i.e., $\sigma_C \xleftarrow{\$} \mathsf{Sig.Sign}(sk_C, \ell_{14} \parallel H_9)$ with

$$H_9 = H(\mathtt{CH} \parallel \mathtt{CKS} \parallel \mathtt{SH} \parallel \mathtt{SKS} \parallel \mathtt{EE} \parallel \mathtt{CR}^* \parallel \mathtt{SCRT} \parallel \mathtt{SCV} \parallel \mathtt{SF} \parallel \mathtt{CCRT}^*).$$

ClientFinished (CF): The last handshake message is the finished message of the client. As for the ServerFinished message this message contains a MAC over every message computed and received so far by the client. The client derives the *client finished key* fk_C from hts_C as $fk_C := \mathbf{Expand}(hts_C, \ell_6)$ and then, computes

$$fin_C := \mathsf{HMAC}(fk_C, H_6)$$

with $H_6 = H(\mathtt{CH} \parallel \mathtt{CKS} \parallel \mathtt{SH} \parallel \mathtt{SKS} \parallel \mathtt{EE} \parallel \mathtt{CR}^* \parallel \mathtt{SCRT} \parallel \mathtt{SCV} \parallel \mathtt{SF} \parallel \mathtt{CCRT}^* \parallel \mathtt{CCV}^*)$.

After receiving and decrypting of (CCRT*, CCV*, CF), the server first checks whether the signature and MAC contained in the CCV message and CF message, respectively, are valid. To that end, it retrieves the client's public key from the CCRT message (if present), derives the client finished key based on hts_C, and recomputes the hashes H_9 and H_6 with the messages it received. If one of the checks fails, the server aborts.

Finally, both parties derive *resumption master secret rms* from the master secret derived earlier and the handshake transcript up to the ClientFinished message:

$$rms := \mathbf{Expand}(ms, \ell_{10} \parallel H_7).$$

where $H_7 = H(\mathtt{CH} \parallel \mathtt{CKS} \parallel \mathtt{SH} \parallel \mathtt{SKS} \parallel \mathtt{EE} \parallel \mathtt{CR}^* \parallel \mathtt{SCRT} \parallel \mathtt{SCV} \parallel \mathtt{SF} \parallel \mathtt{CCRT}^* \parallel \mathtt{CCV}^* \parallel \mathtt{CF})$.

6.5 TLS 1.3 PSK-only/PSK-(EC)DHE Handshake

Similarly to Section 6.4, we describe the messages exchanged in the TLS 1.3 PSK mode of the handshake protocol in this section in detail. Since there are a lot of things shared between the full and the PSK handshake, we do not repeat shared aspects of the two protocol modes, but only discuss their differences. We describe the two variants of the PSK mode, PSK-only and PSK-(EC)DHE, in parallel and highlight how the two modes differ. An overview of the PSK mode of the TLS 1.3 handshake protocol is depicted in Figure 6.2.

The main difference between the TLS 1.3 full handshake and the PSK mode is that signature-based authentication is skipped in the PSK mode to allow for an abbreviated handshake. Authentication then is intuitively established via the PSK. Another aspect that distinguishes the PSK mode clearly from the full handshake is that the PSK mode allows for 0-RTT application data to be sent with the first flight of the client. Similarly to the long-term key pair for the full handshake, we assume that client and server know the PSK they use for the session before-hand. That is, we do not capture negotiation of the PSK in our view. Even though this does not affect the protocol description given in this section, we would like to highlight that we only consider PSKs that were established in a previous TLS 1.3 session, and do not cover PSKs negotiated out-of-band.

Deriving the PSK. The TLS 1.3 standard [Res18] defines PSKs derived from secrets exchanged in a previous TLS 1.3 session as follows. As this is a detail of the implementation of the server, we assume that there is a mapping defined between the pre-shared key identifier *pskid* and the actual pre-shared key *psk* out-of-band. As mentioned above, we only cover pre-shared keys established in previous TLS session and not pre-shared keys that were established externally. Therefore, we briefly describe how the establishment of these pre-shared keys would work. The last secret established in the TLS 1.3 handshake is the resumption master secret *rms*. Its purpose is to be used as the base key for the derivation of the pre-shared key established between a client and a server to allow for an subsequent abbreviated handshake execution (*session resumption*). The main idea here is that client and server that already successfully took part in a joint execution of the handshake, can skip the authentication part of the handshake in later handshakes. A prerequisite to derive a pre-shared key is that the server sends a *(session) ticket* after a successful execution of the TLS handshake. To this end, the server can send the post-handshake message `NewSessionTicket` [Res18, Sect. 4.6.1] at any time after the `ClientFinished` was received. This message contains a ticket that together with the resumption master secret *rms* uniquely defines a pre-shared key. A client can then use this ticket and present it in the `pre_shared_shared` extension (see below), which we denote as `ClientPreSharedKey` in this work. The content of the ticket depends on the implementation of the server and the respective pre-shared key *psk* is defined as follows:

$$psk := \mathbf{Expand}(rms, \texttt{"resumption"}, \texttt{ticket\_nonce})$$

where `ticket_nonce` is a nonce that is part of the `NewSessionTicket` message. In this work, we do not cover the exact derivation of the pre-shared key as described before. In particular, we do not cover the exchange of the `NewSessionTicket` message as this is encrypted under the application traffic secret (i.e., send over the record layer). This would make the protocol more complex as the server application traffic secret then would be used inside of the protocol and not only outside of it. That is, from the view of a MSKE protocol, it would be an internal rather then an external key.

Differences between the full and the PSK handshake. In the following, we discuss the differences between the handshake modes of TLS 1.3 in detail. To this end, we pick the messages that are different and discuss the differences carefully.

`ClientHello` and `ServerHello`. In the PSK-(EC)DHE, the `ClientHello` is very similarly structured to the full handshake presented in the previous section. The PSK handshake only adds two additional mandatory extensions:

- `psk_key_exchange_modes` [Res18, Sect. 4.2.9]: This extension does, similar to the `ciphersuites` extension, contain all PSK modes the client supports. In particular, this means whether it supports the key exchange that includes PSKs only or whether it (also) supports the PSK handshake with an additional (EC)DHE exchange to provide forward secrecy. Since we always consider the TLS handshake with a single configuration in isolation, we assume that this extension only contains either `psk_ke` for the PSK-only handshake or `psk_dhe_ke` for the PSK-(EC)DHE handshake.
- `pre_shared_key` [Res18, Sect. 4.2.11]: This extension contains all the ticket identities that the client wants to present to the server. For each of these ticket identities, the client in additional has to present a PSK binder value [Res18, Sect. 4.2.11.2.]. A binder value is computed exactly as a `Finished` message except that the used finished key fk_B uses the binder key bk as its base key and it is computer over the hash of the truncated `ClientHello` message, which is the `ClientHello` without the list of binder values. The respective binder key is derived from the pre-shared key that corresponds to the the ticket identity for which the binder value is computed. The possibility of presenting multiple pre-shared keys allows the client and server to negotiate a pre-shared key online. Since we do not consider negotiation of parameters in this work, we also do not cover this negotiation. To this end, we here only assume that the PSK used in a PSK handshake is known by the client and server in advance before the execution of the protocol starts and clients only include this pre-shared key in their `pre_shared_key` extension.

Servers also have to include the above mentioned extensions in their `ServerHello` message. Here, they need to announce their selected values from the values presented by the client. As we do not cover negotiation, we assume that servers just echo the values given by the client. Note that the `pre_shared_key` extension

for the server only contains the selected ticket identity, so servers do not send a binder.

Authentication. In the PSK mode of TLS 1.3, the parties are already assumed to be implicitly mutually authenticated via the PSK. This is because only these two parties should know the PSK and thus all the exchanged keys can only be derived by them. In the PSK handshakes, authentication via certificates and digital signatures therefore is omitted and yields an abbreviated handshake.

0-RTT Data. A feature unique to the PSK handshake is that clients can already send data with its first flight, i.e., along with the `ClientHello` message. This data is authenticated and encrypted using the PSK. However, security-wise this "channel" only provides weaker guarantees. First, the data is not forward secret as the key material used to protect the data is solely derived from the PSK, and second, since the data does not depend on the `ServerHello` message including a nonce there can be replays between connections. Possible replay attacks in this context are discussed in [Res18, App. E.5] and ways to counter act these replays are discussed in [Res18, Sect. 8].

The rest of the PSK handshake is analogous to the the full handshake except that some of the transcript hashes are computed over different messages different, because some message are present in the PSK handshake that are not in the full handshake, and vice versa.

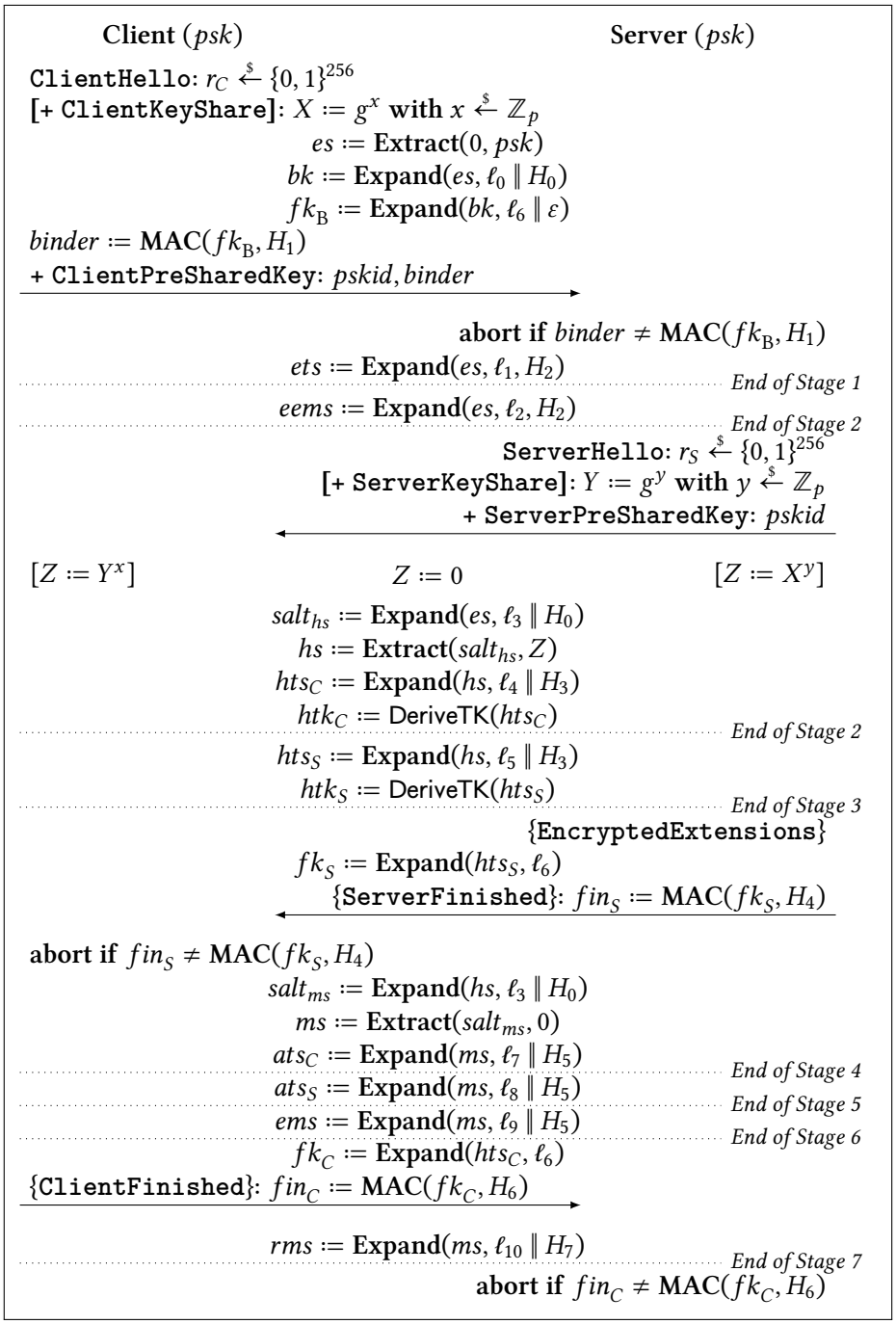

Figure 6.2: TLS 1.3 PSK handshake. Every TLS handshake message is denoted as "`MSG` : C", where C denotes the message's content. Similarly, an extension is denoted by "`+ MSG` : C". Further, we denote by "{`MSG`} : C" messages containing C and being AEAD-encrypted under the handshake traffic key htk. A value "[·]" is only present in the PSK-(EC)DHE handshake. Centered computations are executed by both client and server with their respective messages received, and possibly at different points in time.

7

ABSTRACTING THE TLS KEY SCHEDULE

Author's contribution. The contents presented in this chapter are based on joint work with Hannah Davis, Felix Günther, and Tibor Jager [DDGJ22b, DDGJ22a]. We discussed all aspects of this paper together, and in particular, jointly came up with the vision of a formally justified abstraction of the TLS 1.3 key schedule. This vision stemmed from previous independent and concurrent work by Tibor Jager and the author of this thesis [DJ21], and Hannah Davis and Felix Günther [DG21a], in which both analyzed the tight security of the TLS 1.3 full handshake protocol under rather strong assumptions on the key schedule. The goal of the joint work was to develop a new abstraction of the TLS 1.3 key schedule relying on weaker assumptions. Technically, Hannah Davis suggested the idea of using the indifferentiability framework to justify the newly developed abstraction and developed the architecture of the formal justification. As the results from [DDGJ22b, Sect. 4] focused on the TLS 1.3 PSK handshake, the treatment of the key schedule in this work also focused primarily on the key schedule as it is used in the PSK handshakes. The author of this thesis adapted the result presented in [DDGJ22b, Sect. 4] so that it also applies to the key schedule used in the full handshake. Since the key schedule that is used in the PSK handshake is more general than the one of the full handshake, the overall treatment presented in Sections 7.2 and 7.4 of this thesis in essence is the same and follows along the lines of the proofs from [DDGJ22b, Sect. 4]. Nevertheless, the formal proof requires a careful domain-separation argument for the different uses of the TLS hash function that is presented in [DDGJ22a, App. B] (the full version of [DDGJ22b]). This domain-separation argument is different for the full and the PSK handshake, since it uses the formatting of the handshake messages. Hence, the most notable contribution of the author of this thesis is the extension of the domain-separation argument for the PSK handshake to the full handshake presented in Section 7.5.2 of this work. In this context, smaller inaccuracies with respect to the message formatting in the argument of [DDGJ22b, Sect. 4] presented in Section 7.5.3 were fixed. Overall, the result of this chapter is a similar result as presented in [DDGJ22b, Sect. 4], but extended to all handshake modes.

Contents

7.1 Introduction

In this chapter, we show that the TLS 1.3 key schedule (i.e., the key derivation procedure of TLS 1.3) can be abstracted as a set of *independent* functions. This reduces the complexity of the security proofs for the TLS 1.3 handshake (Chapters 9 and 10) significantly as one can treat every key derived during the handshake as well as the binder (in the PSK mode) and finished MACs as independent functions of the pre-shared key (in the PSK mode), the Diffie–Hellman values (in the full/PSK (EC)DHE mode), protocol message and the randomness of the parties. To formally show this independence, we show over the course of this chapter that under the assumption that the hash function used in TLS is a random oracle, each of the abstract functions behaves like an independent random oracle using the indifferentiability framework by Maurer, Renner, and Holenstein [MRH04].

Motivation for a key schedule abstraction. In their tight(er) analysis of the TLS 1.3 full handshake Diemert and Jager [DJ21] modeled the TLS 1.3 key schedule as four independent random oracles and Davis and Günther [DG21a] concurrently modeled the two subroutines, HKDF.Extract and HKDF.Expand (Section 6.1.2), as two independent random oracles. Both works do not provide formal justification for these assumptions. These two abstractions both do not take into account that all subroutines of the key schedule (i.e., HMAC (Section 6.1.1), HKDF.Extract, and HKDF.Expand denoted by **MAC**, **Extract**, and **Expand**, respectively, in Chapter 6) ultimately rely on the hash function used in TLS 1.3, and that this hash function also is used, for example, to hash transcripts. Recall that HKDF.Extract and HKDF.Expand both basically are HMAC. HKDF.Extract internally forwards its inputs directly to HMAC, and HKDF.Expand also forwards its inputs to HMAC by appending only a 0x01 byte, as it is used in most cases to produce only a single block of length equal to the output length of the hash function.[1] This implies that

[1] The only exception is the derivation of the traffic keys (in DeriveTK introduced in Section 6.4), which consists of two HKDF.Expand calls with output length AEAD key length + AEAD IV length. However,

constructing HKDF.Extract and HKDF.Expand as *independent* random oracles cannot be indifferentiable as they are correlated by definition. Note that this does not necessarily mean that the analyses of Diemert and Jager [DJ21] and Davis and Günther [DG21a] are invalid, rather it might be possible that the two abstractions might not be formally justifiable under the assumption that *only* the TLS hash function is modeled as a random oracle. As Bellare, Davis, and Günther [BDG20] pointed out for the NIST PQC KEMs instantiating independent random oracles with a single hash function might lead to attacks. Davis, Diemert, Günther, and Jager [DDGJ22b] closed this gap and showed for the key schedule used in the TLS 1.3 PSK mode that the key schedule is indifferentiable from 12 independent random oracles. They formally justified this under the assumption that the hash function of TLS is modeled as a random oracle. In this work, we extend this result to the full handshake.

Chapter outline. To this end, we first introduce an abstraction of the key schedule in Section 7.2, give background on the indifferentiability framework in Section 7.3, and finally, prove the abstraction introduced in Section 7.2 in the random oracle model in Section 7.4.

7.2 Abstracted Key Schedule

In this section, we introduce our abstraction of the TLS 1.3 key schedule specified in [Res18, Sect. 7.1] and implicitly presented in Figures 6.1 and 6.2 in Chapter 6. Our abstraction consists of 11 functions, where each function corresponds to a key or a MAC value (binder and `Finished`) derived during the TLS 1.3 handshake (Figures 6.1 and 6.2). The 11 functions have the following signatures and their corresponding pseudocode is given in Figure 7.1.

1. $\mathsf{TKDF}_{binder} : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$
2. $\mathsf{TKDF}_{ets} : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$
3. $\mathsf{TKDF}_{eems} : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$
4. $\mathsf{TKDF}_{htk_C} : \{0,1\}^\lambda \times \mathbb{G} \times \{0,1\}^\lambda \to \{0,1\}^{l+d}$
5. $\mathsf{TKDF}_{fin_C} : \{0,1\}^\lambda \times \mathbb{G} \times \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$
6. $\mathsf{TKDF}_{htk_S} : \{0,1\}^\lambda \times \mathbb{G} \times \{0,1\}^\lambda \to \{0,1\}^{l+d}$
7. $\mathsf{TKDF}_{fin_S} : \{0,1\}^\lambda \times \mathbb{G} \times \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$
8. $\mathsf{TKDF}_{ats_C} : \{0,1\}^\lambda \times \mathbb{G} \times \{0,1\}^\lambda \to \{0,1\}^\lambda$
9. $\mathsf{TKDF}_{ats_S} : \{0,1\}^\lambda \times \mathbb{G} \times \{0,1\}^\lambda \to \{0,1\}^\lambda$
10. $\mathsf{TKDF}_{ems} : \{0,1\}^\lambda \times \mathbb{G} \times \{0,1\}^\lambda \to \{0,1\}^\lambda$

both of these quantities are for all possible AEAD schemes at most the output length of the hash function. That is, one can view these derivations of two derivations of output length of the hash function that are then trimmed down to the appropriate length.

$$11.\ \mathsf{TKDF}_{rms} \qquad : \{0,1\}^{\lambda} \times \mathbb{G} \times \{0,1\}^{\lambda} \to \{0,1\}^{\lambda}$$

Here, λ denotes the output length of the TLS hash function, and l and d denote the key length and IV length, respectively, of the AEAD scheme. Introducing these functions allows us to abstract away the intermediate computations. Note that these functions do not change the protocol as presented in Chapter 6. Every function TKDF_x captures the same steps that are required to derive the respective key/MAC value x. However, as shown in the definition of the functions in Figure 7.1, the 11 functions induce redundancy as every value is derived independently and therefore computes intermediate values such as *es* and *hs* multiple times over the execution of the handshake. Fortunately, this overhead is only conceptual. All of the intermediate value are derived deterministically, thus they could be cached in an actual implementation.

Note that the above abstraction is valid for all three modes of the TLS 1.3 handshake protocol. In the full handshake (Figure 6.1), we set the PSK to 0. Similarly, for the PSK-only mode (Figure 6.2), we set the DH secret Z to 0. This directly implements what the standard mandates: "If a given secret is not available, then the 0-value consisting of a string of Hash.length bytes set to zeros is used." [Res18, Sect. 7.1]. The TLS handshake using the abstracted key schedule then is defined as shown in Figure 7.2. Our main security proofs presented in Sections 9.3 and 10.3 will give a tight security bound for these abstracted handshakes. With the results of this chapter and Chapter 8, we then deduce a tight security bound for the ("unabstracted") TLS handshakes shown in Figures 6.1 and 6.2.

7.3 Indifferentiability

In Section 4.1.2, we discussed the random oracle model (ROM). The random oracle assumption on its own is quite strong, as it is well-known that random oracles can only exist as a theoretical concept rather than actually being instantiated in practice simply due to the size of its description. When applying the random oracle paradigm, it is often the case that the random oracle is replaced by something that is not a monolithic object, but a construction itself. Here you can think of a hash function constructed iteratively via the well-known Merkle–Damgård construction. In this case, it would be desirable if we could weaken our assumption that the hash function is ideal (i.e., a random oracle), to the assumption that the compression function used in the construction of the hash function is ideal. This would overall result in that we do not need to require a random oracle with an infinite domain anymore, but only one with a fixed input length (i.e., a finite domain). Even though, we still would be in the ROM, we weakened the assumptions.

At this point comes the indifferentiability framework originally proposed by Maurer, Renner, and Holenstein [MRH04] into play. This framework allows us to show that a construction that is based on some ideal function (e.g., a random oracle or ideal cipher) is indistinguishable (or *indifferentiable*) from an ideal function. Let us illustrate this idea using the example of the Merkle–Damgård hash function $H : \{0,1\}^{*} \to \{0,1\}^{\lambda}$ using compression function $h : \{0,1\}^{\kappa+\lambda} \to \{0,1\}^{\lambda}$ denoted as $H[h]$. Here, we are able to show with aid of the indifferentiability framework that under the assumption that h is a

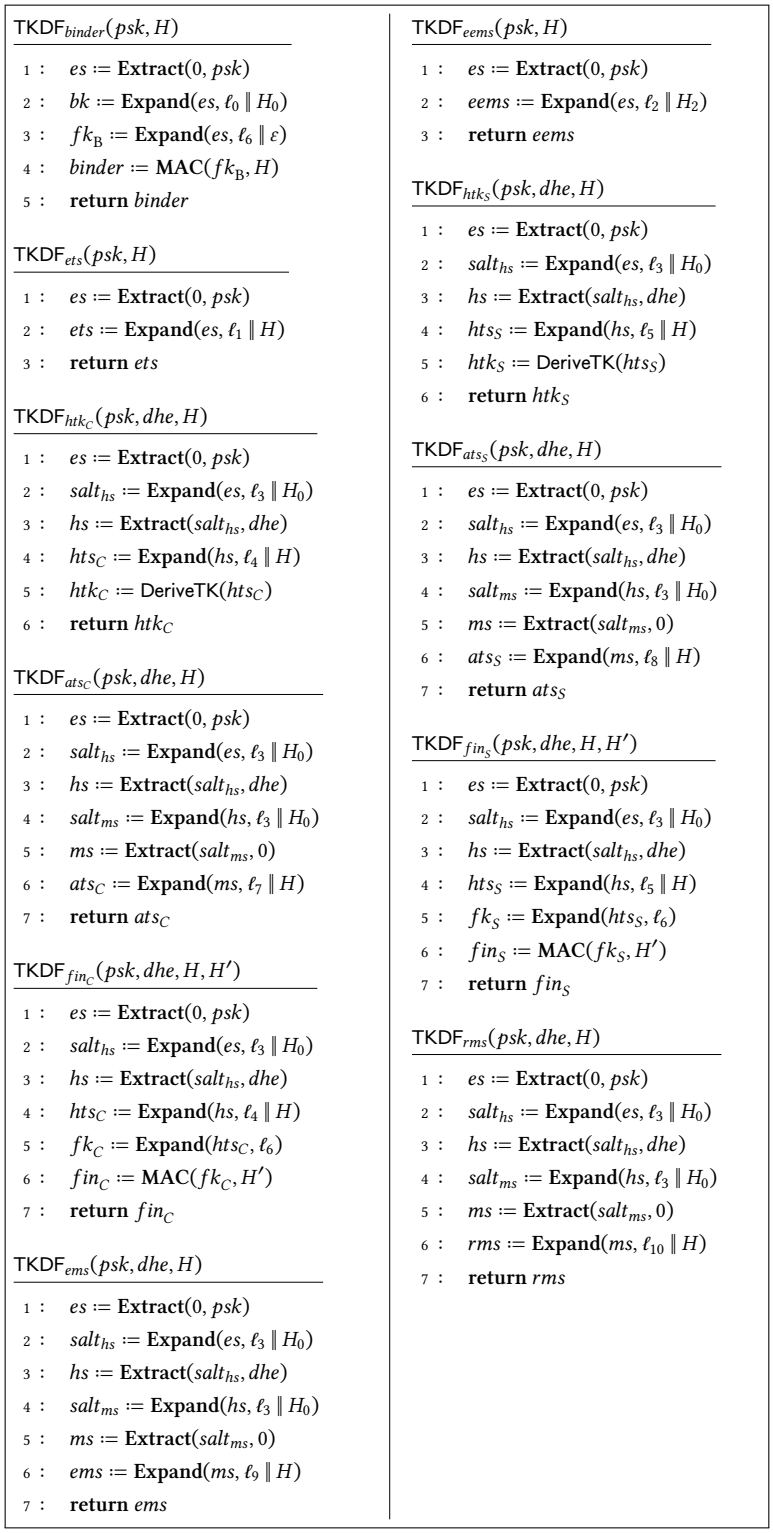

Figure 7.1: Code of the TKDF functions. The subroutines **Extract**, **Expand** and **MAC** are subroutines used in the TLS 1.3 handshake, and the labels and hashes used are defined in Table 6.1.

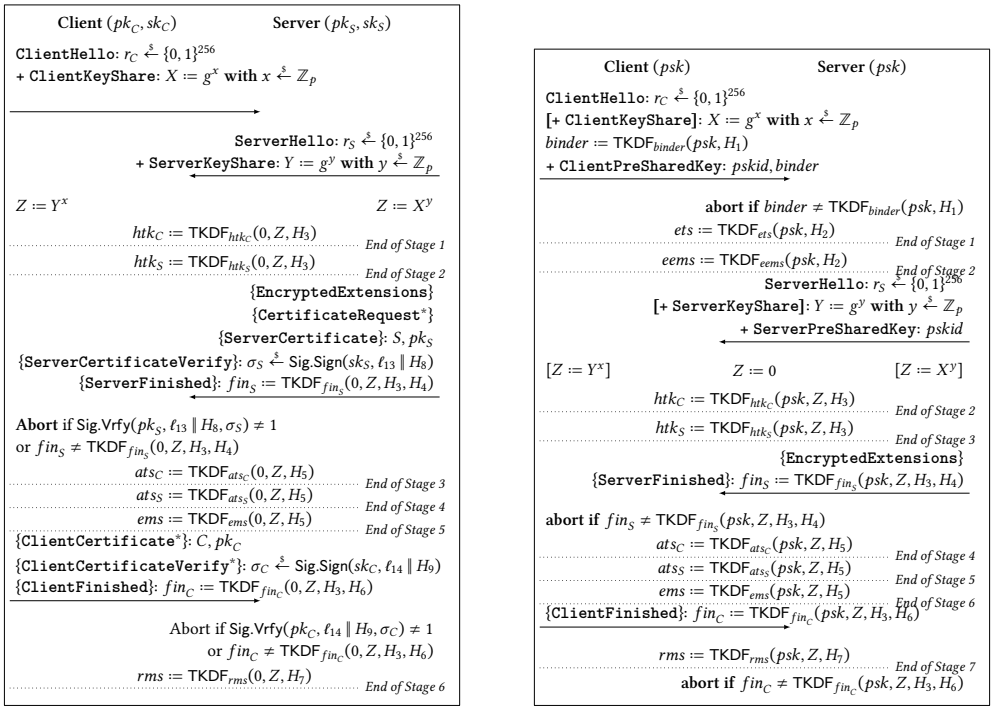

Figure 7.2: TLS 1.3 full (left-hand side) and PSK (right-hand side) handshake with abstracted key schedule.

random oracle (with fixed input length), the construction $H[h]$ is "indistinguishable" from a random oracle $\mathsf{RO} : \{0,1\}^* \to \{0,1\}^\lambda$ with infinite domain. That is, we can use RO when constructing the cryptographic scheme instead of inlining the iterative construction of the hash function H, but only need to assume the existence of an ideal h (resp. a random oracle with finite domain).

In this thesis, we make use the indifferentiability framework to justify our abstraction of the TLS 1.3 key schedule presented in Section 7.2. The abstraction allows us to tame the complexity of the proof and the indifferentiability frameworks provides the tools to back this up by a formal treatment. In contrast, we assumed in [DJ21] that parts of the TLS 1.3 key schedule behave like a random oracle without formal justification. Using the indifferentiability framework, we were able to prove in [DDGJ22b] that under the assumption that the TLS hash function H is a random oracle, each of the TKDF functions introduced Section 7.2 behaves like an independent random oracles.

Before we present the aforementioned result, we first give a brief introduction to the indifferentiability framework originally introduced by Maurer, Renner, and Holenstein [MRH04]. For further details on the framework, we refer to the works cited in this section and also, for example, the book by Mittelbach and Fischlin [MF21a]. In the following, we roughly follow the notation used in [BDG20, DDGJ22b].

Notation. We call a set of functions F such that all functions in this set have the same domain and range a *function space*. We view a *construction* $\mathbf{C}$ as a mapping $\mathsf{S} \to \mathsf{E}$, where S ("starting space") and E ("ending space") are function spaces. More precisely, a construction $\mathbf{C}$ is a deterministic algorithm that given oracle access to a function $s \in \mathsf{S}$ defines a function $\mathbf{C}[s] \in \mathsf{E}$. Here, $\mathbf{C}[\cdot]$ denotes that $\mathbf{C}$ "is built from" another function. To illustrate this, one can think of $\mathbf{C}[\cdot]$ being the Merkle–Damgård construction and S being a set of compression function such that $\mathbf{C}[s]$ is an instantiation of Merkle–Darmgård with a certain compression function. For simplicity, we refer to the functions in the function spaces as random oracles despite the indifferentiability framework being more general and could be used with other ideal objects (e.g., ideal ciphers), as well. We only aim to show that a construction based on a random oracle from a starting space S "can be replaced securely" by a random oracle from an ending space E and thus focusing on random oracles is sufficient for this thesis.

Definition. In general, the notion of *indifferentiability* generalizes the classical notion of *indistinguishability* by adding another oracle (called *public* in [MRH04]). Recall that for indistinguishability the adversary informally is challenged to distinguish the *real world* from the *random (or ideal) world*. Here, one can think for example of the definition of PRFs given above in Definition 4.2. In the real world, the adversary gets oracle access to the actual PRF (with a random key). While in the ideal world, it gets oracles access to a random function. For indifferentiability, we now aim to show that for some function space S and E and some random oracles $\mathsf{RO}_\mathsf{S} \in \mathsf{S}$ and $\mathsf{RO}_\mathsf{E} \in \mathsf{E}$, the construction $\mathbf{C}[\mathsf{RO}_\mathsf{S}]$ behaves indistiguishably from RO_E. Here, we also have a separation of a real and an ideal world. Namely, in the real world the adversary gets oracle access to $\mathbf{C}[\mathsf{RO}_\mathsf{S}]$ and in the ideal world it gets oracle access to RO_E. Now, we have that the construction $\mathbf{C}$ is built from random oracle RO_S thus in the indifferentiability definition the adversary also gets oracle access to RO_S. However, this is not the complete picture as the adversary now could simply distinguish the two worlds by just counting its oracles. This is overcome in the definition of indifferentiability by introducing a *simulator* Sim in the ideal world that aims to behave indistinguishably from RO_S, but maintains consistency with RO_E. That is, in conclusion for indifferentiability the adversary has to distinguish the real world, in which it gets oracle access to $\mathbf{C}[\mathsf{RO}_\mathsf{S}]$ and RO_S, from the ideal world, in which it gets oracle access to RO_E and $\mathsf{Sim}[\mathsf{RO}_\mathsf{E}]$. In the terminology of [MRH04], the oracles $\mathbf{C}[\mathsf{RO}_\mathsf{S}]$ and RO_E are the *private interface* of their respective worlds. Similarly, the oracles RO_S and Sim are the *public interface* of the respective worlds. Intuitively, the terms "private" and "public" are more relatable if one thinks of a different adversary than the one that tries to distinguish the two worlds. One should rather think of an adversary that tries to break the construction $\mathbf{C}$, or a cryptographic system that uses $\mathbf{C}$ as a building block. With this rational the terms private and public interface become clearer. The public interface reflects a classical random oracle that all parties have access to, i.e., the ideal primitive RO_S the construction $\mathbf{C}$ is built from. The private interface, which corresponds to the actual construction $\mathbf{C}$, in the context of the cryptographic system is not publicly available. It is rather the contrary, only the cryptographic system has access to building

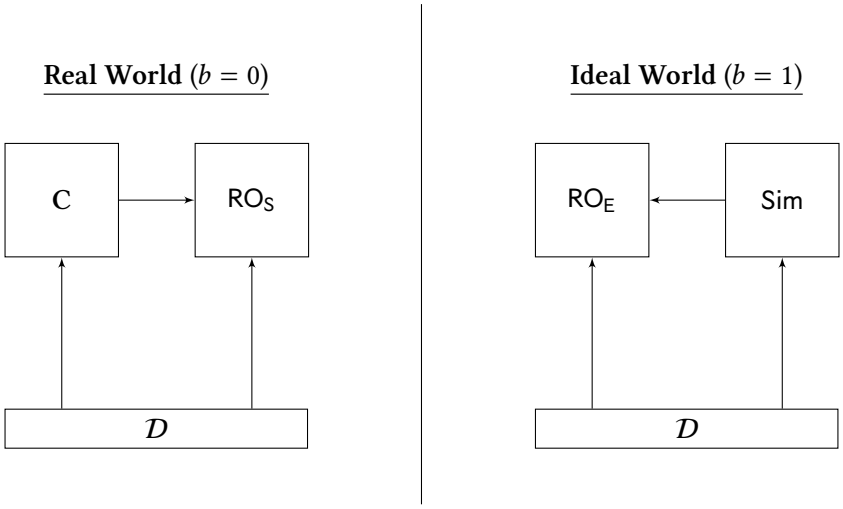

Figure 7.3: Interaction between the distiguisher and the oracles in the indifferentiability framework.

block interface of the construction and an adversary attacking a certain system usually only gets (if at all) access to outputs of the whole system rather of certain building blocks. Formally, we define indifferentiability as follows, which is adapted from [DDGJ22b]. The main interaction between the oracles and the distinguisher in both worlds is often (e.g., see [MRH04]) illustrated as shown in Figure 7.3.

Definition 7.1 (Indifferentiability). Let S and E be function spaces and let $\mathbf{C}$ be a construction for E from S. For any simulator Sim and any adversary $\mathcal{D}$ making q_{Priv} queries to oracle Priv and q_{Pub} to oracle Pub, the *advantage of $\mathcal{D}$ to indifferentiate* $\mathbf{C}$ is

$$\mathrm{Adv}^{\mathsf{indiff}}_{\mathbf{C},\mathsf{Sim},\mathsf{S},\mathsf{E}}(\mathcal{D}) := |\Pr[\mathrm{Exp}^{\mathsf{indiff},1}_{\mathbf{C},\mathsf{Sim},\mathsf{S},\mathsf{E}}(\mathcal{D}) = 1] - \Pr[\mathrm{Exp}^{\mathsf{indiff},0}_{\mathbf{C},\mathsf{Sim},\mathsf{S},\mathsf{E}}(\mathcal{D}) = 1]|$$

where $\mathrm{Exp}^{\mathsf{indiff},b}_{\mathbf{C},\mathsf{Sim},\mathsf{S},\mathsf{E}}(\mathcal{D})$ is defined in Figure 7.4.

Indifferentiability composition theorem. Informally, Definition 7.1 only captures the advantage of an distinguisher in detecting whether it interacts with the actual construction or a random oracle. Unfortunately, this is only one side of the coin. Usually, the construction will be part of a bigger cryptographic system. That is, it is only a building block of another system. Here, one can think of the TKDF functions shown in Section 7.2 as the construction and the TLS handshake as the cryptographic system the construction is used in. Now, what we actually want is given that our construction can be shown to be indifferentiable from a random oracle is to *replace* the construction in the cryptographic system by a random oracle. Recall that the overall goal of this section is to show that the TLS key schedule (i.e., the TKDF functions) behaves like many independent random oracles. With that goal in mind the actual power of the indifferentiability framework becomes clearer. Namely, the framework provides a *composition theorem* originally proven

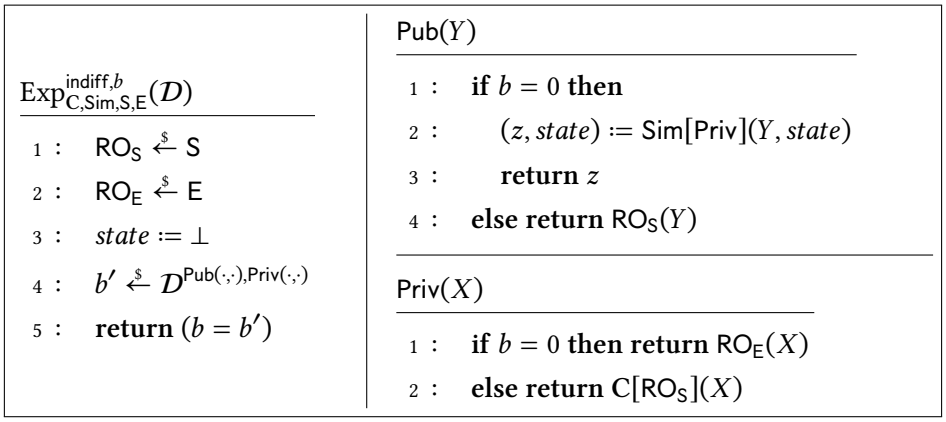

Figure 7.4: Security experiment for indifferentiability.

by Maurer, Renner, and Holenstein [MRH04] and later revisited by Ristenpart, Shacham, and Shrimpton [RSS11][2] that intuitively provides the following:

> *If a construction* $\mathbf{C}[\cdot]$ *is indifferentiable from a random oracle* RO*, then we can* securely *replace* any *application of* RO *in* any *cryptographic system by applications of* $\mathbf{C}[\cdot]$.

In other words, if we can show that $\mathbf{C}$ is indifferentiable from RO, we can prove security for a cryptographic system using RO (probably reducing the proof's complexity) and then inherently have provable security for the cryptographic system using $\mathbf{C}$.

In this thesis, we only need the composition theorem for MSKE protocols, and therefore only give the specific version of the indifferentiability composition theorem, even though the theorem holds in a more generalized form for any "single-stage"[3] security experiment according to [RSS11]. The next theorem is originally from [DDGJ22b].

Theorem 7.1. *Let* KE *be a MSKE protocol using function space* E*. Further, let* $\mathbf{C}$ *be a construction of space* E *from space* S *and let* Sim *be a simulator. The MSKE protocol* KE' *using function space* S *is defined as follows:* KE' *emulates* KE*, but whenever* KE *would compute its function from* E*,* KE' *instead computes* $\mathbf{C}$ *using a function from* S*. Then, for any adversary* $\mathcal{A}$*, we can construct an adversary* $\mathcal{B}$ *and a distinguisher* $\mathcal{D}$ *such that*

$$\mathrm{Adv}^{\Pi}_{\mathsf{KE}'}(\mathcal{A}) \leq \mathrm{Adv}^{\Pi}_{\mathsf{KE}}(\mathcal{B}) + \mathrm{Adv}^{\mathsf{indiff}}_{\mathsf{C},\mathsf{Sim},\mathsf{S},\mathsf{E}}(\mathcal{D})$$

where $\Pi \in \{\mathsf{pMSKE}, \mathsf{sMSKE}\}$ *depending on the MSKE (public or pre-shared key) variant of protocol* KE*.*

[2] Ristenpart, Shacham, and Shrimpton discovered that the theorem only holds for security experiments that they refer to as "single-stage", i.e., the adversary cannot be split up into different algorithms. Note that the multi-stage key exchange model, we are considering actually is a single-stage experiment in this terminology.

[3] Recall that the term "single-stage" security game in the terminology used by Ristenpart, Shacham, and Shrimpton [RSS11] refers to security games in which the adversary is only run once as opposed to in multiple stages as, for example, in a "find-then-guess" [BDJR97] kind of security experiment. This is in contrast to the stages (i.e., keys derived) that we have in the MSKE model.

In this work, we only sketch the rough idea of the proof of Theorem 7.1 for a detailed treatment of the indifferentiability composition, we refer to [RSS11, MRH04].

Proof Sketch. The adversary $\mathcal{B}$ and distinguisher $\mathcal{D}$ are defined as follows. Adversary $\mathcal{B}$ works exactly as adversary $\mathcal{A}$ except when $\mathcal{A}$ queries its random oracle from S, $\mathcal{B}$ responds to this query by running the simulator Sim instantiated with its random oracle from E. The distinguisher $\mathcal{D}$ simulates the experiment $\mathrm{Exp}^{\Pi}_{\mathsf{KE}}$ for adversary $\mathcal{A}$. Here, $\mathcal{D}$ instantiates $\mathcal{A}$'s random oracle with the Pub oracle from the indifferentiability game, and the random oracle from E that is used by KE is instantiated using the Priv oracle. Now, observe that if the distinguisher $\mathcal{D}$ runs in the real indifferentiability experiment (i.e., $b = 1$), then Pub computes construction C and Priv is just a random oracle from S. Thus, $\mathcal{D}$ perfectly simulates $\mathrm{Exp}^{\Pi}_{\mathsf{KE}'}$ for $\mathcal{A}$. If the distinguisher runs in the ideal indifferentibility experiment (i.e., $b = 0$), then Pub runs the simulator instantiated with Priv, which is a random oracle from E. Note that this is exactly what $\mathcal{B}$ would do. Thus, $\mathcal{D}$ perfectly simulates $\mathrm{Exp}^{\Pi}_{\mathsf{KE}}$ for $\mathcal{B}$. □

Read-only indifferentiability. Many cryptographic schemes in the random oracle model assume not only one, but multiple random oracles. At some point, during implementation at latest, these random oracles have to be constructed from a single random oracle (resp. a hash function). Bellare, Davis, and Günther [BDG20] refer to the process of constructing many random oracles out of a single one as *oracle cloning*. The easiest construction is the identity (cloning) functor [BDG20], in which random oracles $\mathsf{RO}_1, \mathsf{RO}_2, \ldots$ are constructed by just forwarding all queries of RO_i to a "base random oracle" RO. Unfortunately, this construction is only indifferentiable if the domains to the oracle RO_i are disjoint. In many cases, we face the problem that the oracles RO_i have overlapping domains, and even worse, all have domain $\{0,1\}^*$. Bellare, Davis, and Günther [BDG20] introduced a variant of the standard indifferentiability discussed above that is called *read-only indifferentiability*. In this variant, they introduce a *working domain* W for the functions and in the security experiment, the adversary is only allowed to query the working domain even though the function is defined on a broader space. This enables to prove that the above discussed identity functor is read-only indifferentiable [BDG20, Thm. 1] if the working domains of the oracles RO_i are (pairwise) disjoint, but the full domains might overlap.

Formally, the read-only indifferentiability is different from the definition above not only because of the working domain of the functions, but also because of the restriction that the simulator is not allowed to write its state. In the indifferentiability experiment shown in Figure 7.4, the simulator starts with an empty state and can read and write it with every execution. In contrast, the read-only indifferentiability splits the (read-only) simulator up into two algorithms: $\mathsf{Sim.Setup}$ generating the (static and experiment-maintained) state and $\mathsf{Sim.Eval}$ receives as input the state and oracle access to Priv, and defines a function $\mathsf{Sim.Eval}[\mathsf{Priv}](\cdot, \mathit{state})$. We define read-only indifferentiability following [BDG20] as follows.

Definition 7.2. Let S and E be function spaces, let W be the working domain of E (i.e., W is a subset of the domain of the functions contained in E) and let C be a construction for

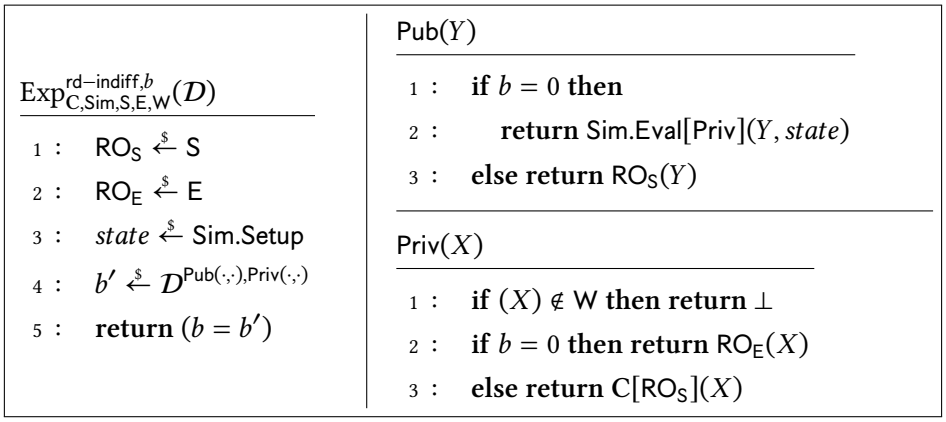

Figure 7.5: Security experiment for read-only indifferentiability.

E from S. For any (read-only) simulator $\mathsf{Sim} = (\mathsf{Sim.Setup}, \mathsf{Sim.Eval})$ and any adversary $\mathcal{D}$ making q_{Priv} queries to oracle Priv and q_{Pub} to oracle Pub, the advantage of $\mathcal{D}$ to (read-only) indifferentiate C is

$$\mathrm{Adv}^{\mathsf{rd-indiff}}_{\mathsf{C},\mathsf{Sim},\mathsf{S},\mathsf{E},\mathsf{W}}(\mathcal{D}) := |\Pr[\mathrm{Exp}^{\mathsf{rd-indiff},1}_{\mathsf{C},\mathsf{Sim},\mathsf{S},\mathsf{E},\mathsf{W}}(\mathcal{D}) = 1] - \Pr[\mathrm{Exp}^{\mathsf{rd-indiff},0}_{\mathsf{C},\mathsf{Sim},\mathsf{S},\mathsf{E},\mathsf{W}}(\mathcal{D}) = 1]|$$

where $\mathrm{Exp}^{\mathsf{rd-indiff},b}_{\mathsf{C},\mathsf{Sim},\mathsf{S},\mathsf{E},\mathsf{W}}(\mathcal{D})$ is defined in Figure 7.5.

Simiarly to the indifferentiability composition theorem, the read-only indifferentiability also provides a composition theorem. Bellare, Davis, and Günther [BDG20] prove the special case of that composition theorem for the IND-CCA-security of KEMs. Even though, read-only indifferentiability implies "classical" indifferentiability when considering full domains taking into account working domains requires a new formal treatment. Bellare, Davis, and Günther claim that their composition theorem not only holds for single-stage but also for multi-stage security games (opposed to the classical indifferentiability composition theorem). Since we only focus on MSKE in this work, we adapt the composition theorem of [BDG20] to the MSKE setting. The proof works analogously to the proof of Theorem 7.1 sketched above.

Theorem 7.2. *Let* KE *be a MSKE protocol using function space* E *with working domain* W. *Further, let* C *be a construction of space* E *from space* S *and let* Sim *be a (read-only) simulator. The MSKE protocol* KE′ *using function space* S *is defined as follows:* KE′ *emulates* KE*, but whenever* KE *would compute its function from* E*,* KE′ *instead computes* C *using a function from* S*. Then, for any adversary* $\mathcal{A}$*, we can construct an adversary* $\mathcal{B}$ *and a distinguisher* $\mathcal{D}$ *such that*

$$\mathrm{Adv}^{\Pi}_{\mathsf{KE}'}(\mathcal{A}) \leq \mathrm{Adv}^{\Pi}_{\mathsf{KE}}(\mathcal{B}) + \mathrm{Adv}^{\mathsf{rd-indiff}}_{\mathsf{C},\mathsf{Sim},\mathsf{S},\mathsf{E},\mathsf{W}}(\mathcal{D})$$

where $\Pi \in \{\mathsf{pMSKE}, \mathsf{sMSKE}\}$ *depending on the MSKE (public or pre-shared key) variant of protocol* KE*.*

7.4 Proving the TLS 1.3 Key Schedule Indifferentiable

In this section, we prove that the abstraction introduced in Section 7.2 is indifferentiable from the TLS 1.3 key schedule in the random oracle model. By the indifferentiability composition theorem, this implies that in the ROM the TLS 1.3 handshake shown in Figures 6.1 and 6.2 is "at least as secure" as the abstracted TLS 1.3 handshake shown in Figure 7.2. Note that this holds for each of the three handshake variants (full, PSK-(EC)DHE, and PSK-only) of the handshake, given that the hash function **H** used in TLS is modeled as a random oracle. Formally, we prove the following two theorems, where the latter is from [DDGJ22b].

Theorem 7.3. *Let* $\mathsf{RO}_H : \{0,1\}^* \to \{0,1\}^\lambda$ *be a random oracle, where* λ *is the output length of the hash function used in TLS. Let* KE *be the TLS 1.3 full handshake protocol described in Figure 6.1 with* $\mathbf{H} := \mathsf{RO}_H$ *and* **MAC**, **Extract**, *and* **Expand** *defined from* **H** *as in Section 6.3. Let* KE' *be the corresponding abstracted TLS 1.3 full handshake protocol described on the left-hand side of Figure 7.2, with* $\mathbf{H} := \mathsf{RO}_{\mathsf{Th}}$ *and* $\mathsf{TKDF}_x := \mathsf{RO}_x$*, where* $\mathsf{RO}_{\mathsf{Th}}$, RO_{htk_C}, …, RO_{rms} *are random oracles with the appropriate signatures. Then, for any adversary* $\mathcal{A}$ *against* KE*, we can construct an adversary* $\mathcal{B}$ *against* KE' *such that*

$$\mathrm{Adv}^{\mathsf{pMSKE}}_{\mathsf{KE}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{pMSKE}}_{\mathsf{KE}'}(\mathcal{B}) + \frac{2(13q_{\mathsf{Send}} + q_{\mathsf{RO}})^2}{2^\lambda} + \frac{2q_{\mathsf{RO}}^2}{2^\lambda} + \frac{8(q_{\mathsf{RO}} + 36q_{\mathsf{Send}})^2}{2^\lambda}$$

where q_{RO} *and* q_{Send} *are number of queries issued by* $\mathcal{A}$ *to the random oracle and oracle* q_{Send}*, respectively.*

Theorem 7.4. *Let* $\mathsf{RO}_H : \{0,1\}^* \to \{0,1\}^\lambda$ *be a random oracle, where* λ *is the output length of the hash function used in TLS. Let* KE *be the TLS 1.3 PSK-only or PSK-(EC)DHE handshake protocol described in Figure 6.2 with* $\mathbf{H} := \mathsf{RO}_H$ *and* **MAC**, **Extract**, *and* **Expand** *defined from* **H** *as in . Let* KE' *be the corresponding abstracted PSK-only or PSK-(EC)DHE handshake protocol described on the right-hand side of Figure 7.2, with* $\mathbf{H} := \mathsf{RO}_{\mathsf{Th}}$ *and* $\mathsf{TKDF}_x := \mathsf{RO}_x$*, where* $\mathsf{RO}_{\mathsf{Th}}$, RO_{binder}, …, RO_{rms} *are random oracles with the appropriate signatures. Then, for any adversary* $\mathcal{A}$ *against* KE*, we can construct an adversary* $\mathcal{B}$ *against* KE' *such that*

$$\mathrm{Adv}^{\mathsf{sMSKE}}_{\mathsf{KE}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{sMSKE}}_{\mathsf{KE}'}(\mathcal{B}) + \frac{2(12q_{\mathsf{Send}} + q_{\mathsf{RO}})^2}{2^\lambda} + \frac{2q_{\mathsf{RO}}^2}{2^\lambda} + \frac{8(q_{\mathsf{RO}} + 36q_{\mathsf{Send}})^2}{2^\lambda}$$

where q_{RO} *and* q_{Send} *are number of queries issued by* $\mathcal{A}$ *to the random oracle and oracle* q_{Send}*, respectively.*

To prove these two theorems, we proceed in three steps. Each step is an indifferentiability step abstracting the use of the TLS hash function incrementally until it finally abstracts the whole key schedule (resp. the TKDF functions defined in Section 7.2). Note that each step results in a new intermediate protocol. Initially, we start with the protocol as it is defined in Figure 6.1 for the full handshake and in Figure 6.2 for the PSK handshake. The hash function **H** is initially modeled as a random oracle RO_H.

Before we tackle the proofs formally, let us first outline the general idea of the three steps. As mentioned before the goal is to incrementally abstract the use of the hash

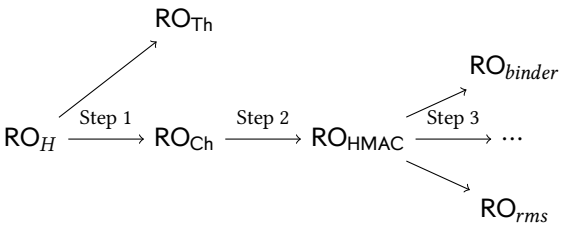

Figure 7.6: Abstraction of the TLS 1.3 key schedule from one to 12 random oracles.

function in the TLS 1.3 handshake such that in the end, we are able to abstract each key derivation and each MAC value computation during the handshake via an independent function (resp. random oracle) as presented in Section 7.2, and in particular, as shown in Figure 7.2. We highlight that the three steps are almost identical for the full handshake (Theorem 7.3) and the PSK handshake (Theorem 7.4). The overall outline remains the same, but the arguments differ only in detail. Consider Figure 7.6, for a visualization of how the random oracles change after each step.

Step 1: Separating the use of the hash function. First, observe that the hash function **H** in TLS is used for multiple purposes. It is used to compute transcript hashes as well as as a subroutine of **MAC** (i.e., HMAC), which is not only used to compute the finished MAC, but also as a subroutine of **Extract** and **Expand**. In the first step, we show that these two applications of the hash function are distinct. That is, we can define two functions Th ("transcript hash") and Ch ("component hash"), which represent each use of the hash function in TLS. Then, we can show using (read-only) indifferentiability (Section 7.3) that under the assumption that **H** is modeled as a random oracle RO_H, the functions Th and Ch can be modeled as two independent random oracles $\mathsf{RO}_{\mathsf{Th}}$ and $\mathsf{RO}_{\mathsf{Ch}}$.

In this step, we define the functions $\mathsf{Th}[\mathsf{RO}_H] \coloneqq \mathsf{RO}_H$ and $\mathsf{Ch}[\mathsf{RO}_H] \coloneqq \mathsf{RO}_H$. In the protocol, we replace calls of hash function **H** (resp. RO_H) to compute transcript hashes by a call of Th and instantiate **MAC** with Ch (inherently defining **Extract** and **Expand** over Ch). Note that the TLS handshake using Th and Ch is by definition (in either variant) identical to the corresponding protocol as defined in Figures 6.1 and 7.2 and this is merely a change in notation. The protocol using two independent random oracle $\mathsf{RO}_{\mathsf{Th}}$ and $\mathsf{RO}_{\mathsf{Ch}}$, however, induces the first intermediate protocol.

Step 2: Abstracting HMAC**.** With the hash function **H** being abstracted as two *independent* random oracles $\mathsf{RO}_{\mathsf{Th}}$ and $\mathsf{RO}_{\mathsf{Ch}}$, we are able to abstract function **MAC** (implemented as $\mathsf{HMAC}[\mathsf{RO}_{\mathsf{Ch}}]$) as its own random oracle. Note that we cannot do this abstraction right away, as **MAC** prior to Step 1 is implemented using **H** that is also used to hash transcripts during the handshake. Modeling **H** as the random oracle RO_H and modeling **MAC** as an independent random oracle might neglect possible dependencies between these two oracles. Therefore, Step 1 is crucial to have a formally-sound result, as it clearly separates the different applications of the hash function as independent. To this end, we replace the implementation of $\mathbf{MAC} = \mathsf{HMAC}[\mathsf{RO}_{\mathsf{Ch}}]$ in this step by **MAC** calling a single random

oracle $\mathsf{RO}_{\mathsf{HMAC}}$. Fortunately, we can rely on the result by Dodis, Ristenpart, Steinberger, and Tessaro [DRST12] to show that $\mathsf{HMAC}[\mathsf{RO}_{\mathsf{Ch}}]$ is indifferentiable from $\mathsf{RO}_{\mathsf{HMAC}}$. This result only applies if the HMAC key is of a fixed length less than the block length of the underlying hash function. TLS 1.3 supports SHA256 and SHA384, which have a block size of 512 and 1024 bits, respectively. As TLS 1.3 uses HMAC keys only of the same length as the output length of the hash function, i.e., 256 bits for SHA256 and 384 bits for SHA384, this requirement is satisfied. There is an exception that are pre-shared keys that are negotiated out-of-band, which might have a different length. In this work, we do not consider out-of-band pre-shared keys, therefore we do not cover them in this analysis.

The random oracle $\mathsf{RO}_{\mathsf{HMAC}}$ subsumes the random oracle $\mathsf{RO}_{\mathsf{Ch}}$ as Ch is only used as a subroutine of HMAC. The random oracle $\mathsf{RO}_{\mathsf{Th}}$ remains such that the new abstraction now consists of the random oracles $\mathsf{RO}_{\mathsf{Th}}$ and $\mathsf{RO}_{\mathsf{HMAC}}$.

Step 3: Final abstraction. In the final abstraction step, we move to the key schedule abstraction as depicted in Figure 7.2. After this indifferentiability step, we result in 12 independent random oracles, which consist of the oracle $\mathsf{RO}_{\mathsf{Th}}$ introduced in Step 1 and 11 random oracles that represent each key and MAC value computation during the handshake, called RO_{binder}, RO_{ets}, RO_{eems}, RO_{htk_C}, RO_{fin_C}, RO_{htk_S}, RO_{fin_S}, RO_{ats_C}, RO_{ats_S}, RO_{ems}, and RO_{rms}. For this, we show that the TKDF functions implemented using $\mathsf{RO}_{\mathsf{HMAC}}$ are indifferentiable from the aforementioned random oracles. Here, TKDF_x is a construction for RO_x.

Outline of this section. Applying these three steps combined to the key schedule used in the TLS 1.3 full and PSK handshake results in Theorems 7.3 and 7.4, respectively. This then forms the formal foundation to only work with the abstracted key schedule for subsequent analyses. In the remainder of this section, we formally prove the three steps outlined below. Recall that the general structure is the same for the full and PSK handshake, but the details, in particular for Step 1, differ. We will present a combined analysis in the next section and highlight the aspects that differ in the the various modes.

7.4.1 Step 1: Separating the Use of the Hash Function

In this section, we show that the two different ways TLS uses its hash function can be replaced by two independent random oracles. Recall that TLS uses its hash function for computing transcript hashes and as a subroutine of **MAC**, **Extract**, and **Expand** (Section 6.3). To separate these two applications of the hash function, we refer to the former as Th ("transcript hash") and to the latter as Ch ("component hash"). These two functions are just defined as $\mathsf{Th}[\mathbf{H}] := \mathbf{H}$ and $\mathsf{Ch}[\mathbf{H}] := \mathbf{H}$, so they are only conceptual. On a high level, we now want to show that these two uses of the hash function actually are *distinct*. That is, TLS never computes a transcript and component hash of the same input value. More formally, this means that we show that we can define domains $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$ for Th and Ch, respectively, such that $(\mathcal{D}_{\mathsf{Th}}, \mathcal{D}_{\mathsf{Ch}})$ forms a partition of $\mathbf{H}$'s domain $\{0,1\}^*$ (i.e., $\mathcal{D}_{\mathsf{Th}} \cap \mathcal{D}_{\mathsf{Ch}} = \emptyset$ and $\mathcal{D}_{\mathsf{Th}} \cup \mathcal{D}_{\mathsf{Ch}} = \{0,1\}^*$). With these domains defined, we can define

random oracles

$$\mathsf{RO}_{\mathsf{Th}} : \mathcal{D}_{\mathsf{Th}} \to \{0,1\}^{\lambda} \qquad \text{and} \qquad \mathsf{RO}_{\mathsf{Ch}} : \mathcal{D}_{\mathsf{Ch}} \to \{0,1\}^{\lambda}$$

and leverage the a construction that is called the *identity cloning functor* by Bellare, Davis, and Günther [BDG20]. This construction constructs multiple random oracles $(\mathsf{RO}_i)_i$ from a "base random oracle" RO and $\mathsf{RO}_i(x) = \mathsf{RO}(x)$ for all i, i.e., every query to oracle RO_i is forwarded to RO without change. This construction clearly is only indifferentiable if inputs x for different i are distinct (i.e., the domains of RO_i to not overlap). Otherwise, these oracles cannot be independent.

However, formally defining the domains $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$ is very complex, as they are only implicitly defined by the standard. Hence, it would be desirable to define the random oracles $\mathsf{RO}_{\mathsf{Th}}$ and $\mathsf{RO}_{\mathsf{Ch}}$ over the full domain $\{0,1\}^*$. Unfortunately, as mentioned above we are then not able to prove these indifferentiable. To circumvent this, we can make use of the working domains defined by Bellare, Davis, and Günther [BDG20] and prove read-only indifferentiability (Definition 7.2) instead. In this setting, random oracles can be defined over overlapping domains as long as they are only queried on the working domain during the security experiment. That is, $\mathsf{RO}_{\mathsf{Th}}$ and $\mathsf{RO}_{\mathsf{Ch}}$ can be defined on domain $\{0,1\}^*$, but have working domains $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$, respectively. The intuition behind this is that $\mathsf{RO}_{\mathsf{Th}}$ and $\mathsf{RO}_{\mathsf{Ch}}$ cannot be indiffierentiable in general, but only if queried on distinct domains.

Proof strategy. To prove this, we proceed as follows. First, we prove Lemma 7.1 given below stating that if $\mathbf{H}$ is modeled as a random oracle RO_H, then the way how $\mathbf{H}$ is used during the TLS handshake is read-only indifferentiable from two independent random oracles $\mathsf{RO}_{\mathsf{Th}}$ and $\mathsf{RO}_{\mathsf{Ch}}$ under the condition that the *working* domains of $\mathsf{RO}_{\mathsf{Th}}$ and $\mathsf{RO}_{\mathsf{Ch}}$ are disjoint. Using Lemma 7.1, we then can deduce from the (read-only) indifferentiability composition theorem Theorem 7.2 that the TLS 1.3 full handshake as well as the TLS 1.3 PSK handshakes preserves MSKE-security if it uses the abstraction of two random oracles representing the hash function. This is formalized in Lemmas 7.2 and 7.3. Note that the above mentioned results up to this point only are valid under the assumption that we can define (working) domains $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$. We already mentioned that defining these is complex. Therefore, we present a careful domain separation argument separately for the TLS 1.3 full handshake protocol and the TLS 1.3 PSK-(EC)DHE/PSK-only handshake protocol. Since this argument is so complex, we focus in this section on proving the indifferentiability, and the defer the discussion on domain separation as well as the definition of the working domains to Section 7.5.

Indifferentiability of the TLS hash function. Above we referred to the different applications for $\mathbf{H}$ as Th and Ch to highlight the overall idea. Formally, we need to express this as outlined above using the identity cloning functor by Bellare, Davis, and Günther [BDG20]. That is, the hash function construction $\mathbf{C}_1$ from $\mathbf{H}$ used during the TLS handshake can be formalized as follows: Let $\mathsf{S}_1 = \mathcal{F}(\{0,1\}^*, \{0,1\}^{\lambda})$ and $\mathsf{E}_1 = \mathcal{F}(\{\mathsf{Th}, \mathsf{Ch}\} \times$

$\mathsf{Sim}_1.\mathsf{Eval}[\mathsf{Priv}](Y, state)$

1 : **if** $Y \in \mathcal{D}_{\mathsf{Th}}$ **then** $i = \mathsf{Th}$ **else** $i = \mathsf{Ch}$
2 : **return** $\mathsf{Priv}(i, Y)$

Figure 7.7: Read-only simulator Sim_1 for Lemma 7.1. The simulator does not hold a state, i.e., $\mathsf{Sim}_1.\mathsf{Setup} = \bot$.

$\{0,1\}^*, \{0,1\}^\lambda)$. We define construction $\mathbf{C}_1$ for E_1 from S_1 as

$$\mathbf{C}_1[s](i, x) := s(x) \qquad \text{for } i \in \{\mathsf{Th}, \mathsf{Ch}\}.$$

For the above notation, we then have for the TLS handshake that $\mathsf{Th}(\cdot) = \mathbf{C}_1[\mathbf{H}](\mathsf{Th}, \cdot)$ and $\mathsf{Ch}(\cdot) = \mathbf{C}_1[\mathbf{H}](\mathsf{Ch}, \cdot)$. Next, we prove the following lemma showing that construction $\mathbf{C}_1$ is (read-only) indifferentiable from two independent random oracles with disjoint working domains. Implicitly this result was already given in [DDGJ22b] and adapts the the proof of [BDG20, Thm. 1].

Lemma 7.1. *Let* $\mathsf{S}_1 = \mathcal{F}(\{0,1\}^*, \{0,1\}^\lambda)$ *be the starting function space and* $\mathsf{E}_1 = \mathcal{F}(\{\mathsf{Th}, \mathsf{Ch}\} \times \{0,1\}^*, \{0,1\}^\lambda)$ *be the ending function space. Let* $\mathsf{W} := (\{\mathsf{Th}\} \times \mathcal{D}_{\mathsf{Th}}) \cup (\{\mathsf{Ch}\} \times \mathcal{D}_{\mathsf{Ch}})$ *be the working domain, where* $\mathcal{D}_{\mathsf{Th}}$ *and* $\mathcal{D}_{\mathsf{Ch}}$ *are disjoint sets with* $\mathcal{D}_{\mathsf{Th}} \cup \mathcal{D}_{\mathsf{Ch}} = \{0,1\}^*$. *Further, let construction* $\mathbf{C}_1$ *for* E_1 *from* S_1 *be the construction defined above. Define read-only simulator* Sim_1 *as shown in Figure 7.7. Then, for any adversary* $\mathcal{D}$*, it holds*

$$\mathrm{Adv}^{\mathsf{rd-indiff}}_{\mathbf{C}_1,\mathsf{Sim}_1,\mathsf{S}_1,\mathsf{E}_1,\mathsf{W}}(\mathcal{D}) = 0.$$

Proof. First, recall that

$$\mathrm{Adv}^{\mathsf{rd-indiff}}_{\mathbf{C}_1,\mathsf{Sim}_1,\mathsf{S}_1,\mathsf{E}_1,\mathsf{W}}(\mathcal{D}) = |\Pr[\mathrm{Exp}^{\mathsf{rd-indiff},1}_{\mathbf{C}_1,\mathsf{Sim}_1,\mathsf{S}_1,\mathsf{E}_1,\mathsf{W}}(\mathcal{D}) = 1] - \Pr[\mathrm{Exp}^{\mathsf{rd-indiff},0}_{\mathbf{C}_1,\mathsf{Sim}_1,\mathsf{S}_1,\mathsf{E}_1,\mathsf{W}}(\mathcal{D}) = 1]|$$

where $\mathrm{Exp}^{\mathsf{rd-indiff},b}_{\mathbf{C}_1,\mathsf{Sim},1\mathsf{S}_1,\mathsf{E}_1,\mathsf{W}}(\mathcal{D})$ is defined in Figure 7.5. We show that it holds

$$\Pr[\mathrm{Exp}^{\mathsf{rd-indiff},1}_{\mathbf{C}_1,\mathsf{Sim}_1,\mathsf{S}_1,\mathsf{E}_1,\mathsf{W}}(\mathcal{D}) = 1] = \Pr[\mathrm{Exp}^{\mathsf{rd-indiff},0}_{\mathbf{C}_1,\mathsf{Sim}_1,\mathsf{S}_1,\mathsf{E}_1,\mathsf{W}}(\mathcal{D}) = 1]$$

for Sim_1 given in Figure 7.7 and any adversary $\mathcal{D}$. To that end, we argue that the output of the oracles Priv and Pub in the read-only indifferentiability security experiment are distributed identically independent of bit b. As a reminder, if $b = 1$ then $\mathsf{Priv}(i, X) = \mathbf{C}_1[s](i, X) = s(X)$ for $(i, X) \in \mathsf{W}$ and $\mathsf{Priv}(i, X) = \bot$ otherwise, and $\mathsf{Pub}(Y) = s(Y)$ for $s \xleftarrow{\$} \mathsf{S}_\mathbf{I}$. If $b = 0$, then $\mathsf{Priv}(i, X) = e(i, X)$ for $(i, X) \in \mathsf{W}$ and $\mathsf{Priv}(i, X) = \bot$ otherwise, and $\mathsf{Pub}(Y) = \mathsf{Sim}_1.\mathsf{Eval}[e](Y)$ for $e \xleftarrow{\$} \mathsf{E}_\mathbf{I}$. Now, observe that both functions s (if $b = 1$) and e (if $b = 0$), are random functions with range $\{0,1\}^\lambda$. They only differ in their domain: function s has domain $\{0,1\}^*$ and function e has domain $\{\mathsf{Th}, \mathsf{Ch}\} \times \{0,1\}^*$. That is, function e intuitively only adds a parameter for the intended use of the resulting hash. Since functions s and e both are random functions with range $\{0,1\}^\lambda$, this implies that oracle Priv by definition, independent of bit b, implements a random function with

range $\{0,1\}^\lambda$, as well, if queried on the working domain. For Pub the same holds if $b = 1$. If $b = 0$, Pub is implemented using the simulator Sim_1. By definition, $\mathsf{Sim}_1.\mathsf{Eval}[e](Y)$ computes $e(i, Y)$ with $i = \mathsf{Th}$ if $Y \in \mathcal{D}_{\mathsf{Th}}$ and $i = \mathsf{Ch}$ otherwise. This means effectively, Priv and Pub for $b = 0$ are the same function, but with a different interface. Hence, Pub and Priv (for the working domain) independent of bit b implement a random function with range $\{0,1\}^\lambda$. This means for all inputs queried to Pub and Priv, respectively, for the first time, the output will be a uniformly distributed string on $\{0,1\}^\lambda$. Therefore, $\Pr[\mathrm{Exp}^{\mathsf{rd-indiff},1}_{\mathsf{C}_1,\mathsf{Sim}_!,\mathsf{S}_1,\mathsf{E}_1,\mathsf{W}}(\mathcal{D}) = 1] = \Pr[\mathrm{Exp}^{\mathsf{rd-indiff},0}_{\mathsf{C}_1,\mathsf{Sim}_1,\mathsf{S}_1,\mathsf{E}_1,\mathsf{W}}(\mathcal{D}) = 1]$ and the lemma follows. □

Remark 7.1. Note that the read-only simulator Sim_1 (Figure 7.7) used for Lemma 7.1 requires a membership function for set $\mathcal{D}_{\mathsf{Th}}$. The simulator thus asymptotically can only be efficient if such a membership function exist efficiently.

Lemma 7.1 on a high-level now gives us that if we are able to define sets $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$, then we can split-up a random oracle $\mathsf{RO}_H : \{0,1\}^* \to \{0,1\}^\lambda$ into two independent random oracles $\mathsf{RO}_{\mathsf{Ch}} : \{0,1\}^* \to \{0,1\}^\lambda$ and $\mathsf{RO}_{\mathsf{Ch}} : \{0,1\}^* \to \{0,1\}^\lambda$ with the restriction that $\mathsf{RO}_{\mathsf{Th}}$ and $\mathsf{RO}_{\mathsf{Ch}}$ are only queried on $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$, respectively. This intuition together with the (read-only) indifferentiability composition theorem Theorem 7.2 directly implies the following two lemmas formalizing that the TLS handshake protocol preserves MSKE-security if $\mathsf{RO}_{\mathsf{Th}}$ and $\mathsf{RO}_{\mathsf{Ch}}$ are used in place of RO_H. Lemma 7.3 was already given and shown in [DDGJ22b].

Lemma 7.2. *Let* KE *be the TLS 1.3 full handshake protocol as defined in Theorem 7.3. Let* $\mathsf{RO}_{\mathsf{Th}}, \mathsf{RO}_{\mathsf{Ch}} : \{0,1\}^* \to \{0,1\}^\lambda$ *be two independent random oracles. Let* KE_1 *be the TLS 1.3 full handshake protocol as defined in Figure 6.1, where*

- $\mathbf{H} := \mathsf{RO}_{\mathsf{Th}}$,
- $\mathbf{MAC} := \mathsf{HMAC}[\mathsf{RO}_{\mathsf{Ch}}]$,

and **Extract** *and* **Expand** *are defined using* **MAC** *as in* KE*. Further, let* $\mathcal{D}_{\mathsf{Th}}$ *and* $\mathcal{D}_{\mathsf{Ch}}$ *are disjoint sets with* $\mathcal{D}_{\mathsf{Th}} \cup \mathcal{D}_{\mathsf{Ch}} = \{0,1\}^*$*. Then, for any adversary* $\mathcal{A}$ *against the* pMSKE *security of* KE*, we can construct an adversary* $\mathcal{B}$ *against* KE_1 *such that*

$$\mathrm{Adv}^{\mathsf{pMSKE}}_{\mathsf{KE}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{pMSKE}}_{\mathsf{KE}_1}(\mathcal{B}).$$

Lemma 7.3. *Let* KE *be the TLS 1.3 PSK-only or PSK-(EC)DHE handshake protocol as defined in Theorem 7.4. Let* $\mathsf{RO}_{\mathsf{Th}}, \mathsf{RO}_{\mathsf{Ch}} : \{0,1\}^* \to \{0,1\}^\lambda$ *be two independent random oracles. Let* KE_1 *be the TLS 1.3 PSK-only or PSK-(EC)DHE handshake protocol as defined in Figure 6.2, where*

- $\mathbf{H} := \mathsf{RO}_{\mathsf{Th}}$,
- $\mathbf{MAC} := \mathsf{HMAC}[\mathsf{RO}_{\mathsf{Ch}}]$,

and **Extract** *and* **Expand** *are defined using* **MAC** *as in* KE*. Further, let* $\mathcal{D}_{\mathsf{Th}}$ *and* $\mathcal{D}_{\mathsf{Ch}}$ *are disjoint sets with* $\mathcal{D}_{\mathsf{Th}} \cup \mathcal{D}_{\mathsf{Ch}} = \{0,1\}^*$*. Then, for any adversary* $\mathcal{A}$ *against the* sMSKE *security of* KE*, we can construct an adversary* $\mathcal{B}$ *against* KE_1 *such that*

$$\mathrm{Adv}^{\mathsf{sMSKE}}_{\mathsf{KE}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{sMSKE}}_{\mathsf{KE}_1}(\mathcal{B}).$$

Remark 7.2. We highlight that Lemmas 7.2 and 7.3 only are valid under the assumption that the working domains $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$ can be defined. The definition is quite involved and requires a careful inspection of the format of the messages input to the hash function **H**. For better readability, we define these domains separately in Section 7.5 and focus on the actual proof of the key-schedule indifferentiability in this section. Here, it is sufficient to know that these domain can be defined.

7.4.2 Step 2: Separating the Use of the Hash Function

In the previous step (Section 7.4.1), we abstracted the hash function (resp. the random oracle RO_H) used in the TLS 1.3 handshake by two independent random oracles $\mathsf{RO}_{\mathsf{Th}}$ and $\mathsf{RO}_{\mathsf{Ch}}$, one for each use of the hash function. $\mathsf{RO}_{\mathsf{Th}}$ is used to hash transcripts, and $\mathsf{RO}_{\mathsf{Ch}}$ is used to instantiate **MAC**, **Extract** and **Expand**. In this section, we keep oracle $\mathsf{RO}_{\mathsf{Th}}$ as it is, and abstract oracle $\mathsf{RO}_{\mathsf{Ch}}$ even further. In our current abstraction of the TLS 1.3 handshake, which we referred to as KE_1 in Lemmas 7.2 and 7.3, **MAC** is defined as $\mathsf{HMAC}[\mathsf{RO}_{\mathsf{Ch}}]$, i.e., HMAC instantiated with $\mathsf{RO}_{\mathsf{Ch}}$ as hash function, and **Extract** and **Expand** are defined using **MAC**. Note that $\mathsf{RO}_{\mathsf{Ch}}$ in KE_1 is solely used to instantiate **MAC**. Therefore, the goal of this step is to replace $\mathsf{HMAC}[\mathsf{RO}_{\mathsf{Ch}}]$ by a random oracle $\mathsf{RO}_{\mathsf{HMAC}}$. To this end, we need to show that HMAC is indifferentiable. Fortunately, the indifferentiability of HMAC has already been studied by Dodis, Ristenpart, Steinberger, and Tessaro [DRST12]. Their result (given in [DRST12, Thm. 3] and [DRST13, Thm. 4.3]) shows indifferentiability for HMAC when the underlying hash function is modeled as a random oracle and the key length is less than the block size of the underlying hash function. However, this is not a big restriction in the context of TLS 1.3 as TLS 1.3 only allows for the hash function to be SHA256 and SHA384, which have a block size of 512 and 1024 bits, respectively. Therefore, the key size of HMAC used in TLS is always small enough as it is equal to the output length of the hash function, which is either 256 or 384 bits. The only exception might be if one would consider out-of-band PSKs that might deviate from this length requirement. We do not consider out-of-band PSKs in this work, so this is no restriction for our perspective.

Next, we show indifferentiability of HMAC as it is used in TLS, and then apply this result to the TLS full and PSK handshake, respectively. The following lemma was already implicitly given in [DDGJ22b] and the proof follows from [DRST13, Thm. 4.3].

Lemma 7.4. *Let* $\mathsf{S}_2 = \mathcal{F}(\{\mathsf{Th},\mathsf{Ch}\}\times\{0,1\}^*,\{0,1\}^\lambda)$ *be the starting function space and* $\mathsf{E}_2 = \mathcal{F}((\{\mathsf{Th}\}\times\{0,1\}^*)\cup(\{\mathsf{HMAC}\}\times\{0,1\}^\lambda\times\{0,1\}^*),\{0,1\}^\lambda)$ *be the ending function space. Further, let construction* $\mathbf{C}_2$ *for* E_2 *from* S_2 *be the construction defined as follows:*

$$\mathbf{C}_2[s](\mathsf{Th},x) = s(\mathsf{Th},x) \qquad \textit{and} \qquad \mathbf{C}_2[s](\mathsf{HMAC},k,x) = \mathsf{HMAC}[s](k,x).$$

Define simulator Sim_2 *as shown in Figure 7.8. Then, for any adversary* $\mathcal{D}$*, it holds*

$$\mathrm{Adv}^{\mathsf{indiff}}_{\mathbf{C}_2,\mathsf{Sim}_2,\mathsf{S}_2,\mathsf{E}_2}(\mathcal{D}) \leq \frac{2q^2}{2^\lambda}$$

where q *is the total number of oracle queries issued by* $\mathcal{D}$*.*

$\mathsf{Sim}_2[\mathsf{Priv}](i, U, state)$

1 : **if** $i = \mathsf{Th}$ **then return** $\mathsf{Priv}(\mathsf{Th}, U)$
2 : **else** // $i = \mathsf{Ch}$
3 : Parse U as $X \| Y$ with $|X|$ = block size
4 : $K \coloneqq \mathsf{GetKey}(X)$ // remove ipad or opad from X
5 : **if** $\mathsf{isOuter}(X) = 0$ **then** // "inner query"
6 : $V \xleftarrow{\$} \{0,1\}^\lambda$
7 : $state.F[K, V] \coloneqq Y$
8 : **return** V
9 : // "outer query" and "inner" made before
10 : **if** $\mathsf{isOuter}(X) = 1 \wedge state.F[K, Y] \neq \bot$ **then**
11 : **return** $\mathsf{Priv}(\mathsf{HMAC}, K, state.F[K, Y])$
12 : **return** $(R \xleftarrow{\$} \{0,1\}^\lambda)$

Figure 7.8: Simulator Sim_2 for Lemma 7.4. The construction is an augmented version of the simulator used in the proof of [DRST13, Thm. 4.3]. We augment the simulator to be able to process transcript hash queries.

Proof. Consider simulator Sim_2 defined in Figure 7.8. The general idea is simple. Given access to a Priv oracle in the indifferentiability experiment (Figure 7.4) that is either $\mathsf{RO}_{\mathsf{Th}}$ or $\mathsf{RO}_{\mathsf{HMAC}}$ the simulator has to simulate $\mathsf{RO}_{\mathsf{Th}}$ and $\mathsf{RO}_{\mathsf{Ch}}$. As we do not touch $\mathsf{RO}_{\mathsf{Th}}$ for this abstraction step and it is only considered for completeness, Sim_2 just forwards every Th query to $\mathsf{RO}_{\mathsf{Th}}$ as expected. To simulate $\mathsf{RO}_{\mathsf{Ch}}$ simulator Sim_2 is exactly the simulator defined by Dodis, Ristenpart, Steinberger, and Tessaro in the proof of [DRST13, Thm. 4.3]. Their simulator just distinguishes whether the queried string U resembles an "inner" or an "outer" oracle call in the computation of HMAC. As a reminder, recall the definition of the HMAC function (cf. Section 6.1.1):

$$H\left((k \oplus \mathsf{opad}) \| H((k \oplus \mathsf{ipad}) \| m)\right).$$

Here, we refer to $H\left((k \oplus \mathsf{opad}) \| Y\right)$ as an "outer" call and to $H((k \oplus \mathsf{ipad}) \| Y)$ as an "inner" call. To be able to distinguish these two kind of queries, the simulator assumes the existence of a predicate isOuter that on input $k \oplus \mathsf{opad}$ outputs 1 and on input $k \oplus \mathsf{ipad}$ outputs 0. Note that we consider a less general version of HMAC here compared to Dodis, Ristenpart, Steinberger, and Tessaro. We only consider a fixed key length of λ (i.e., the output length of the configured TLS hash function), which is for the two candidates SHA256 and SHA386 much smaller than their respective block size. In this case, the predicate isOuter can easily be implemented. HMAC pads the used key with 0 bits to the block size of the hash function. As discussed before, the key length is much smaller than the block size, and thus one can easily identify an "outer" or "inner" call as it either contains an $\mathsf{opad} = \mathtt{0x5c}$ segment for an "outer" call and an $\mathsf{ipad} = \mathtt{0x36}$ segment for an

"inner" call. With this the simulator, then is easily able to obtain the key for an input X by first calling $\mathsf{isOuter}$ and then computing $K = X \oplus \mathsf{opad}$ if $\mathsf{isOuter} = 1$ and $K = X \oplus \mathsf{ipad}$ otherwise. Now, if the simulator is queried for an inner query, it just chooses a random response and keeps this in a look-up table in its state. An outer response is responded by first looking-up whether the simulator was already queried for a suitable inner query. If this is the case, the simulator answers with the HMAC random oracle keyed with the computed key K and the corresponding inner query from its state. In any other case, the simulator just responds with a random string.

Now, note that transcript hash queries are identical in the real and ideal world as the oracle $\mathsf{RO}_{\mathsf{Th}}$ is present in both worlds. Therefore, our change to the simulator compared to Dodis, Ristenpart, Steinberger, and Tessaro does not affect their result. Hence, we get by applying [DRST13, Thm. 4.3] that

$$\mathrm{Adv}^{\mathsf{indiff}}_{\mathsf{C}_2,\mathsf{Sim}_2,\mathsf{S}_2,\mathsf{E}_2}(\mathcal{D}) \leq \frac{2q^2}{2^\lambda}$$

where q is the total number of queries (i.e., $q = q_{\mathsf{Priv}} + q_{\mathsf{Pub}}$) issued by $\mathcal{D}$. □

Next, we can use the indifferentiability composition theorem (Theorem 7.1) to apply Lemma 7.4 to the TLS handshake protocol.

Lemma 7.5. *Let* KE_1 *be the TLS 1.3 full handshake protocol as defined in Lemma 7.2. Let* $\mathsf{RO}_{\mathsf{Th}} : \{0,1\}^* \to \{0,1\}^\lambda$ *and* $\mathsf{RO}_{\mathsf{Th}} : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$ *be two independent random oracles. Let* KE_2 *be the TLS 1.3 full handshake protocol as defined in Figure 6.1, where*

- $\mathbf{H} := \mathsf{RO}_{\mathsf{Th}}$,
- $\mathbf{MAC} := \mathsf{RO}_{\mathsf{HMAC}}$,

and **Extract** *and* **Expand** *are defined using* **MAC** *as defined in Chapter 6. Then, for any adversary* $\mathcal{A}$ *against the* pMSKE *security of* KE_1*, we can construct an adversary* $\mathcal{B}$ *against* KE_2 *such that*

$$\mathrm{Adv}^{\mathsf{pMSKE}}_{\mathsf{KE}_1}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{pMSKE}}_{\mathsf{KE}_2}(\mathcal{B}) + \frac{2(13q_{\mathsf{Send}} + q_{\mathsf{RO}})^2}{2^\lambda}$$

where q_{Send} *and* q_{RO} *are the total amount of queries issued by* $\mathcal{A}$ *to the* Send *oracle and its random oracles, respectively.*

Proof. As mentioned above the lemma follows from the indifferentiability composition theorem (Theorem 7.1) and Lemma 7.4. The only thing that remains to be justified is that the distinguisher constructed in the reduction for the composition theorem makes a total amount of $13q_{\mathsf{Send}} + q_{\mathsf{RO}}$ queries to Priv and Pub. Note that in the proof of the composition theorem, the distinguisher $\mathcal{D}$ simulates the full TLS handshake for $\mathcal{A}$ using the oracles Priv and Pub instead of $\mathsf{RO}_{\mathsf{Th}}$ and $\mathsf{HMAC}[\mathsf{RO}_{\mathsf{Ch}}]$. Hence, we need to count what is the maximum number of HMAC and transcript hash computations for a message computation in the TLS full handshake. Considering the TLS full handshake (Figure 6.1) carefully one observes that the maximum number of computations occurs if a server receives the `ClientHello` message. Then, server computes 10 HMAC values (1 per

MAC, **Extract**, and **Expand**, respectively, and 2 per DeriveTK) and computes 3 transcript hashes resulting in a total amount of at most 13 Priv queries per run of a session. In addition, the distinguisher has to make 1 Pub query per query of $\mathcal{A}$ to either of the random oracles. Hence, overall we have a total amount of oracles queries issued by $\mathcal{D}$ of $13q_{\mathsf{Send}} + q_{\mathsf{RO}}$, where q_{Send} and q_{RO} are the total amount of queries issued by $\mathcal{A}$ to the Send oracle and its random oracles, respectively. □

Lemma 7.6. *Let* KE_1 *be the TLS 1.3 PSK-only or PSK-(EC)DHE handshake protocol as defined inLemma 7.3. Let* $\mathsf{RO}_{\mathsf{Th}}, \mathsf{RO}_{\mathsf{Ch}} : \{0,1\}^* \to \{0,1\}^\lambda$ *be two independent random oracles. Let* $\mathsf{RO}_{\mathsf{Th}} : \{0,1\}^* \to \{0,1\}^\lambda$ *and* $\mathsf{RO}_{\mathsf{Th}} : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$ *be two independent random oracles. Let* KE_2 *be the TLS 1.3 PSK-only or PSK-(EC)DHE handshake protocol as defined in Figure 6.2, where*

- $\mathbf{H} \coloneqq \mathsf{RO}_{\mathsf{Th}}$,
- $\mathbf{MAC} \coloneqq \mathsf{RO}_{\mathsf{HMAC}}$,

and **Extract** *and* **Expand** *are defined using* **MAC** *as defined in Chapter 6. Then, for any adversary* $\mathcal{A}$ *against the* sMSKE *security of* KE_1*, we can construct an adversary* $\mathcal{B}$ *against* KE_2 *such that*

$$\mathrm{Adv}^{\mathsf{sMSKE}}_{\mathsf{KE}_1}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{sMSKE}}_{\mathsf{KE}_2}(\mathcal{B}) + \frac{2(12q_{\mathsf{Send}} + q_{\mathsf{RO}})^2}{2^\lambda}$$

where q_{Send} *and* q_{RO} *are the total amount of queries issued by* $\mathcal{A}$ *to the* Send *oracle and its random oracles, respectively.*

Proof. Similar to the proof of Lemma 7.5, we only need to justify that the distinguisher constructed in the context of the composition issues at most $12q_{\mathsf{Send}} + q_{\mathsf{RO}}$ many queries. Considering the TLS PSK handshake (Figure 6.2) carefully one observes that upon receiving the `ClientHello` message (independent of the PSK mode), the server computes 10 HMAC values (1 per **MAC**, **Extract**, and **Expand**, respectively, and 2 per DeriveTK) and computes 2 transcript hashes resulting in a total amount of at most 12 Priv queries per run of a session. In addition, the distinguisher has to make 1 Pub query per query of $\mathcal{A}$ to either of the random oracles. Hence, overall we have a total amount of oracles queries issued by $\mathcal{D}$ of $12q_{\mathsf{Send}} + q_{\mathsf{RO}}$, where q_{Send} and q_{RO} are the total amount of queries issued by $\mathcal{A}$ to the Send oracle and its random oracles, respectively. □

7.4.3 Step 3: Introducing the Key Schedule Abstraction

After abstracting HMAC as its own random oracle, we are now on an abstraction level of the TLS 1.3 handshake protocol that uses two different random oracles, one for computing transcript hashes, which we denote by $\mathsf{RO}_{\mathsf{Th}}$, and one that we use in place of HMAC, which we denote by $\mathsf{RO}_{\mathsf{HMAC}}$. In the next and last step, we move to the abstraction of the TLS 1.3 handshake protocol as depicted in Figure 7.2 and replace the HMAC random oracle by 11 independent random oracles, where each random oracle represents each of the TKDF functions defined in Section 7.2. We start by showing that the abstraction of the TLS 1.3 with 11 independent random oracles is indifferentiable from the abstraction presented

in the previous step. Subsequently, we conclude with the indifferentiability composition theorem that the three handshake variants of TLS 1.3 maintain security with the new abstraction.

Lemma 7.7. *Let* $\mathsf{S}_3 = \mathcal{F}((\{\mathsf{Th}\}\times\{0,1\}^*)\cup(\{\mathsf{HMAC}\}\times\{0,1\}^\lambda\times\{0,1\}^*),\{0,1\}^\lambda)$ *be the starting function space and let*

$$\mathsf{E}_3 = \mathcal{F}(\{\mathsf{Th}\}\times\{0,1\}^*,\{0,1\}^\lambda)\times\mathcal{F}_{binder}\times\mathcal{F}_{ets}\times\cdots\times\mathcal{F}_{rms}$$

be the ending function space, where $\mathcal{F}_\ell$ *denotes the space of all functions with domain and range as* TKDF_ℓ *as defined in Section 7.2. Further, let construction* $\mathbf{C}_3$ *for* E_3 *from* S_3 *be the construction defined as follows:*

$$\mathbf{C}_3[s](\mathsf{Th},x) = s(\mathsf{Th},x) \qquad \textit{and} \qquad \mathbf{C}_3[s](\ell,k,x) = \mathsf{TKDF}_\ell[s](k,x)$$

with $\ell \in \{binder,\ldots,rms\}$. *Define simulator* Sim_3 *as shown in Figure 7.9. Then, for any adversary* $\mathcal{D}$*, it holds*

$$\mathrm{Adv}^{\mathsf{indiff}}_{\mathbf{C}_3,\mathsf{Sim}_3,\mathsf{S}_3,\mathsf{E}_3}(\mathcal{D}) \le \frac{2q_{\mathsf{Pub}}^2}{2^\lambda} + \frac{8(q_{\mathsf{Pub}}+6q_{\mathsf{Priv}})^2}{2^\lambda}$$

where q *is the total number of oracle queries issued by* $\mathcal{D}$.

Proof. In this proof, we bound the advantage of any distinguisher $\mathcal{D}$ against the simulator Sim_3. The construction of Sim_3 is rather complex, therefore we start the proof by providing an intuition on why the simulator is constructed as it is. After that we prove the bound given above in a sequence of games [Sho04].

Construction of simulator Sim_3**.** Consider the construction of simulator Sim_3 presented in Figure 7.9. The simulator Sim_3 takes as input $i \in \{\mathsf{Th},\mathsf{HMAC}\}$, a string $s \in \{0,1\}^*$, and a state. Further, it gets oracle access to a random oracle $\mathsf{RO} \in \mathsf{E}_3$ that represents 12 independent random oracles: $\mathsf{RO}_{\mathsf{Th}}$, RO_{binder}, …, RO_{rms}. Each random oracle RO_x can be accessed via RO by providing a prefix, i.e., $\mathsf{RO}_x(\cdot) := \mathsf{RO}(x,\cdot)$. Intuitively, the overall goal of the simulator is to simulate random oracle $\mathsf{RO}_{\mathsf{HMAC}}$ by using only the 11 random oracles RO_{binder}, …, RO_{rms} representing the TKDF functions. The random oracle $\mathsf{RO}_{\mathsf{Th}}$ is not abstracted in this step, therefore it remains unchanged. To this end, the simulator Sim_3 will always directly answer every query (Th,s) by simply forwarding it to the random oracle RO, and outputs $\mathsf{RO}(\mathsf{Th},s)$. HMAC oracle queries (i.e., inputs of the form $(\mathsf{HMAC},\cdot,\cdot)$) are more complex to respond to as we need to assure consistency with the other 11 "TKDF oracles". On input (HMAC,s), the simulator first parses every input s as a pair $(K,Y) \in \{0,1\}^\lambda\times\{0,1\}^*$ and then simulates $\mathsf{RO}_{\mathsf{HMAC}}(K,Y)$ as follows.

The simulator Sim_3 first starts by checking whether an HMAC query has already been issued before and answers repeated queries consistently using a look-up table M that it keeps in its state. If the input has not been queried before, it first samples its response y uniformly at random. Subsequently, the simulator ensures that is remains consistent with its random oracle RO. Recall that in the indifferentiablity game, we have two worlds.

$\mathsf{Sim}[\mathsf{RO}](i, s, state)$

```
1 :  M, T := state
2 :  // Th queries are just forwarded
3 :  if i = Th then return RO(Th, s)
4 :  // Otherwise, HMAC query
5 :  if M[s] ≠ ⊥ then return M[s]
6 :  parse s as (K, Y) ∈ {0,1}^λ × {0,1}*
7 :  // Sample random response
8 :  y ←$ {0,1}^λ
9 :  // computation for es
10 : if K = 0 then T_psk[y] := Y
11 : // computation for ms
12 : else if Y = 0 then T_saltMS[y] := K
13 : // MAC key K is already defined?
14 : else if T_BK/HTS[K] ≠ ⊥ then
15 :    es := T_es[T_BK/HTS[K]]
16 :    psk := T_psk[es]
17 :    // binder computation
18 :    if psk ≠ ⊥ then
19 :       y := RO(binder, psk, Y)
20 :    // fin_C/S computation
21 :    hts := T_BK/HTS[K]
22 :    (ℓ', hs, H) := T_hs/H[hts]
23 :    (salt_hs, Z) := T_saltHS/DHE[hs]
24 :    psk := T_psk[T_es/hs[salt_hs]]
25 :    if psk ≠ ⊥ then
26 :       (·, x) := ℓ'
27 :       y := RO(x, psk, Z, H, Y)
28 : // query is hs computation
29 : else T_saltHS/DHE[y] := (K, Y)
30 : // Is Y a HkdfLabel?
31 : if Y ≠ ⟨λ⟩_2 ‖ ⟨i⟩_1 ‖ "tls13 "
32 :       ‖ ℓ ‖ ··· ‖ 0x01 with i ∉ [8, 18]
33 :       ∧ |ℓ| ∉ [16, 96] then
34 : // No, output current response
35 :    M[s] := y; return y
36 : // Yes, parse Expand format
37 : parse ℓ from Y
38 : parse last λ + 1 bytes of Y
39 :       as H ‖ 0x01
```

$\mathsf{Sim}[\mathsf{RO}](i, s, state)$ // continued ...

```
40 : // query is bk computation
41 : if ℓ = ℓ_0 ∧ H = H_0 then
42 :    T_es[y] := K
43 : // query is salt_hs/ms computation
44 : else if ℓ = ℓ_3 ∧ H = H_0 then
45 :    T_es/hs[y] := K
46 : // query is hts_C/S computation
47 : else if ℓ ∈ {ℓ_4, ℓ_5} then
48 :    T_hs/H[y] := (L(ℓ), K, H)
49 : // query is ets/eems computation
50 : else if ℓ ∈ {ℓ_1, ℓ_2} ∧ T_psk[K] ≠ ⊥ then
51 :    if ℓ = ℓ_1 then x = ets
52 :    else x = eems
53 :    y := RO(x, T_psk[K], H)
54 : // query is ats_C/S/ems/rms computation
55 : else if ℓ ∈ {ℓ_7, ℓ_8, ℓ_9, ℓ_10} then
56 :    (salt_hs, Z) := T_saltHS/DHE[T_es/hs[
57 :                          T_saltMS[K]]]
58 :    psk := T_psk[T_es/hs[salt_hs]]
59 : // x is ats_C/S/ems/rms depending on ℓ
60 :    if psk ≠ ⊥ then
61 :       y := RO(x, psk, Z, H)
62 : // query is fk computation
63 : else if ℓ = ℓ_6 ∧ H = "" then
64 :    T_BK/HTS[y] := K
65 : // query is htk computation
66 : else if ℓ ∈ {"tls13 key",
67 :       "tls13 iv"} ∧ H = H_0 then
68 :    (ℓ', hs, H') := T_hs/H[K]
69 :    (salt_hs, Z) := T_saltHS/DHE[hs]
70 :    psk := T_psk[T_es/hs[salt_hs]]
71 :    if psk ≠ ⊥ then
72 :       (x, ·) := ℓ'
73 :       (y_0, y_1) := RO(x, psk, Z, H')
74 :       y := y_L(ℓ)
75 : M[s] := y; return y
```

Figure 7.9: Simulator Sim_3 for Lemma 7.7. The construction of the simulator is adapted from [DDGJ22b, DDGJ22a]. The label translator $\mathcal{L}$ outputs on input ℓ_4 the pair (htk_C, fin_C); on input ℓ_5 the pair (htk_S, fin_S); on input "tls13 key" 0; on input "tls13 iv" 1; and $\perp$ otherwise.

In the real world, an adversary has oracle access to the construction $\mathbf{C}_3$ and a random oracle $\mathsf{RO}_S \in \mathsf{S}_3$, which $\mathbf{C}_3$ is defined upon. In the ideal world, in turn it has oracle access to a random oracle $\mathsf{RO} \in \mathsf{E}_3$ and the simulator Sim_3, as without the simulator informally speaking an adversary can win by just counting its oracles. Now, a query to Sim_3 ultimately could be part of computing $\mathbf{C}_3$ using the Sim_3 and comparing it to RO in order to determine in which "world" an adversary is. Due to this possible attack, it is key for the simulator Sim_3 to be consistent with the random oracle RO. Otherwise, distinguishing the ideal from the real world might be trivial. Thus, simulator Sim_3 cannot always answer with a uniformly random y. Rather, it potentially needs to override the uniform y by the an output of its random oracle RO to keep consistent. For illustration, think about the underlying construction of TKDF, which informally is just a special way of calling HMAC. The output of TKDF ultimately is always from an HMAC call. Since Sim_3 basically simulates $\mathsf{RO}_{\mathsf{HMAC}}$, it is clear that all the "last" HMAC computations of each TKDF function (cf. Figure 7.1) has to consistent as the corresponding RO query (resp. TKDF computation). Since Sim_3 simulates a random oracle, one can view this "overwriting" as programming of the simulated random oracle. As mentioned before, only all the "last" computations of TKDF have to be consistent with the simulation. Therefore, only the keys derived during the handshake and the MAC values need to be programmed. All the intermediate values that we abstracted away using the introduction of TKDF (e.g., *es*, *hs*, etc.), the simulator stills need to keep track of using a look-up table T kept in the state for possible later programming. As sometimes multiple intermediate values are possible in certain situations, we have combined look-up tables in these cases. As an example take the computation of $salt_{hs}$ and $salt_{ms}$ (cf. Figure 7.1 for the precise definition). Here, the computation of these two intermediate values only differs in the key they are derived from, which is the early secret *es* for $salt_{hs}$ and the handshake secret *hs* for $salt_{ms}$. The context H_0 and the respective label are identical, therefore a key K in a query (HMAC, K, Y) could be both *es* or *hs*. Therefore, we cannot say with certainty whether the query belongs to a computation of $salt_{hs}$ and $salt_{ms}$, hence we store the respective key K under the response y to come back later to this query when more context is given. Note that each intermediate value should only ever appear in a single key derivation step except if there is a collision in $\mathsf{RO}_{\mathsf{HMAC}}$. Therefore, if no collision occurs we will find the right use (as *es* or *hs*) for the stored key later.

Overall, the simulator Sim_3 can be divided up into two phases. In the first phase, it checks whether the query corresponds to a "plain" HMAC computation (Figure 7.9, Lines 10–29). These occur in the TLS 1.3 key schedule whenever HMAC (for the MAC values) or **Extract** is computed. Recall that informally, **Extract** represents $\mathsf{HKDF.Extract}$, which is just another name for the HMAC function. These queries do not use labels, so potentially multiple values are possible. In the second phase, the simulator Sim_3 checks whether the input string Y follows the specific structure of the **Expand** calls used in TLS 1.3. We discuss this in more detail in the next section, when we have a closer look at the formatting of hash computation. However, the important thing to observe here is that **Expand** is also HMAC, but the input string Y has a specific structure to it. First of all, it always ends with $\mathtt{0x01}$ as all **Expand** queries used in TLS 1.3 have output length of at most the output length of the underlying hash function. That is, all **Expand** calls

only ever compute a single output block (cf. Section 6.1.2 for more details). The rest of Y is a special structure called `HkdfLabel` in the TLS 1.3 standard [Res18, Sect. 7.1], which is structured as follows:

$$\texttt{HkdfLabel} = \langle\lambda\rangle_2 \,\|\, \langle i\rangle_1 \,\|\, \texttt{"tls13 "} \,\|\, \ell \,\|\, \langle\lambda\rangle_1 \,\|\, H$$

where λ is the output length of the hash function configured in TLS, $\ell \,\|\, H$ is the input to **Expand** consisting of label and a hash, $i \in [8, 18]$, and $\langle\cdot\rangle_j$ denotes the encoding of $\cdot$ in i bytes. This can then be used by the simulator to precisely determine which key derivation the query might belong to. In summary, the simulator uses a look-up table to backtrack through potential computations of intermediate values using queries to ultimately find the inputs ensuring consistency with RO. Using this backtracking allows the simulator to find the inputs to TKDF to derive a certain key.

Bounding the advantage. Let $\mathcal{D}$ be any distinguisher. In the following, we construct a sequence of games ([Sho04]) to bound the advantage $\mathrm{Adv}^{\mathsf{indiff}}_{\mathsf{C}_3,\mathsf{Sim}_3,\mathsf{S}_3,\mathsf{E}_3}(\mathcal{D})$. Let Game_δ denote the event that Game δ outputs 1. The sequence of games presented below was already given in a similar form in [DDGJ22b, DDGJ22a].

GAME 0. The initial Game 0 is the "ideal" indifferentiability game $\mathrm{Exp}^{\mathsf{indiff},0}_{\mathsf{C}_3,\mathsf{Sim}_3,\mathsf{S}_3,\mathsf{E}_3}(\mathcal{D})$ defined in Figure 7.4. This means the distinguisher $\mathcal{D}$ has oracle access to a Priv oracle using a random oracle $\mathsf{RO} \xleftarrow{\$} \mathsf{E}_3$ and a Pub oracle using $\mathsf{Sim}_3[\mathsf{RO}]$. In this game, it holds

$$\Pr[\mathsf{Game}_0] = \Pr[\mathrm{Exp}^{\mathsf{indiff},0}_{\mathsf{C}_3,\mathsf{Sim}_3,\mathsf{S}_3,\mathsf{E}_3}(\mathcal{D}) = 1].$$

GAME 1. In Game 1, we ensure that the response y sampled by Sim_3 (Line 8, Figure 7.9) does not collide with neither the input nor the output of any previous query to Sim_3. In particular, we require for inputs $(\mathsf{HMAC}, s, \cdot)$ with s parsed as $(K, Y) \in \{0,1\}^\lambda \times \{0,1\}^*$ that y does not collide with neither K nor Y. If this event occurs, we set a flag bad_C and if bad_C is set in the end of the game, we always output 0. If distinguisher $\mathcal{D}$ issues at most q_{Pub} many queries to oracle Pub (resp. Sim_3), there are in any query at most $2q_{\mathsf{Pub}}$ values that string $y \xleftarrow{\$} \{0,1\}^\lambda$ can collide with. Applying the union bound, the probability that such a collision occurs and flag bad_C is set, is bounded from above by $2q^2_{\mathsf{Pub}}/2^\lambda$. Then, it holds (by the Difference Lemma [Sho04] or the Fundamental Lemma of game playing [BR06]) that

$$\Pr[\mathsf{Game}_1] \leq \Pr[\mathsf{Game}_0] + \Pr[\mathsf{bad}_C] \leq \Pr[\mathsf{Game}_0] + \frac{2q^2_{\mathsf{Pub}}}{2^\lambda}.$$

GAME 2. In Game 2, we recompute every query issued to oracle Priv using TKDF[Pub] before the game computes its final output. That is, we store every query (r, X) issued to oracle Priv and in the end of the game we compute $\mathsf{TKDF}_r[\mathsf{Pub}](X)$ for all (r, X) stored before. The result of each computation is discarded. Since we ignore the outputs, this change only has an effect on the flag bad_C being set, because any of the new computations might ask a new query to oracle Pub that induces a collision. Recalling the definition of TKDF (Figure 7.1), we observe that each of the functions call the subroutines **Extract**,

Expand, and **MAC** at most 6 times in total. Since all of these functions correspond to a single HMAC computation, this change induces at most $6 \cdot q_{\mathsf{Priv}}$ additional Pub queries (i.e., at most 6 additional Pub per Priv query). Thus, this change only increases the probability that flag bad_C is set, which can be upper-bounded as follows:

$$\Pr[\mathsf{Game}_2] \leq \Pr[\mathsf{Game}_1] + \frac{2(q_{\mathsf{Pub}} + 6q_{\mathsf{Priv}})^2}{2^\lambda}.$$

GAME 3. In Game 3, we move the TKDF computations introduced in Game 2 directly to the Priv oracle. That is, on query (r, X) oracle Priv first computes $\mathsf{TKDF}_r[\mathsf{Pub}](X)$, discards the output and then outputs $\mathsf{RO}(r, X)$. Note that this does not change the view of the adversary as $\mathsf{Priv}(r, X) = \mathsf{RO}(r, X)$ still holds. To be precise, if $r = \mathsf{Th}$ then TKDF is not computed and the input is directly forwarded to RO. However, this change induces that now not only oracle Pub uses the simulator Sim_3, but also Priv implicitly uses it because TKDF has oracle access to Pub. This might change the state of the simulator and even though the result is discarded the corresponding Pub queries might cause the flag bad_C to be set. Let us briefly elaborate on the consequences. Recall that RO is a "classical" random oracle, i.e., a random function. Every output to new query is a string sampled uniformly at random, and repeated queries will be answered consistently. The simulator Sim_3 on the other hand also "simulates" a random oracle ($\mathsf{RO}_{\mathsf{HMAC}}$), but checks the structure of queries and backtracks to potential previous computations to keep consistency with the random oracle RO. By that we add computations of $\mathsf{TKDF}_r[\mathsf{Pub}](X)$ for any query to oracle Priv, every Pub query that would be necessary to ultimately compute the output of TKDF has already been made. This of course changes the order of Pub queries, as the adversary not necessarily queries all intermediate queries one after the other and also does not necessarily query all of them. This has two major consequences: (1) the flag bad_C might be set in Game 3 when it was not set in Game 2, and (2) a Priv query now has an effect on the look-up table T used to backtrack through computations to decide whether all necessary queries have been made and consistency with RO has to be assured. Thus, in Game 3 the simulator Sim_3 might "program" responses not programmed in Game 2.

Next, let us analyze the difference of Game 3 and Game 2. As already mentioned above the changes introduced in this game do not change the adversary's view when it comes to the Priv oracle. For the Pub oracle, we claim that the responses are distributed identically except when either Game 3 or Game 2 sets bad_C. Formally, let break denote the event that bad_C is set in either Game 3 or Game 2. First, we prove that the above claim holds and then conclude that it holds that

$$\Pr[\mathsf{Game}_3] \leq \Pr[\mathsf{Game}_2] + \Pr[\mathsf{break}].$$

We prove the above claim that if break does not occur then oracle Pub answers identically (for the same randomness) in Game 3 and Game 2 by contradiction. To this end, assume that event break does not occur. Let Pub and Priv behave identically in both Game 3 and Game 2. In particular, Game 3 and Game 2 use the *same* randomness. Further, assume that there exists a query Q issued by the adversary to oracle Pub (for the first time) that is answered differently in Game 3 than in Game 2. As Game 3 and Game 2 use the same

randomness, this implies that the simulator Sim_3 had to program the response in at least one game. The simulator Sim_3 only programs a response if the backtracking using the look-up table T results in a consistent history. Thus, let us consider the state of the look-up tables in Games 2 and 3 when query Q is issued. Here, let T_2 denote the look-up table in Game 2 and let T_3 denote the look-up table in Game 3. Recall that the index of the look-up table always is the uniformly sampled response y. That is, entries can only be overwritten in a look-up table if a collision occurs and event break occurs. Now, recall that Q is the first query that results in a different answer of the oracle Pub. This implies that T_3 has to contain at least all the entries of T_2. Since T decides if (and how) a response in programmed in Sim_3, we have that every query prior to Q and Q itself that is programmed in Game 2, has to be programmed to the same response in Game 3. Equivalently, any response in Game 3 that has not been programmed (i.e., answered uniformly at random), is also not programmed in Game 2.

Since the response to query Q now is different by assumption, it cannot be the case that it was programmed in Game 2 or it was answered randomly in Game 3. Therefore, query Q must have been answered randomly in Game 2, and programmed in Game 3. Now, recall that in Game 3 TKDF is computed for every Priv query using Pub. This computation induces a (consecutive) sequence of queries to Pub corresponding to a full execution of the function. These queries have to store entries in T_3 corresponding to the intermediate values computed. Since Q is programmed in Game 3, it has to be a "final" query of a TKDF computation. Let $Q_1, \ldots, Q_i$ denote the sequence of queries corresponding to entries stored in T_3 used to program Q. In each query, the simulator either stores an entry in table T or programs a response y. Therefore, there has to exist a r such that Q_r that was answered by a random response. As each step of the TKDF function computation depends on the previous, we have that the response of query Q_j in the sequence has to be contained in some way in the input of query Q_{j+1}. Since Q_i is the second last query for the computation of TKDF, Q can be seen as Q_{i+1}.

In Game 2, it has to be the case that there is an entry missing in table T_2 as otherwise, the value would have been programmed. That is, there exists at least one query that has not been issued before query Q in Game 2. Instead, this would be one of the queries made at the end of Game 2, which in Game 3 is moved directly in the Priv oracle. Thus, this is exactly the spot where the order of Pub query was changed. Let us denote this query by Q_j. Observe that all other queries related to that TKDF were already been made before. Now, if Q_j is sampled at random it has to collide with (some part of) the input to query Q_{j+1} because otherwise $Q_1, \ldots, Q_i$ would not be a valid sequence of queries computation TKDF. However, this would cause bad_C to be set and ultimately causes event break to occur, which is a contradiction to our assumption that break does not occur. Therefore, we can conclude that Priv is identically distributed in both games and Pub is identical only if break does not occur. Overall, this implies that Game 3 and Game 2 are identical if break does not occur.

It remains to analyze the probability that break occurs. The number of queries Pub queries, $q_{\mathsf{Pub}} + 6q_{\mathsf{Priv}}$ is identical in Game 3 and Game 2, as they only ask their queries in a different order. Each sets the flag bad_C with probability $\frac{2(q_{\mathsf{Pub}}+6q_{\mathsf{Priv}})^2}{2^\lambda}$, and thus it holds

by the union bound

$$\Pr[\mathsf{Game}_3] \leq \Pr[\mathsf{Game}_2] + \Pr[\mathsf{break}] \leq \Pr[\mathsf{Game}_2] + \frac{4(q_{\mathsf{Pub}} + 6q_{\mathsf{Priv}})^2}{2^\lambda}.$$

GAME 4. In Game 4, we change the implementation of the Priv oracle. Instead of computing TKDF and discarding the output, in this game we output the result instead of the output of RO. Note that for completeness, we replace $\mathsf{Priv}(r, X)$ by $\mathbf{C}[\mathsf{Pub}](r, X)$ to include Th queries, which is merely a matter of syntax. We claim that in this game it holds

$$\Pr[\mathsf{Game}_4] = \Pr[\mathsf{Game}_3].$$

To see this, observe that the construction $\mathbf{C}_3$ when it computes TKDF always makes a complete and well-formed sequence of queries to compute TKDF. The last query always programs the response to ensure consistency with RO (or if it is repeated it is answered using the RO table M). As discussed earlier, if it happens that queries are issued out of order, a collision will be unavoidable as the output of the previous query will be (part of) the input of the subsequent query. So as long as bad_C is not set the games are identical. Now, since setting of bad_C induces a winning probability of 0 in both Game 4 and Game 3, it follows that the winning probability is identical.

GAME 5. In Game 5, we remove that the game outputs 0 if bad_C is set at the end of the game. This just increases by the collision term that we already discussed multiple times. The number of Pub in Game 5 still is $q_{\mathsf{Pub}} + 6q_{\mathsf{Priv}}$. Therefore,

$$\Pr[\mathsf{Game}_5] \leq \Pr[\mathsf{Game}_4] + \frac{2(q_{\mathsf{Pub}} + 6q_{\mathsf{Priv}})^2}{2^\lambda}$$

GAME 6. After having removed random oracle RO from the Priv oracle, random oracle RO is only queried by Sim_3. That is, we do not need to ensure consistency between Sim_3 and RO anymore. Hence, we implement oracle Pub using a random function from S_3 implemented via lazy sampling instead of using the simulator. The adversary cannot detect this change, as it does not have access to RO anymore. Then, it holds that

$$\Pr[\mathsf{Game}_6] = \Pr[\mathsf{Game}_5].$$

Finally, observe that Game 6 is equal to "real" indifferentiability game $\mathrm{Exp}^{\mathsf{indiff},1}_{\mathbf{C}_3,\mathsf{Sim}_3,\mathsf{S}_3,\mathsf{E}_3}(\mathcal{D})$. Collecting all the bound implies the lemma. □

Note that the result of Lemma 7.7 covers the whole definition of the TKDF as it is used in the PSK handshakes. In the full handshake, we have that the pre-shared key, for example, is constantly set to 0, and the functions to derive the binder MAC, the early traffic secret *ets* and the early exporter master secret *eems* are omitted. Nevertheless, this does not change that the result remains applicable for the full handshake as the version proven above is more general. This is why we chose the modular approach of first proving the indifferentiability separately.

Next, we apply Lemma 7.7 to the different handshake modes. The following two lemmas directly follow from the indifferentiability composition theorem Theorem 7.1 and Lemma 7.7. Note that for both the abstracted full and PSK handshake shown in Figure 7.2 for each flight of the protocol, function TKDF only has to be computed at most 6 times. This implies that for every q_{Send} issued by the adversary $\mathcal{A}$, the distinguisher $\mathcal{D}$ constructed in the proof of Theorem 7.1 has to issued at most 6 Priv queries, i.e., $q_{\mathsf{Priv}} = 6q_{\mathsf{Send}}$.

Lemma 7.8. *Let* KE_2 *be the TLS 1.3 full handshake protocol as defined in Lemma 7.5 and let* KE' *be the abstracted full handshake protocol as defined in Theorem 7.3. Then, for any adversary* $\mathcal{A}$ *against the* pMSKE *security of* KE_2*, we can construct an adversary* $\mathcal{B}$ *against* KE' *such that*

$$\mathrm{Adv}^{\mathsf{pMSKE}}_{\mathsf{KE}_2}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{pMSKE}}_{\mathsf{KE}'}(\mathcal{B}) + \frac{2q^2_{\mathsf{RO}}}{2^\lambda} + \frac{8(q_{\mathsf{RO}} + 36q_{\mathsf{Send}})^2}{2^\lambda}$$

where q_{RO} *and* q_{Send} *are number of queries issued by* $\mathcal{A}$ *to the random oracle* RO *(in total) and oracle* q_{Send}*, respectively.*

Lemma 7.9. *Let* KE_2 *be the TLS 1.3 full handshake protocol as defined in Lemma 7.6 and let* KE' *be the abstracted full handshake protocol as defined in Theorem 7.4. Then, for any adversary* $\mathcal{A}$ *against the* sMSKE *security of* KE_2*, we can construct an adversary* $\mathcal{B}$ *against* KE' *such that*

$$\mathrm{Adv}^{\mathsf{sMSKE}}_{\mathsf{KE}_2}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{sMSKE}}_{\mathsf{KE}'}(\mathcal{B}) + \frac{2q^2_{\mathsf{Pub}}}{2^\lambda} + \frac{8(q_{\mathsf{RO}} + 36q_{\mathsf{Send}})^2}{2^\lambda}$$

where q_{RO} *and* q_{Send} *are number of queries issued by* $\mathcal{A}$ *to the random oracle* RO *(in total) and oracle* q_{Send}*, respectively.*

Theorem 7.3 and Theorem 7.4 are a direct consequence of Lemmas 7.2, 7.5 and 7.8, and Lemmas 7.3, 7.6 and 7.9, respectively. With these results, we can reduce the complexity of our (tight) security proofs for the TLS 1.3 handshakes presented in Chapter 6 as it suffices to only prove security for the abstracted TLS 1.3 handshakes as presented in Figure 7.2.

7.5 Defining the Domains $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$

Lemmas 7.1 to 7.3 highly rely on the assumption that we can define a set $\mathcal{D}_{\mathsf{Th}}$ that defines all possible strings representing all possible transcripts to be hashed during the TLS handshake and a set $\mathcal{D}_{\mathsf{Ch}}$ that defines all possible inputs to the hash function when it is used as a subroutine of HMAC during the TLS handshake. Clearly, formally defining these sets and even writing them out is complex. However, the above mentioned lemmas can only be valid if these sets exist. In this section, we define these sets based on formatting of the TLS messages defined in the standard RFC 8446 [Res18] and show that they are indeed disjoint. Since the full handshake and the PSK handshake of TLS 1.3 are different in some aspects, we analyse these sets separately for each of the two variants. We start

with the full handshake, which is a new addition in this work and follow-up with the PSK mode of the TLS 1.3 handshake, which is based on [DDGJ22b] (the details presented here are only present in the full-version [DDGJ22a]). Before we actually define these sets, we first discuss assumptions that we make, and more importantly define the format of the possible inputs to the hash function.

7.5.1 Assumptions and Hash Query Types

In the following, we need to define disjoint sets $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$ such that an honest execution of the TLS 1.3 handshake only queries the random oracle RO_H on $\mathcal{D}_{\mathsf{Ch}}$ to compute HMAC and on $\mathcal{D}_{\mathsf{Th}}$ otherwise. To this end, we make a couple of assumptions.

Assumptions

The first assumption is on how messages are processed by honest sessions. We assume that a server only responds (resp. continues the execution of the protocol) if the received `ClientHello` message contains valid encodings of all the *mandatory* parameters of TLS 1.3. If a server receives only invalid encodings, we assume that the server will abort. Note that this does not exclude malformed `ClientHello` messages whatsoever, but rather mandates that *only* valid ones will be processed. As we will see in Section 7.5.3, we have to strengthen this assumption a bit to be able to separate the domains for PSK-only with SHA384. Here, we need to require that the cipher suite values and extensions presented in a `ClientHello` consist only of standardized values. For details, we refer to Section 7.5.3. Further, the TLS 1.3 standard gives the field length always in (full) bytes. That is, in contrast to the majority of this thesis, we consider the byte length of strings in this section. We denote the output length in bytes of random oracle RO_H by $\Lambda := \lambda/8$ with λ being the output length of the configured hash function (i.e., either SHA256 or SHA384). For the PSK mode, we always assume that the pre-shared key is of the same length as the output length of RO_H, i.e., of length Λ bytes. Furthermore, if a Diffie–Hellman group is used, we assume that it is one of the groups that are standardized (either elliptic curve or finite field; cf. [Res18, Sect. 4.2.7]). Recall that in Chapter 6, we presented our view on the TLS 1.3 handshake and stated the assumption that we do not consider parameter negotiation. We consider the cipher suite, algebraic group, and signature scheme fixed once and for all. That is we consider all combinations of these parameters, but in isolation. To reflect this in this section, we assume that an honest client at least presents the fixed parameters, and the server always selects the fixed parameters and ignores the others. To be precise, this means, for example, that a client can present multiple cipher suites in the `ClientHello` message, but the one that will be selected by the receiving server is known before hand. This applies to all other parameters listed above.

Hash Query Types

The TLS 1.3 handshake does not provide *intentional* domain separation for the use its hash function. Therefore, we need to analyze the formatting of the RO_H queries made by honest executions during the protocol run to find *unintentional* domain separation.

Allowing us to separate the sets $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$. Note that the following analysis does not apply to the RO_H queries issued by the adversary. The adversary can query the random oracle RO_H arbitrary. The exact formatting of each input to the random oracle RO_H highly depends on the considered mode of the TLS 1.3 handshake (i.e., full, PSK-only, or PSK-(EC)DHE) and the selected cipher suite, in particular, the choice of hash function. Therefore, we analyze each combination of mode and hash function separately. However, the types of random oracle queries are shared between all of these cases. To this end, we first define all types of queries to random oracle RO_H appearing in the TLS 1.3 handshake and then analyze each configuration separately in Sections 7.5.2 and 7.5.3 below.

Types of RO_H queries. In general, we classify queries to random oracle RO_H into two types:

- **Type 1 (component hash)**: These kind of queries are used to compute **Extract**, **Expand**, or **MAC**, which ultimately are used to compute an HMAC.
- **Type 2 (transcript hash)**: These kind of queries are used to compute digests of protocol transcripts (or the empty string).

Further, these two types of queries can be split into seven subtypes: five subtypes of Type 1 queries and two subtypes of Type 2 queries. Before we define the subtypes, first recall the definition of **Extract**, **Expand**, and **MAC**. Either of these functions is defined in Section 6.3 in terms of **MAC**, where **MAC** $:=$ HMAC. **Extract** (abstracting HKDF.Extract; cf. Section 6.1.2) is the same function as **MAC**, i.e., **Extract** $:=$ HMAC. **Expand** (abstracting HKDF.Expand; cf. Section 6.1.2) is almost identical to the HMAC function, but adds a trailing 0x01 byte to the end of the input. Formally, for a key k and an input s, it holds **Expand**$(k, s) = \mathsf{HMAC}(k, s \parallel \texttt{0x01})$. Let us elaborate, why **Expand** will always compute a single block. For this, recall that HKDF.Expand as defined in Section 6.1.2 is a variable output length function. Informally, this means that it takes a length parameter L and produces $\lceil L/\lambda \rceil$ many blocks of length λ each resulting from a HMAC computation, where λ is the output length of the function underlying HMAC, and then trims down the last block to yield the desired output length of L. Note that for the TLS 1.3 handshake all HKDF.Expand calls have a output length of at most the output length of the underlying hash function, i.e., the hash function from the cipher suite. This results then in HKDF.Expand only computing a single output block, thus being computed as a single HMAC call where the input is padded with a 0x01 byte. In particular, this also holds for the two **Expand** calls in the conceptual subroutine DeriveTK used to compute the handshake traffic keys. The function DeriveTK has a output length of $l + d$, where l is the encryption key length and d is the IV length of the configured AEAD scheme, respectively. Internally, DeriveTK makes two **Expand** calls: one to compute the AEAD key with output length l and one to compute the AEAD IV with output length d. Now, checking the standard [Res18, App. B.4] one observes that for each standardized AEAD scheme defined for TLS 1.3, the key length is at most 256 bit and the IV length is always 96 bit (for references, see [McG08, Sect. 5], [NL18, Sect. 2.8], and [MB12, Sect. 6.1]). That is, both l and d are at most the output length λ of configured hash function for both

Table 7.1: Table displaying the standardized groups for use with TLS 1.3, their encodings in the `NamedGroup` enum, and the length of an encoded group element in bytes. This table is taken from [DDGJ22a].

Group name	`NamedGroup` value	Enc. length $\lvert\mathbb{G}\rvert/8$
`secp256r1` [Nat13]	`0x0017`	32
`secp384r1` [Nat13]	`0x0018`	48
`secp521r1` [Nat13]	`0x0019`	66
`x25519` [LHT16]	`0x001d`	32
`x448` [LHT16]	`0x001E`	56
`ffdhe2048` [Gil16]	`0x0100`	128
`ffdhe3072` [Gil16]	`0x0101`	192
`ffdhe4096` [Gil16]	`0x0102`	256
`ffdhe6144` [Gil16]	`0x0103`	384
`ffdhe8192` [Gil16]	`0x0104`	512

SHA256 and SHA384. Hence, without loss of generality, we can assume that the **Expand** calls in the computation of DeriveTK also derive λ many output bits, and only use the first l or d many bits, respectively.

Since each of the Type 1 queries boils down to an HMAC computation, we briefly recall its definition. The function $\mathsf{HMAC}[\mathsf{RO}_H](k, s)$ is defined over the random oracle RO_H and takes as input a key k of length Λ bytes. The key k is then padded with zeroes to the block size B of the underlying function, we call the padded key k'. The block size of SHA256 and SHA384 are 64 and 128 bytes, respectively. Thus, the key k will in TLS 1.3 always be padded with $B - \Lambda > 0$ many `0x00` bytes. To compute its output $\mathsf{HMAC}[\mathsf{RO}_H](k, s)$ queries RO_H twice:

1. $h := \mathsf{RO}_H(k' \oplus \mathsf{ipad} \parallel s)$
2. $\mathsf{HMAC}[\mathsf{RO}_H](k, s) = \mathsf{RO}_H(k' \oplus \mathsf{opad} \parallel h)$

where ipad and opad are byte strings of length B, where each byte is fixed to `0x36` and `0x5c`, respectively. Since the padded key k' always ends with a segment of $B - \Lambda > 0$ many `0x00` bytes, $k' \oplus \mathsf{pad}$ with $\mathsf{pad} \in \{\mathsf{ipad}, \mathsf{opad}\}$ every Type 1 query has a segment of $B - \Lambda$ many bytes fixed to either `0x36` or `0x5c`.

The seven subtypes of RO_H queries are then defined as follows:

1. **Outer HMAC queries**. We call the second query ($\mathsf{RO}_H(k' \oplus \mathsf{opad} \parallel h)$) made to compute HMAC an outer HMAC query. The first part of the input has length B bytes, and ends with a segment of $B - \Lambda$ many `0x5c` bytes. The second part is a digest h of length Λ bytes. We call the segment at the end of the first part the "fixed region". For SHA256, the fixed region is 32 bytes, and for SHA384 the fixed region is 80 bytes. This yields a total input length (independent of the considered handshake mode) of 96 bytes for SHA256 and 176 bytes for SHA384 for outer HMAC queries.

2. **Inner HMAC queries**. The next four subtypes are variants of the first query ($h \coloneqq \mathsf{RO}_H(k' \oplus \mathsf{ipad} \parallel s)$) made to compute HMAC. The first variant is called an inner HMAC query and occurs if the input string s is an arbitrary byte string of length Λ. Inner HMAC queries are formatted identically as outer HMAC queries except that the fixed region is filled with `0x36` instead of `0x5c` bytes. These kind of queries occur in the computation of finished and binder MACs, the computation of the early and master secret in all modes, and only in the PSK-only mode, also for the computation of the handshake secret. For completeness, this yields a total input length (independent of the considered handshake mode) of 96 bytes for SHA256 and 176 bytes for SHA384 for inner HMAC queries.

3. **Diffie–Hellman HMAC queries**. The second subtype of the first HMAC query, is called a Diffie–Hellman HMAC query. This kind of query occurs if the input string s is an encoded Diffie–Hellman key share. It appears in the full and PSK-(EC)DHE mode of the handshake protocol to compute the handshake secret. The format is similar to inner HMAC queries, but the length of the second part (i.e., following the fixed region) has the length of an encoded Diffie–Hellman group element. We denote this length (in bytes) by $|\mathbb{G}|/8$ and give the corresponding values for each standardized group in Table 7.1. Given that the first part of the input has length block size B of the hash function, this yields a total input length of a Diffie–Hellman HMAC query of $64 + |\mathbb{G}|/8$ bytes for SHA256 and $128 + |\mathbb{G}|/8$ bytes for SHA384.

   We assume that an honest execution of TLS 1.3 only queries group elements in this context that correspond to the group fixed for the considered instance of TLS 1.3. Recall we consider each configuration of TLS 1.3 in isolation and all clients and server use the same group.

4. **`Derive-Secret` queries**. The TLS 1.3 key schedule [Res18, Sect. 7.1] defines a function `Derived-Secret(Secret, Label, Messages)`, which is ultimately a particular way of calling HKDF.Expand. We abstracted this in our view by writing it out using HKDF.Expand. The inputs are a secret `Secret` of length Λ bytes, a label string of 2 to 12 ASCII characters (i.e., each character is encoded by one byte), and an (arbitrary) string `Messages`. `Derive-Secret` performs the following computations:

   a) It queries RO_H(`Messages`) to hash down the input string; this is a transcript hash.

   b) It queries RO_H with a special inner HMAC query. We call this query the `Derive-Secret` query. The format of the query is identical to the inner HMAC query and the Diffie–Hellman query except that the input string is the struct `HkdfLabel` followed by a `0x01` byte. We already mentioned this struct briefly in Section 6.3. The struct is defined as follows (cf. [Res18, Sect. 7.1]):

      - 2 bytes encoding the integer value of Λ

- A variable-length vector consisting of a 1 byte length field followed by the string `"tls13 "` encoded in 6 bytes and the string `Label` encoded in 2 to 12 bytes[4]
- A variable-length vector consisting of a 1 byte length field and a byte vector of length Λ (the hash of `Messages`)

The total length of the struct `HkdfLabel` including the trailing `0x01` byte is therefore at least $\Lambda + 13$ bytes and at most $\Lambda + 23$ bytes.

c) Lastly, it queries an outer HMAC query with the padded value of `Secret` as key and the result of the `Derive-Secret` query of b).

Resulting in an overall input length of `Derive-Secret` query (described in b)) of 109–119 bytes for SHA256 and 189–199 bytes for SHA384.

5. **`Finished` key queries**. To derive the finished keys to compute the `SF` message, the `CF` message, and the binder value in the PSK mode, a special `Derive-Secret` query is used. Namely, the label string `Label` is fixed to `"finished"` and the transcript hash (i.e., the final vector before the `0x01` byte) is replaced by a vector of length 0 (i.e., the empty string ε). This results in a label string encoded in 8 bytes and an overall length of the respective `HkdfLabel` struct of 19 bytes. Therefore, the total input length of a `Finished` key query is 83 bytes for SHA256 and 147 bytes for SHA384. This kind of query is the last variant of a first HMAC query and also concludes the subtypes of component hash queries (Type 1).
6. **Empty transcript queries**. We call the query $\mathsf{RO}_H(\varepsilon)$ an empty transcript query.
7. **Transcript queries**. All query (sub-)types defined before are valid independent of the considered mode. This is different for the RO_H queries that are used to condense transcripts, which we call transcript (hash) queries. Here, we need to distinguish between the full and PSK handshake as the minimal transcript that is hashed down during the respective handshakes differ.

Full handshake. In the TLS 1.3 full handshake, the smallest transcript that needs to be hashed down (other than the empty transcript ε) is the hash that we denote by H_3, which consists of the `ClientHello` messages followed by the `ServerHello` message (including all extensions appended to these messages). In the description of TLS 1.3 we omitted a couple of fields to reduce complexity as they did not effect our view when it comes to the security proof. In this section, we need to be a bit more careful and consider fields that we omitted before, to ensure a sound domain separation treatment. Let us consider the minimal size of a `ClientHello` message and `ServerHello` message, respectively, in the full handshake:

The `ClientHello` message is minimally structured (including all mandatory extensions) in the full handshake as follows:

- 1 byte message type (`0x01` for `client_hello`)

[4] The label string in TLS are no longer than 12 ASCII characters. (cf. [Res18, Sect. 7.1])

- 3 bytes message length field
- 2 bytes `legacy_version` fixed to `0x0303` (indicating TLS 1.2)
- 32 bytes `random`
- 1 byte `legacy_session_id` (empty vector with 1 byte length field; we do not consider compatibility with previous versions of TLS, so the standard mandates that this must be an zero-length vector [Res18, Sect. 4.1.2])
- 4 bytes `cipher_suites` (2 bytes length field + 2 bytes cipher suite value, e.g., `0x1303` for `TLS_AES_128_GCM_SHA256`)
- 2 bytes `legacy_compression_methods` (1 byte length field + 1 byte fixed to `0x00` for "null" compression method; TLS 1.3 `ClientHello` messages need to set this exactly like this [Res18, Sect. 4.1.2])
- 2 bytes length field of the `extensions` vector
- 7 bytes of `supported_versions` extension (2 bytes extension type + 2 bytes extension length field + 1 byte length field + 2 byte fixed to `0x0304` for TLS 1.3)
- 8 bytes `signature_algorithms` extension (2 bytes extension type + 2 bytes extension length field + 2 byte length field + 2 bytes `SignatureScheme` value [Res18, Sect. 4.2.3])
- 8 bytes `supported_groups` extension (2 bytes extension type + 2 bytes extension length field + 2 bytes length field + 2 bytes `NamedGroup` value [Res18, Sect. 4.2.7])
- $10 + |\mathbb{G}|/8$ bytes `key_share` extension (2 bytes extension type + 2 bytes extension length field + 2 bytes length field + 2 bytes `NamedGroup` value [Res18, Sect. 4.2.7] + 2 bytes length field + $|\mathbb{G}|/8$ bytes encoding of a `NamedGroup` element (cf. Table 7.1)[5]

Note that according to [Res18, Sect. 9.2], the extensions presented above (`supported_versions`, `signature_algorithms`, `supported_groups`, and `key_share`) are mandatory for (`ClientHello`) messages for (EC)DHE key exchange and certificate authentication negotiating TLS 1.3, which all applies to the full handshake. Also, note that the `key_share` extension might be sent empty to force a `HelloRetryRequest`, we do not cover this in our view, so we require the `key_share` extension to contain at least a single key share entry (i.e., a pair of group and group element). All of the extensions mentioned above can be ordered arbitrarily. The above minimal format of a `ClientHello` message in the full handshake results in a minimum length of $80 + |\mathbb{G}|/8$ bytes.

The `ServerHello` message is minimally structured (including all mandatory extensions) in the full handshake as follows:

[5] Note that we do not cover negotiation in this work. Therefore, we assume that the group that is presented here is known in at advance and corresponds to the group $\mathbb{G}$ considered to be configured for the protocol.

- 1 byte message type (`0x02` for `server_hello`)
- 3 bytes message length field
- 2 bytes `legacy_version` fixed to `0x0303` (indicating TLS 1.2)
- 32 bytes `random`
- 1 byte `legacy_session_id_echo` (empty vector with 1 byte length field)
- 2 bytes `cipher_suite` (2 bytes cipher suite value)
- 1 bytes `legacy_compression_method` (1 byte fixed to `0x00` for "null" compression method)
- 2 bytes length field of the `extensions` vector
- 6 bytes of `supported_versions` extension (2 bytes extension type + 2 bytes extension length field + 2 byte fixed to `0x0304` for TLS 1.3)
- $8 + |\mathbb{G}|/8$ bytes `key_share` extension (2 bytes extension type + 2 bytes extension length field + 2 bytes `NamedGroup` value + 2 byte length field + $|\mathbb{G}|/8$ bytes encoding of a `NamedGroup` element (cf. Table 7.1)

The above minimal format of a `ServerHello` message in the full handshake results in a minimum length of $58 + |\mathbb{G}|/8$ bytes. This yields a minimal input length of a transcript query in the full handshake to $138 + |\mathbb{G}|/8$ bytes.

In remains to determine an upper bound of the transcript length to determine the maximum input length of a transcript hash query. Note that the `ServerHello` message cannot vary in length as it only contains selections from the options the client presented for the negotiation of the cryptographic parameters. Therefore, the only thing that can be changed in the size is the extension field. This is bounded by a maximum size of 2^{16} bytes. Hence, the maximum $44 + 2^{16}$ bytes (including the encoding of the group element selected for key exchange). The maximum length of the `ClientHello` message is more complex as the `ClientHello` includes a number of variable-length vectors. These vectors are `cipher_suites`, `legacy_compression_methods`, and similar to the `ServerHello` the extension field. The vector of cipher suites and the extension field are each upper bounded by 2^{16} bytes. The compression methods can hold a vector of up to 2^8 bytes. However, the standard mandates that all TLS 1.3 `ClientHello` messages set this vector must contain "[…] exactly one byte, set to zero, which corresponds to the 'null' compression method […]" [Res18, Sect. 4.1.2]. Since we only consider negotiation of TLS 1.3 we adapt this here. With these considerations we obtain an upper-bound for the `ClientHello` message in the full handshake of $45 + 2^{17}$ bytes. This yields an overall upper-bound of $99 + 3 \cdot 2^{16}$ bytes for a transcript containing a `ClientHello` and `ServerHello` message.

Note that all other transcripts to be hash will only be strictly longer and always contain the prefix `ClientHello` and `ServerHello`. To this end, we only consider the smallest transcript as a "transcript hash". All arguments presented for this kind

of query in particular also holds for all other transcript hashes H_i computed in the full handshake (Table 6.1).

PSK-only handshake. In the PSK handshakes, each transcript contains at least a *partial* `ClientHello` (i.e., a `ClientHello` message without the binder value/list). Up to the `supported_versions` extension this partial `ClientHello` message is structured identically to the `ClientHello` message in the full handshake described above. We do not have authentication via certificates and thus the `signature_algorithms` extension is never present. Instead two other extensions are mandatory: (1) `psk_key_exchange_modes` [Res18, Sect. 4.2.9] indicating all supported PSK modes (i.e., PSK-only (`0x00`) or PSK-(EC)DHE (`0x01`)) and (2) `pre_shared_key` [Res18, Sect. 4.2.11] containing a (list of) PSK(s). In the PSK-only mode, the extensions `supported_groups` and `key_share` are also not required compared to the full handshake. Therefore, a minimal partial `ClientHello` message in the PSK-only mode includes the following fields:

- 1 byte message type (`0x01` for `client_hello`)
- 3 bytes message length field
- 2 bytes `legacy_version` fixed to `0x0303` (indicating TLS 1.2)
- 32 bytes `random`
- 1 byte `legacy_session_id` (empty vector with 1 byte length field; we do not consider compatibility with previous versions of TLS, so the standard mandates that this must be a zero-length vector [Res18, Sect. 4.1.2])
- 4 bytes `cipher_suites` (2 bytes length field + 2 bytes cipher suite value, e.g., `0x1303` for `TLS_AES_128_GCM_SHA256`)
- 2 bytes `legacy_compression_methods` (1 byte length field + 1 byte fixed to `0x00` for "null" compression method; TLS 1.3 `ClientHello` messages need to set this exactly like this [Res18, Sect. 4.1.2])
- 2 bytes length field of the `extensions` vector
- 7 bytes of `supported_versions` extension (2 bytes extension type + 2 bytes extension length field + 1 byte length field + 2 byte fixed to `0x0304` for TLS 1.3)
- 6 bytes of `psk_key_exchange_modes` [Res18, Sect. 4.2.9] extension (2 bytes extension type + 2 bytes extension length field + 1 byte length field + 1 byte `PskKeyExchangeMode` value (`0x00` for PSK-only).
- 13 bytes of `pre_shared_key` [Res18, Sect. 4.2.11] extension (2 bytes extension type + 2 bytes extension length field + 4 byte length field + 1 byte PSK identity + 4 byte "obfuscated ticket age"; this extension must always be the last; note that this is the partial version excluding the binder list)

Therefore, we get an minimal length of a partial `ClientHello` in the PSK-only mode of at least 73 bytes. For an upper bound, recall that the fields `cipher_suites`

and `extensions`. As already discussed above both fields have a maximum length of 2^{16} bytes. Since the fields up to the extensions are identical to the full handshake, we also get an upper bound of $45+2^{17}$ for the partial `ClientHello` in the PSK-only mode.

PSK-(EC)DHE handshake. To mitigate repetition, we only briefly discuss the changes for a minimal transcript in the PSK-(EC)DHE handshake. As mentioned above, each transcript hashed in the PSK-(EC)DHE handshake contains at an partial `ClientHello` message. In this mode, the `ClientHello` message is more or less a hybrid out of the PSK-only (partial) `ClientHello` and the `ClientHello` message presented above for the full handshake. In detail, this means that the partial `ClientHello` is similar to the one in the PSK-only handshake, but two additional extensions (for the DH key exchange) are mandatory. Namely, the `key_share` and `supported_groups` extensions have to be added. The minimal length of these extensions is identical to the full handshake such that $18 + |\mathbb{G}|/8$ bytes need to be added. The rest of the partial `ClientHello` is identical to the PSK-only `ClientHello`. That is, we have a minimal length of $73+18+|\mathbb{G}|/8 = 91 + |\mathbb{G}|/8$ bytes. Since only the extensions are different, and their length is upper bounded by 2^{16} bytes, the maximum length is $45 + 2^{17}$ bytes in the PSK-(EC)DHE handshake, as well.

7.5.2 Domain Separation in the TLS 1.3 Full Handshake

After defining all the possible queries that might be issued to the random oracle RO_H, let us now start defining the domains $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$. As the formatting of the message highly depends on the selected parameters, as already outline above, we now consider the full handshake and separate two cases. First of all, we fix the signature algorithm, the cipher suite, and the group that will be negotiated. This implies that the `ClientHello` at least has to contain these parameters and the server will always selected the fixed values. However, we allow clients to repeat values or add additional values, but require that the server ignores these, since we do not capture parameter negotiation. Since the length of the queries as discussed in the previous part, highly depends on the hash function used, we consider the choice of hash function as an individual case. Recall that TLS 1.3 allows hash function SHA256 (with output length $\lambda = 32$ bytes and block size $B = 64$ bytes) and SHA384 (with output length $\lambda = 48$ bytes and block size $B = 128$ bytes).

Full Handshake with SHA256

Let us start with the case in which SHA256 is the configured hash function that is modeled as RO_H. In Table 7.2, we show the minimum and maximum input length for each of the query types, we defined above. Next, we have to show for each type with overlapping length that we can use the structure of the query to separate the query types. First note that outer HMAC, inner HMAC, and DH HMAC queries could potentially collide for a choice of $\mathbb{G}$ with encoding length of 32 bytes (e.g., the mandatory-to-implement group

Table 7.2: Minimal input length of the hash function calls made by TLS 1.3 in the full handshake with SHA256.

Query type	Min. length (in bytes)	Max. length (in bytes)
Outer HMAC	96	96
Inner HMAC	96	96
DH HMAC	$64 + \lvert\mathbb{G}\rvert/8$	$64 + \lvert\mathbb{G}\rvert/8$
`Derive-Secret`	109	119
`Finished` key	83	83
Empty transcript	0	0
Transcript	$138 + 2\lvert\mathbb{G}\rvert/8$	$89 + 3 \cdot 2^{16}$

`secp256r1`; cf. Table 7.1 and [Res18, Sect. 9.1]) because all then have a length of 96 bytes. Moreover, the input length of DH HMAC queries and `Derive-Secret` queries might overlap for a choice of $\mathbb{G}$ with encoding length 48 bytes (e.g., `secp384r1`; cf. Table 7.1) with an input length of 112 bytes. Now, observe that all of the mentioned queries are component hash queries (Type 1). That is, all belong to the same domain, and a collision would not be a problem. Furthermore, transcript queries with an input length of at least $138 + 2|\mathbb{G}|/8$ will always be longer than any of the possible component hash queries. Therefore, separating Type 2 queries (either of length 0 or at least $138 + 2|\mathbb{G}|/8$) from Type 1 queries is easy. However, we want to be more precise, so let us have a deeper look on the queries.

First, recall that HMAC random oracle queries have a segment that we referred to has the "fixed region". This region for SHA256 starts with byte 33 and ends with byte 64 of the input. Outer and inner HMAC queries can always be easily separated by this fixed region. Namely, in an outer query, all bytes in the fixed region are set to `0x5c`, and in an inner query all bytes in this region are set to `0x36`. Second, recall that inner HMAC, DH HMAC, and `Derive-Secret` all are variants of the first random oracle query used to compute HMAC. That is, all have the same fixed region set to `0x36`, but the segment following has different semantics. As a reminder, for inner HMAC this segment can be arbitrary, for DH HMAC this needs to be an encoding of group element and for `Derive-Secret` this is a special struct. Unfortunately, if for inner HMAC and DH HMAC, and DH HMAC and `Derive-Secret`, respectively, the input length collide we cannot further separate. But as noted above this is not a problem as all of these queries are Type 1 queries that ultimately belong to the same domain.

Next, let us have a closer look at transcript hash queries. As noted above transcript queries can easily be separated from component hash queries by length. For any choice of group, they will be longer. However, a transcript hash query has a particular structure since it always starts with a `ClientHello`. Namely, through the `legacy_session_id` it is structured as follows:

$$\texttt{0x01} \,\|\, \underbrace{\cdots}_{\text{3 bytes}} \|\texttt{0x0303} \,\|\, \underbrace{\cdots}_{\text{32 bytes}} \|\texttt{0x00}$$

Table 7.3: Minimal input length of the hash function calls made by TLS 1.3 in the full handshake with SHA384.

Query type	Min. length (in bytes)	Max. length (in bytes)
Outer HMAC	176	176
Inner HMAC	176	176
DH HMAC	$128 + \lvert\mathbb{G}\rvert/8$	$128 + \lvert\mathbb{G}\rvert/8$
`Derive-Secret`	189	199
`Finished` key	147	147
Empty transcript	0	0
Transcript	$138 + 2\lvert\mathbb{G}\rvert/8$	$89 + 3 \cdot 2^{16}$

Now, since we only consider TLS 1.3 `ClientHello` messages, the `legacy_session_id` always will only consists of an empty vector with a length field set to `0x00`. That is, at byte 39 of a transcript hash there will `always` be a `0x00` byte. If we now compare this to the component hash queries we observe that byte 39 is exactly in the fixed region of the HMAC queries. That is for all other query types (i.e., 1–5), the 39-th byte will always be either `0x5c` or `0x36`.[6]

With these considerations, we can precisely define the domains $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$ for the full handshake protocol configured with SHA256 as follows:

> The domain of all transcripts $\mathcal{D}_{\mathsf{Th}}$ is the set that contains the empty string and all strings in $\{0,1\}^*$ with byte length at least $138+2\lvert\mathbb{G}\rvert/8$, where the 39-th byte is not equal to `0x5c` or `0x36`. All other strings in $\{0,1\}^*$ are contained in $\mathcal{D}_{\mathsf{Ch}}$.

Note that this definition depends on the choice of group considered for the full handshake. As mentioned before, we assume that the cryptographic parameters such as signature algorithm, cipher suite, and group are fixed, and do not consider negotiation. That is, the domains $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$ are defined differently for different combinations of cryptographic parameters, but for all parameters we can define such domains. Therefore, we have shown that Lemma 7.2 is valid for all combinations of signature algorithm, cipher suite with SHA256, and group in isolation.

Full Handshake with SHA384

Next, we turn to the case in which SHA384 is the configured hash function that is modeled as RO_H. Consider the minimum and maximum input length for each query type in Table 7.3. Recall that the output length for SHA384 is 48 bytes and its block size is 128 bytes. This implies that the fixed region for the HMAC queries now is 80 bytes and starts in byte 49. That is, outer and inner HMAC queries still can be identified by their

[6] Note that even if we would allow a legacy session id this would have a maximum length of 32 bytes according to [Res18, Sect. 4.1.2], which corresponds to length field value `0x20`. That is, even in this case, byte 39 would never be either `0x5c` or `0x36`.

fixed region being either `0x5c` or `0x36`. Collisions between inner HMAC and DH HMAC, and DH HMAC and `Derive-Secret`, respectively, still are not obviously to separate due to lack of structure. But as already noted above, this is not a problem as all of these queries belong to the same domain (component hashes).

Transcript hashes in this case can again be easily separated from component hashes due to their length. Even for the smallest possible group, a transcript hash query is of input length at 202 bytes, which is larger than any component hash query. For SHA256, we were able to give a bit more structure to transcript hash queries by using the "fixed prefix" and the position of the `legacy_session_id`. This is unfortunately more difficult for the case of SHA384. Let us elaborate on this. Every transcript starts with a `Client Hello` message that in turn starts with a prefix of 39 bytes (from the message type through the legacy session id) followed by 2 bytes for the length field for the cipher suite vector and at least 2 bytes (up 2^{16}) cipher suite value(s). The cipher suites are followed by 4 bytes (legacy compression method and extension length) before (at least) $33 + |\mathbb{G}|/8$ bytes for the *mandatory* extension data follow. Now, the fixed region of any HMAC-related query starts in byte 49 and ends in byte 80. The problem for SHA384 now is that the fixed region starts after the cipher suite length field. Since the cipher suite field is of variable-length, we cannot say with certainty where it will end, as clients could present multiple cipher suites (or even repeat values). We only require that the cipher suite we are considering is present. Therefore, we cannot say for sure whether any byte in the region of byte 49–80 is different from the fixed region. However, to be clear this does not effect the possibility to define the domain of transcripts as transcript for the full handshake fortunately are already separated by length. We only have to be more general compared to SHA256.

> For the full handshake protocol configured with SHA384, we define the domain $\mathcal{D}_{\mathsf{Th}}$ as the set containing the empty string and all string of $\{0,1\}^*$ with byte length of at least $138 + |\mathbb{G}|/8$. All other strings of $\{0,1\}^*$ are contained in $\mathcal{D}_{\mathsf{Ch}}$. Recall that these sets highly depend on the considered group.

Closing Remarks on Domain Separation for the Full handshake

As discussed above we were able to define separate domains $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$ such that the TLS 1.3 full handshake instantiated with fixed cryptographic parameters only queries its hash function (resp. the random oracle) on $\mathcal{D}_{\mathsf{Th}}$ when computing transcript hashes and on $\mathcal{D}_{\mathsf{Ch}}$ when computing HMAC (and related functions). For the full handshake, the minimal byte length of a transcript is already long enough to separate transcripts by length. Therefore, explicit intentional domain separation is not necessarily required as the length of the inputs already can be leveraged for separation. However, having intentional domain separation for each use of the hash function would be the cleanest and safest solution, and should be considered in future revision of the protocol.

Table 7.4: Minimal input length of the hash function calls made by TLS 1.3 in the PSK handshakes with SHA256.

Query type	Min. length (in bytes)	Max. length (in bytes)
Outer HMAC	96	96
Inner HMAC	96	96
DH HMAC (only in PSK-(EC)DHE)	$64 + \lvert\mathbb{G}\rvert/8$	$64 + \lvert\mathbb{G}\rvert/8$
`Derive-Secret`	109	119
`Finished` key	83	83
Empty transcript	0	0
Transcript (PSK-only)	73	$45 + 2^{17}$
Transcript (PSK-(EC)DHE)	$91 + \lvert\mathbb{G}\rvert/8$	$45 + 2^{17}$

7.5.3 Domain-separation in the TLS 1.3 PSK Handshake

Next, let us turn to defining the domain $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$ for the PSK handshakes. Similar to the treatment of the full handshake, we fix the cipher suite, the PSK mode, and (in the PSK-(EC)DHE mode) the group that will be negotiated. That is, we assume that in all `ClientHello` at least these parameters are listed and even if other parameters are present all servers will always pick the fixed parameters from the respective list. We consider four major cases in the section. Namely, we analyze the two PSK modes, PSK-only and the PSK-(EC)DHE, separately when configured with SHA256 or SHA384, respectively. Since there is a lot of overlap with the full handshake, we consider the two cases of PSK-only and PSK-(EC)DHE configured with SHA256 together. The argument is almost identical to the full handshake and only the domains defined differ. The case for SHA384 is more complex and therefore, we consider it for each PSK mode separately.

In this section, we only focus on collisions between transcript hashes and component hashes. In the treatment of the full handshake, we discussed how some of the component hashes can be separated by making use of their structure. Recall that this did not effect the definitions of $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$ and merely served as a side remark. This discussion remains valid for the PSK handshakes as it is only effected by the choice of hash function rather the considered handshake mode. So, we refer for more insights on that to Section 7.5.2.

PSK Handshakes with SHA256

In Table 7.4, we show the minimal input length of the hash function calls made by TLS 1.3 when SHA256 is configured as the hash function in the PSK handshakes. Observe that for the PSK-only handshake the length ranges of a transcript hash query overlaps with all component hash query types (i.e., outer and inner HMAC, `Derive-Secret`, and `Finished` key). While the length range of a transcript hash query in the PSK-(EC)DHE do not overlap with any of the component hash query types for all choices of the group $\mathbb{G}$. According to Table 7.4, the minimal length of a transcript hash query is $91 + |\mathbb{G}|/8$ bytes,

where $|\mathbb{G}|/8 \geq 32$ for all standardized groups. That is, the minimum length is at least 123 bytes, which is larger than any of the maximum length (for the same choice of group $\mathbb{G}$) given for the component hash types. Next, we give a separation argument for the PSK-only handshake. This argument is similar to the argument given for the full handshake and only uses the "fixed prefix" of the `ClientHello` message up to the `legacy_session_id`. In particular, the same argument applies to the `ClientHello` message in the PSK-(EC)DHE handshake and allows us to be more specific about the format of transcript hashes, even though in this mode transcript and component hashes could already be separated by length.

Recall that every `ClientHello` (also the partial one) starts with the same structure. Namely, 1 byte of message type (`0x01`), 3 bytes of `ClientHello` message length, 2 bytes set to `0x0303`, 32 bytes client nonce, and finally, 1 byte `legacy_session_id` = `0x00`. That is, byte 39 is always `0x00`. Now recall that every kind of the component hash query types represents either a first or second HMAC random oracle call. That is, it will start with 32 bytes incorporating the HMAC key followed by the "fixed region" that is 32 bytes of either `0x5c` or `0x36`. In particular, this means that for all component hash queries the 39-th byte will be different from `0x00`. This can be leveraged to separate Type 1 and Type 2 queries. For more details on this refer to the treatment of the full handshake in Section 7.5.2 above.

With this insight, we can define the domain $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$ for the PSK handshakes configured with SHA256 as follows:

- **PSK-only handshake with SHA256**: The domain of all transcripts $\mathcal{D}_{\mathsf{Th}}$ is the set that contains the empty string and all strings in $\{0,1\}^*$ with byte length at least 73, where the 39-th byte is not equal to `0x5c` or `0x36`. All other strings in $\{0,1\}^*$ are contained in $\mathcal{D}_{\mathsf{Ch}}$.
- **PSK-(EC)DHE handshake with SHA256**: The domain of all transcripts $\mathcal{D}_{\mathsf{Th}}$ is the set that contains the empty string and all strings in $\{0,1\}^*$ with byte length at least $91 + |\mathbb{G}|/8$, where the 39-th byte is not equal to `0x5c` or `0x36`. All other strings in $\{0,1\}^*$ are contained in $\mathcal{D}_{\mathsf{Ch}}$.

PSK-only handshake with SHA384.

In Table 7.5, we show the minimal input length of the hash function calls made by TLS 1.3 when SHA384 is configured as the hash function in the PSK-only handshake. The length range of transcript hash query overlap with all types of component hash queries. Let us compare a transcript to an outer HMAC call for demonstration. The other types can be treated analogously. Outer HMAC queries are in this configuration 176 bytes long. That is, we have to compare it to a (partial) `ClientHello` message of length 176 bytes. Let us first recall the mandatory segments. The first 39 bytes must appear in exactly that order (up to the `legacy_session_id`). These bytes are followed by 2 bytes of cipher suite length field and a cipher suites vector of length at least 2 bytes. Finally, there are 4 bytes of extension field length and at least 26 bytes of mandatory extensions. We refer to the definition of the minimal partial `ClientHello` message given in Section 7.5.1 for

Table 7.5: Minimal input length of the hash function calls made by TLS 1.3 in the PSK-only handshake with SHA384.

Query type	Min. length (in bytes)	Max. length (in bytes)
Outer HMAC	176	176
Inner HMAC	176	176
`Derive-Secret`	189	199
`Finished` key	147	147
Empty transcript	0	0
Transcript	73	$45 + 2^{17}$

details. Thus, for a `ClientHello` of length 176 there are 103 bytes undefined. Recall that the `cipher suites` and `extensions` field are variable-length vectors and can be extended. Schematically, a transcript is the structured as follows:

$$39 \text{ bytes} \parallel 4 \text{ bytes} \parallel \leftrightarrow \parallel 4 \text{ bytes} \parallel \leftrightarrow \parallel 26 \text{ bytes}$$

where $\leftrightarrow$ indicates that these segments could vary in length. Note that for the 26 bytes of mandatory extensions, the `pre_shared_key` extension always has to be the last extension of the extension field. The other could be reordered arbitrarily and therefore we only have a segment of variable-length before the mandatory extensions. To be more precise, actually there is a variable-length segment before and after each of the mandatory extensions, except for the `pre_shared_key` extension that always is last and thus only has a variable–length segment before. However, since the above is only an illustration, we decided to keep it simple to just highlight that at the start of the extensions field further extensions could be added to lengthen the transcript. In comparison, an outer HMAC query is structured as follows:

$$48 \text{ bytes} \parallel (\texttt{0x5c})^{80} \parallel 48 \text{ bytes}$$

where $(\texttt{0x5c})^{80}$ denotes the 80-byte string where all bytes are set to `0x5c`. Now, if one strictly follows the standard one can easily construct a collision between these two strings. Namely, the standard allows for, for example, the cipher suites field to contain values that are not standardized and servers should ignore all unknown/malformed values (cf. [Res18, Sect. 4.2.1]). Therefore, it is sufficient for a `ClientHello` message to be still considered valid if at least one valid value is contained in the cipher suites vector. Hence, a `ClientHello` message with the following cipher suite vector would be indistinguishable from an outer HMAC query. Define the cipher suite vector $\texttt{0x1302} \parallel (\texttt{0x5c})^{103}$, which yields a `ClientHello` message of length 176 bytes, containing a valid cipher suite `0x1302` for `TLS_AES_256_GCM_SHA384`. Now, observe that bytes 43–146 are fixed to `0x5c`, which in particular covers the fixed region of 80 `0x5c` in an outer HMAC call. Note that the same construction also works for all variants of first HMAC queries by replacing `0x5c` by `0x36` and adapting the factor 103 so that the respective length is achieved. A similar collision can be constructed when using extensions that are not specified. Here, servers

also are required to ignore unrecognized extensions (cf. [Res18, Sect. 4.1.2]) and as long as all mandatory extensions are present, the handshake can still be continued.

These kind of collisions seem unavoidable as long as it is permitted to arbitrarily choose the cipher suite vector and use unspecified extensions. We would like to highlight that in our view as described in Chapter 6, we only consider clients that suggest only a single cipher suite as we do not cover negotiation of cryptographic parameters. Nevertheless, we would like to be as close as possible to the standard, therefore we did not make any assumptions about the choice of cipher suites before as we were able to achieve the desired results even with this more general approach. Moreover, we consider it to be valuable independent of our analyses given in this work to understand the structure underlying the random oracle queries. Unfortunately, this setting seems to be too strong for the PSK-only mode with SHA384, because of the short minimal length of the partial `ClientHello` message and the resulting gap in length between the component and transcript hashes. This allows for too many bytes to be chosen freely and favors constructing collisions. Given that TLS 1.3 in our abstraction only will ever include a single (valid) cipher suite in the cipher suite vector, we consider it a reasonable assumption to only consider `ClientHello` messages that contain standardized cipher suites only. This is still more general than assuming the cipher suites field to be only a single element, and allows us to find a way to separate the two types of hash queries. We also make a similar assumption about the extensions in this mode. We assume that only extensions will be present in transcripts that are specified in the standard and no undefined extensions occur. Similar to the cipher suites, this is even more general than the view we actually considering as we only include mandatory extensions as described in Chapter 6.

We would like to highlight that even though no honest client would ever create a `ClientHello` with values that are not standardized in our view, an adversary could tamper with the message adding arbitrary, unstandardized values, for example, in the list of cipher suites, in transit. If the server forwards this `ClientHello` to an honest server, the honest server will query a transcript hash query that might cause a collision with a component hash. Nevertheless, this tampered value will never result in a valid honest execution, because the client that output the `ClientHello` will abort since its transcript will be different from the server's transcript, because the server received a different `ClientHello` than the client sent. Hence, the assumption that we are making to allow for domain separation in PSK-only with SHA384 basically is that the adversary does not tamper with `ClientHello` messages in the sense that it adds additional values in the variable length vectors of a valid, honest `ClientHello` message. To exclude this exact attack.

Under the assumption of `ClientHello` messages only including standardized cipher suite values as defined in [Res18, App. B.4], i.e., `0x130i` for `i` $\in \{1, 2, 3, 4, 5\}$ and only specified extensions [Res18, Sect. 4.2] are allowed, we can domain separate PSK-only with SHA384 as follows. To separate transcript hashes from component hashes (resp. HMAC queries) we need to be sure that the fixed region of the HMAC queries (i.e., the 80 byte range 49–128) overlaps with a segment of transcript in which we can say with certainty that it cannot contain the byte `0x5c` (when comparing to an outer HMAC query)

or byte 0x36 (when comparing to any of the other component hash subtypes). Now, let us demonstrate the domain separation by comparing a transcript hash to an outer HMAC query. The other types follow analogously. Recall that in the transcript hash, we have 73 bytes that have to be present and 103 bytes (to get to the input length of 176 bytes of an outer HMAC query) that can be chosen freely. The only fields in a (partial) `Client Hello` message that are of variable length are the cipher suites and the extensions field.[7] Now, there can essentially be two cases to yield the 176 bytes of `ClientHello`: (1) the cipher suites vector overlaps with the fixed region, or (2) the cipher suites vector does not overlap with the fixed region, but then the extensions vector has to overlap the fixed region. These two cases naturally arise from the structure of mandatory segments of a (partial) `ClientHello` message and that 103 bytes in the cipher suites and extensions field can be added in total to achieve a length of 176 bytes. That is, to achieve the desired length it is not possible that neither of the two do not overlap the fixed region of an outer HMAC query. Next, let us be more precise on how the above considered cases can actually occur. Recall that the fixed region is from bytes 49 to 128. Thus, if the cipher suites field goes through byte 49 cases (1) occurs and the cipher suites vector overlaps with the fixed region. If we now recall the structure of a `ClientHello`, we get that the cipher suites vector starts in byte 42 because it is prepended byte the 39-byte "fixed" prefix of the `ClientHello` and a 2 byte length field of the cipher suites vector. Now, every possible standardized cipher suite value is 2 bytes. That is, to have an overlap with the fixed region and the cipher suite vector overlaps byte 49, the vector has to contain at least 4 cipher suite values (i.e., 8 bytes). This byte 49 then allows us to separate. Namely, byte 49 in this case will be the second byte of a cipher suite value, which is one of the bytes `0x0i` for $\texttt{i} \in \{1, 2, 3, 4, 5\}$. In particular, we have under the assumption that only standardized cipher suites are used in the cipher suite vector that byte 49 if the cipher suite vector overlaps the fixed region of an outer HMAC query (i.e., bytes 49 to 128) that this byte always is different from 0x5c. The same argument applies, of course, to all the variants of the first HMAC query, since byte 49 also is always different from 0x36. The only thing that changes are the number of bytes that can be freely chosen. The remaining case to consider now is the case that the cipher suites vector does not overlap the fixed region. This implies that the cipher suite vector has to contain strictly less than 4 standardized cipher suite values. In this case, the cipher suite vector will end in byte 47 and 2 bytes of extension length will follow. That is, in this case byte 49 will be the second byte of the extensions length field. Unfortunately, this could take value 0x5c (or 0x36) so that is not the position that we can use to separate. Now, since at most 4 bytes of the bytes that can be freely chosen are occupied by the cipher suite vector (at most 6 byte cipher suite vector, where 2 are mandatory), the remaining (at least) 99 bytes (up to 103 bytes) have to be filled up with additional extensions. That is, in byte 50 the extensions field starts and in particular the first extension starts here. Recall that every extension starts with 2 bytes extension type and 2 bytes length field of

[7] Note that technically also the legacy session id and the legacy compression methods could be longer, but since we only negotiate TLS 1.3 in this work and do not consider compatibility to pre-TLS 1.3 versions these values are fixed as described above.

the extension followed by the actual extension data (cf. [Res18, Sect. 4.2]). Fortunately, all `ExtentionType` values defined for TLS 1.3 are different from `0x5c` (and `0x36`). In decimal, this would correspond to 92 (and 54), and there is no extension specified that corresponds to value 92.[8] In fact, all values are strictly smaller than 255 and therefore all values will start with a `0x00` byte. Hence, both byte 50 and 51 are different from `0x5c` (or `0x36`) with certainty under the assumption that only specified extensions are allowed in this case. That is, transcripts of length exactly 176 bytes will have at least two bytes different from from `0x5c` (or `0x36`) in the range of bytes 49–128 and therefore can be separated from HMAC queries by structure.

Overall, we have seen that for the PSK-only handshake configured with SHA384 transcript hash queries can be separated by structure under the assumption that the cipher suite vector only contains cipher suites value and the extension field only contains extensions that are standardized. In the previous paragraph, we have show that for all possible cases for filling up the 73 bytes of a minimal `ClientHello` to match the length of 176 bytes of an outer HMAC query, it has to hold for a transcript that there is at least one byte different from `0x5c` in the region of byte 49–128. This region is the fixed region of an outer HMAC query, where all bytes are set to `0x5c`. This argument easily can be adapted to the three other types of HMAC query (i.e., inner HMAC, `Derive-Secret`, and `Finished` key) by replacing `0x5c` by `0x36` in the analysis and adapting the filled up bytes such that the length of the transcript matches the respective input length of these types. Finally, we can define the domains $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$ as follows:

> For the PSK-only handshake protocol configured with SHA384, we define the domain $\mathcal{D}_{\mathsf{Th}}$ as the set containing the empty string and all string of $\{0,1\}^*$ with byte length of at least 73 such that there exists at least one byte in the range of byte 49 through 128 that is different from `0x5c` and `0x36`. All other strings of $\{0,1\}^*$ are contained in $\mathcal{D}_{\mathsf{Ch}}$.

PSK-(EC)DHE handshake with SHA384.

Finally, it remains to define domain $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$ for the PSK-(EC)DHE handshake configured with SHA384. As in the previous section, in which we analyzed the PSK-only handshake with SHA384, we have an overlap of the length range between transcript hashes and all component hash types as shown in Table 7.6. The fixed region for all the component hash queries is 80 bytes long and ranges from byte 49 through byte 128. Let us first have a look at the structure of a (partial) `ClientHello` in the PSK-(EC)DHE handshake. In comparison to the `ClientHello` in the PSK-only handshake, here $18 + |\mathbb{G}|/8$ additional bytes are mandatory to accommodate for the configured group and the corresponding key share. This results in the following illustration of a `ClientHello`:

$$39 \text{ bytes} \,\|\, 4 \text{ bytes} \,\|\, \leftrightarrow \,\|\, 4 \text{ bytes} \,\|\, \leftrightarrow \,\|\, (44 + |\mathbb{G}|/8) \text{ bytes}$$

where $\leftrightarrow$ indicate segments that can vary in length. Recall that a `ClientHello` has to end with the `pre_shared_key` extension and all other extensions might be reordered

[8] As a side remark, in DTLS [RM12] (not considered in this thesis), there is an extension with the value 54 specified (`connection_id` [RTFK22]).

Table 7.6: Minimal input length of the hash function calls made by TLS 1.3 in the PSK-(EC)DHE handshake with SHA384.

Query type	Min. length (in bytes)	Max. length (in bytes)
Outer HMAC	176	176
Inner HMAC	176	176
DH HMAC	$128 + \lvert\mathbb{G}\rvert/8$	$128 + \lvert\mathbb{G}\rvert/8$
`Derive-Secret`	189	199
`Finished` key	147	147
Empty transcript	0	0
Transcript	$91 + \lvert\mathbb{G}\rvert/8$	$45 + 2^{17}$

arbitrarily. The two segments that can vary in length are on the one hand the cipher suites vector and the extensions field can be extended by further extensions. We show that a transcript hash query cannot collide with any of the component hash queries by contradiction. To this end, we assume that there is a transcript that collides with a component hash query. As a reminder, we recap the structure of a component hash query:

$$48 \text{ bytes} \parallel (\texttt{pad})^{80} \parallel k \text{ bytes}$$

where `pad` is `0x5c` for outer HMAC queries and `0x36`, otherwise, and k is 48 bytes for both inner and outer HMAC queries, $|\mathbb{G}|/8$ bytes for DH HMAC queries, (at most) 71 bytes for `Derive-Secret` queries, and 19 bytes for `Finished` key queries. As illustrated above the mandatory extensions are $(44+|\mathbb{G}|/8)$ bytes, which is at least 76 bytes for the smallest choice of $\mathbb{G}$. That is, in the region after the 80 bytes of `pad` is not enough space to fit all mandatory extensions for *any* of the query types. Now, we have the case that either one of the mandatory extensions has to start in the fixed region or before the fixed region. Recall that all extensions start with 2 byte each of extension type and extension length. Since none of the standardized extensions of TLS 1.3, and thus in particular not the mandatory ones have an extension type of `0x5c5c` or `0x3636`, they cannot start in the fixed region as otherwise we would not have a collision. Thus, such an above collision is only possible if any mandatory extension starts outside of the fixed region of $(\texttt{pad})^{80}$ and its extension data contains an 80 byte `pad` vector. This generally would be possible as all mandatory extensions include a variable length vector. Next, let us have a look at the region before the fixed region. All `ClientHello` messages start with 39 bytes that always are ordered in the same way ending with the `legacy_session_id` that we already leveraged to domain separate the handshake modes with SHA256 before. These are followed by 2 bytes of cipher suite vector length and at least 2 bytes for a mandatory cipher suite value, and 4 bytes of the extensions field length. This already results in a *mandatory* prefix of 47 bytes. Now every extension starts with 2 byte of extension type and 2 byte of extension data length. At this point, the contradiction becomes very clear. The extension type field now overlaps the fixed region, as the second byte is byte 49, which is the first byte of the fixed region. Recall that none of the extension

types actually contains `0x5c` or `0x36`. Hence, byte 49 `cannot` be `pad`. Therefore, the mandatory extensions cannot start inside of the fixed region nor before it. Thus, it is impossible to fit 80 bytes of `pad` in a transcript and also find enough space for all mandatory extensions for any of the component hash queries. Let us be more precise about this. The mandatory extensions include $44 + |\mathbb{G}|/8$ bytes, the mandatory prefix is already 47 bytes. That is, there are only 1 byte before, and k bytes (depending on the query type) after the fixed region to fit in the mandatory extensions, when also wanting to fit in a fixed region of 80 bytes of `pad`. For $k = |\mathbb{G}|/8$, there are missing 43 bytes. For the other choices of k with $k \leq 71$, it even holds for the smallest group that $|\mathbb{G}|/8 = 32$. This means that there are at least 4 bytes missing to encode all mandatory extensions. Consequently, there has to be at least one byte in the range from 49 through 128 in a transcript that is different from `0x5c` or `0x36`.

Overall, we define the domains for PSK-(EC)DHE with SHA384 as follows:

> For the PSK-(EC)DHE handshake protocol configured with SHA384, we define the domain $\mathcal{D}_{\mathsf{Th}}$ as the set containing the empty string and all string of $\{0,1\}^*$ with byte length of at least $91 + |\mathbb{G}|/8$ such that there exists at least one byte in the range of byte 49 through 128 that is different from `0x5c` and `0x36`. All other strings of $\{0,1\}^*$ are contained in $\mathcal{D}_{\mathsf{Ch}}$.

Closing Remarks on Domain Separation for the PSK handshakes.

As discussed above we were able to define separate domains $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$ such that the TLS 1.3 PSK handshakes instantiated with fixed cryptographic parameters only queries its hash function (resp. the random oracle) on $\mathcal{D}_{\mathsf{Th}}$ when computing transcript hashes and on $\mathcal{D}_{\mathsf{Ch}}$ when computing HMAC (and related functions). The analysis presented here relies on the domain separation analysis first presented in [DDGJ22b, DDGJ22a]. In the originally published version, there are a couple of missing bytes (e.g., message type and message length) which we corrected in the presented version above.[9] Also, as these results presented originally domain separation for PSK-only with SHA384 was not possible. Due to the shift by the aforementioned bytes, we were able to domain separate the PSK-only handshake with SHA384 with an additional assumption. We assumed that the cipher suite vector and the extensions vector only contains values that are specified in the standard. Even though we believe that this assumption is not too strong despite it does not reflect the standard entirely. In our formal model, TLS 1.3 will only use mandatory extensions and only a single valid cipher suite value anyway. Thus, we never run into the problem that TLS 1.3 as we consider it formally could construct such a collision during execution. Hence, TLS 1.3 will never consider such artificial string as a transcript in our abstraction and therefore the domains are well-defined for our use. Nevertheless, we decided to keep this analysis as general as possible to give insides about the domain separation unintentionally employed by the TLS 1.3 standard.

[9] The authors of [DDGJ22b, DDGJ22a] would like to thank Robert Merget for spotting this and notifying us immediately.

7.6 Discussion

In this chapter, we presented a new abstraction of the TLS 1.3 key schedule in the ROM, which holds under the assumption that the TLS hash function is a random oracle. This abstraction allows it to consider the TLS key schedule and the TLS hash function as 12 independent random oracles and thus tames the complexity of further analyses in a modular fashion. In particular, one does not need to deal with the interleaved computations of key schedule in, for example, the subsequent key exchange proof. As we will see in Chapters 9 and 10 this abstraction can be leveraged for a tight security proof for the TLS 1.3 handshakes.

In an earlier version [DDGJ22b, DDGJ22a] of the results presented in this chapter that focused solely on the PSK handshakes of TLS 1.3 instead of all handshake modes, it was not possible to prove indifferentiability for the PSK-only handshake configured with SHA384. The reason is that it was not possible to define the *disjoint* working domains $\mathcal{D}_{\mathsf{Th}}$ and $\mathcal{D}_{\mathsf{Ch}}$ that were necessary for the first abstraction step (from one hash random oracle to a transcript hash and component hash random oracle) to hold. In this work, we fixed small inaccuracies with the formatting of the messages, which induced a shift of the bytes in the domain-separation argument (cf. Section 7.5). This in combination with an additional assumption allowed us to separate the domains for the PSK-only handshake with SHA384. In essence, this assumption restricts the server in the PSK-only mode with SHA384 to process `ClientHello` messages that contain unstandardized cipher suite values or unstandardized extensions. Note that this assumption is not necessary for the other handshake modes. Here, it is irrelevant if undefined values are contained in these messages as long as at least one standardized cipher suite value is present and all mandatory extensions for the mode are present. We would like to highlight, again, that we believe that this is not a strong assumption for our perspective. Honest clients would never add undefined values in our perspective on TLS, and rather only would configure the bare minimum. That is, these `ClientHello` messages can only be received by servers if an adversary tampered with it. However, the presence of this undefined values in our perspective would uncover the adversarially-chosen messages. Therefore, we consider it reasonable to consider only protocol messages as valid that can be output by honest sessions. In our perspective, we have a fixed cipher suite, a fixed hash function, a fixed signature algorithm, and a fixed group. Therefore, honest clients and servers only configure the values and these values alone. Nevertheless, we wanted to keep our analysis as general as possible to understand the unintentional domain separation and thus wanted to make as little assumptions as possible.

Open Questions. Even though, we have formal justification for our key-schedule abstraction and that is sufficient for our perspective on the TLS 1.3 protocol, we consider the following direction valuable for future work. First, we consider it an interesting open question for future work to investigate the possibility of dropping the assumption made in the domain-separation discussion for the PSK-only mode with SHA384. That is, it would be interesting whether domain separation for transcript and component hashes can be shown if one considers `ClientHello` messages that are allowed to contain undefined

values. Recall that the standard considers messages as valid as long as at least one recognized value in every vector is present and it mandates to simply ignore undefined values [Res18, Sect. 4.2.1]. This is particularly interesting in a formal model that allows for negotiation of the cipher suites, which we exclude. We highlight that our result still applies to a model in which honest clients present only standardized values for the cipher suites. However, as soon as it is allowed to add undefined values, our treatment needs to be revisited. This might be achieved through revisiting the domain-separation argument presented in Section 7.5, finding a different approach to formally justify the key schedule abstraction presented in Section 7.2 that avoids such a domain-separation argument, or even to revisit the domain separation for the uses of the hash function in future revisions of the standard.

Another interesting avenue is to extend our treatment by out-of-band PSKs. In this work, we excluded out-of-band PSK entirely, mostly because they are hard to capture formally. The length of the externally established PSKs might be application dependent, and even the provided entropy is unclear. Our treatment, especially in this chapter, highly relies on the length of the PSK being equal to the output length of the hash function. In future work, it would be interesting to investigate whether there is a way to incorporate out-of-band PSKs into the results presented here.

8

MODULARIZING HANDSHAKE ENCRYPTION

Author's contribution. The contents of this chapter are based on joint work with Hannah Davis, Felix Günther and Tibor Jager [DDGJ22b, DDGJ22a]. While we discussed all aspects of this paper together, the main idea to abstract the handshake encryption of the TLS 1.3 protocol (resp. the use of internal keys for MSKE protocols, in general) and the corresponding technical treatment is mainly due to Hannah Davis and Felix Günther. The author of this thesis extended the result by a theorem for the full handshake (Theorem 8.2), which is a *direct* implication from the considerations of [DDGJ22b, DDGJ22a]. The result, in particular, the transformation presented in Section 8.1 and the corresponding analysis (Theorem 8.1) are almost identical as presented in [DDGJ22b, DDGJ22a].

Contents

8.1 Introduction

In this chapter, we address that with respect to key indistinguishability the encryption used in the handshake protocol of TLS 1.3 does not contribute to security. That is, the handshake would still provide indistinguishable keys even without the handshake messages starting from the `EncryptedExtensions` message being encrypted under the handshake traffic key (cf. Chapter 6). The handshake encryption mainly is used for privacy to hide the identity of the client and server, e.g., contained in the certificates. We only focus on the key exchange security of the TLS 1.3 handshake and do not cover privacy. For information about the privacy of TLS 1.3, we refer to [Arf+19]. Since it does not contribute to the key exchange security, it is desirable to abstract the handshake encryption as it only increases complexity. In particular, recall that for MSKE-security (Chapter 5) we distinguish between "internal" keys that are potentially used inside the protocol and

"external" keys that may only be used outside of the MSKE protocol. The mentioned handshake traffic key of the TLS 1.3 handshake is an internal key due to the handshake encryption. Internal keys need to be handled with more care than external keys, because testing of these keys potentially could open a side channel for the adversary. Recalling the TLS 1.3 handshake encryption an adversary simply could get a real or random key (i.e., testing a key) and then try to decrypt an encrypted TLS message with this key. If it is successful, then distinguishing is easy since only real keys yield a meaningful decryption. Therefore, internal keys can only be tested by the adversary right after they have been accepted and before they have been used in the protocol. Once tested, the internal key gets replaced by the real or random test key to address the above mentioned side channel. On the contrary, external keys can be tested as usual. For more details, we refer to Chapter 5 and in particular to the definition of the oracle Test.

Therefore, removing the handshake encryption from the analyzed protocol and by this removing the necessity of handling internal keys tames the complexity of our analysis. To cover the TLS 1.3 handshake as it is defined, namely with handshake encryption, we formally show how to define a protocol that is identical to TLS 1.3, but does not use handshake encryption and all key derived during the protocol are considered to be external. As this result might also benefit other security analysis of MSKE protocols, we present this in a general way for any MSKE protocol. This result originally was presented in [DDGJ22b].

8.2 Handshake Encryption as a Modular Transformation

In this section, we define a transformation from a MSKE protocol that uses only external keys to a MSKE protocol that works the same, but where a subset of stage keys is used to transform messages before/after they are send/received. Here, one can think of this transformation, for example, as encryption and decryption using an internal key.

Transformation. Let $\mathsf{KE}_2 = (\mathsf{Gen}, \mathsf{Activate}, \mathsf{Run})$ be a MSKE protocol with no internal keys, i.e., $\mathsf{INT}[s] = \mathsf{false}$ for all stages s of KE_2. Define KE_1 parameterized by two (deterministic) algorithms $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$, and a set $\mathsf{KS}_{\mathsf{Transform}} \subseteq [\mathsf{STAGES}]$, where STAGES denotes the number of stages of KE_2, as follows. The algorithms $\mathsf{KE}_1.\mathsf{Gen}$ and $\mathsf{KE}_1.\mathsf{Activate}$ are identical to their counterparts of KE_2. Algorithm $\mathsf{KE}_1.\mathsf{Run}$ is different and its definition is shown in Figure 8.1. Intuitively, $\mathsf{KE}_1.\mathsf{Run}$ applies the transformation $\mathsf{Transform}_{\mathsf{Recv}}$ to its input (i.e., when receiving a message), then runs $\mathsf{KE}_2.\mathsf{Run}$ on the transformed input, and finally outputs the result of the transformation $\mathsf{Transform}_{\mathsf{Send}}$ applied to the output of $\mathsf{KE}_2.\mathsf{Run}$ (i.e., before sending a message). $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$ receives the list of all stage keys corresponding to the stages in $\mathsf{KS}_{\mathsf{Transform}}$ All the protocol-specific and session-specific properties for KE_1 and KE_2 are identical except for the vector of internal stages INT, which sets $\mathsf{INT}[s'] = \mathsf{true}$ for all $s' \in \mathsf{KS}_{\mathsf{Transform}}$.

Correctness. Clearly, there exist implementations of algorithms $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$ that do not yield the intuition described above. In particular, as defined

$\mathsf{KE}_1.\mathsf{Run}(v, \pi_u^i, m)$

1 : $\mathsf{keys} := (\pi_u^i.skey[s])_{s \in \mathsf{KS}_{\mathsf{Transform}}}$
2 : $\mathsf{acc} := (\pi_u^i.\mathsf{accepted}[s] \neq \infty)_{s \in \mathsf{KS}_{\mathsf{Transform}}}$
3 : $\tilde{m} := \mathsf{Transform}_{\mathsf{Recv}}(\mathsf{keys}, \pi_u^i.role, \mathsf{acc}, m)$
4 : $(\pi_u^i, \tilde{M}) := \mathsf{KE}_2.\mathsf{Run}(v, \tilde{m})$
5 : // Run might change the accepted keys/stages
6 : $\mathsf{keys} := (\pi_u^i.skey[s])_{s \in \mathsf{KS}_{\mathsf{Transform}}}$
7 : $\mathsf{acc} := (\pi_u^i.\mathsf{accepted}[s] \neq \infty)_{s \in \mathsf{KS}_{\mathsf{Transform}}}$
8 : $M := \mathsf{Transform}_{\mathsf{Send}}(\mathsf{keys}, \pi_u^i.role, \mathsf{acc}, \tilde{M})$
9 : **return** (π_u^i, M)

Figure 8.1: Run procedure of the transformed key exchange protocol KE_1. Depending of the MSKE variant, $v = (\mathsf{pkeys}, u, sk_u)$ for pMSKE and $v = (u, psk)$ for sMSKE.

above $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$ might be defined in such a way that they alter underlying protocol messages so that keys that are accepted without the transformation are no longer accepted with the transformation. Therefore, we need to define correctness for the transformation to ensure that it is meaningful. We require for correctness that if two sessions honestly run protocol KE_2 without presence of an adversary will accept a key for stage s with probability p, then two sessions honestly running protocol KE_1 without presence of an adversary will accept a key for stage s with probability p, as well. Because of the absence of the adversary, it is ensured that all messages are sent honestly and the only changes made are the ones induced by $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$.

For TLS 1.3, one can think of $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$ as the encryption and decryption algorithm of AEAD scheme configured for the handshake. These algorithms are perfectly correct, and hence correctness of $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$ follows.

Security. Our goal is that KE_1 is a secure MSKE protocol if KE_2 is one. This should hold independently of the transformation algorithms.

The following theorem was originally proven for sMSKE in [DDGJ22b]. We state a more general version including pMSKE protocols in this work, and recap the proof.

Theorem 8.1. *Let* KE_2 *be a MSKE protocol with* STAGES *stages and* $\mathsf{KE}_2.\mathsf{INT}[s] = \mathsf{false}$ *for all stages. Let* $\mathcal{M}$ *be a session matching algorithm for* KE_2 *(Definition 5.4). Further, let* $\mathsf{Transform}_{\mathsf{Send}}$ *and* $\mathsf{Transform}_{\mathsf{Recv}}$ *be transformation algorithms and* $\mathsf{KS}_{\mathsf{Transform}} \subseteq [\mathsf{STAGES}]$. *Define* KE_1 *exactly as* KE_2 *but with* $\mathsf{KE}_1.\mathsf{Run}$ *defined as in Figure 8.1 and* $\mathsf{KE}_1.\mathsf{INT}[s] = \mathsf{true}$ *for all* $s \in \mathsf{KS}_{\mathsf{Transform}}$.

For any adversary $\mathcal{A}$ *against the MSKE security of* KE_1*, we can construct an adversary* $\mathcal{B}$ *such that*

$$\mathrm{Adv}^{\Pi}_{\mathsf{KE}_1}(\mathcal{A}) \leq \mathrm{Adv}^{\Pi}_{\mathsf{KE}_2}(\mathcal{B})$$

where $\Pi \in \{\mathsf{pMSKE}, \mathsf{sMSKE}\}$.

Proof. To prove this theorem, we give a construction of adversary $\mathcal{B}$ using $\mathcal{A}$ as a subroutine and relate the MSKE advantage of the two algorithms afterwards.

Construction of adversary B**.** Adversary $\mathcal{B}$ simulates the MSKE experiment for $\mathcal{A}$ and forwards every query except Send queries to its own corresponding oracle. It maintains its own counter time and increments it upon every query. Further, it maintains for each session π_u^i a list keys_u^i and a list $\mathsf{acc}_u^i[s]$ which is set to false for every $s \in \mathsf{KS}_{\mathsf{Transform}}$. To simulate $\mathsf{Send}(u, i, m)$ for $\mathcal{A}$, adversary $\mathcal{B}$ first checks for which $s \in \mathsf{KS}_{\mathsf{Transform}}$ with $\mathsf{acc}_u^i[s]$ it holds $\pi_u^i.\mathsf{accepted}[s] \neq \infty$. Then, for each of these stages s satisfying the condition, it checks whether π_u^i has been tested or revealed in stage s (i.e., if $\pi_u^i.\mathsf{tested}[s]$ or $\pi_u^i.\mathsf{revealed}[s]$). Additionally, it runs the session matching algorithm $\mathcal{M}$ for stage s to determine whether π_u^i has a partnered session, and repeats the two previous checks for the partner. If any of the conditions above holds, then adversary $\mathcal{B}$ can determine the session key $\pi_u^i.skey[s]$. Note that all stages $s \in \mathsf{KS}_{\mathsf{Transform}}$ are considered internal from $\mathcal{A}$'s view. That is, if an internal stage s key is tested it will be overwritten by the key output by Test. This needs to be handled by $\mathcal{B}$ during the simulation of the experiment for $\mathcal{A}$, and hence it knows $\pi_u^i.skey[s]$. If the conditions are not true, then $\mathcal{B}$ queries an additional $\mathsf{RevSessionKey}(u, i, s)$ query to obtain the key. In either case it adds $\pi_u^i.skey[s]$ to keys_u^i, sets $\mathsf{acc}_u^i[s] := \mathsf{true}$, and computes

$$\tilde{m} := \mathsf{Transform}_{\mathsf{Recv}}(\mathsf{keys}, \pi_u^i.role, \mathsf{acc}, m).$$

Then, it queries $\mathsf{Send}(u, i, \tilde{m})$ to its own oracle, retrieves the response $\tilde{M}$, and returns

$$M := \mathsf{Transform}_{\mathsf{Send}}(\mathsf{keys}, \pi_u^i.role, \mathsf{acc}, \tilde{M})$$

to adversary $\mathcal{A}$.

Analysis. Note that KE_1 and KE_2 are identical except for the transformation applied to the exchanged messages and that the stages in $\mathsf{KS}_{\mathsf{Transform}}$ are internal in KE_1. This is exactly what $\mathcal{B}$ takes into account. Whenever adversary $\mathcal{A}$ sends a message, adversary $\mathcal{B}$ determines the key(s) that potentially could be used in the transformation and then applies the transformation. Thus, it is easy to verify that adversary $\mathcal{B}$ perfectly simulates $\mathrm{Adv}^{\Pi}_{\mathsf{KE}_1}(\mathcal{A})$. Therefore, it only remains to argue that if $\mathcal{A}$ wins the MSKE experiment, $\mathcal{B}$ also wins its experiment. Recall from the definition of the MSKE experiment and winning the experiment can happen in three ways: (1) violating the Sound predicate, (2) violating the ExplAuth predicate, or (3) satisfying the Fresh predicate and guessing the bit b correctly. We can handle the first two cases together. Note that the conditions to check violation of either the Sound or ExplAuth predicate depends only on variables maintained by the MSKE security experiment. That is, if $\mathcal{A}$ wins by violating Sound or ExplAuth in the experiment simulated by $\mathcal{B}$, then Sound or ExplAuth also have to be violated in the MSKE experiment for KE_2 adversary $\mathcal{B}$ is running in. Thus, $\mathcal{B}$ would also win.

In the third case, $\mathcal{A}$ wins by guessing the bit b correctly if predicate Fresh is not violated. First of all, the bit b is determined by the MSKE experiment $\mathcal{B}$ is running in,

so if $\mathcal{A}$ wins, then inherently $\mathcal{B}$ wins as well. However, recall that $\mathcal{B}$ sometimes need to query RevSessionKey (at most q_{Send} times). Since one of the conditions of the Fresh predicate is that no session is tested and revealed at the same time, it might happen that even though $\mathcal{A}$ does not violate Fresh, but $\mathcal{B}$ does. Now, in the construction of $\mathcal{B}$ we make sure that we only reveal a session key if it has not been tested before. Therefore, the critical case is when $\mathcal{A}$ later tests the keys revealed by $\mathcal{B}$. However, note that all of these keys are in $\mathsf{KS}_{\mathsf{Transform}}$ and therefore considered internal in $\mathcal{A}$'s protocol KE_1. These keys are only allowed to be tested right after acceptance and not if the protocol execution was already continued, where this key might already been used. Adversary $\mathcal{B}$ now will only reveal keys (other than the ones queried by $\mathcal{A}$) if the respective stage has already been accepted and the protocol moved on. This is because there already has been a Send query, where the state of this stage was already set to accepted, and thus $\mathcal{A}$ cannot make any later Test query to that session and stage. The same arguments hold for any partnered session. Finally, the Fresh predicate can be violated by breaking the conditions of forward secrecy. These conditions are defined over the contributive identifiers of sessions and the timings of corruption. The time of corruption is maintained by the security experiment and is not influenced by $\mathcal{B}$. The contributive identifiers are defined identically for KE_1 and KE_2. This means that if two session are contributively partnered in the simulated experiment for KE_1, they also are contributively partnered in the experiment that $\mathcal{B}$ runs in for KE_2.

Hence, if $\mathcal{A}$ wins, then also $\mathcal{B}$ wins, which implies the theorem. □

For the TLS 1.3 handshake, Theorem 8.1 implies that transforming the handshake messages using the AEAD algorithms does not impact the MSKE security of the handshake protocol. Therefore, it is sufficient to analyze the security of the simpler protocol not using handshake encryption and only uses external keys. For the full handshake and the PSK handshakes, we then get the following two results, where the latter was already given in [DDGJ22b].

Theorem 8.2. *Let* KE_1 *be the TLS 1.3 full handshake protocol with handshake encryption (i.e., stages* 1 *and* 2 *are internal) defined in the left-hand side of Figure 7.2. Let* KE_2 *be the same protocol, but without handshake and encryption and all stages being external. Further, let* $\mathsf{Transform}_{\mathsf{Send}}$ *and* $\mathsf{Transform}_{\mathsf{Recv}}$ *be the encryption and decryption algorithm that is configured for the considered instance of the protocol, and let* $\mathsf{KS}_{\mathsf{Transform}} = \{1, 2\}$. *Then, for any adversary* $\mathcal{A}$ *we can construct an adversary* $\mathcal{B}$ *such that*

$$\mathrm{Adv}^{\mathsf{pMSKE}}_{\mathsf{KE}_1}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{pMSKE}}_{\mathsf{KE}_2}(\mathcal{B}).$$

Theorem 8.3. *Let* KE_1 *be either the TLS 1.3 PSK-only or PSK-(EC)DHE handshake protocol with handshake encryption (i.e., stages* 3 *and* 4 *are internal) defined in the right-hand side of Figure 7.2. Let* KE_2 *be the same protocol, but without handshake and encryption and all stages being external. Further, let* $\mathsf{Transform}_{\mathsf{Send}}$ *and* $\mathsf{Transform}_{\mathsf{Recv}}$ *be the encryption and decryption algorithm that is configured for the considered instance of the protocol, and let* $\mathsf{KS}_{\mathsf{Transform}} = \{3, 4\}$. *Then, for any adversary* $\mathcal{A}$ *we can construct an adversary* $\mathcal{B}$ *such that*

$$\mathrm{Adv}^{\mathsf{sMSKE}}_{\mathsf{KE}_1}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{sMSKE}}_{\mathsf{KE}_2}(\mathcal{B}).$$

Theorems 8.2 and 8.3 are given for the *abstracted* handshake given in Figure 7.2. Note that we could also have stated it for the "regular version" of the handshakes as presented in Chapter 6. However, we abstracted the "regular" handshake in Chapter 7 and in this chapter to prepare for the security proofs given in Chapters 9 and 10. The abstraction of the handshake encryption completes the preparation of the handshake protocol for the security proofs, and the MSKE security of the protocol(s) KE_2 given in Theorems 8.2 and 8.3 will be subject of the subsequent chapters.

9

TIGHT SECURITY OF THE TLS FULL HANDSHAKE

Author's contribution. The contents of this chapter are a new contribution added in this thesis. The proof presented in Section 9.3 was solely developed by the author of this thesis. The proof technique is based on a number of previous works. The general idea, and particularly, leveraging the combination of the strong Diffie–Hellman (SDH) assumption and a key derivation function that is modeled as a random oracle to prove tight security of Diffie–Hellman (DH) based key exchange protocols is due to Cohn-Gordon et al. [Coh+19]. In early work [DJ21], which was joint work with Tibor Jager, we proved a tight security bound for the MSKE-security of the full TLS 1.3 handshake under strong assumptions about the key schedule. The author of this thesis developed the architecture of the aforementioned analysis that implicitly also finds application in the analysis of Section 9.3. In concurrent and independent work [DG21a], Hannah Davis and Felix Günther, also gave a tight security bound for TLS 1.3 full handshake in a weaker security model with only a single stage and also under similarly strong assumptions about the key schedule. The results of this chapter were also inspired by their work. Lastly, the proof presented in Section 9.3, follows the example of the tight security proof for the PSK handshakes presented in Chapter 10, which originated from joint work with Hannah Davis, Felix Günther, and Tibor Jager [DDGJ22b, DDGJ22a]. The proof of Section 9.3 uses the key schedule abstraction introduced in Chapter 7, which also originated from this joint work, and can be seen as an extension to the full handshake.

Contents

9.1 Introduction

In this chapter, we present our tight security bound for the TLS 1.3 full (EC)DHE handshake protocol. We begin by briefly outlining our approach. In this context, we discuss how we circumvent the difficulties of tightly-secure key exchange discussed in Section 1.5. To this end, we first sketch our high-level approach to prove tight security, then we recall the commitment problem and how to avoid it. Finally, we outline how we can leverage the techniques to avoid the commitment problem for a tight security proof for the TLS 1.3 handshake protocol.

9.1.1 Technical Approach

The overall high-level idea to obtain our tight security bound for TLS 1.3 full handshake is to take advantage of the abstraction for the TLS 1.3 key schedule (Chapter 7) and the abstraction of the handshake encryption (Chapter 8) to reduce the overall complexity of the handshake protocol. These abstractions allow us to focus on giving a tight security bound for the abstracted TLS 1.3 full handshake as shown on left-hand side of Figure 7.2 without handshake encryption. With a tight bound for the abstracted TLS 1.3 full handshake, we can use Theorems 7.3 and 8.2 to conclude tight security for the TLS 1.3 full handshake without any abstractions.

The commitment problem and how to avoid it. To prove a tight bound for the abstracted TLS 1.3 full handshake, we first need to solve the main challenge to prove tight security, in particular of Diffie–Hellman-based, key exchange protocols, namely the *commitment problem* outlined in Section 1.5. In the following, we sketch how to approach a tight security proof for a Diffie–Hellman-based key exchange straightforwardly and recap the core of the commitment problem. Then, we address how to solve the commitment problem technically to yield a tight key exchange proof.

To reduce the security of a key exchange protocol to a Diffie–Hellman problem, one usually leverages the random self-reducibility of this class of problems. This allows a reduction to create arbitrary many Diffie–Hellman challenges from a single one that all look like fresh challenges. Recall that the straightforward approach for a Diffie–Hellman-based key exchange proof is to guess a "tested" session and its partner that remain uncompromised until the end. In these two sessions the reduction can then embed its challenge. This straightforward idea, unfortunately, induces a quadric loss in the number of sessions. Using the random self-reducibility enables to evade such guessing, because the reduction can just embed a challenge in basically every session. However, with this approach new challenges evolve. Namely, by embeding a Diffie–Hellman challenge in every session, the reduction *commits* to not knowing the secret Diffie–Hellman value for any session. Now, a simple attack is possible that breaks the reduction: Assume that we embed in any session a rerandomized Diffie–Hellman challenge and for illustration think of the classical Diffie–Hellman key exchange (Section 3.2). Now, if an initiator outputs such a value, the adversary could simply answer with a tampered Diffie–Hellman value, for which the reduction does not know the corresponding secret value. The adversary

now simply breaks the reduction by asking the initiator to reveal its key, which is a common feature in key exchange models (cf. Chapter 5). The underlying problem here is that the initiator committed on not knowing the secret Diffie–Hellman value for the value that it output. Since the intiator cannot decide at the time it chooses its Diffie–Hellman value, whether it will receive an honest or a dishonest Diffie–Hellman value, this seems unavoidable.

Gjøsteen and Jager [GJ18] solved this by introducing an additional first message by the intiator that can be seen as a reprogrammable commitment. In this approach, the initiator can basically revoke to embed a rerandomized challenge by reprogramming the first message and thus choose a fresh Diffie–Hellman value with a corresponding secret to be able to compute a key. Unfortunately, this technique only applies to new protocols that particularly aim for tight security. In existing (real-world) protocols like TLS 1.3 we cannot add additional messages to prove tight security. Cohn-Gordon et al. [Coh+19] proposed a technique that is applicable for a large class of protocols. They propose to switch the underlying computational problem from the commonly used DDH (Definition 3.4) problem to the SDH (Definition 3.5) problem. This in combination with a key derivation function that is modeled as a random oracle can be leveraged to achieve a tight security proof for a large class of Diffie–Hellman based key exchange protocols. The key derivation function needs to receives the Diffie–Hellman key, and ideally also, the Diffie–Hellman values of both the initiator and responder computing the respective key as input for this technique to work. The idea is then the following. Intuitively, the challenge for SDH is solve to CDH (Section 3.2) with the help of a DDH oracle, where the first component is fixed. This evades the attack described above as follows. If the intiator receives a Diffie–Hellman value for which it is not able to compute a key, we basically have two cases. Either the adversary has already queried the random oracle for the respective Diffie–Hellman key or not. If it queried the random oracle, we are able to use the DDH oracle and the context, i.e., the Diffie–Hellman values output and received by the intiator, to recognize the Diffie–Hellman key the intiator is not able to compute, and set the key accordingly, not breaking the reduction. This makes use of the reduction having full control over the random oracle. If there has been no query to the random oracle, the adversary also does not know the key, so it is consistent for the initiator to simply guess it. We need to be careful, and reprogram the random oracle later if necessary to ensure consistency. Morever, we can use a similar technique to argue that the adversary in honest sessions where it did not interfere, is only able to get any information about these honest keys, if it is able to solve SDH for the reduction. Namely, it would need to query the random oracle with a Diffie–Hellman key by only observing the (rerandomized) Diffie–Hellman values (cf. CDH Section 3.2). We can again recognize such random oracle queries by using the DDH oracle and simple algebra. For responder sessions this works analogously. Overall, with this technique we are able to circumvent the commitment problem efficiently and prove tight security.

Avoiding the commitment problem in TLS 1.3. Fortunately, the TLS 1.3 full handshake is at its core only a Diffie–Hellman key exchange. Thus, we are able to adopt

the technique by Cohn-Gordon et al. [Coh+19] to the full handshake of TLS 1.3. The main challenge for this adoption is that in TLS 1.3 the key schedule does not include the context of the used Diffie–Hellman key into the same function call as the corresponding Diffie–Hellman key that than could serve as the random oracle outlined above. In particular, recall that in the TLS 1.3 key schedule (Figure 6.1) the Diffie–Hellman key Z is used to derive the handshake secret $hs := \mathbf{Extract}(salt_{hs}, Z)$, where $salt_{hs}$ is a constant in the full handshake. This handshake secret hs then is used in further derivation to derive the handshake traffic secrets and the master secret, among others. Only in these further derivation the Diffie–Hellman key Z is put into context by adding a context hash that contains the Diffie–Hellman key shares. That is, as TLS 1.3 is standardized the technique by Cohn-Gordon et al. [Coh+19] is not directly applicable. In early work, Diemert and Jager [DJ21] abstracted the key schedule of TLS 1.3 as multiple random oracles by summarizing certain steps. Here, they introduce four random oracles such that one derives the handshake traffic secrets, one the application traffic secrets, one the exporter master secret and one the resumption master secret. By doing so, these four key derivations could be expressed as a function of the Diffie–Hellman key Z and the context for the respective secrets containing the Diffie–Hellman values. This ultimately allowed to apply the technique to TLS 1.3. In independent and concurrent work, Davis and Günther [DG21a] modeled the subroutines HKDF.Extract and HKDF.Expand as independent random oracles, and applied a careful bookkeeping technique to overcome the above problem and also were able to apply the technique by Cohn-Gordon et al. [Coh+19]. Unfortunately, both of these techniques were not ideal, because they do not provide a formal justification of this abstraction as already discussed in Chapter 7. Our abstraction of the TLS 1.3 key schedule presented in Chapter 7 and originally published in [DDGJ22b] closes this gap and provides a formally justified abstraction that allows for the Cohn-Gordon et al. [Coh+19] technique to be applied in a natural way.

Chapter outline. To this end, we first present in Section 9.2 the protocol-specific properties of the TLS 1.3 full handshake such that it makes up a public-key MSKE protocol as defined in Chapter 5. Then, we prove in Section 9.3 the tight security bound for the TLS 1.3 full handshake protocol in its abstracted form as obtained by the insights presented in Chapters 7 and 8, i.e., we prove the security bound for the protocol as presented on the left-hand side of Figure 7.2 without handshake encryption. Finally, we apply the main results of the key schedule abstraction (Theorem 7.3 of Chapter 7) and modularization of the handshake encryption (Theorem 8.2 of Chapter 8) to obtain the final bound for the TLS 1.3 full handshake protocol as described in Section 6.4.

9.2 TLS 1.3 Full (EC)DHE Handshake as an MSKE Protocol

We begin by capturing the TLS 1.3 full (EC)DHE handshake protocol, specified in Figures 6.1 and 7.2, formally as an pMSKE protocol. To this end, we must explicitly define the variables discussed in Chapter 5. In particular, we have to define the stages themselves,

which stages are internal and which replayable, the session and contributive identifiers, when stages receive authentication, and when stages become forward secret.

Stages. The TLS 1.3 (EC)DHE handshake protocol has six stages (i.e., $\mathsf{STAGES} = 6$), corresponding to the keys htk_C, htk_S, ats_C, ats_S, ems, and rms in that order. The set INT of internal keys contains htk_C and htk_S, the handshake traffic encryption keys. None of the stages are replayable, i.e., for all $s \in [\mathsf{STAGES}]$ it holds that $\mathsf{REPLAY}[s] = \mathsf{false}$.

Session and contributive identifiers. The session and contributive identifiers for stage s are tuples $(label_s, ctxt)$, where $label_s$ is a unique label identifying stage s, and $ctxt$ is the transcript that enters the key derivation. The session identifiers $(sid[s])_{s\in\{1,\ldots,6\}}$ are defined as follows:[1]

$$
\begin{aligned}
sid[1] &= (\text{“}htk_C\text{”}, (\texttt{CH}, \texttt{CKS}, \texttt{SH}, \texttt{SKS}))\,,\\
sid[2] &= (\text{“}htk_S\text{”}, (\texttt{CH}, \texttt{CKS}, \texttt{SH}, \texttt{SKS}))\,,\\
sid[3] &= (\text{“}ats_C\text{”}, (\texttt{CH}, \texttt{CKS}, \texttt{SH}, \texttt{SKS}, \texttt{EE}, \texttt{CR}^*, \texttt{SCRT}, \texttt{SCV}, \texttt{SF}))\,,\\
sid[4] &= (\text{“}ats_S\text{”}, (\texttt{CH}, \texttt{CKS}, \texttt{SH}, \texttt{SKS}, \texttt{EE}, \texttt{CR}^*, \texttt{SCRT}, \texttt{SCV}, \texttt{SF}))\,,\\
sid[5] &= (\text{“}ems\text{”}, (\texttt{CH}, \texttt{CKS}, \texttt{SH}, \texttt{SKS}, \texttt{EE}, \texttt{CR}^*, \texttt{SCRT}, \texttt{SCV}, \texttt{SF}))\,, \quad \text{and}\\
sid[6] &= (\text{“}rms\text{”}, (\texttt{CH}, \texttt{CKS}, \texttt{SH}, \texttt{SKS}, \texttt{EE}, \texttt{CR}^*, \texttt{SCRT}, \texttt{SCV}, \texttt{SF}, \texttt{CCRT}^*, \texttt{CCV}^*, \texttt{CF}))\,.
\end{aligned}
$$

To make sure that a server that received untampered `ClientHello` and `ClientKeyShare` can be tested in (the unauthenticated) stages 1 and 2, even if the sending client did not receive the server's answer, we set the contributive identifiers of stages 1 and 2 such that cid_{role} reflects the messages that a session in role $role$ must have honestly received for testing to be allowed. Namely, we let clients (resp. servers) upon sending (resp. receiving) the messages $(\texttt{CH}, \texttt{CKS})$ set

$$cid_{\mathsf{responder}}[1] = (\text{“}htk_C\text{”}, (\texttt{CH}, \texttt{CKS})) \quad \text{and} \quad cid_{\mathsf{responder}}[2] = (\text{“}htk_S\text{”}, (\texttt{CH}, \texttt{CKS}))\,.$$

Further, when the client receives (resp. the server sends) the message $(\texttt{SH}, \texttt{SKS})$, they set

$$cid_{\mathsf{initiator}}[1] = sid[1] \quad \text{and} \quad cid_{\mathsf{initiator}}[2] = sid[2].$$

For all other stages $s \in \{3, 4, 5, 6\}$, $cid_{\mathsf{initiator}}[s] = cid_{\mathsf{responder}}[s] = sid[s]$ is set upon acceptance of the respective stage (i.e., when $sid[s]$ is set as well).

Authentication. Note that in the TLS 1.3 full handshake the server always authenticates via a certificate and it allows for optional client authentication via a certificate. The stage 1 and 2 keys htk_C and htk_S are initially unauthenticated. Upon acceptance of stage 3, they become unilaterally authenticated by verifying the server signature from the `ServerCertificateVerify` message. If client authentication is desired, then htk_C and htk_S become mutually authenticated upon acceptance of stage 6 after the client signature from the `ClientCertificateVerify` successfully was verified. All other keys,

[1] Components marked with * are only present if the client authenticates.

i.e., ats_C, ats_S, *ems*, *rms*, become unilaterally authenticated upon acceptance of their respective stage, and in case of client authenticated being executed, they become mutually authenticated upon acceptance of stage 6.

For explicit authentication, initiators receive explicit authentication always upon acceptance of stage 3 after successfully verifying the `ServerFinished` message. Even though stages $s > 3$ of initiators actually receive explicit authentication upon acceptance, they receive it because of acceptance of stage 3. Responders only receive explicit authentication if client authentication is done, and therefore receive explicit authentication upon acceptance of stage 6 after after successfully verifying the `ClientFinished` message. That is, $\mathsf{EAUTH}[\mathsf{initiator}, s] = 3$ for all stages and only if client authentication is done then $\mathsf{EAUTH}[\mathsf{responder}, s] = 6$ for all stages.

Formally, we define the following set AUTH:

$$\{((3, m, 3, m), (3, m, 3, m), (3, m, 3, m), (4, m, 3, m), (5, m, 3, m), (6, m, 3, m)) \mid m \in \{6, \infty\}\}$$

Forward secrecy. In the full (EC)DHE handshake protocol, all keys are forward secret upon acceptance because of the ephemeral Diffie–Hellman key that basically is the root of the key derivation. Therefore, it holds $\mathsf{FS}[r, s, lvl] = 1$ for all $r \in \{\mathsf{initiator}, \mathsf{responder}\}$, all stages $s \in [\mathsf{STAGES}]$ and all $lvl \in \{\mathsf{wfs2}, \mathsf{fs}\}$.

9.3 Tight Security of the TLS 1.3 Full (EC)DHE Handshake

We start this section by proving a tight security bound for the abstracted TLS 1.3 full handshake. With this bound, we successively apply the results from Chapters 7 and 8 to obtain a bound for the TLS 1.3 handshake protocol as specified in Figure 6.1.

9.3.1 Tight Security Bound for the Abstracted Full Handshake

In the first step, we prove the following theorem.

Theorem 9.1. *Let TLS-DHE be the (abstracted) TLS 1.3 full (EC)DHE handshake protocol as specified on the left-hand side of Figure 7.2 without handshake encryption (i.e.,* $\mathsf{INT}[s] = \mathsf{false}$ *for all stages s). Let* $\mathbb{G}$ *be a standardized group with prime-order p. Let* Sig *be a standardized signature scheme. Let* $\lambda \in \mathbb{N}$ *be the output length in bits of* **H**. *Further, let* **H** *and* TKDF_x *for each* $x \in \{htk_C, fin_C, htk_S, fin_S, ats_C, ats_S, ems, rms\}$ *be modeled as* 9 *independent random oracles* $\mathsf{RO}_{\mathsf{Th}}, \mathsf{RO}_{htk_C}, \ldots, \mathsf{RO}_{rms}$. *Then, for any adversary* $\mathcal{A}$, *we can construct adversaries* $\mathcal{B}$ *and* $\mathcal{F}$ *such that*

$$\mathrm{Adv}^{\mathsf{pMSKE}}_{\text{TLS-DHE}}(\mathcal{A}) \leq \frac{2q^2_{\mathsf{Send}}}{2^{256} \cdot p} + \frac{(q_{\mathsf{RO}} + 7q_{\mathsf{Send}})^2}{2^{\lambda}} + \mathrm{Adv}^{\mathsf{SDH}}_{\mathbb{G},p}(\mathcal{B}) + \mathrm{Adv}^{\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathsf{corr}}}_{\mathsf{Sig}}(\mathcal{F}) + \frac{q_{\mathsf{Send}}}{2^{\lambda}}.$$

where q_{Send} *is the number of* Send *and* q_{RO} *the (total) number of random oracle queries issued by adversary* $\mathcal{A}$, *respectively.*

Proof. To prove the bound, we make an incremental series of changes [Sho04] to the public-key MSKE experiment $\mathrm{Exp}^{\mathsf{pMSKE}}_{\mathtt{TLS-DHE}}(\mathcal{A})$ defined in Figure 5.9. Let Game_δ denote the event that $\mathcal{A}$ in Game δ wins the game.

GAME 0. The initial game Game 0 is the public-key MSKE game $\mathrm{Exp}^{\mathsf{pMSKE}}_{\mathtt{TLS-DHE}}(\mathcal{A})$. Since the hash function $\mathbf{H}$ and the functions TKDF_x are modeled as a random oracle, we assume that each of these random oracles RO_x is implemented using a look-up table ROList_x.[2] By definition, we have $\Pr[\mathsf{Game}_0] := \Pr[\mathrm{Exp}^{\mathsf{pMSKE}}_{\mathtt{TLS-DHE}}(\mathcal{A}) = 1]$.

GAME 1. In Game 1, we make sure that collisions among the random nonces and group elements computed by honest sessions are excluded. To that end, we introduce two flags bad_C and $\mathsf{bad}_{C'}$, and abort the game if either of the two is set. These flags are set if the following conditions are satisfied:

- bad_C is set when honest sessions choose the same nonce (i.e., r_C or r_S and group element (i.e., X or Y) in their `Hello` message, and
- $\mathsf{bad}_{C'}$ is set when an honest responder samples some nonce and group element that have already been received by another session. This case reflects a collision with a nonce and group element chosen by the adversary.

Then, it holds that

$$\Pr[\mathsf{Game}_0] \leq \Pr[\mathsf{Game}_1] + \Pr[\mathsf{bad}_C] + \Pr[\mathsf{bad}_{C'}].$$

Let us analyze the probability that bad_C and $\mathsf{bad}_{C'}$ are set in Game 1. First, note that in each Send query at most one session will choose a nonce $r \xleftarrow{\$} \{0,1\}^{256}$ and a group element $G \xleftarrow{\$} \mathbb{G}$. If bad_C is set, then there is a Send query such that a session π^i_u is activated via $\mathsf{Activate}$ and this session samples (r, G) which were previously sampled by another session $\pi^{i'}_{u'}$. Thus, the probability for bad_C to be set is the probability that a collision among the (up to) q_{Send} pairs (r, G) occurs. This is bounded from above by the birthday bound, i.e.,

$$\Pr[\mathsf{bad}_C] \leq \frac{q^2_{\mathsf{Send}}}{2^{256} \cdot p}$$

where q_{Send} denotes the number of queries issued to Send by $\mathcal{A}$.

Next, if $\mathsf{bad}_{C'}$ is set, then there is a Send query that creates a new responder session π^j_v. This session samples a nonce $r_S \xleftarrow{\$} \{0,1\}^{256}$ and a group element $Y \xleftarrow{\$} \mathbb{G}$, which were already received by another session π^i_u. There are at most q_{Send} many sessions, so there are no more than q_{Send} pairs received by sessions that could collide with (r_C, Y). By the union bound, we get that the probability that the nonce and group element sampled by π^j_v collide is bounded from above by $q_{\mathsf{Send}}/(2^{256} \cdot p)$. Overall, we get, again by the union bound, that there is a collision for any π^j_v with probability

$$\Pr[\mathsf{bad}_O] \leq q_{\mathsf{Send}} \cdot \frac{q_{\mathsf{Send}}}{2^{256} \cdot p} = \frac{q^2_{\mathsf{Send}}}{2^{256} \cdot p}.$$

[2] For convenience, we assume that each of these look-up tables is implemented using a data structure that allows for constant time access when indexed either by inputs or outputs. An example for this data structure would be a hash table.

Hence, we have that

$$\Pr[\mathsf{Game}_0] \leq \Pr[\mathsf{Game}_1] + \frac{2q_{\mathsf{Send}}^2}{2^{256} \cdot p}.$$

Predicate Sound **cannot be violated.** At this point, we argue that in Game 1 and all subsequent games, adversary $\mathcal{A}$ cannot violate the Sound predicate without the game being aborted. If any of the conditions of the Sound predicate (defined in Figure 5.6) is satisfied, either of the flags introduced above will also be set and the game will be aborted. Recall that Sound includes 7 events that cause it to output false. We argue that none of these events can occur in Game 1 and thus Sound = true holds from Game 1 on.

1. *There are two sessions partnered in some stage that do not hold the same key in that stage.*

   First, recall that according to Definition 5.1, it holds that two sessions are partnered in a stage if and only if they agree on the session identifier in that stage. For stage 1, we have that $sid[1]$ uniquely defines the ephemeral DH key Z as $sid[1]$ contains the key shares of the client and server in the CKS and SKS messages, respectively. Further, Z and the messages contained in $sid[1]$, i.e., $\mathtt{CH} \parallel \cdots \parallel \mathtt{SKS}$, uniquely define the stage-1 key htk_C. That is, for stage 1 agreeing on $sid[1]$ implies agreement of the key. For the remaining stages, first observe that $sid[2]$ only distinguishes from $sid[1]$ by unique label and the messages contained are identical. The messages contained in $sid[1]$ (resp. $sid[2]$) always are a prefix of the messages contained in $sid[s]$ for $s \in \{3, 4, 5, 6\}$. This implies that all session identifiers contain `Client KeyShare` and `ServerKeyShare`, and thus uniquely define Z. Now, observe that the stage s key is always defined by the session identifier $sid[s] = (label_s, ctxt)$ as $\mathsf{TKDF}_{label_s}(0, Z, \mathbf{H}(ctxt))$. Since all session identifiers uniquely define Z as discussed before, the session identifier for any stage uniquely defines the respective stage key implying that agreement on a session identifier implies agreement on the respective stage key.

2. *There are two sessions partnered (in a non-replayable stage), which have the same role.*

   In the full handshake, none onf the stages are replayable. Therefore, we need to consider all stages. All session identifiers contain the messages CH, CKS, SH, and SKS by defintion. In these four messages there are two pairs of nonce and group element included. One contributed by the client in (CH, CKS) and one contributed by the server in (SH, SKS). Given that there are two honest sessions that share the same *sid* and the same role, they would need to agree on (CH, CKS, SH, SKS) and they would need to share the same nonce and group element. However, this would cause flag bad_C to be set as this would be a collision among two honest sessions. We excluded this to occur in Game 1.

3. *There are session partnered in a stage that aim for a different authentication level in that stage.*

Note that the authentication level is unambiguously determined by presence/absence of the context-dependent messages. Stages 1 and 2 are initially always unauthenticated and become always unilaterally authenticated with the acceptance of stage 3. This is indicated by the messages SCRT and SCV appearing in the *sid* from stage 3 on. In fact, all other stages become unilaterally authenticated upon acceptance. The only difference for the authentication type is whether client authentication is performed and mutual authentication is achieved. Mutual authentication always is achieved in stage 6 for all stages (if at all). Here, the presence of the messages CR, CCRT, and CCV clearly indicates that the client authenticated. Hence, the *sid* unambiguously reflects the authentication level and agreement on the *sid* (i.e., partnering) implies agreement on the authentication level.

4. *There are sessions partnered in a stage that are not contributively partnered in that stage.*

   This event can never occur, since our definition of contributive identifiers (cf. Section 9.2) and session identifiers for any stage defines both equal as soon as the session identifier is set.

5. *There are partnered session that disagree on the identity of their peer.*

   Note that we only consider honest sessions in the context of the Sound predicate. Honest session always only send certificates that contain their owners identity. Responders always send the server certificate SCRT as all stages are unilaterally authenticated upon acceptance of stage 3 (perhaps retroactively). Intiators only send the client certificate CCRT if the server desires client authentication, and mutual authentication is then achieved for all stages (perhaps retroactively) upon acceptance of stage 6. The respective peer learns (and sets) the identity according to the content of the Certificate message. That is, if a client and server aim for unilateral authentication agreement on the stage-3 session identifier $sid[3]$ implies the client will set the server identity as its peer identity. Similarly, if they aim for mutual authentication agreement on the stage-6 session identifier $sid[6]$ implies mutual agreement on the peer. Note that $sid[6]$ contains both SCRT and CCRT containing both identities.

6. *There are two stages with the same session identifier.*

   This event cannot occur as by definition of the session identifiers each identifier is prefixed by a unique label defining the key they define.

7. *In a non-replayable stage, there are three sessions sharing the same session identifier for that stage.*

   Recall that each session identifier held by an honest session contains its pair of nonce and DH key share. For a threefold collision, there have to exist two honest sessions that sample the same pair of nonce and DH key share. This would cause flag bad_C to be set in Game 1 and the game would be aborted. Hence, this cannot occur in Game 1.

GAME 2. In Game 2, we set a flag bad_H if there are two distinct queries to random oracle $\mathsf{RO}_{\mathsf{Th}}$ that result in the same output. If flag bad_H is set, we abort the game. This change ensures that each transcript has a unique hash. To implement this, we introduce a look-up table $\mathsf{CollList}_{\mathsf{Th}}$ in random oracle $\mathsf{RO}_{\mathsf{Th}}$. Whenever $\mathsf{RO}_{\mathsf{Th}}$ computes a hash d for some input s, we log $\mathsf{CollList}_{\mathsf{Th}}[d] := s$. Therefore, $\mathsf{RO}_{\mathsf{Th}}$ can check whenever it computes a hash d on some input s' whether $\mathsf{CollList}_{\mathsf{Th}}[d] = s \neq \bot$ is already set. If it is already set, then we have found a collision. Namely, if s' would have been queried before, d would not have been computed, but just answered consistent with the previous query s'. Thus, we have $s \neq s'$ such that $\mathsf{RO}_{\mathsf{Th}}(s) = \mathsf{RO}_{\mathsf{Th}}(s') = d$.

There are at most q_{RO} queries to $\mathsf{RO}_{\mathsf{Th}}$ issued by the adversary $\mathcal{A}$ and in every protocol execution there are up to 7 *distinct* transcript hashes computed. That is, in total there are at most $q_{\mathsf{RO}} + 7q_{\mathsf{Send}}$ queries issued to $\mathsf{RO}_{\mathsf{Th}}$. By the birthday bound, we get that the probability that a collision occurs among the $q_{\mathsf{RO}} + 7q_{\mathsf{Send}}$ uniform and independent samples from $\{0,1\}^\lambda$ is bounded from above as

$$\Pr[\mathsf{bad}_H] \leq \frac{(q_{\mathsf{RO}} + 7q_{\mathsf{Send}})^2}{2^\lambda}.$$

Then, it holds that

$$\Pr[\mathsf{Game}_1] \leq \Pr[\mathsf{Game}_2] + \Pr[\mathsf{bad}_H] \leq \Pr[\mathsf{Game}_2] + \frac{(q_{\mathsf{RO}} + 7q_{\mathsf{Send}})^2}{2^\lambda}$$

Changing the way partners compute their session keys, and `Finished` tags. In the next games, we make a series of changes such that in partnered session the keys and tags are only computed once and the partner computing these values later just copies the respective values. Since in Game 2, Sound is always true, we can use here that all sessions that are partnered (resp. agree on the *sid* for a certain stage) will derive the same key in that stage. Thus, copying the value from the partner is consistent. This approach follows the tight key exchange security proof by technique by Cohn-Gordon et al. [Coh+19], which was, e.g., also adapted by Davis and Günther [DG21a] in their tighter proof for SIGMA and TLS 1.3. This allows us to ensure consistency between partners more easily in later games.

GAME 3. In Game 3, we let honest sessions log the keys and tags they computed. In addition, we let honest initiator sessions log their first messages, so that a responder can use this to check whether the received `Hello` message has an honest origin. To this end, we introduce a set CHs to log the `CH` messages output by honest sessions, a look-up table SKEYS to log the session keys computed by honest session under their respective session identifier, and a look-up table TAGS to log the `Finished` tags computed by honest sessions under the respetive context. In the following, we define how these tables are filled, and finally argue that this is not detectable for the adversary $\mathcal{A}$.

Log honest `ClientHello` messages. First, we log for all honest initiator sessions the messages $\mathtt{CH} \| \mathtt{CKS}$ in the set CHs before returning it to the adversary. Note that all values logged in CHs will only be logged once, because otherwise there would be two honest

initiator session outputting the same `Hello` message. Due to Game 1, this cannot occur as two identical `Hello` messages, in particular, share the same nonce and group element.

Log session keys computed by honest sessions. Second, recall that initiator sessions compute the *rms* before their partner and responder sessions compute htk_C, htk_S, ats_C, ats_S, and *ems* before their partner.[3] Therefore, we let honest initiator sessions π_u^i create a new entry in SKEYS upon acceptance of stage 8 (*rms*), i.e., $\mathsf{SKEYS}[\pi_u^i.sid[8]] \coloneqq \pi_u^i.skey[8]$. Honest responder sessions π_v^j create a new entry in SKEYS when accepting in stages 1 through 5, i.e.,

$$\mathsf{SKEYS}[\pi_v^j.sid[s]] \coloneqq \pi_v^j.skey[s]$$

for $s \in \{1, \dots, 5\}$. Note that no two sessions will ever log keys in the table SKEYS under the same session identifier *sid*. Since Sound cannot be violated in Game 3 anymore, we know that in any (non-replayable, i.e., all) stage s only one initiator and one responder will share the same $sid[s]$. Thus, if two sessions share a *sid* one of them will only log the key.

Log `Finished` tags computed by honest sessions. Third, recall that the `ServerFinished` and `ClientFinished` messages, as their names already suggest, are computed first by the server (responder) and the client (initiator), respectively. Therefore, we log whenever an honest session computes its finished message and to this end queries RO_x with $x \in \{fin_C, fin_S\}$, the following value

$$\mathsf{TAGS}[x, Z, d_1, d_2] \coloneqq \mathsf{RO}_x(0, Z, d_1, d_2)$$

where $(0, Z, d_1, d_2)$ constitutes the query issued by an honest session to compute its finished message. Note that we perform this logging in the session and not in the random oracle. Therefore, TAGS only contains tags computed by honest sessions. As `Finished` tags do not have a session identifier that they are defined over, we need to log the hashes used to compute them. Note that for all transcripts, these are unique due to Game 2.

In this game, we only introduce bookkeeping. Thus, from the view of the adversary nothing changes. That is,

$$\Pr[\mathsf{Game}_2] = \Pr[\mathsf{Game}_3].$$

GAME 4. In Game 4, we change the way sessions compute their keys and `Finished` tags. Namely, if a session has an honest partner in stage s it copies the stage s key that was already computed by its partner before instead of computing it itself. This can be done using the table SKEYS introduced in Game 3. Note that only honest session log their keys there, so if the session identifier of a session matches an entry in the look-up table, it knows that it is honestly partnered.

[3] We remark here that in our abstraction without handshake encryption, we have no internal keys. Therefore, servers always compute the listed keys before their partner client. When htk_C and htk_S were internal as in the unabstracted version of the TLS 1.3 handshake, there could be situations in which the a server session is "on pause" waiting to be tested and a client then might compute these keys before the server.

Honest server sessions. An honest server session π_v^j only copies (if any) the stage-6 key (*rms*). That is, upon receiving CF the session π_v^j additionally checks before it computes *rms* itself whether there is an entry in SKEYS indexed by $\pi_v^j.sid[6]$. If so, it sets *rms* to that value, and otherwise, proceeds as in Game 3.

Honest client sessions. An honest initiator session π_u^i potentially copies all keys except the last, *rms*. To this end, upon receiving (SH, SKS) it sets the session identifier for stages 1 (htk_C) and 2 (htk_S), and then looks for entries for $\pi_u^i.sid[1]$ and $\pi_u^i.sid[2]$ in SKEYS. In case, these values are set it copies the entry logged under $\pi_u^i.sid[1]$ as its htk_C and the entry logged under $\pi_u^i.sid[2]$ as its htk_S. If there are no consistent entries, it continues as in Game 3. Upon receiving (EE, CR*, SCRT, SF), it sets the session identifier for stages 3–5. Then, it checks whether there are entries indexed with the respective session identifiers. If so, it copies its stage-*s* key from $\mathsf{SKEYS}[\pi_u^i.sid[s]]$. Otherwise, it just continues as in Game 3.

Computation of MAC tags. All honest session (i.e., both client and server) which would query RO_x to compute a MAC (either for its Finished message or to verify one), first check the table TAGS whether there is a consistent entry for them. If so, they copy instead of making the query themselves.

It remains to argue that copying the keys from a partnered session is consistent with computing the keys as in Game 3. First, recall that Sound is always true in Game 4. That is, all sessions partnered in a stage *s*, i.e., sharing the same $sid[s]$, agree on their stage-*s* key $skey[s]$. Thus, it does not matter whether a session copies the key from its partner or computes it itself, the value of the key remains equal. Note that in Game 4 tampering is still possible. In particular, the adversary could tamper with the signatures or the Finished messages. Therefore, if, for example, an initiator copies their stage-1 and 2 keys because it is partnered in these stages (i.e., there is a consistent entry in SKEYS) this does not necessarily mean it copies its remaining keys. But in these cases, the session just act like in Game 3.

For Finished tags, we have that TAGS only caches responses to the random oracle queries that would be necessary to compute the respective tag values anyway. That is, if a session copies the value from TAGS instead of making the query itself, the concrete value would not change. This is because an entry in TAGS consistent with the inputs the session would hand to the random oracle implies that the corresponding random oracle query has already been made. Therefore, the random oracle would only answer consistently with the previous query anyway.

Hence, the view of the adversary is identical in Game 4 compared to Game 3 and it holds that

$$\Pr[\mathsf{Game}_3] = \Pr[\mathsf{Game}_4].$$

GAME 5. In Game 5, we move the random sampling of the session keys and tag values computed by honest sessions from the random oracles into the sessions. To ensure consistency, we program the random oracles retroactively if later random oracle queries are issued. To this end, we change the implementation as follows.

Random oracles. We introduce a new look-up table $\mathsf{ProgList}$ to keep track of values that might need later programming. To this end, we introduce a list $\mathsf{ProgList}_x$ for each random oracle RO_x. Additionally, we change the implementation of the random oracles RO_x. For any query, we check in RO_x if there already exists an entry indexed by that query in $\mathsf{ProgList}_x$. If so, we copy it in the random oracle table ROList_x indexed by the query and then proceed in RO_x as in Game 4. This ensures consistency between the keys sampled at random (as described below) and the random oracles.

Honest server sessions. All honest server sessions sample the keys htk_C, htk_S, ats_C, ats_S, and *ems* uniformly at random from the respective range instead of querying the corresponding oracle. This is under the condition that the value of these keys has not been fixed by queries to the corresponding random oracle with appropriate input before. If a later random oracle query occurs it sets the random oracle table ROList_x of the corresponding oracle RO_x to the previously sampled value. Let us demonstrate how this is concretely implemented for htk_C; for the other keys this is analogous. If a server session in Game 4 would query $\mathsf{RO}_{htk_C}(0, Z, H_3)$ to compute htk_C, we first check whether the value is already fixed by checking $\mathsf{ROList}_{htk_C}[0, Z, H_3] \neq \bot$. If the random oracle value is already fixed, we continue as in Game 4. Otherwise, we sample the value of htk_C uniformly at random from $\{0,1\}^\lambda$ and log $\mathsf{ProgList}_{htk_C}[0, Z, H_3] := htk_C$ for possible later reprogramming of RO_x as described above. Then, it continues as in Game 4 and logs the "computed" key so that a partnered initiator could copy it. The computation of htk_S, ats_C, ats_S, and *ems* follows analogously.

Moreover, honest server sessions that do not copy *rms* from SKEYS do the same as described above for *rms*. With respect to the `Finished` tags, honest server sessions compute their own `ServerFinished` message analogously to the the session keys, and in particular, keep consistency by checking ROList and maintaining $\mathsf{ProgList}$. To verify `ClientFinished` it first checks TAGS as in Game 4 and if a value is set, it proceeds as before. However, if there is no value set, the server samples the `Finished` value fin_C for verification at random (unless there is a suitable query in RO_{fin_C} ensuring consistency as before).

Honest client sessions. Honest client sessions that did not copy htk_C, htk_S, ats_C, ats_S, and *ems* do the same as described for the server session. Moreover, all honest client sessions compute their resumption master secret *rms* in the same way. The `Finished` tags are handled analogously to server sessions.

We claim that these changes are unobservable for the adversary $\mathcal{A}$. As already mentioned above the essence of this change is that we move the lazy sampling from the random oracles into the sessions if the values were not already fixed by the random oracles. As we later program the random oracles when a suitable query arrives consistency between all values is guaranteed. Therefore, partnered sessions still copy the keys from their partner and all other sessions choose random keys consistent with the respective random oracle. Thus, the distribution of keys did not alter from Game 4 to Game 5 and it holds

$$\Pr[\mathsf{Game}_4] = \Pr[\mathsf{Game}_5].$$

GAME 6. In Game 6, we stop maintaining the consistency between the keys and tag values computed and their corresponding random oracles in sessions honestly partnered in the first two stages. In particular, we change the implementation of honest server sessions that receive a `ClientHello` message that is contained in CHs, i.e., it originates from an honest client. These sessions only sample the keys htk_C, htk_S, ats_C, ats_S, and ems uniformly at random without checking the random oracle table or maintaining $\mathsf{ProgList}$. The same holds if this session does not copy rms. This also applies to honest clients that are partnered in stages 1 and 2 (i.e., they received an honest `ServerHello` message), but do not copy the stage-3, -4, or -5 key. These clients sample ats_C, ats_S, and ems uniformly at random without checking the random oracle table or maintaining $\mathsf{ProgList}$. Moreover, they sample rms inconsistently. The `Finished` tags are also derived inconsistently with the random oracles in these sessions. Here, honest server sessions that receive an honest `CH` sample their tag fin_S uniformly at random without checking the random oracle or maintaining $\mathsf{ProgList}$. If these sessions receive a `CF` message, and there is no consistent entry for verification in TAGS, then it samples this value at random similar to fin_S. Honest clients receiving an honest `SH` message, proceed similarly for their fin_C value and to verify `SF`. With these changes, we have that sessions partnered honestly in the first two stages will have derived keys inconsistently with the respective random oracles. Now, note that partnering in the first two stages implies agreement on the `CH` and `SH` messages. Agreement on these messages implies that these session derive the same ephemeral Diffie–Hellman key Z.

An adversary can only detect these inconsistencies if it is able to issue a query that would have been used to derive one of the inconsistent keys. Note that we make sure that the adversary could not have tampered with the messages `CH` and `SH` without detection using our bookkeeping. Therefore, we are certain that the adversary does not know the secret exponents x and y (i.e., the DLOGs of the client and server key share, respectively). That is, the adversary can only issue such a query if it would be able to compute g^{xy} by only observing $X = g^x$ (contained in `CKS`) and $Y = g^y$ (contained in `SKS`). Intuitively, if the adversary could do this, we could use it to solve the CDH problem (Definition 3.3).

Formally, we set a flag $\mathsf{bad}_{\mathrm{DHE}}$ if at any point in the game the adversary $\mathcal{A}$ queries one of the following random oracle queries:

1. For any $(s, x) \in \{(1, htk_C), (2, htk_S), (3, ats_C), (4, ats_S), (5, ems)\}$, there is a query
$$\mathsf{RO}_x(0, Z, \mathsf{RO}_{\mathsf{Th}}(ctxt[s]))$$
and there exists an honest server session π_v^j such that
   - π_v^j holds session identifier $sid[s] = (\text{“}x\text{”}, ctxt[s])$ in stage s,
   - π_v^j received a `ClientHello` message with $\mathtt{CH} \parallel \mathtt{CKS} \in \mathsf{CHs}$ (i.e., it originated from an honest client), and
   - $Z = g^{xy}$ with $\mathtt{CKS} = g^x$ and $\mathtt{SKS} = g^y$.
2. There is a query
$$\mathsf{RO}_{rms}(0, Z, \mathsf{RO}_{\mathsf{Th}}(ctxt[6]))$$
and there exists an honest server session π_v^j such that

- π_v^j holds session identifier $sid[6] = (\text{“}rms\text{”}, ctxt[6])$ in stage s,
- π_v^j received a `ClientHello` message with $\texttt{CH} \parallel \texttt{CKS} \in \mathsf{CHs}$ (i.e., it originated from an honest client), but there is no honest client session that shares $sid[6]$, i.e., π_v^j did not copy from SKEYS, and
- $Z = g^{xy}$ with $\texttt{CKS} = g^x$ and $\texttt{SKS} = g^y$.

3. There is a query
$$\mathsf{RO}_{rms}(0, Z, \mathsf{RO}_{\mathsf{Th}}(ctxt[6]))$$
and there exists an honest client session π_u^i such that
   - π_u^i holds session identifier $sid[6] = (\text{“}rms\text{”}, ctxt[6])$ in stage s,
   - π_u^i received a `ServerHello` message such that there is an entry for $\pi_u^i.sid[1]$ in SKEYS, and
   - $Z = g^{xy}$ with $\texttt{CKS} = g^x$ and $\texttt{SKS} = g^y$.
4. For any $(s, x) \in \{(1, htk_C), (2, htk_S), (3, ats_C), (4, ats_S), (5, ems)\}$, there is a query
$$\mathsf{RO}_x(0, Z, \mathsf{RO}_{\mathsf{Th}}(ctxt[s]))$$
and there exists an honest client session π_u^i such that
   - π_u^i holds session identifier $sid[s] = (\text{“}x\text{”}, ctxt[s])$ in stage s,
   - π_u^i received a `ServerHello` message such that there is an entry for $\pi_u^i.sid[1]$ in SKEYS, but there is no honest client session that shares $sid[s]$, i.e., π_u^i did not copy x from SKEYS, and
   - $Z = g^{xy}$ with $\texttt{CKS} = g^x$ and $\texttt{SKS} = g^y$.
5. There is a query
$$\mathsf{RO}_{fin_S}(0, Z, \mathsf{RO}_{\mathsf{Th}}(ctxt[1]), \mathsf{RO}_{\mathsf{Th}}(\texttt{CH} \parallel \cdots \parallel \texttt{SCV}))$$
and there exists an honest server session π_v^j such that
   - π_v^j holds session identifier $sid[1] = (\text{“}htk_C\text{”}, ctxt[1])$ in stage 1 and computed messages the message `EE`, `CR`, `SCRT`, and `SCV`,
   - π_v^j received a `ClientHello` message with $\texttt{CH} \parallel \texttt{CKS} \in \mathsf{CHs}$ (i.e., it originated from an honest client), and
   - $Z = g^{xy}$ with $\texttt{CKS} = g^x$ and $\texttt{SKS} = g^y$,
6. There is a query
$$\mathsf{RO}_{fin_C}(0, Z, \mathsf{RO}_{\mathsf{Th}}(ctxt[1]), \mathsf{RO}_{\mathsf{Th}}(\texttt{CH} \parallel \cdots \parallel \texttt{CCV}^*))$$
and there exists an honest server session π_v^j such that
   - π_v^j holds session identifier $sid[1] = (\text{“}htk_C\text{”}, ctxt[1])$ in stage 1, computed the message `EE`, $\texttt{CR}^*$, `SCRT`, and `SCV`,

- π_v^j received a `ClientHello` message with $\texttt{CH} \parallel \texttt{CKS} \in \mathsf{CHs}$ (i.e., it originated from an honest client), and received $\texttt{CCRT}^*$ and $\texttt{CCV}^*$ not (if at all) output by this honest client, and
- $Z = g^{xy}$ with $\texttt{CKS} = g^x$ and $\texttt{SKS} = g^y$,

7. There is a query

$$\mathsf{RO}_{fin_S}(0, Z, \mathsf{RO}_{\mathsf{Th}}(ctxt[1]), \mathsf{RO}_{\mathsf{Th}}(\texttt{CH} \parallel \cdots \parallel \texttt{SCV}))$$

and there is an honest client session π_u^i such that

- π_u^i holds session identifier $sid[1] = (\text{“}htk_C\text{”}, ctxt[1])$ in stage 1,
- π_u^i received a `ServerHello` message such that there is an entry for $\pi_u^i.sid[1]$ in SKEYS, i.e., there is an honest partner server, and it received $(\texttt{EE}, \texttt{CR}^*, \texttt{SCRT}, \texttt{SCV})$ not output by this honest server session, and
- $Z = g^{xy}$ with $\texttt{CKS} = g^x$ and $\texttt{SKS} = g^y$.

8. There is a query

$$\mathsf{RO}_{fin_C}(0, Z, \mathsf{RO}_{\mathsf{Th}}(ctxt[1]), \mathsf{RO}_{\mathsf{Th}}(\texttt{CH} \parallel \cdots \parallel \texttt{CCV}^*))$$

and there is an honest client session π_u^i such that

- π_u^i holds session identifier $sid[1] = (\text{“}htk_C\text{”}, ctxt[1])$ in stage 1, it received the message $\texttt{EE}$, $\texttt{CR}^*$, $\texttt{SCRT}$, $\texttt{SCV}$, and SF, and computed $\texttt{CCRT}^*$, $\texttt{CCV}^*$,
- π_u^i received a `ServerHello` message such that there is an entry for $\pi_u^i.sid[1]$ in SKEYS, and
- $Z = g^{xy}$ with $\texttt{CKS} = g^x$ and $\texttt{SKS} = g^y$.

Each of these queries could disclose an inconsistency introduced by the changes of Game 6. We bound the probability of flag $\mathsf{bad}_{\mathrm{DHE}}$ being set with a reduction $\mathcal{B}$ to the strong Diffie–Hellman problem (Definition 3.5) in group $\mathbb{G}$. Reduction $\mathcal{B}$ simulates Game 6 for adversary $\mathcal{A}$, and it wins whenever the simulated game would set the flag $\mathsf{bad}_{\mathrm{DHE}}$. Consider the following construction of reduction $\mathcal{B}$.

Construction of reduction $\mathcal{B}$. Reduction $\mathcal{B}$ gets as input a SDH challenge $(A = g^a, B = g^b)$ as well as access to the DDH oracle $\mathrm{DDH}_a(\cdot,\cdot) = \mathrm{DDH}(g^a, \cdot, \cdot)$ with the first argument fixed. It simulates the oracles $\mathsf{RevLongTermKey}$, $\mathsf{RevSessionKey}$ and Test as in Game 6, managing all the game variables itself. The Send oracle, i.e., the execution of the sessions, and the random oracles are handled differently. Let us first describe the idea on a high level and then elaborate on the details. When adversary $\mathcal{A}$ issues a Send queries the messages are delivered in the same way as in Game 6. However, the sessions handle them differently. On a high level, adversary $\mathcal{B}$ embeds a rerandomization of its SDH challenge in every client and in every server that receive an honest `Hello` message. Note that it does not know a or b, so it is not able to compute the DH key for these sessions. If $\mathsf{bad}_{\mathrm{DHE}}$ is set, then $\mathcal{B}$ learns the DH key corresponding to some of the embedded

rerandomizations. From this it can then extract the solution for the SDH challenge by basic algebra.

Next, we describe the additional look-up tables used by $\mathcal{B}$ and then describe how client and server sessions are implemented in detail.

- The look-up table KSRnd is maintained by all (honest) sessions. It holds the random exponent τ used by an honest session to randomize their key share G indexed by its nonce and key share (r, G). Note that due to Game 1 every nonce and group element pair is unique for a session and therefore we can use it to uniquely identify the session using τ.
- Each random oracle RO_x maintains a look-up table $\mathsf{DHEList}_x$. It holds for each query $\mathsf{RO}_x(0, Z, d)$, the value of Z indexed by d.
- For each random oracle RO_x, we maintain a look-up table $\mathsf{RndList}_x$. It maps a transcript hash d to a tuple of client key share randomizer τ_u^i, server key share randomizer τ_v^j, context $ctxt$ (i.e., a transcript such that $d = \mathsf{RO}_{\mathsf{Th}}(ctxt)$), and potentially a key key. This look-up table serves multiple purposes for random oracle RO_x. First, it can be used to check in RO_x upon an incoming query $(0, Z, d)$ whether this query sets $\mathsf{bad}_{\mathrm{DHE}}$. Second, it can be the case that honest client sessions receive a tampered $\mathtt{SH}$ message, but already committed to not knowing the DLOG of their share. In this case, we need to be able to (1) recognize a query $(0, Z, d)$ that requires consistency and (2) program the response so that RO_x and key of the client are consistent. Also, sometimes we have the case that $\tau_v^j = \bot$ when there is no honest server partnered to the client, or $key = \bot$ if we are certain that there never has to be a reprogramming of that value, e.g., if the corresponding sessions are honestly partnered in stage 1.

Implementation of honest server sessions. Consider any server session π_v^j. $\mathcal{B}$ implements this session as follows.

1. Upon receiving $(\mathtt{CH}, \mathtt{CKS})$, the reduction first checks whether these messages have an honest origin by checking $\mathtt{CH} \parallel \mathtt{CKS} \in \mathsf{CHs}$. If not, it will simulate π_v^j just as in Game 6. In particular, this means it will not embed a challenge value (in $\mathtt{SKS}$) and maintains consistency with the random oracles. For the rest of this discussion for server sessions, we assume that the origin, i.e., client session π_u^i, of the $\mathtt{Hello}$ message is honest.

   If there is an honest partner client, the session π_v^j generates its key share by randomizing the challenge DH value B. All the remaining values are chosen as in Game 6. To choose the key share, it chooses a randomizer $\tau_v^j \xleftarrow{\$} \mathbb{Z}_p$ uniformly at random, sets $Y := B \cdot g^{\tau_v^j}$, and logs $\mathsf{KSRnd}[r_S, Y] := \tau_v^j$.
2. Before it outputs $(\mathtt{SH}, \mathtt{SKS})$, it computes the keys htk_C and htk_S. Now, we cannot compute the DHE key Z because we embedded rerandomizations of the group elements B and A in this server session and, skipping slightly ahead, in the (honest) client session that output the $\mathtt{CH}$ session π_v^j received, respectively. We distinguish

two cases whether there already was a query for π_v^j that triggers $\mathsf{bad}_{\mathrm{DHE}}$ or not (yet), we present the procedure for htk_C and for htk_S it is analogous. To this end, we first check whether event $\mathsf{bad}_{\mathrm{DHE}}$ is triggered for π_v^j and a previous query to RO_{htk_C}. If so, $\mathcal{B}$ can already solve its SDH challenge. If not, there are currently no inconsistencies observable for the adversary $\mathcal{A}$ and we choose the key uniformly at random. Additionally, we store the current context for later to check in the random oracle RO_{htk_C} whether a later query causes the flag $\mathsf{bad}_{\mathrm{DHE}}$ to be set. For this method, we utilize the look-up tables $\mathsf{DHEList}_{htk_C}$ and $\mathsf{RndList}_{htk_C}$ as follows. First, $\mathcal{B}$ computes the transcript hash for htk_C, i.e., $d = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \cdots \parallel \mathtt{SH})$. Then, it checks whether $\mathsf{DHEList}_{htk_C}[d] \neq \emptyset$. If so, i.e., there was a query to $\mathsf{RO}_{htk_C}(0, \cdot, d)$, the reduction $\mathcal{B}$ fetches the key share randomizer $\tau_u^i = \mathsf{KSRnd}[r_C, X]$, where r_C and X are the nonce and key share of its honest partner client, and queries

$$\mathrm{DDH}_a(Y, Z \cdot Y^{-\tau_u^i})$$

for all $Z \in \mathsf{DHEList}_{htk_C}$ and for Y being the key share of server π_v^j. If any of these queries is answered positively, then there was a query that triggered $\mathsf{bad}_{\mathrm{DHE}}$ (query Type 1 according to the above list). That is, there is a $Z^* \in \mathsf{DHEList}_{htk_C}[d]$ with $Z^* \cdot Y^{\tau_u^i} = Y^a$, where a is the DLOG (wrt. to generator g) of the challenge value A. This then implies that $Z^* = Y^{a+\tau_u^i} = X^{b+\tau_v^j}$, where a and b are the DLOGs of the SDH challenges A and B, respectively, and $Y = B \cdot g^{\tau_v^j}$ and $X = A \cdot g^{\tau_v^j}$ are the key shares of server session π_v^j and client session π_u^i, respectively. This would exactly be the DHE key computed by π_v^j. We skip slightly ahead here, but we embed a rerandomization of A in *every* honest client session. The reduction $\mathcal{B}$ then submits the following solution to SDH to its game:

$$Z^* \cdot Y^{\tau_u^i} \cdot A^{-\tau_v^j} = Y^a \cdot A^{-\tau_v^j} = (g^a)^{b+\tau_v^j} \cdot (g^a)^{-\tau_v^j} = g^{ab}.$$

Note that here setting of $\mathsf{bad}_{\mathrm{DHE}}$ implies solving the SDH problem.

If none of the entries in $\mathsf{DHEList}_{htk_C}[d]$ is answered positively or $\mathsf{DHEList}_{htk_C}[d]$ is empty, there has not been a query triggering $\mathsf{bad}_{\mathrm{DHE}}$ (so far). Hence, we sample htk_C uniformly at random and log htk_C under $\pi_v^j.sid[1]$ in SKEYS. Further, we need to prepare that the random oracle RO_{htk_C} can check upon later queries whether $\mathsf{bad}_{\mathrm{DHE}}$ is set for this query by logging:

$$\mathsf{RndList}_{htk_C}[d] := (\tau_u^i, \tau_v^j, (\mathtt{CH}, \mathtt{CKS}, \mathtt{SH}, \mathtt{SKS}), \bot).$$

Note that we do not need to store the key here, as server π_v^j is partnered with an honest client in stage 1 and thus we never program RO_{htk_C}. We only need to observe the random oracle, whether $\mathcal{A}$ queries a tuple that could uncover this inconsistency.

For htk_S, π_v^j proceeds analogously.

3. Next, the server π_v^j computes the messages $\mathtt{EE}$, $\mathtt{CR}$, $\mathtt{SCRT}$, and $\mathtt{SCV}$ as in Game 6. Note that `CertificateRequest` is context dependent and is only sent if mutual authentication is desired.

4. For computing the `ServerFinished` message, the reduction proceeds similarly as described above for htk_C, but the lists are indexed slightly differently as the signature of the oracle RO_{fin_S} is different. Namely, RO_{fin_S} takes as input two hashes d_1 and d_2. For the computation of the server finished tag fin_S, these values are defined as follows:

$$d_1 = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \cdots \parallel \mathtt{SKS}) \quad \text{and} \quad d_2 = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \cdots \parallel \mathtt{SCV}).$$

For value fin_S it proceeds as described above for htk_C, but using lists $\mathsf{DHEList}_{fin_S}$ and $\mathsf{RndList}_{fin_S}$ indexed with (d_1, d_2), and random oracle RO_{fin_S}. Further, we store the context in the list $\mathsf{RndList}_{fin_S}$ as a tuple $((\mathtt{CH} \parallel \cdots \parallel \mathtt{SKS}), (\mathtt{CH} \parallel \cdots \parallel \mathtt{SCV}))$. If there is no (suitable) entry in $\mathsf{DHEList}_{fin_S}$, the value for fin_S is chosen uniformly at random and logged in TAGS instead of SKEYS. Note that TAGS requires the DHE key, which we are not able to compute. Therefore, we log the following entry instead:

$$\mathsf{TAGS}[fin_S, (\tau_u^i, \tau_v^j, (\mathtt{CH}, \mathtt{CKS}, \mathtt{SH}, \mathtt{SKS})), d_1, d_2] := fin_S.$$

5. Then, the server computes ats_C, ats_S, and ems. Here, it proceeds for each key exactly as in Step 2, but uses a different random oracle (RO_{ats_C}, RO_{ats_S}, or RO_{ems}) and a different context $d = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \| \cdots \| \mathtt{SF})$. If $\mathsf{bad}_{\mathrm{DHE}}$ is not triggered for a certain stage key, the respective key is logged in SKEYS under the corresponding session identifier.

   The server then outputs $(\mathtt{EE}, \mathtt{CR}^*, \mathtt{SCRT}, \mathtt{SCV}, \mathtt{SF})$.

6. Upon receiving $(\mathtt{CCRT}^*, \mathtt{CCV}^*, \mathtt{CF})$, the server first checks (if client authentication is performed, i.e., $(\mathtt{CCRT}^*, \mathtt{CCV}^*)$ are present) the signature contained in `CCV` and aborts if it is invalid for the announced public key `CCRT`. If not, it checks whether there is an entry

$$\mathsf{TAGS}[fin_C, (\tau_u^i, \tau_v^j, (\mathtt{CH}, \mathtt{CKS}, \mathtt{SH}, \mathtt{SKS})), d_1, d_2]$$

   with

$$d_1 = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \cdots \parallel \mathtt{SKS}) \quad \text{and} \quad d_2 = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \cdots \parallel \mathtt{CCV}^*)$$

   that is consistent with the messages it sent and received. If so, it compares this value to the tag fin'_C contained in `CF` and terminates if these values are different. If there is no consistent entry in TAGS, fin'_C was not computed by π_v^j's honest first stage partner client π_u^i. (Note that we did not exclude signature or MAC forgeries in this game so far.) Nevertheless, since π_v^j has an honest first stage partner, its DHE key used to derive fin_C is honestly established. Therefore, the server proceeds for fin_C analogously to the computation of fin_S in Step 4, but does not log the entry in TAGS if $\mathsf{bad}_{\mathrm{DHE}}$ was not triggered by RO_{fin_C} so far. It only samples fin_C uniformly and without maintaining consistency. Now, it checks fin_C against the tag fin'_C contained in `CF`. It they are different (which is quite likely), session π_v^j terminates. Otherwise, it proceeds to the last step.

7. Finally, π_v^j checks whether there is a consistent entry for *rms* in SKEYS and if so, copies it and terminates. Otherwise, there has been tampering by the adversary, but π_v^j had an honest first stage partner, so it computes *rms* exactly as htk_C described in Step 2, but with context $\mathtt{CH} \parallel \cdots \parallel \mathtt{CCV}^* \parallel \mathtt{CF}$ and without logging *rms* in SKEYS. If $\mathsf{bad}_{\mathrm{DHE}}$ was not triggered, it accepts *rms* and terminates.

Implementation of honest client sessions. Consider any client session π_u^i. Reduction $\mathcal{B}$ implements this session as follows.

1. $\mathcal{B}$ proceeds exactly as in Game 6 until it chooses π_u^i's key share. Instead of choosing a fresh and random key share X in $\mathbb{G}$, it chooses X as a rerandomization of A. That is, it chooses $\tau_u^i \xleftarrow{\$} \mathbb{Z}_p$ and sets $X := A \cdot g^{\tau_u^i}$. Then it logs τ_u^i in $\mathsf{KSRnd}[r_C, X]$, where r_C is the nonce chosen by π_u^i. The rest is identical to Game 6 and it outputs $(\mathtt{CH}, \mathtt{CKS})$.

2. Upon receiving $(\mathtt{SH}, \mathtt{SKS})$, it checks whether π_u^i has an honest first stage partner, by setting the stage-1 session identifier $sid[1]$ and checking whether

$$\mathsf{SKEYS}[sid[1]] \neq \bot.$$

   If so, there is an honest server that output the received `Hello` message and $\mathcal{B}$ copies htk_C from SKEYS as well as htk_S from $\mathsf{SKEYS}[sid[2]]$ after setting the stage-2 *sid*.

   If not, the adversary $\mathcal{A}$ did tamper with the received `Hello` message. Now, there is the problem that in Step 1, reduction $\mathcal{B}$ already committed to not knowing the DLOG of its key share by outputting a rerandomization of the challenge A. Thus, it cannot compute the DHE key that is required to maintain consistency with respective random oracles of the stage-1 and -2 keys. Note that in Game 6 we are only inconsistent if sessions are partnered honestly in the first (two) stage(s). To be consistent, we first need to check whether there already has been a random oracle query that fixed htk_C or htk_S to ensure consistency. This can be done in a similar way as we checked whether $\mathsf{bad}_{\mathrm{DHE}}$ was set in the implementation of honest server sessions above. We will elaborate on this below. If there has not been such a query to either of the respective random oracles, we can sample the key at random and store it in $\mathsf{RndList}_x$ for later reprogramming and thus ensuring consistency.

   Concretely, we do the following for $x \in \{htk_C, htk_S\}$. First, compute $d = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \cdots \parallel \mathtt{SKS})$ and then for all $Z \in \mathsf{DHEList}_x[d]$, query

$$\mathrm{DDH}_a(\tilde{Y}, Z \cdot \tilde{Y}^{-\tau_u^i})$$

   where $\tilde{Y}$ is the dishonest key share contained in $\mathtt{SKS}$. If there exists a value $Z^* \in \mathsf{DHEList}_x[d]$ such that the above query returns 1, we set the key $x := \mathsf{RO}_x(0, Z^*, d)$ as this value is already defined. For further information, what the above DDH query implies, see the implementation of honest server sessions above. If there is

no such Z^*, we choose key uniformly at random, set $x := key$ and store key for later reprogramming. That is, it logs

$$\mathsf{RndList}_x[d] := (\tau_u^i, \bot, (\mathtt{CH}, \mathtt{CKS}, \mathtt{SH}, \mathtt{SKS}), key)$$

where the server key share randomizer is $\bot$ as the key share in SKS is dishonest.

At this point, π_u^i accepts both stage 1 and 2, because it either copied or computed the respective keys.

3. Upon receiving $(\mathtt{EE}, \mathtt{CR}^*, \mathtt{SCRT}, \mathtt{SCV}, \mathtt{SF})$, we distinguish, whether π_u^i has an honest partner in Step 1.

   If there was no honest server session in the first (two) stage(s), then there will also be no honest server session in all subsequent stages, because otherwise $\mathsf{bad}_{C'}$ would occur, which is excluded in Game 1. This is due to the reason that the stage 1 transcript is a prefix of all transcripts contained in the *sid* of all subsequent stages. Therefore, π_u^i will also not copy the keys of stages 3, 4 and 5. Session π_u^i processes the messages as follows. It first verifies the received SCV message and aborts if it is invalid. Then, it has no honest partner in stage 1, so there will be no entry consistent entry in $\mathsf{TAGS}[fin_S, \cdot]$ to verify the SF message. Therefore, it samples the fin_S value at random unless there is a consistent RO_{fin_S} query. To check this it proceeds exactly as in Step 2 with $x = fin_S$ except that `Finished` tags require two transcript hashes d_1 and d_2, as well as two context strings. After this step, either fin_S is set consistently with RO_{fin_S} or chosen uniformly at random. Next, it verifies the SF tag against fin_S and aborts if they are different. Otherwise, π_u^i proceeds for $x \in \{ats_C, ats_S, ems\}$ exactly as described in Step 2, but with context $\mathtt{CH} \parallel \cdots \parallel \mathtt{SF}$.

   If there was an honest server session, then we first check whether the adversary $\mathcal{A}$ tampered with one of the messages $(\mathtt{EE}, \mathtt{CR}^*, \mathtt{SCRT}, \mathtt{SCV}, \mathtt{SF})$. To this end, it first sets the session identifier for stages 3, 4 and 5. Then, session π_u^i can check whether tampering occurred by checking

   $$\mathsf{SKEYS}[sid[3]] = \bot.$$

   First, note that (ignoring the labels) $sid[3] = sid[4] = sid[5]$, which means that if there is an honest partner in stage 3, then there is also one in stages 4 and 5. Further, note that π_u^i is honestly partnered in stage 1 and that SKEYS is only filled by honest sessions. Due to the changes of Game 1, there can only be exactly one honest server session that π_u^i is partnered with in stage 3 and this is the server session it is partnered with in the first (two) stage(s). Hence, if SKEYS is set for $sid[3]$, then no tampering occurred. If no tampering occurred, there will also be an entry in TAGS for fin_S consistent with π_u^i. Therefore, it checks for completeness whether the SCV and SF messages are valid, but since no tampering occurred the checks will go through. Then, it copies ats_C, ats_S, and ems from SKEYS and proceeds to the next step.

If there was tampering by the adversary, then we need to be more careful. Session π_u^i is partnered in the first stage, so the DHE key it would compute in Game 6, would be an honest one. For the simulated game, this means that it cannot be computed, since we embedded the SDH challenge in the key share of the two peers. Thus, the adversary could trigger $\mathsf{bad}_{\mathrm{DHE}}$ for this tampered handshake. Note that we did not exclude signature forgeries up to now, so we first check whether the SCV message is valid for the announced SCRT. If not, we terminate the session. Otherwise, we need to fetch fin_S for checking the SF message. Note that the adversary could tampered with each of the messages (EE, CR*, SCRT, SCV, SF). Since if it tampered with (EE, CR*), it also would need to forge a signature as we already excluded invalid signatures. So, we focus on the cases that it tampered with either the server signature or the server finished tag only, or both. If it tampered with the SF message only, then there will be an entry in TAGS for fin_S consistent with π_u^i's view that will uncover the tampering, and will cause π_u^i to terminate. Because the SF message depends only on the transcript up to (and including) the SCV message. If it tampered with the signature only, then this immediately implies that it also would need to tamper with the server finished tag as the honest server session π_v^j would not verify a tampered signature. Hence, the only interesting case is when it tampered with both and there is no honest session that seeks to tag the transcript that π_u^i would expect to be in the SF message. Now, note that we are inconsistent with fin_S for this value, as the session is honestly partnered in stage 1. Here, we proceed exactly as described in Step 6 of the server implementation, where the server received a tampered CF. After that, $\mathsf{bad}_{\mathrm{DHE}}$ was either triggered or fin_S was chosen at random. With this fin_S value, the session verifies the SF message, and terminates if it is different. If it is valid, we need to choose the keys ats_C, ats_S, and *ems* inconsistently with the random oracle. Here, we proceed exactly as in Step 4 of the implementation of server session, but without storing the final keys in SKEYS as there will be no partner that might copy these values.

4. Finally, it remains to compute (the client authentication messages and) the CF message, and to compute *rms*. If client authentication is required it computes the corresponding messages (CCRT, CCV) as in Game 6. The ClientFinished message needs to be computed in either of the two following ways. If π_u^i has been an honest first stage partner it computes the tag fin_C analogously to fin_S in Step 4 of the implementation of honest server sessions. Next, to compute *rms* it proceeds analogously as described in Step 2 of the implementation of honest server. Note that we do not need to distinct here whether (EE, CR*, SCRT, SCV, SF) were tampered with. Since π_u^i has an honest partner in stage 1 we are inconsistent anyway and $\mathsf{bad}_{\mathrm{DHE}}$ still could be triggered. The only thing to note here if there has not been a partner in stage 3, the key *rms* will of course never be copied from SKEYS.

   If there has been no honest first stage partner, we need to ensure consistency and so we compute Step 2 for $x \in \{fin_C, rms\}$ ensuring consistency with the respective random oracles.

Then, it accepts *rms* and terminates.

Implementation of random oracle RO_x. If $x = \mathsf{Th}$, $\mathsf{RO}_{\mathsf{Th}}$ works just as in Game 6. For $x \neq \mathsf{Th}$, that is for any of the TKDF oracles, we proceed as follows. If RO_x receives a query that was already answered before, we answer consistently. If there is a new query (p, Z, d) with $p \neq 0$, we answer as in Game 6. If $x \notin \{fin_C, fin_S\}$ and $p = 0$, i.e., for the random oracles computing stage keys, we process new queries (p, Z, d) as follows. First, RO_x logs $\mathsf{DHEList}_x[d] := \mathsf{DHEList}_x[d] \cup Z$, so that we can check in the session if $\mathsf{bad}_{\mathrm{DHE}}$ occurred. If $\mathsf{RndList}_x[d] \neq \bot$, we know that there already was an honest session using context hash d to compute a key x without knowing the respective DHE key. If not, we proceed as in Game 6. Let $(\tau_u^i, \tau_v^j, ctxt, key) = \mathsf{RndList}_x[d]$ be the tuple, where the first two components are the randomness used by a client and server to randomize the SDH challenge, $d = \mathsf{RO}_{\mathsf{Th}}(ctxt)$, and key denotes the (potentially undefined) value of the computed key x for reprogramming, because RO_x has not fixed the value for x at the time of computation. Now, we can check whether Z is the right value for the query logged in $\mathsf{RndList}_x[d]$ by querying $\mathrm{DDH}_a(Y, Z \cdot Y^{-\tau_u^i})$. If this query is answered positively, we know that Z is a valid Z for computing key x with context d by the session that logged d. Next, we need to check whether Z is in fact a solution to SDH and $\mathsf{bad}_{\mathrm{DHE}}$ is triggered, or whether we need to program RO_x for consistency. Recall that we only ensure consistency if there was no honest stage 1 partner. For server session, we can react adaptively as we can decide to embed the challenge and thus not knowing the DHE key on demand when receiving the CH message. For client (initiator) sessions, this is different. Here, we always embed and need to react when the SH is received, because clients always commit on not knowing the DHE key. Therefore, we only need to ensure consistency "manually" for honest initiator sessions without a first stage partner. We can identify these logs in $\mathsf{RndList}$ as these sessions always set $\tau_v^j = \bot$ (i.e., no honest server present). By the implementation of honest client sessions, the logs will also always set $key \neq \bot$. Thus, we program the $\mathsf{ROList}_x[0, Z, d] := key$ and answer accordingly. If τ_v^j is defined, we know that the respective query belongs to a pair of honestly partnered client and server, and this triggers $\mathsf{bad}_{\mathrm{DHE}}$. Hence, $\mathcal{B}$ submits $Z^* \cdot Y^{\tau_u^i} \cdot A^{-\tau_v^j}$ to the SDH game and will win (as discussed above).

If $x \in \{fin_C, fin_S\}$ and $p = 0$, the random oracle RO_x proceeds similarly as described in the previous paragraph, but this computation required two hashes d_1 and d_2 instead of a single hash d.

Unless, $\mathcal{B}$ solved the SDH challenge above RO_x outputs $\mathsf{ROList}_x[0, Z, d]$.

Analysis of reduction $\mathcal{B}$. By the considerations above, we have that if $\mathsf{bad}_{\mathrm{DHE}}$ would be set in Game 6, we are able to find a *valid* solution for $\mathcal{B}$'s SDH challenge. Further, note that $\mathcal{B}$ queries at most one DDH oracle query for every random oracle query issued by the adversary. An honest server that received an honest CH queries the DDH oracle at most once for every $Z \in \mathsf{DHEList}_x[d]$ when computing key x, where d is the context hash of the server for key x. Due to Games 1 and 2, and the context hash is unique for this server. Note that by the definition of the session identifiers there are multiple keys sharing the same context and thus the same context hash, but they belong to the *same* session.

That is, only this server session will iterate $\mathsf{DHEList}_x[d]$ (for all x and corresponding d). Every entry $Z \in \mathsf{DHEList}_x[d]$ implies that there was a query $\mathsf{RO}_x(0, Z, d)$. Hence, a server queries DDH for every of these random oracle queries at most once. This is an upper-bound, because it does not necessarily query DDH for *rms* and fin_C always, but only if the adversary tampered with the last messages of the client. An honest client session that has an honest partner server would share the same context hashes with that server, but due to the existence of the honest session it will only copy the stage 1–5 keys from SKEYS and only queries DDH for *rms* and fin_C. Even though the partner server would share the context for these values as well if it receives the honest values, it does not query DDH again and uses SKEYS and TAGS instead. If the adversary tampered with either the SH message or the authentication messages, the client will use DDH in the worst case for all stage-1–6 keys and MAC values. However, due to Game 1, the CH of this client and also the received SH (potentially dishonest) are unique among the honest sessions. That is, the context hashes of this client are unique as well (cf. Game 2) and thus, as for the server, only this client will iterate $\mathsf{DHEList}_x[d]$ (for all x and corresponding d). Hence, informally every client and every server will query DDH for their own (disjoint) set of random oracle queries $(x, 0, Z, d)$ and only once per element. It remains to argue about the random oracle RO_x. RO_x queries DDH in a query $(0, Z, d)$ once and only if $\mathsf{RndList}_x[d]$ is set. If this is the case, we have that there already was an honest session with context d for x, and that $\mathsf{bad}_{\mathrm{DHE}}$ was not triggered and no suitable value Z' was queried before. Consequently, as d is unique to that session (and its partner, which only copies if it exists), Z will never be queried in a session again since that session already accepted x. Overall, we conclude that every random oracle query induces at most one query to DDH, i.e., $\mathcal{B}$ issues at most q_{RO} DDH queries. That is, it holds that

$$\Pr[\mathsf{Game}_5] \leq \Pr[\mathsf{Game}_6] + \Pr[\mathsf{bad}_{\mathrm{DHE}}] \leq \Pr[\mathsf{Game}_6] + \mathrm{Adv}^{\mathsf{SDH}}_{\mathbb{G},p}(\mathcal{B}).$$

Now, if the game is not aborted because $\mathsf{bad}_{\mathrm{DHE}}$ is set, we know that the adversary $\mathcal{A}$ does not learn anything about the session keys and tags values computed by sessions that partnered in the first stage. In the next two games, we will exclude forgeries of the authentication messages of honest sessions and then will conclude that the adversary $\mathcal{A}$ cannot win the resulting game anymore.

GAME 7. In Game 7, we add a flag bad_S and abort the game if it is set. The flag bad_S is set if there is an honest session that receives a valid signature under an uncorrupted public key and it authenticates a message that no honest session signed. We bound the probability that flag bad_S is set by a reduction to the multi-user security (Definition 4.7) of the signature scheme Sig configured for the considered instance of the TLS 1.3 handshake.

Construction of reduction $\mathcal{F}$. The reduction $\mathcal{F}$ receives no input and gets oracle access to New, Sign, and Corrupt (as defined in Figure 4.4). It simulates Game 7 for the adversary $\mathcal{A}$ except it changes the implementation of the NewUser oracle (cf. Figure 5.1) to add new users, the RevLongTermKey oracle to corrupt the long term key of users and the way how honest sessions compute their signature for the SCV/CCV message. In the NewUser oracle, reduction $\mathcal{F}$ queries the oracle New to obtain a new public key

pk_{users} instead of running the key generation algorithm Sig.Gen of Sig. Note that with this implementation of NewUser reduction $\mathcal{F}$ is not able to compute the signatures of the sessions anymore because it does not know the respective secret (signing) keys. Therefore, it uses its signing oracle Sign to compute the signature in a session instead of using the algorithm Sig.Sign. Concretely, if a session of user u would compute $\sigma \xleftarrow{\$} \mathsf{Sig.Sign}(sk_u, m)$ in Game 7, reduction $\mathcal{F}$ queries $\sigma \xleftarrow{\$} \mathsf{Sign}(u, m)$. As mentioned above, $\mathcal{F}$ does not know the the secret keys anymore, therefore it also needs to implement the oracle RevLongTermKey, which allows the adversary to corrupt the long term keys (i.e., the signature secret keys). To this end, it fetches $sk_u := \mathsf{Corrupt}(u)$ upon a query $\mathsf{RevLongTermKey}(u)$ and implements the rest exactly as in Game 7. We further introduce bookkeeping for all messages signed by honest sessions. To this end, any honest session π_u^i logs (u, m) in a set SIGS if it computes a signature on message m. This then can be used to easily check whether bad_S would have been set in Game 7: After a session verified the signature σ for some message m received by its peer *pid* successfully, the reduction $\mathcal{F}$ can check whether

$$\mathsf{revsk}[pid] = \infty \land (pid, m) \notin \mathsf{SIGS}$$

holds. If so, bad_S would be set in Game 7, since this session received a signature that is valid for the expected public key of *pid* such that *pid* is not corrupted and none of *pid*'s sessions signed the message m the signature is valid for. Hence, $\mathcal{F}$ found a forgery (pid, m, σ) and outputs it to its game. Note that this holds independently of the peer's role. We assume that sessions only receive signatures that they expect. In particular, we assume that the adversary does not send a client signature to a server that did not sent a `CertificateRequest`.

Analysis of reduction $\mathcal{F}$. First of all, note that if bad_S is set, $\mathcal{F}$ will output a valid forgery for Sig. This is because how bad_S is chosen. The flag ensures that the signature received is valid for an uncorrupted public key and that the message was not output by any of the sessions of the owner of that uncorrupted public key. As we compute all honest signatures using the signing oracle, there cannot be a query to the signing oracle for the respective user and message. Otherwise, there would be an entry in SIGS. That is, if bad_S is set, then the winning condition of the multi-user security experiment is met by the tuple (pid, m, σ) that induced flag bad_S to be set. Then, it holds

$$\Pr[\mathsf{bad}_S] \leq \mathrm{Adv}^{\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathrm{corr}}}_{\mathsf{Sig}}(\mathcal{F}).$$

Note that adversary $\mathcal{F}$ issues q_{NewUser} queries to New, q_{Send} queries to Sign (as every session queries the Sign oracle at most once), and $q_{\mathsf{RevLongTermKey}}$ queries to Corrupt. Its runtime is approximately the same as the run time of $\mathcal{A}$.

Overall, it holds for Game 7 that

$$\Pr[\mathsf{Game}_6] \leq \Pr[\mathsf{Game}_7] + \Pr[\mathsf{bad}_S] \leq \Pr[\mathsf{Game}_7] + \mathrm{Adv}^{\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathrm{corr}}}_{\mathsf{Sig}}(\mathcal{F}).$$

From this game on, we excluded that the adversary can tamper with the signature (up to a certain degree briefly outlined next). Note that Game 7 only excludes that the

adversary cannot compute a signature under an uncorrupted public key for a message that the owner of that public key did not intend to sign. This does not exclude that the adversary may output a new signature for a message actually signed by an honest session that is different from the one output by that session. To exclude this, we would need to reduce the security to the strong unforgeability (cf. Remark 4.2) of the signature scheme Sig. However, this requirement is unnecessarily strong as we can exclude this in the next step by excluding MAC forgeries in honest sessions.

GAME 8. In Game 8, we add a flag bad_M and abort the game if it is set. The flag bad_M is set if there is an honest session with an honest partner in the first stage that receives a valid `Finished` message that was not by any honest session. Next, we analyze the probability that flag bad_M is set. Note that due to the change of Game 6 all sessions that are partnered in the first two stages (i.e., honest servers that received an honest `ClientHello` and honest clients that receive an honest `ServerHello` from a server that received their `ClientHello`) choose their keys inconsistently with the random oracle without the adversary being able to detect this change. Therefore, all keys and all tags computed by these sessions are distributed uniformly at random from the view of the adversary. Recall that due to the changes of Game 6, if there is such a honest session that receives a `Finished` message not output by an honest session, i.e., there is no consistent entry in TAGS, then this session will sample the value it verifies the `Finished` message against uniformly (and independently) at random. Therefore, for any fixed session with a first stage partner that receives a dishonest `Finished` message, it holds that the probability of this MAC being verified successfully is $1/2^\lambda$. By the union bound, we can bound this probability for any session with a first stage partner from above and thus also upper-bounding the probability of bad_M being set as follows:

$$\Pr[\mathsf{bad}_S] \leq \frac{q_{\mathsf{Send}}}{2^\lambda}$$

where q_{Send} is the number of Send queries, i.e., an upper-bound on honest session activations. Hence, we overall have that

$$\Pr[\mathsf{Game}_7] \leq \Pr[\mathsf{Game}_8] + \Pr[\mathsf{bad}_M] \leq \Pr[\mathsf{Game}_8] + \frac{q_{\mathsf{Send}}}{2^\lambda}.$$

ExplAuth cannot be violated. Next, we argue that in Game 8, predicate ExplAuth cannot be violated by the adversary anymore, i.e., ExplAuth = true with certainty in Game 8. Predicate ExplAuth is set to false if there is a session π_u^i accepting an explicitly authenticated stage s, whose intended peer $\pi_u^i.pid = v$ was not corrupted at the time of π_u^i accepting the stage s' in which it received (perhaps retroactively) explicit authentication, and (1) there is no honest session π_v^j partnered to π_u^i in stage s', (2) there is an honest session π_v^j partnered to π_u^i in stage s', but it accepts with a peer identity $w \neq v$, or it is not partnered to π_u^i in stage s.

Recall that we always at least have unilateral (server-only) authentication. Therefore, client (initiator) sessions receive explicit authentication with acceptance of stage 3, after the `ServerFinished` message successfully verified. That is, stages 1 and 2 receive

explicit authentication retroactively upon acceptance of stage 3 and stages 3–6 received explicit authentication upon acceptance, but due to acceptance of stage 3. For server (responder) sessions, we have that they receive (retroactively) explicit authentication upon acceptance of stage 6, after the `ClientFinished` successfully verified.

We first argue why (1) cannot occur in Game 8. Let us start with client sessions. In the following, we explain why in Game 8 every honest initiator π_u^i that accepts stage 3 with an uncorrupted peer will have an honest partner server π_v^j in stage 3. The crucial argument here is that the adversary $\mathcal{A}$ cannot tamper with `ServerHello` message in Game 8 anymore as we excluded signature forgeries in Game 7. That is, if an honest initiator π_u^i receives ($\mathtt{SH}, \mathtt{SKS}, \mathtt{EE}, \mathtt{CR}^*, \mathtt{SCRT}, \mathtt{SCV}, \mathtt{SF}$) it can be certain about the following things:

1. If $\mathtt{SCV}$ is valid for $m = \ell_{13} \| H_8$, where ℓ_{13} is defined in Table 6.1 and $H_8 = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \| \cdots \| \mathtt{SH} \| \mathtt{SKS} \| \mathtt{EE} \| \mathtt{CR}^* \| \mathtt{SCRT})$ for the messages sent and received by π_u^i, under the uncorrupted public key of the peer identity v announced in $\mathtt{SCRT}$, then
   a) There is an honest session π_v^j owned by v that signed H_8, which due to Game 2 is unique to the messages $\mathtt{CH} \| \cdots \| \mathtt{SH} \| \mathtt{SKS} \| \mathtt{EE} \| \mathtt{SCRT}$.
   b) Session π_v^j received the $\mathtt{CH}$ output by π_u^i as it (1) due to Game 1 is unique to π_u^i and (2) otherwise it would not have included it in the transcript to be signed.
   c) The messages ($\mathtt{SH}, \mathtt{SKS}, \mathtt{EE}, \mathtt{SCRT}$) were computed by π_v^j as otherwise it would not have signed these messages, and due to Game 1, ($\mathtt{SH}, \mathtt{SKS}$) are unique to π_v^j, thus only (the unique) session π_v^j would seek to sign this transcript.

   Note that in Game 7, we only ensure that all sessions receive a signature that is valid for a message that was actually signed by an honest session. However, this does not include if the adversary is able to compute a new signature from the signature output (e.g., by rerandomizing it). Intuitively it only ensures that the adversary cannot compute signatures for messages no honest session would intent to sign. Therefore, the MAC tag contained in the `Finished` message further is required to ensure integrity of the $\mathtt{SCV}$ message.
2. Due to Game 8, there is an entry in TAGS for π_u^i's view (i.e., its DHE key Z and the hashes $d_1 = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \| \cdots \| \mathtt{SKS})$ and $d_2 = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \| \cdots \| \mathtt{SCV})$, which are unique due to Game 2) that is consistent with the value fin_S contained in $\mathtt{SF}$ such that fin_S is valid and consequently $\mathtt{SCV}$ was output by an honest session, which is by the considerations above the (unique) session π_v^j.

Since π_v^j is unique, it also has to be the honest session that computed the tag in TAGS as no other session would have seeked to compute this value with the context. Hence, π_v^j is partnered to π_u^i in stage 3.

Server (responder) sessions π_u^i can only be explicitly authenticated if client authentication is performed. In this case, we can argue similarly as for client sessions that there always has to be an honest client session π_v^j that needs to be partnered to π_u^i in stage 6 in which the server receives explicit authentication for all stages. If this session

π_u^i accepts stage 6 with an uncorrupted peer, then this implies that if the signature CCV received by π_u^i has to be signed by an honest session π_v^j of the peer announced in the CCRT message. The signature CCV authenticated the message $m = \ell_{14} \| H_9$, where ℓ_{13} is defined in Table 6.1 and $H_8 = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \| \cdots \| \mathtt{SH} \| \mathtt{SKS} \| \mathtt{EE} \| \mathtt{CR} \| \mathtt{SCRT} \| \mathtt{SCV} \| \mathtt{SF} \| \mathtt{CCRT})$ for the messages sent and received by π_u^i. Following a similar line of arguments as above, we conclude (due to Games 1, 2, and 7) that there has to be a unique client session π_v^j owned by v that sent and received these messages. Again by Game 8, there has to be an entry in TAGS for π_u^i that proves validity of fin_C contained in CF received by π_u^i and implies that CCV has to be computed by the same honest (unique) client session π_v^j that signed the above transcript. Due to the uniqueness of this session, it also has to be the session that computed the entry in TAGS, the CF message, and hence π_v^j is partnered to π_u^i in stage 6.

Next, let us address the possibility of case (2). Recall that a server session receives (if client authentication is performed) retroactively explicit authentication upon acceptance of stage 6. Now, if a server session π_u^i has a partner in stage 6, i.e., there is a session π_v^j with $\pi_u^i.sid[6] = \pi_v^j.sid[6]$, then these sessions also have to be partnered in all stages $s < 6$, because the transcripts contained in the session identifiers of stages $s < 6$ are all "sub-transcripts" of the session identifier of stage 6. Further, as π_v^j has set its session identifier in stage 6, it also must have accepted stage 6 implying that it also had accepted stages $s < 6$. As in Game 8 predicate Sound cannot be violated anymore, we know by Property 5 that all partnered sessions agree on the peer identity. Hence, for a server session case (2) is impossible to be violated. For a client session, we have that stages 1 and 2 receives explicit authentication retroactively upon acceptance of stage 3, and all stages $s > 3$ receive explicit authentication upon acceptance, but because of acceptance of stage 3. For stages $s \leq 3$, we can argue exactly as for the server session before. Further, the peer is set upon verification of the authentication messages. As server signatures and Finished messages are verified right before acceptance of stage 3, the peer identity will not change anymore. However, the session identifier of stages 4–6 might. For stages 4 and 5, the argument that this cannot occur is easy: the transcripts contained in $sid[s]$ for $s = 3, 4, 5$ are identical (except for the labels). Therefore, partnering in stage 3 implies partnering (upon acceptance) of stage 4 and 5, respectively. The session identifier in stage 6 differs from the stage 5 session identifier only by the client authentication messages (i.e., CCRT, CCV, and CF). Note that the CCRT and CCV are only sent if client authentication is performed. So, to violate case (2) and thus ExplAuth the adversary needs to ensure that the client does not have a partner in stage 6 even though it had one stage 5. To this end, it can either drop the authentication messages from the client and thus the partner server will never accept, or it has to forge, in particular, the Client Finished message. Note that if it forges the client signature, then it also has to forge the CF message. Dropping the messages, would induce that there is no partner for the client in stage 6, but this case seems to be unavoidable as it is the last protocol message. In our definition, we address this problem by requiring that the session only needs to be

partnered with its stage s' partner in stage s *if* the partner accepted stage s'.[4] Hence, the adversary can only violate ExplAuth, when it makes the partner server accept a forged `ClientFinished`, which is not possible due to Game 8.

If Fresh = true, then the adversary can only guess. Finally, we argue that the adversary $\mathcal{A}$ can only win Game 8 with probability at most $\frac{1}{2}$ if the predicate Fresh is not violated. Recall that if Fresh is not violated, then no session could have been tested and revealed in the same stage, and a tested session's partner may also be neither tested nor revealed in that stage. In the following, we assume that this did not occur. Further, the Fresh predicate depends on the levels of forward secrecy reached at the time of the Test query. Recall that all stages in the TLS 1.3 full handshake receive forward secrecy upon acceptance. That is, if a session π_u^i is tested on a stage s it remains fresh (i.e., Fresh is not set to false) if the owner of the session's peer was corrupted only after acceptance of stage s or if there is a contributive partner in that stage. Next, we argue that all tested session keys in Game 8 such that the corresponding Test query does not violate Fresh are distributed uniformly (and independently of the challenge bit b) at random from the view of the adversary. Assume that session π_u^i is any session that is tested in some stage s. Further, let the owner of π_u^i's peer be corrupted only after acceptance of stage s. This implies that the π_u^i accepted with an uncorrupted peer. In Game 8, due to the changes introduced in Games 7 and 8, π_u^i could only have received honest and untampered messages as signature and MAC forgeries are excluded, and the session would terminate after detection of this tampering. If $s = 1, 2$, then we can detect whether tampering occurs using the lists CHs and SKEYS. Therefore, there has to be a session π_v^j that computed the messages received by π_u^i and this session has to be unique (by Games 1 and 2). In fact, π_v^j is a contributive partner of π_u^i in stage s. Hence, due to Games 4–6, π_u^i (depending on its role and s) either samples its stage s uniformly at random inconsistently with the random oracle or copies the (uniformly random) stage s key from SKEYS computed by π_v^j. Consequently, the key returned by Test is a uniformly random key independent of the challenge bit b. Therefore, the adversary cannot learn anything about the stage s key of any session accepting stage s with an uncorrupted peer. As π_v^j is a contributive partner of π_u^i, we can use the same arguments given above to argue that the key returned by Test is distributed uniformly at random independent of the challenge bit b if the session's peer was corrupted before accepting stage s, but there is an honest contributive partner.

Overall, we have that the adversary $\mathcal{A}$ cannot learn anything about the challenge bit b from Test queries without violating Sound, ExplAuth, and Fresh. Thus, the probability to win in this game is no greater than $\frac{1}{2}$, i.e., $\Pr[\mathsf{Game}_8] \leq 1/2$.

Collecting all the term, we get the following:

$$\Pr[\mathrm{Exp}^{\mathsf{pMSKE}}_{\mathtt{TLS\text{-}DHE}}(\mathcal{A}) = 1] \leq \frac{2q_{\mathsf{Send}}^2}{2^{256} \cdot p} + \frac{(q_{\mathsf{RO}} + 7q_{\mathsf{Send}})^2}{2^{\lambda}} + \mathrm{Adv}^{\mathsf{SDH}}_{\mathbb{G},p}(\mathcal{B})$$

[4] For a detailed discussion, we refer to [FGSW16], which formally treat key confirmation in key exchange and introduced the notion of *almost-full key confirmation* to circumvent such issues, which is closely related to explicit authentication (cf. [dFW20]).

$$+ \mathrm{Adv}^{\text{MU-EUF-CMA}^{\text{corr}}}_{\mathsf{Sig}}(\mathcal{F}) + \frac{q_{\mathsf{Send}}}{2^{\lambda}} + \frac{1}{2}.$$

Hence, by Definition 5.2 it finally holds

$$\mathrm{Adv}^{\mathsf{pMSKE}}_{\mathtt{TLS-DHE}}(\mathcal{A}) \leq \frac{2q^2_{\mathsf{Send}}}{2^{256} \cdot p} + \frac{(q_{\mathsf{RO}} + 7q_{\mathsf{Send}})^2}{2^{\lambda}} + \mathrm{Adv}^{\mathsf{SDH}}_{\mathbb{G},p}(\mathcal{B}) + \mathrm{Adv}^{\text{MU-EUF-CMA}^{\text{corr}}}_{\mathsf{Sig}}(\mathcal{F}) + \frac{q_{\mathsf{Send}}}{2^{\lambda}}.$$

□

9.3.2 Tight Security Bound for the Full Handshake

With the tight security bound for the abstracted handshake, we can now deduce a tight security bound for the full (EC)DHE handshake as defined in Figure 6.1. In particular, this bound applies to the TLS 1.3 protocol that uses handshake traffic encryption and internal key (htk_C and htk_S), and only the hash function $\mathbf{H}$ is modeled as a random oracle RO_H. To deduce the this bound, we define the following intermediate protocols:

- KE is the protocol defined in Theorem 7.3, i.e., the TLS 1.3 full handshake protocol described in Figure 6.1 with $\mathbf{H} := \mathsf{RO}_H$ and **MAC**, **Extract**, and **Expand** defined from $\mathbf{H}$ as in Chapter 6 (with handshake encryption). This is the protocol, we give the final bound for.
- KE_1 is the protocol defined in Theorem 7.3 (as KE'), i.e., full handshake protocol described on the left-hand side of Figure 7.2, with $\mathbf{H} := \mathsf{RO}_{\mathsf{Th}}$ and $\mathsf{TKDF}_x := \mathsf{RO}_x$, where $\mathsf{RO}_{\mathsf{Th}}$, RO_{htk_C}, …, RO_{rms} are random oracles (with handshake encryption).
- KE_2 is the protocol defined in Theorem 8.2, i.e., full handshake protocol described on the left-hand side of Figure 7.2, with $\mathbf{H} := \mathsf{RO}_{\mathsf{Th}}$ and $\mathsf{TKDF}_x := \mathsf{RO}_x$, where $\mathsf{RO}_{\mathsf{Th}}$, RO_{htk_C}, …, RO_{rms} are random oracles as KE_1, but without handshake encryption. This is the protocol, we have proven tightly secure in Theorem 9.1.

Let $\mathcal{A}$ be any adversary against the pMSKE security of KE, then Theorem 7.3 grants that we can construct an adversary $\mathcal{B}_1$ such that

$$\mathrm{Adv}^{\mathsf{pMSKE}}_{\mathsf{KE}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{pMSKE}}_{\mathsf{KE}_1}(\mathcal{B}_1) + \frac{2(13q_{\mathsf{Send}} + q_{\mathsf{RO}})^2}{2^{\lambda}} + \frac{2q^2_{\mathsf{RO}}}{2^{\lambda}} + \frac{8(q_{\mathsf{RO}} + 36q_{\mathsf{Send}})^2}{2^{\lambda}}.$$

Next, we can apply Theorem 8.2, which gives us that we can construct an adversary $\mathcal{B}_2$ from $\mathcal{B}_1$ such that

$$\mathrm{Adv}^{\mathsf{pMSKE}}_{\mathsf{KE}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{pMSKE}}_{\mathsf{KE}_2}(\mathcal{B}_2) + \frac{2(13q_{\mathsf{Send}} + q_{\mathsf{RO}})^2}{2^{\lambda}} + \frac{2q^2_{\mathsf{RO}}}{2^{\lambda}} + \frac{8(q_{\mathsf{RO}} + 36q_{\mathsf{Send}})^2}{2^{\lambda}}.$$

Finally, we can use Theorem 9.1, which bounds the pMSKE security of KE_2 as follows. This means, we can construct adversaries $\mathcal{B}$ and $\mathcal{F}$ (from $\mathcal{B}_2$) such that

$$\mathrm{Adv}^{\mathsf{pMSKE}}_{\mathsf{KE}}(\mathcal{A}) \leq \frac{2q^2_{\mathsf{Send}}}{2^{256} \cdot p} + \frac{(q_{\mathsf{RO}} + 7q_{\mathsf{Send}})^2}{2^{\lambda}} + \mathrm{Adv}^{\mathsf{SDH}}_{\mathbb{G},p}(\mathcal{B}) + \mathrm{Adv}^{\text{MU-EUF-CMA}^{\text{corr}}}_{\mathsf{Sig}}(\mathcal{F})$$

$$
\begin{aligned}
&+ \frac{q_{\mathsf{Send}}}{2^\lambda} + \frac{2(13q_{\mathsf{Send}} + q_{\mathsf{RO}})^2}{2^\lambda} + \frac{2q_{\mathsf{RO}}^2}{2^\lambda} + \frac{8(q_{\mathsf{RO}} + 36q_{\mathsf{Send}})^2}{2^\lambda} \\
&= \mathrm{Adv}^{\mathsf{MU\text{-}EUF\text{-}CMA}}_{\mathsf{Sig}}(\mathcal{F}) + \mathrm{Adv}^{\mathsf{SDH}}_{\mathbb{G},p}(\mathcal{B}) + \frac{2q_{\mathsf{Send}}^2}{2^{256} \cdot p} \\
&+ \frac{q_{\mathsf{Send}} + (q_{\mathsf{RO}} + 7q_{\mathsf{Send}})^2 + 2(13q_{\mathsf{Send}} + q_{\mathsf{RO}})^2 + 2q_{\mathsf{RO}}^2 + 8(q_{\mathsf{RO}} + 36q_{\mathsf{Send}})^2}{2^\lambda}
\end{aligned}
$$

This yields the following final tight security bound for the TLS 1.3 full (EC)DHE handshake.

Corollary 9.1. *Let* KE *be the TLS 1.3 full (EC)DHE handshake protocol as specified in Figure 6.1. Let* $\mathbb{G}$ *be standardized group with order* p*. Let* Sig *be a standardized signature scheme. Let* $\lambda \in \mathbb{N}$ *be the output length in bits of the hash function* **H***. Further, let* **H** *be modeled as a random oracle* RO_H *and let* **MAC**, **Extract**, *and* **Expand** *defined from* **H** *as in Chapter 6. Then, for any adversary* $\mathcal{A}$*, we can construct adversaries* $\mathcal{B}$ *and* $\mathcal{F}$ *such that*

$$
\begin{aligned}
\mathrm{Adv}^{\mathsf{pMSKE}}_{\mathsf{KE}}(\mathcal{A}) \leq\ & \mathrm{Adv}^{\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathsf{corr}}}_{\mathsf{Sig}}(\mathcal{F}) + \mathrm{Adv}^{\mathsf{SDH}}_{\mathbb{G},p}(\mathcal{B}) + \frac{2q_{\mathsf{Send}}^2}{2^{256} \cdot p} \\
&+ \frac{q_{\mathsf{Send}} + (q_{\mathsf{RO}} + 7q_{\mathsf{Send}})^2 + 2(13q_{\mathsf{Send}} + q_{\mathsf{RO}})^2 + 2q_{\mathsf{RO}}^2 + 8(q_{\mathsf{RO}} + 36q_{\mathsf{Send}})^2}{2^\lambda}
\end{aligned}
$$

where q_{Send} *is the number of* Send *and* q_{RO} *the (total) number of random oracle queries issued by adversary* $\mathcal{A}$*, respectively.*

Remark 9.1. In the bound given in Corollary 9.1 one could further replace the SDH advantage of $\mathcal{B}$ by a generic group model (GGM) [Sho97, Mau05] bound. Abdalla, Bellare, and Rogaway [ABR01] originally proved in the GGM that any adversary making at most t group operation and DDH oracle queries have an advantage $\mathcal{O}(t^2/p)$, where p the prime order of the respective group. Davis and Günther [DG21a, DG20] revisited this result and bounded the SDH advantage from above by $4t^2/p$. In this work, we choose to give a more general bound, therefore we did not replace it above, but still we wanted to point to these results.

9.4 Discussion

In this chapter, we have proven a tight security bound for the TLS 1.3 full (EC)DHE handshake protocol as described in Figure 6.1. The proof is in the random oracle model (Section 4.1.2) and reduces the pMSKE-security (Definition 5.2) of TLS 1.3 full handshake to the multi-user EUF-CMA-security with adaptive corruptions (Definition 4.7) of the signature scheme used for authentication and to the SDH problem (Definition 3.5) in the Diffie–Hellman group. In the remainder, we discuss the result of this chapter and point to interesting open questions.

On the tightness of the standardized signatures. Unfortunately, even though the bound we are proving is tight, the TLS 1.3 full handshake is not fully tight. The reason for

this is that the standardized signature schemes supported by TLS 1.3 are not tightly multi-user-secure with adaptive corruptions. They are only single-user-secure and therefore there is an implicit loss that is linear in the number of users, because for any of the standardizes scheme it holds that for any (multi-user) adversary $\mathcal{F}$, we can construct an single-user adversary $\mathcal{F}'$ such that

$$\mathrm{Adv}^{\text{MU-EUF-CMA}^{\text{corr}}}_{\mathsf{Sig}}(\mathcal{F}) \leq \frac{1}{q_{\mathsf{New}}} \cdot \mathrm{Adv}^{\text{EUF-CMA}}_{\mathsf{Sig}}(\mathcal{F}')$$

where q_{New} is the number of users generated by $\mathcal{F}$. As already outlined in the introduction (Section 1.4), all schemes have unique keys (in the sense of [BJLS16]) and hence this loss seems to be unavoidable according to the impossibility result by Bader, Jager, Li, and Schäge [BJLS16]. In Part IV of this thesis, we present a signature scheme that is multi-user-secure with adaptive corruptions with a tight reduction in the random oracle model. When instantiated with DDH (Section 14.1), we result in a scheme with short signatures (only 3 $\mathbb{Z}_p$ elements). However, this scheme is not standardized and cannot be used with TLS 1.3 in practice. For future revisions of TLS it would be interesting to consider such signatures schemes, as well. Nevertheless, we consider it an interesting open question for future work to investigate whether it is possible (under which assumptions) to circumvent the impossibility result by [BJLS16]. A possible direction would be to try to find a relaxed notion of signature-security for which the impossibility result does not apply, but that still gives rise to a tight-security proof. Such a model could, for example, use that TLS 1.3 always signs messages with a special structure, i.e., hashes of transcripts. Clearly, this security notion needs to be strong enough to still give rise to a tight security proof for the TLS 1.3 handshake and thus had to be incorporated in our proof outline above.

On getting rid of the random oracle. Our proof is in the random oracle model, which is a commonly accepted heuristic for practical cryptographic schemes as discussed in Section 4.1.2. However, it would be interesting from a theoretical standpoint whether it is possible to achieve a tight security bound for the TLS 1.3 handshake protocol without relying on the random oracle model. Previous computational analyses of the TLS handshake (most prominently, for version 1.2 [JKSS12, KPW13] and for version 1.3 [DFGS21]), based security of the TLS handshake on the hardness of the PRF oracle-Diffie–Hellman (PRF-ODH) problem introduced by Jager, Kohlar, Schäge, and Schwenk [JKSS12] for the analysis of TLS-DHE version 1.2. The PRF-ODH problem is a variant of the oracle-Diffie–Hellman (ODH) problem introduced by Abdalla, Bellare, and Rogaway [ABR01]. Brendel, Fischlin, Günther, and Janson [BFGJ17] study the PRF-ODH problem in detail and even generalize it. In this context, they showed that the PRF-ODH assumption as introduced by in [JKSS12] can be instantiated with the SDH assumption in the random oracle model. Brendel, Fischlin, Günther, and Janson also claim that instantiating PRF-ODH without a random oracle might be challenging based on an impossibility result they are proving. However, this demonstrates that up to now we do not know whether the previous (non-tight) analyses of the TLS handshake that appear to be in the standard model (as opposed to the random oracle model) might actually be also implicitly

in the random oracle model. For future work, it would be interesting to investiagate this further. This could be done in various ways. First, it would be valuable to understand the relation between PRF-ODH and the random oracle model better, and follow-up on the work by Brendel, Fischlin, Günther, and Janson [BFGJ17]. Then, it is interesting to investigate the (im)possibility of tight reductions for TLS 1.3 in the standard model, in general. However, these two approaches might be linked, because previous analyses (e.g., [JKSS12, KPW13, DFGS21]) already indicate that PRF-ODH actually is exactly what we require from the TLS key derivation.

10

TIGHT SECURITY OF THE TLS-PSK HANDSHAKES

Author's contribution. The contents of this chapter are based on joint work with Hannah Davis, Felix Günther, and Tibor Jager [DDGJ22b, DDGJ22a]. While we discussed all aspects of this paper together, the author of this paper mainly developed the architecture of the proof presented in Section 10.3 and worked out most of its details. Nevertheless, Hannah Davis revised some parts of the proof. The proof of Section 10.3 incorporates techniques from [Coh+19, DJ21, DG21a] as already outlined for the author's contribution in Chapter 9 and uses the abstraction presented in Chapter 7. Since the majority of the proof of [DDGJ22a, Sect. 7.1] was written by the author of this thesis, the proof presented in Section 10.3 is almost verbatim from [DDGJ22a]. The same holds for the contents of Section 10.2, in which we model the PSK handshake as MSKE protocols.

Contents

10.1 Introduction

In this chapter, we present our tight security bound for the TLS 1.3 PSK handshakes. The technical approach is similar to the one we presented in Section 9.1.1 and only differs in details of the proof that are specific to the PSK mode of TLS 1.3.

Chapter outline. Analogously, to the treatment of the full handshake in Chapter 9, we first present in Section 10.2 the protocol-specific properties of the TLS 1.3 PSK-only and PSK-(EC)DHE handshakes to model them as symmetric-key MSKE (sMSKE) protocols as defined in Chapter 5. Subsequently, we prove in Section 10.3 our tight security bound for the TLS 1.3 PSK-(EC)DHE handshake in its abstracted form as obtained by the insights presented in Chapters 7 and 8, i.e., we prove the security bound for the protocol as presented on the right-hand side of Figure 7.2 without handshake encryption. Then, again applying Theorems 7.3 and 8.2 in Section 10.3.2 we obtain the final bound for the TLS 1.3 PSK-(EC)DHE handshake protocol as described in Section 6.4. Finally, we argue in Section 10.4 how the analysis given for the PSK-(EC)DHE would change for the PSK-only handshake without providing a full proof due to their similarity.

10.2 TLS 1.3 PSK-only and PSK-(EC)DHE Handshake as an MSKE Protocol

We begin by capturing the TLS 1.3 PSK-only and PSK-(EC)DHE handshake protocols, specified in Figures 6.2 and 7.2, formally as a sMSKE protocol. To this end, we must explicitly define the variables discussed in Chapter 5. In particular, we have to define the stages themselves, which stages are internal and which replayable, the session and contributive identifiers, when stages receive authentication, and when stages become forward secret.

Stages. The TLS 1.3 PSK-only/PSK-(EC)DHE handshake protocol has eight stages (i.e., $\mathsf{STAGES} = 8$), corresponding to the keys *ets*, *eems*, htk_S, htk_C, ats_C, ats_S, *ems*, and *rms* in that order. The set INT of internal keys contains htk_C and htk_S, the handshake traffic encryption keys. Stages *ets* and *eems* are replayable: $\mathsf{REPLAY}[s]$ is true for $s \in \{1, 2\}$ and false for all others.

Session and contributive identifiers. The session and contributive identifiers for stages are tuples $(label_s, ctxt)$, where $label_s$ is a unique label identifying stage s, and $ctxt$ is the transcript that enters key's derivation. The session identifiers $(sid[s])_{s\in\{1,\ldots,8\}}$ are defined as follows:[1]

$$
\begin{aligned}
sid[1] &= \left(\text{“}ets\text{”}, (\mathtt{CH}, \mathtt{CKS}^{\dagger}, \mathtt{CPSK})\right),\\
sid[2] &= \left(\text{“}eems\text{”}, (\mathtt{CH}, \mathtt{CKS}^{\dagger}, \mathtt{CPSK})\right),\\
sid[3] &= \left(\text{“}htk_C\text{”}, (\mathtt{CH}, \mathtt{CKS}^{\dagger}, \mathtt{CPSK}, \mathtt{SH}, \mathtt{SKS}^{\dagger}, \mathtt{SPSK})\right),\\
sid[4] &= \left(\text{“}htk_S\text{”}, (\mathtt{CH}, \mathtt{CKS}^{\dagger}, \mathtt{CPSK}, \mathtt{SH}, \mathtt{SKS}^{\dagger}, \mathtt{SPSK})\right),\\
sid[5] &= \left(\text{“}ats_C\text{”}, (\mathtt{CH}, \mathtt{CKS}^{\dagger}, \mathtt{CPSK}, \mathtt{SH}, \mathtt{SKS}^{\dagger}, \mathtt{SPSK}, \mathtt{EE}, \mathtt{SF})\right),\\
sid[6] &= \left(\text{“}ats_S\text{”}, (\mathtt{CH}, \mathtt{CKS}^{\dagger}, \mathtt{CPSK}, \mathtt{SH}, \mathtt{SKS}^{\dagger}, \mathtt{SPSK}, \mathtt{EE}, \mathtt{SF})\right),\\
sid[7] &= \left(\text{“}ems\text{”}, (\mathtt{CH}, \mathtt{CKS}^{\dagger}, \mathtt{CPSK}, \mathtt{SH}, \mathtt{SKS}^{\dagger}, \mathtt{SPSK}, \mathtt{EE}, \mathtt{SF})\right), \text{ and}
\end{aligned}
$$

[1] Components marked with † are only part of the TLS 1.3 PSK-(EC)DHE handshake.

$$sid[8] = \big(\text{“}rms\text{”}, (\texttt{CH}, \texttt{CKS}^{\dagger}, \texttt{CPSK}, \texttt{SH}, \texttt{SKS}^{\dagger}, \texttt{SPSK}, \texttt{EE}, \texttt{SF}, \texttt{CF})\big)\,.$$

To make sure that a server that received `ClientHello`, `ClientKeyShare`†, and `CPSK` untampered can be tested in stages 3 and 4, even if the sending client did not receive the server's answer, we set the contributive identifiers of stages 3 and 4 such that cid_{role} reflects the messages that a session in role *role* must have honestly received for testing to be allowed. Namely, we let clients (resp. servers) upon sending (resp. receiving) the messages $(\texttt{CH}, \texttt{CKS}^{\dagger}, \texttt{CPSK})$ set

$$cid_{\mathsf{responder}}[3] = \big(\text{“}htk_C\text{”}, (\texttt{CH}, \texttt{CKS}^{\dagger}, \texttt{CPSK})\big) \text{ and}$$
$$cid_{\mathsf{responder}}[4] = \big(\text{“}htk_S\text{”}, (\texttt{CH}, \texttt{CKS}^{\dagger}, \texttt{CPSK})\big)\,.$$

Further, when the client receives (resp. the server sends) the message $(\texttt{SH}, \texttt{SKS}^{\dagger}, \texttt{SPSK})$, they set

$$cid_{\mathsf{initiator}}[3] = sid[3] \quad \text{and} \quad cid_{\mathsf{initiator}}[4] = sid[4].$$

For all other stages $s \in \{1, 2, 5, 6, 7, 8\}$, $cid_{\mathsf{initiator}}[s] = cid_{\mathsf{responder}}[s] = sid[s]$ is set upon acceptance of the respective stage (i.e., when $sid[s]$ is set as well).

Authentication. As the PSK handshakes rely on a pre-shared key, which we assume to be exchanged in a previous TLS 1.3 session, we already have that every stage is at least implicitly mutually authenticated upon acceptance. This is because only the parties that took part in the previous session can know the PSK and therefore only the intended partner of a session knows any of its established key (except for potential corruption of the PSK). Thus, we only focus on explicit authentication in the PSK protocols.

For initiator sessions, all stages achieve explicit authentication after successfully verifying the `ServerFinished` message. This happens right before stage 5 (i.e., ats_C) is accepted. That is, upon accepting stage 5 all previous stages receive explicit authentication retroactively and all following stages are explicitly authenticated upon acceptance. Formally, we set $\mathsf{EAUTH}[\mathsf{initiator}, s] = 5$ for all stages $s \in \{1, \dots, 8\}$.

For responder session, all stages receive explicit authentication upon (successful) verification of the `ClientFinished` message. This occurs right before the acceptance of stage 8 (i.e., *rms*). Similar to initiators, responders receive explicit authentication for all stages upon acceptance of stage 8 since this is the last stage of the protocol. Accordingly, we set $\mathsf{EAUTH}[\mathsf{responder}, s] = 8$ for all stages $s \in \{1, \dots, 8\}$.

Forward secrecy. Only keys dependent on a Diffie–Hellman secret achieve forward secrecy, so all stages s of the PSK-only handshake have $\mathsf{FS}[r, s, \mathsf{fs}] = \mathsf{FS}[r, s, \mathsf{wfs2}] = \infty$ for both roles $r \in \{\mathsf{initiator}, \mathsf{responder}\}$. In the PSK-(EC)DHE handshake, full forward secrecy is achieved at the same stage as explicit authentication for all keys except *ets* and *eems*, which are never forward secret. That is, for both roles r and stages $s \in \{3, \dots, 8\}$ we have $\mathsf{FS}[r, s, \mathsf{fs}] = \mathsf{EAUTH}[r, s]$. All keys except *ets* and *eems* possess weak forward secrecy 2 upon acceptance, so we set $\mathsf{FS}[r, s, \mathsf{wfs2}] = s$ for stages $s \in \{3, \dots, 8\}$. Finally, as stages 1 and 2 (i.e., *ets* and *eems*) never achieve forward secrecy we set $\mathsf{FS}[r, s, \mathsf{fs}] = \mathsf{FS}[r, s, \mathsf{wfs2}] = \infty$ for both roles r and stages $s \in \{1, 2\}$.

10.3 Tight Security of TLS 1.3 PSK-(EC)DHE Handshake

We proceed analogously to Section 9.3. That is, we start this section by proving a tight security bound for the abstracted TLS 1.3 PSK-(EC)DHE handshake. With this bound, we successively apply the results from Chapters 7 and 8 to obtain a bound for the TLS 1.3 PSK-(EC)DHE handshake protocol as specified in Figure 6.2.

10.3.1 Tight Security Bound for the Abstracted PSK-(EC)DHE Handshake

In the first step, we prove the following theorem.

Theorem 10.1. *Let TLS-PSK-DHE be the TLS 1.3 PSK-(EC)DHE handshake protocol (with optional 0-RTT) as specified on the right-hand side in Figure 7.2 without handshake encryption. Let $\mathbb{G}$ be a standardized group of order p. Let λ be the output length in bits of $\mathbf{H}$, and let the pre-shared key space be $\mathsf{KE.PSKS} = \{0,1\}^\lambda$. Further, let $\mathbf{H}$ and TKDF_x for each $x \in \{binder, \ldots, rms\}$ be modeled as 12 independent random oracles $\mathsf{RO}_{\mathsf{Th}}, \mathsf{RO}_{binder}, \ldots, \mathsf{RO}_{rms}$. Then, for any adversary $\mathcal{A}$, we can construct an adversary $\mathcal{B}$ such that*

$$\mathrm{Adv}^{\mathsf{sMSKE}}_{\mathit{TLS-PSK-DHE}}(\mathcal{A}) \leq \frac{2q^2_{\mathsf{Send}}}{2^{256} \cdot p} + \frac{(q_{\mathsf{RO}} + q_{\mathsf{Send}})^2}{2^\lambda} + \frac{q^2_{\mathsf{NewSecret}}}{2^\lambda} + \frac{(q_{\mathsf{RO}} + 6q_{\mathsf{Send}})^2}{2^\lambda} + \frac{q_{\mathsf{RO}} \cdot q_{\mathsf{NewSecret}}}{2^\lambda} + \frac{q_{\mathsf{Send}}}{2^\lambda} + \mathrm{Adv}^{\mathsf{SDH}}_{\mathbb{G},p}(\mathcal{B})$$

where q_{Send} is the number of Send *and q_{RO} the (total) number of random oracle queries issued by adversary $\mathcal{A}$, respectively.*

Proof. To prove the bound, we make an incremental series of changes to the symmetric-key MSKE experiment $\mathrm{Exp}^{\mathsf{sMSKE}}_{\mathtt{TLS-PSK-DHE}}(\mathcal{A})$ defined in Figure 5.10. Due to the length of the upcoming proof, we divide the proof into three phases reflecting the three ways the adversary can win the security experiment:

1. We establish that the adversary $\mathcal{A}$ cannot violate predicate Sound.
2. We establish that the adversary $\mathcal{A}$ cannot violate predicate ExplAuth.
3. Finally, we ensure that all Test queries return uniformly random keys independent of the challenge bit b if predicate Fresh is not violated.

Then, we can conclude that the adversary $\mathcal{A}$ cannot do better then random guessing to win the security experiment implying an advantage of 0. In the following, let Game_δ denote the event that $\mathcal{A}$ in Game δ wins the game.

GAME 0. The initial game Game 0 is the sMSKE security experiment $\mathrm{Exp}^{\mathsf{sMSKE}}_{\mathtt{TLS-PSK-DHE}}(\mathcal{A})$ played for the TLS 1.3 PSK-(EC)DHE handshake (with optional 0-RTT) as specified in Figure 6.2 (right), but without handshake encryption. Note that the functions $\mathbf{H}$ and TKDF_x for $x \in \{binder, \ldots, rms\}$ are modeled as 12 independent random oracles $\mathsf{RO}_{\mathsf{Th}}, \mathsf{RO}_{binder}$, $\ldots, \mathsf{RO}_{rms}$. We implement random oracle RO_x by a look-up table ROList_x assigning inputs

to outputs.[2] By definition, we have

$$\Pr[\mathsf{Game}_0] := \Pr[\mathrm{Exp}^{\mathsf{sMSKE}}_{\mathtt{TLS-PSK-DHE}}(\mathcal{A}) = 1].$$

Phase 1: Ensuring Predicate Sound **cannot be violated**

GAME 1. In Game 1, we eliminate collisions among nonces and group elements computed by honest sessions via two new flags bad_C and $\mathsf{bad}_{C'}$, and abort the game if either is set. The flags bad_C and $\mathsf{bad}_{C'}$ are defined identically as in Game 1 defined in the proof of Theorem 9.1. Using an analogous line of argument, we get that

$$\Pr[\mathsf{Game}_0] \leq \Pr[\mathsf{Game}_1] + \frac{2q^2_{\mathsf{Send}}}{2^{256} \cdot p}. \tag{10.1}$$

GAME 2. In Game 2, we abort the game if there is a collision among the binder values computed by any honest session. Whenever two distinct queries to RO_{binder} return the same value, we set a flag bad_{binder} and abort the game if this flag is set.

To implement this, we add a table $\mathsf{CollList}_{binder}$ to the random oracle RO_{binder} (this table is currently redundant to the table implementing RO_{binder}, but will be useful in later game hops, where we will introduce changes such that it is not guaranteed anymore that all *binder* values will be contained in the RO_{binder} table). Whenever RO_{binder} computes a binder value $b = \mathsf{RO}_{binder}(psk, ctxt)$, we log $\mathsf{CollList}_{binder}[b] := (psk, ctxt)$. Now, whenever RO_{binder} computes some binder b for some tuple s and $\mathsf{CollList}_{binder}[b]$ is not empty, there has to be a tuple $s' = (psk, ctxt)$ with $\mathsf{RO}_{binder}(psk, ctxt) = b$ queried before and we have found a collision if $s \neq s'$. In this case we set bad_{binder}.

Then, it holds that

$$\Pr[\mathsf{Game}_1] \leq \Pr[\mathsf{Game}_2] + \Pr[\mathsf{bad}_{binder}].$$

Next, we bound the probability that the game sets flag bad_{binder}. If bad_{binder} is set, then this means that there is a collision in random oracle RO_{binder}. That is, there is a pair (s, s') with $s \neq s'$ such that $\mathsf{RO}_{binder}(s) = \mathsf{RO}_{binder}(s')$. We bound this probability using the birthday bound. There are at most $q_{\mathsf{RO}} + q_{\mathsf{Send}}$ many queries issued to RO_{binder}, because the adversary $\mathcal{A}$ issues at most q_{RO} queries to any of the random oracles and the protocol computes at most 1 binder value per Send query (i.e., per honest session). Therefore, we have at most $q_{\mathsf{RO}} + q_{\mathsf{Send}}$ many uniform and independent samples from $\{0,1\}^\lambda$ in RO_{binder}. Thus, we get that

$$\Pr[\mathsf{bad}_{binder}] \leq \frac{(q_{\mathsf{RO}} + q_{\mathsf{Send}})^2}{2^\lambda}.$$

Hence, we have that

$$\Pr[\mathsf{Game}_1] \leq \Pr[\mathsf{Game}_2] + \frac{(q_{\mathsf{RO}} + q_{\mathsf{Send}})^2}{2^\lambda}. \tag{10.2}$$

[2] As in the proof of Theorem 9.1, we assume that every of these look-up tables is implemented using a data structure that allows for constant time access when indexed either by inputs or outputs. An example for this data structure would be a hash table.

GAME 3. In Game 3, we set a flag bad_{PC} and abort the game whenever the NewSecret oracle samples a previously sampled pre-shared key (again). Formally, we set bad_{PC} if there exist two distinct tuples $(u, v, pskid)$ and $(u', v', pskid')$ with $\mathsf{pskeys}[(u, v, pskid)] = \mathsf{pskeys}[(u', v', pskid')]$. Since the pre-shared keys are uniformly distributed[3] on $\{0,1\}^\lambda$, by the birthday bound Then, we have that

$$\Pr[\mathsf{Game}_2] \leq \Pr[\mathsf{Game}_3] + \Pr[\mathsf{bad}_{PC}] \leq \Pr[\mathsf{Game}_3] + \frac{q^2_{\mathsf{NewSecret}}}{2^\lambda}.$$

Conclusion of Phase 1. At this point, we argue that in Game 3 and any subsequent games, adversary $\mathcal{A}$ cannot violate the Sound predicate without also causing the game to abort. If any check of Sound is satified, one of the flags we introduced in the previous games will be set and the game will be aborted. According to the definition of the MSKE game, there are seven events that cause the predicate Sound to be violated (see Figure 5.6). For the TLS 1.3 PSK handshakes only six of them are relevant, because the PSK provides already implicit mutual authentication and therefore there only is one authentication type. In the following, we argue why each of these events cannot occur in Game 3 and thus $\mathsf{Sound} = \mathsf{true}$ needs to hold from Game 3 on. The order presented above does not reflect order of events as they are presented in Figure 5.6, because some events depend on each other and for readability we decided to move them around.

1. *There are three honest sessions that have the same session identifier at any non-replayable stage.*

   Since the only replayable stages are stages 1 (*ets*) and 2 (*eems*), consider any later stage $s \geq 3$. Recall that session identifiers *sid* for all stages $s \geq 3$ contain a `ClientHello` message containing the initiator session's nonce and group element and a `ServerHello` message containing the responder session's nonce and group element (see Section 10.2). Every session's *sid* therefore contains its own randomly sampled nonce-group element pair. For three sessions to accept the same $sid[s]$ for $s \geq 3$, there must be two honest sessions who have sampled the same nonce and group element. Due to Game 1, this would trigger the bad_C flag, leading to abortion of the game.

2. *There are two sessions with the same session identifier in some non-replayable stage that have the same role.*

   Session identifiers $sid[s]$ for $s \geq 3$ as defined by TLS 1.3 (see Section 10.2) contain only one pair of nonce and group element per initiator and responder. If two honest sessions share a *sid* and a role, they must also share a nonce and group element. This case would also trigger the bad_C flag.

3. *There are two sessions with the same session identifier in some stage that do not share the same contributive identifier in that stage.*

[3] This can be generalized to a different distribution on $\{0,1\}^\lambda$, i.e., for any distribution D on $\{0,1\}^\lambda$, but then the denominator has to be adapted to 2^α, where α is the min-entropy of D.

Once a session holds both a contributive identifier and a session identifier for the same stage, both are equal by our definition (see Section 10.2) of the session and contributive identifiers for TLS 1.3. This case will therefore never occur.

4. *There are two sessions that hold the same session identifier for different stages.*

   This is impossible as the session identifier of stage s begins with the unique label $label_s$ for stage s.

5. *There are two honest sessions with the same session identifier in some stage that disagree on the identity of their peer or their pskid.*

   Two sessions which hold the same session identifier must necessarily agree on the value of the *binder*, which is part of the `ClientHello` message. In Game 2, we required the game is aborted if two queries to the oracle RO_{binder} collide. The two sessions must therefore also agree on the pre-shared key, which they obtained from the list pskeys. From Game 3, we have that the game is aborted if any two distinct entries in pskeys contain the same value. Therefore two sessions can obtain the same pre-shared key from pskeys only if they hold the same tuple $(u, v, pskid)$, meaning they agree on both the peer identities and the pre-shared key identity.

6. *Sessions with the same session identifier in some stage do not hold the same key in that stage.*

   We have just established that two sessions with the same session identifier must agree on the peer identities and *pskid* (contained in `CPSK` and `SPSK`), meaning they also share the same *psk*. Session identifiers for stages whose keys are derived from a Diffie–Hellman secret Z must include both Diffie–Hellman shares X and Y (contained in `CKS` and `SKS`). These shares uniquely determine the DH key Z. Besides that the session identifier also contains the context required to derive the respective stage keys, which then uniquely determines the stage key. Therefore, agreement on a session identifier implies agreement on a stage key.

Phase 2: Ensuring Predicate ExplAuth cannot be violated

GAME 4. In Game 4, we abort the game if two distinct queries to $\mathsf{RO}_{\mathsf{Th}}$ lead to colliding outputs. This ensures that each transcript has a unique hash. When such a collision occurs, we set a new flag bad_H and let the game abort. As in Game 2, we introduce a table $\mathsf{CollList}_{\mathsf{Th}}$ to random oracle $\mathsf{RO}_{\mathsf{Th}}$. Whenever it computes a hash $d = \mathsf{RO}_{\mathsf{Th}}(s)$ for some string s, we log $\mathsf{CollList}_{\mathsf{Th}}[d] := s$. This table then is used to set bad_H as in Game 2.

Analogously to Game 2, we bound the probability of bad_H to be set using the birthday bound. Random oracle $\mathsf{RO}_{\mathsf{Th}}$ is queried at most $q_{\mathsf{RO}} + 6q_{\mathsf{Send}}$ times. Namely at most q_{RO} queries issued by the adversary $\mathcal{A}$ and up to 6 queries for each session to compute the *distinct* transcript hash value. Therefore, we have that

$$\Pr[\mathsf{bad}_H] \leq \frac{(q_{\mathsf{RO}} + 6q_{\mathsf{Send}})^2}{2^\lambda}.$$

and it follows that

$$\Pr[\mathsf{Game}_3] \leq \Pr[\mathsf{Game}_4] + \frac{(q_{\mathsf{RO}} + 6q_{\mathsf{Send}})^2}{2^\lambda}.$$

GAME 5. In Game 5, we abort the game when the adversary queries any random oracle on a pre-shared key *psk before* that key has been corrupted via RevLongTermKey. We introduce some bookkeeping in order to implement this change. First, we add a reverse look-up table PSKList that is maintained by the NewSecret oracle. When $\mathsf{NewSecret}(u, v, pskid)$ samples a fresh pre-shared key *psk*, we log the tuple under index *psk* as $\mathsf{PSKList}[psk] := (u, v, pskid)$. Note that the pre-shared keys might repeat, so we may have multiple entries in PSKList indexed by a single *psk*. Second, we add a time log T to the 12 random oracles RO_x. Each random oracle query containing a pre-shared key *psk* now creates an entry $\mathsf{T}[psk] \leftarrow \mathsf{time}$, where time is the counter maintained by the key exchange experiment, unless $\mathsf{T}[psk]$ already exists.

The actual check whether the adversary queries any random oracle with a *psk* before it was corrupted is performed at the end of the game, when the game determines its output. We set a flag bad_{psk} if $\mathsf{T}(psk) \leq \mathsf{revpsk}_{(u,v,pskid)}$ for any $psk \in \mathsf{T}$ and $(u, v, pskid) \in \mathsf{PSKList}[psk]$. If the bad_{psk} flag was set during this process, then the game aborts (i.e., it outputs 0).

Next, let us analyze the probability that the game is lost due to flag bad_{psk} being set. Each random oracle query could hit one out of $q_{\mathsf{NewSecret}}$ many pre-shared keys. Before a given pre-shared key is corrupted or queried to a random oracle, the adversary learns nothing about its value. Since we assume that pre-shared keys are sampled uniformly at random from $\{0,1\}^\lambda$, the probability to hit a specific one is at most $2^{-\lambda}$.[4] By the union bound, we obtain that the probability that the adversary hits any of the pre-shared keys in a single random oracle query is upper-bounded by $\mathsf{NewSecret} \cdot 2^{-\lambda}$. Thus, the probability that bad_{psk} is set in response to any of the q_{RO} many random oracle queries overall is limited by $q_{\mathsf{RO}} \cdot q_{\mathsf{NewSecret}} \cdot 2^{-\lambda}$. This follows again by applying the union bound.

Hence, we get that

$$\Pr[\mathsf{Game}_4] \leq \Pr[\mathsf{Game}_5] + \Pr[\mathsf{bad}_{psk}] \leq \Pr[\mathsf{Game}_5] + \frac{q_{\mathsf{RO}} \cdot q_{\mathsf{NewSecret}}}{2^\lambda}.$$

In the next two games, we change the way that partnered sessions compute their session keys, *binder* values, and `Finished` MAC tags. Since we have established in Phase 1 that partnered sessions will always share the same key, we can compute these keys only once and let partnered sessions copy the results. This will make it easier to maintain consistency between partners as we change the way we compute keys and tags. This approach follows the tight key exchange security proof techniques of Cohn-Gordon et al. [Coh+19].

GAME 6. First, we will store all session keys in a look-up table SKEYS under their session identifiers. Sessions will be able to use this table to easily check if they share a session identifier with another honest session and thus share a key with a partner.

[4] Note that at this point, we use that the pre-shared key distribution is uniform. As already mentioned before, for any distribution $\mathcal{D}$ on $\{0,1\}^\lambda$, the probability would be $2^{-\alpha}$, where α is the min-entropy of $\mathcal{D}$.

Honest sessions π_u^i in the initiator role (clients) will derive the keys *ets*, *eems*, and *rms* before their partners. In Game 6, when an initiator session accepts in stage 1 (*ets*), 2 (*eems*), or 8 (*rms*) it creates a new entry in SKEYS, i.e.,

$$\mathsf{SKEYS}[\pi_u^i.sid[s]] \coloneqq \pi_u^i.skey[s]$$

for $s \in \{1, 2, 8\}$. Honest responder sessions π_v^j (servers) will derive the keys htk_S, htk_C, ats_C, ats_S, and *ems* before their partners. These sessions also log their keys in SKEYS under the appropriate session identifier:

$$\mathsf{SKEYS}[\pi_v^j.sid[s]] \coloneqq \pi_v^j.skey[s]$$

for $s \in \{3, \ldots, 7\}$.

Note that no two sessions will ever log keys in table SKEYS under the same *sid*. From Sound, we know that only one initiator and one responder session may have the same session identifier $sid[s]$ in any stage s. Note that for the replayable stages 1 and 2 (*ets* and *eems*) we only log once because the messages will only be logged by the initiator that output the replayed messages and not by the receivers that are receiving them.

We also store *binder*, fin_C and fin_S MAC tags. When any honest session queries RO_x with $x \in \{binder, fin_C, fin_S\}$, it logs the response in a second look-up table, TAGS, indexed by x and the inputs to RO_x. That is, for a query (psk, Z, d_1, d_2) to RO_{fin_S}, we log

$$\mathsf{TAGS}[fin_S, psk, Z, d_1, d_2] \coloneqq \mathsf{RO}_{fin_S}(psk, Z, d_1, d_2).$$

Since Game 6 only introduces book-keeping steps, we have that

$$\Pr[\mathsf{Game}_5] = \Pr[\mathsf{Game}_6].$$

GAME 7. In Game 7, we change the way the sessions compute their keys and MAC tags. Namely, if a session has an honest partner in stage s, instead of computing a key itself, it copies the stage-s key already computed by the partner via the table SKEYS introduced in Game 6. Concretely, the sessions compute their keys depending on their role as follows.

Honest server sessions. An honest server session π_v^j, upon receiving (CH, CKS, CPSK), sets its session identifier for stages 1 (*ets*) and 2 (*eems*). It then checks whether keys have been logged in SKEYS under $\pi_v^j.sid[1]$ and $\pi_v^j.sid[2]$. If such log entries exist, then π_v^j has an honest partner in stages 1 and 2, and copies the keys *ets* and *eems* from SKEYS when they would instead be computed directly.

Analogously, upon receiving CF, π_v^j uses SKEYS to check whether there is an honest client session that shares the same stage-8 (*rms*) session identifier $\pi_v^j.sid[8]$, and it copies the *rms* key if this is the case. If there are no entries in SKEYS under the appropriate session identifiers, π_v^j proceeds as in Game 6 and computes its keys using the random oracles.

Honest client sessions. An honest client session π_u^i, upon receiving (SH, SKS, SPSK), sets its session identifiers for stages 3–7, which identify the keys htk_S, htk_C, ats_C, ats_S and

ems. It then searches for entries in SKEYS indexed by $\pi_u^i.sid[s]$ for $s \in \{3, \ldots, 7\}$. If these entries are present for stage s, then π_v^i copies the stage-s keys from SKEYS instead of computing them itself. Otherwise, π_u^i proceeds as in Game 6 and computes the keys using the random oracle in each case.

Computation of MAC tags. Finally, all honest sessions (both client and server) which would query RO_x to compute $x \in \{binder, fin_C, fin_S\}$ in Game 6 first check the look-up table TAGS to see if their query has already been logged. If so, they copy the response from TAGS instead of making the query to RO_x.

It remains to argue that the procedure of copying the keys in partnered sessions described in this game is consistent with computing the keys in Game 6. Recall that sessions which are partnered in stage s must agree on the stage-s key, since the Sound predicate cannot be violated. Consider a session π_u^i which accepts the stage-s key $\pi_u^i.skey[s]$. By Sound, any other session π_v^j in Game 6 which accepts in stage s with $\pi_v^j.sid[s] = \pi_u^i.sid[s]$ must set its stage-s key equal to $\pi_u^i.skey[s]$. Although in Game 7 the session π_v^j may copy $\pi_u^i.skey[s]$ from table SKEYS instead of deriving it directly, the value of $\pi_v^j.skey[s]$ does not change between the two games.

Sessions may also copy queries from look-up table TAGS instead of making the appropriate random oracle query themselves. However, table TAGS simply caches the response to random oracle queries and does not change them. Hence, the view of the adversary is identical. This implies that

$$\Pr[\mathsf{Game}_6] = \Pr[\mathsf{Game}_7].$$

With the next two games, we finalize Phase 2. First, we postpone the sampling of the pre-shared key to the RevLongTermKey oracle such that only corrupted sessions hold pre-shared keys. As a consequence of this change, we can no longer compute session keys and MAC tags using the random oracles. We will instead sample these uniformly at random from their respective range and only program the random oracles upon corruption of the corresponding pre-shared key. After this change, we can show that in order to break explicit authentication, the adversary must predict a uniformly random `Finished` MAC tag, which is unlikely.

GAME 8. In Game 8, we postpone the sampling of pre-shared keys from the NewSecret oracle to the RevLongTermKey oracle (if the pre-shared key gets corrupted) or at the end of the game (if the key remains uncorrupted).

Since we now do not have a PSK anymore for uncorrupted sessions, we cannot use the random oracle to compute keys or MAC tags in those sessions, but instead sample them uniformly at random. If the corresponding pre-shared key is corrupted later and a PSK is chosen (in RevLongTermKey), we will retroactively program the affected random oracles to ensure consistency.

Concretely, we change the implementation of the game as follows. When NewSecret receives a query $(u, v, pskid)$, we set $\mathsf{pskeys}[(u, v, pskid)]$ to a special symbol $\star$ instead of a randomly chosen pre-shared key. The $\star$ serves as a placeholder and signalizes that the NewSecret oracle already received a query $(u, v, pskid)$, but no PSK has been chosen yet.

We add $(u, v, pskid)$ to the set $\mathsf{PSKList}[\star]$ to keep track of all tuples with an undefined PSK.

We let honest sessions whose pre-shared key has not been sampled (yet) but equals $\star$ sample their session keys as well as *binder* and `Finished` MAC tags uniformly at random. Due to the changes introduced in Game 7 we do not need to ensure consistency when sampling, as we sample each value once and partnered sessions copy the suitable value from the tables SKEYS and TAGS. (When sessions would log MAC tags in TAGS under their pre-shared keys in Game 7, those with no pre-shared key use the tuple $(u, v, pskid)$ instead in this game.) We further log the respective random oracle query that sessions would normally have used for the computation in a look-up table $\mathsf{PrgList}_x$ for later programming of the respective random oracle RO_x. Sessions which would log their RO-derived values in tables SKEYS and TAGS now log their randomly chosen values instead. That is, if a session in Game 7 would issue a query $(\star, Z, ctxt)$ (where Z might be $\perp$, e.g., for *ets* and *eems*) to random oracle RO_x to compute a value k, in Game 8 it chooses k uniformly at random from RO_x's range and logs

$$\mathsf{PrgList}_x[(u, v, pskid), Z, ctxt] := k$$

in the look-up table $\mathsf{PrgList}_x$, where $(u, v, pskid)$ uniquely identifies the used *psk*. Note that the table $\mathsf{PrgList}_x$ is closely related to the random oracle table ROList_x for RO_x. Table $\mathsf{PrgList}_x$ is always used when there is no *psk* defined for a session, i.e., it has not (yet) been corrupted. Therefore, we need to make sure that if the *psk* (identified by $(u, v, pskid)$) gets corrupted we are able to reprogram RO_x. Using $\mathsf{PrgList}_x$ we can upon corruption of the pre-shared key associated with $(u, v, pskid)$ efficiently look-up the entries we need to program from $\mathsf{PrgList}_x$ and transfer them to the random oracle table ROList_x after *psk* has been set. We will discuss the precise process below when we describe how to adapt the RevLongTermKey oracle.

We must be particularly careful when $x = binder$, because we still wish to set the bad_{binder} flag when two randomly chosen binder values collide. Therefore, honest sessions still record the sampled binder values in list $\mathsf{CollList}_{binder}$, so that the bad_{binder} flag is set as before. This ensures that the probability of setting the flag does not change.

We also need to adapt the corruption oracle RevLongTermKey. Upon a query $(u, v, pskid)$ for which $\mathsf{pskeys}[(u, v, pskid)] = \star$, we perform the following additional steps: First, we sample a fresh pre-shared key $psk \xleftarrow{\$} \mathsf{KE.PSKS}$ and update pskeys, i.e., set $\mathsf{pskeys}[(u, v, pskid)] := psk$. Next, we need to reprogram the random oracles using the lists $\mathsf{PrgList}_x$ to ensure consistency. Thus, for all x we update the random oracle tables ROList_x for RO_x using $\mathsf{PrgList}_x$ as follows. For every entry $\mathsf{PrgList}_x[((u, v, pskid), Z, ctxt)] = k$, we set

$$\mathsf{ROList}_x[psk, Z, ctxt] := k$$

where ROList_x is the random oracle table of RO_x. Lastly, we remove $(u, v, pskid)$ from the set $\mathsf{PSKList}[\star]$ and add it to $\mathsf{PSKList}[psk]$.

To be able to still set bad_{psk}, we also make sure that at the end of the game every pre-shared key is defined before the check against the random oracle time log T introduced in Game 5. We sample a pre-shared key for every tuple $(u, v, pskid) \in \mathsf{PSKList}[\star]$,

setting $\mathsf{pskeys}[(u, v, pskid)] \xleftarrow{\$} \mathsf{KE.PSKS}$, and update the reverse look-up table PSKList accordingly. As a result, also uncorrupted sessions now have a pre-shared key defined and we can check the condition for bad_{psk} being set as introduced in Game 5.

The changes introduced in Game 8 are unobservable for the adversary as it never queries the random oracle for an uncorrupted pre-shared key, as otherwise the game would be aborted due to bad_{psk} introduced in Game 5. It hence does not matter whether the pre-shared key is already set before or upon corruption, because from the view of the adversary the keys (and the pre-shared key) are uniformly random bitstrings anyway up to this point. Upon corruption of a pre-shared key, we make sure by reprogramming the random oracle that all session keys and MAC tag computations are consistent with sessions that would have otherwise used this pre-shared key but derived all session keys and MAC tags without it. The procedure at the end of the game does not affect the view of the adversary as it only retroactively defines keys on which the adversary cannot get any information about anymore. Consequently,

$$\Pr[\mathsf{Game}_7] = \Pr[\mathsf{Game}_8].$$

GAME 9 (Exclude that honest sessions accept without a partner). In Game 9, we set a flag $\mathsf{bad}_{\mathrm{MAC}}$ and abort the game if any session with an uncorrupted pre-shared key accepts stage 5 (htk_C) as initiator, or stage 8 (rms) as responder, without having a partnered session. Formally, we set $\mathsf{bad}_{\mathrm{MAC}}$ if there is a session π_u^i such that $\pi_u^i.\mathsf{accepted}[s] < \mathsf{revpsk}_{(u,v,\pi_u^i.pskid)}$ with $v = \pi_u^i.pid$ and

$$s = \begin{cases} 5 & \text{if } \pi_u^i.role = \mathsf{initiator} \\ 8 & \text{if } \pi_u^i.role = \mathsf{responder} \end{cases}$$

and there is no session π_v^j with $\pi_u^i.sid[s] = \pi_v^j.sid[s]$ when π_u^i accepts stage s.

Let us analyze the probability $\Pr[\mathsf{bad}_{\mathrm{MAC}}]$. Consider a session π_u^i which triggers the $\mathsf{bad}_{\mathrm{MAC}}$ flag. In the following analysis, let π_u^i be an initiator. For responder sessions the arguments are analogous. The pre-shared key of session π_u^i is uncorrupted, which means that by the changes of Game 8 it has not been sampled. Therefore π_u^i either samples the `ServerFinished` MAC tag uniformly at random or copies it from table TAGS (in which case the MAC tag was uniformly sampled and logged by another honest session). First observe that session π_u^i will not copy the `ServerFinished` MAC tag from table TAGS as this would imply that π_u^i is partnered when it accepts in stage 5. This in turn contradicts that π_u^i has triggered flag $\mathsf{bad}_{\mathrm{MAC}}$. Namely, if π_u^i would be able to copy the `ServerFinished` MAC tag from table TAGS there must have been another honest session that computed the same `ServerFinished` MAC, i.e., using the same tuple $(u, v, pskid)$, DHE secret, and transcript hash. Recall that the session identifier of stage 5 contains both the `ServerFinished` message and the transcript hashed to computed the `ServerFinished` MAC tag. Further, we have that transcript hashes are unique due to Game 4. This implies that the session that logged the `ServerFinished` MAC tag in TAGS needs to have the same stage-5 session identifier than π_u^i meaning π_u^i would be partnered in stage 5.

Thus, if π_u^i triggers $\mathsf{bad}_{\mathrm{MAC}}$, it must have sampled its `ServerFinished` MAC tag at random. Hence, the probability that π_u^i triggers the flag $\mathsf{bad}_{\mathrm{MAC}}$ is bounded by $2^{-\lambda}$. Namely, the probability that the sampled value matches the received `ServerFinished` message. A union bound over all sessions gives

$$\Pr[\mathsf{bad}_{\mathrm{MAC}}] \leq \frac{q_{\mathsf{Send}}}{2^{\lambda}}.$$

Overall, we get that

$$\Pr[\mathsf{Game}_8] \leq \Pr[\mathsf{Game}_9] + \Pr[\mathsf{bad}_{\mathrm{MAC}}] \leq \Pr[\mathsf{Game}_9] + \frac{q_{\mathsf{Send}}}{2^{\lambda}}.$$

Conclusion of Phase 2. At this point, we argue that in Game 9 and any subsequent games, adversary $\mathcal{A}$ cannot violate the ExplAuth predicate without also causing the game to abort. To this end, we argue that ExplAuth = true holds with certainty from Game 9 on.

The predicate ExplAuth is set to false if there is a session π_u^i accepting an explicitly authenticated stage s, whose pre-shared key was not corrupted before accepting the stage s' in which it received (perhaps retroactively) explicit authentication, and (1) there is no honest session π_v^j partnered to π_u^i in stage s', or (2) there is an honest partner session π_v^j for π_u^i in stage s' but it accepts with a peer identity $w \neq u$, with a different pre-shared key identity than π_u^i, i.e. $\pi_v^j.pskid \neq \pi_u^i.pskid$, or with a different stage-s session identifier, i.e. $\pi_v^j.sid[s] \neq \pi_u^i.sid[s]$.

Recall that initiator (resp. responder) sessions receive explicit authentication with acceptance of stage 5 (resp. stage 8) meaning that all previous stages 1–4 (resp. stages 1–7) receive explicit authentication retroactively and all future stages 6–8 upon their acceptance. From Game 9, we have that any initiator session π_u^i accepting stage 5 (resp. any responder session accepting stage 8) with uncorrupted *psk* must have a partnered session in that stage. Consequently, case (1) is impossible to achieve.

We next address the possibility of case (2). To achieve explicit authentication for stage $s \leq 8$, a responder session must have accepted stage 8. From Game 9 on, we know that π_u^i must have a partner with the same stage 8 session identifier. Observe that the transcripts contained in π_u^i's session identifiers for all stages are "sub-transcripts" of the transcript contained in the session identifier of stage 8. Therefore the partner must also have the same stage s session identifier. Now, since the Sound predicate cannot be violated in Game 9, we have that all partnered sessions agree on the peer identity and the pre-shared key identity, so ExplAuth is not violated by session π_u^i. The same property holds for initiator sessions accepting stages $s \leq 5$. So ExplAuth can only be violated if an initiator session's stage-5 partner accepts in stage $s > 5$ with a different peer identity, pre-shared key identifier, or session ID. Since peer and pre-shared key identifiers do not change after they are set, only the session identifiers may not match in stage s. The "sub-transcripts" of stage 6 (ats_C) and 7 (ats_S) session identifiers are identical to those of stage 5, so a partner in stage 5 will also be a partner in stages 6 and 7. Then the only way to violate predicate ExplAuth is to convince the stage-5 partner, a responder session, to accept a forged `ClientFinished` message and accept stage 8. This is impossible because

the partner will verify the received `ClientFinished` message against the message sent by π_u^i, which it copies from table TAGS. It follows that no session, responder or initiator, can violate the ExplAuth predicate.

Phase 3: Ensuring that the Challenge Bit is Independently Random

GAME 10. In this game, we rule out that the adversary manages to guess the DHE secret of two honestly partnered session to learn about the keys they are computing. Here, we only look at those session that have a corrupted pre-shared key, because we already ruled out in Game 5 that the adversary learns something about the keys computed by these sessions. To that end, we add another flag $\mathsf{bad}_{\mathrm{DHE}}$ to the game and abort the game when it is set. Flag $\mathsf{bad}_{\mathrm{DHE}}$ is set if the adversary ever queries a random oracle

$$\mathsf{RO}_x(psk, Z, \mathsf{RO}_{\mathsf{Th}}(sid[s]))$$

for $(x, s) \in \{(htk_C, 3), (htk_S, 4), (fin_S, 5)(ats_C, 5), (ats_S, 6), (ems, 7), (fin_C, 8), (rms, 8)\}$ such that

- psk is corrupted, i.e., the adversary made a prior query $\mathsf{RevLongTermKey}(u, v, pskid)$ with $\mathsf{pskeys}[u, v, pskid] = psk$,
- there are honest sessions π_u^i and π_v^j that are contributively partnered in stage s with $\pi_v^j.cid_{\pi_u^i.role}[s] = \pi_u^i.cid_{\pi_u^i.role}[s] = (\mathtt{CH}, \mathtt{CKS}, \mathtt{CPSK}, \mathtt{SH}, \mathtt{SKS}, \mathtt{SPSK}, \ldots)$, and
- $Z = g^{xy}$ such that $\mathtt{CKS} = g^x$ and $\mathtt{SKS} = g^y$.[5]

We bound the probability of flag $\mathsf{bad}_{\mathrm{DHE}}$ being set via a reduction $\mathcal{B}$ to the strong Diffie–Hellman assumption in group $\mathbb{G}$. Reduction $\mathcal{B}$ simulates Game 10 for $\mathcal{A}$, and it wins the strong Diffie–Hellman whenever the simulated game would set the $\mathsf{bad}_{\mathrm{DHE}}$ flag.

Construction of reduction $\mathcal{B}$. The reduction $\mathcal{B}$ gets as input a strong DH challenge $(A = g^a, B = g^b)$ as well as access to the oracle $\mathrm{DDH}_a := \mathrm{DDH}(g^a, \cdot, \cdot)$ for the Decisional Diffie–Hellman problem with the first argument fixed. Adversary $\mathcal{B}$ then honestly executes the RevSessionKey, Test, and NewSecret oracles as well as the setup and the computation of the output as Game 10 would, managing all game variables itself. We explain in more detail how $\mathcal{A}$ answers Send, RevLongTermKey, and random oracle queries.

When $\mathcal{A}$ makes a query to the Send oracle, $\mathcal{B}$ delivers the message to a protocol session in the same way as Game 10. However, the sessions themselves handle messages quite differently. At a high level, $\mathcal{B}$ embeds its strong DH challenges into the key shares of every initiator session and every partnered responder session. When $\mathsf{bad}_{\mathrm{DHE}}$ is triggered, $\mathcal{B}$ learns the Diffie–Hellman secret Z associated with two of these embedded key shares, and it can extract a solution to the strong DH challenge using some basic algebra. However, $\mathcal{B}$ must take care to appropriately program random oracles queries after corruptions, since it cannot compute Diffie–Hellman secrets for embedded key shares as it does not

[5] Note that the game knows the exponents x and y used by the sessions, but the reduction constructed in the remainder will not.

know the corresponding exponents. Next, we describe how client and server sessions are implemented in Game 10.

But first we explain the (constant-time accessible) look-up tables that are used (or defined) by reduction $\mathcal{B}$ to ensure an efficient implementation:

- The look-up table KSRnd is maintained for all sessions. It holds the random exponent τ used by the honest sessions to randomize their key share G, indexed by the session's nonce and key share (r, G) (see the implementation of the session for further details). To identify a session uniquely we use its nonce r and key share G as the index.
- Each random oracle RO_x maintains a look-up table $\mathsf{DHEList}_x$. For each query $\mathsf{RO}_x(psk, Z, d)$, the table stores the group element Z indexed by psk and d.
- Each random oracle RO_x maintains a look-up table $\mathsf{RndList}_x$. It holds a tuple $(\tau, \tau'$, $ctxt, key)$ indexed by the pair (psk, d). The table holds all necessary information that is required to reprogram of the random oracle RO_x. The fields psk and key can hold special values. If a psk is uncorrupted, we cannot log the information under it because it is not defined. Therefore, we can use the tuple $(u, v, pskid)$ uniquely identifying psk instead. Moreover, key can sometimes be an empty field, because reprogramming of that value will never occur. When this field is empty, it will not be accessed as we instead use the remaining information of $\mathsf{RndList}_x$ to solve the SDH challenge. See the remainder of the proof for details.

Implementation of honest server sessions. Consider any server session π_v^j.

1. Upon receiving $(\mathtt{CH}, \mathtt{CKS}, \mathtt{CPSK})$, the reduction $\mathcal{B}$ first checks whether π_v^j has an honest partner in stages 1 (*ets*) and 2 (*eems*) by checking for entries indexed by $\pi_v^j.sid[1]$ and $\pi_v^j.sid[2]$ in the look-up table SKEYS introduced in Game 6. If no such entries exist, then $\mathcal{B}$ answers this and all future Send queries just as specified in Game 10. For the rest of the discussion, we assume the entries do exist.

   Session π_v^j generates its key share $\mathtt{SKS}$ by randomizing the challenge key share B. Namely, it chooses a randomizer $\tau_v^j \xleftarrow{\$} \mathbb{Z}_p$ uniformly at random and sets $Y := B \cdot g^{\tau_v^j}$. Then, it logs τ_v^j under index (r_S, Y) in the look-up table KSRnd.
2. Before π_v^j outputs $(\mathtt{SH}, \mathtt{SKS}, \mathtt{SPSK})$, it computes the keys htk_C and htk_S. By Game 8, these keys are sampled randomly when psk is uncorrupted and computed using RO_{htk_C}, resp. RO_{htk_S} otherwise. In both cases, $\mathcal{B}$ needs to know the Diffie–Hellman secret Z to log in table $\mathsf{PrgList}_x$ or to query RO_x for $x \in \{htk_C, htk_S\}$. However, $\mathcal{A}$ cannot compute Z because it does not know the discrete logarithms of either $\mathtt{CKS}$ or $\mathtt{SKS}$. Therefore, $\mathcal{B}$ needs to compute the keys without knowing the DHE key using the control over the random oracles. Server π_v^j first computes htk_C. If the pre-shared key has been corrupted, the adversary could potentially have already queried the random oracle RO_{htk_C} with the query π_v^j should make. To that end, $\mathcal{B}$ first checks whether the corresponding query for htk_C was already made to RO_{htk_C}. Concretely, $\mathcal{B}$ computes the context hash $d = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \cdots \parallel \mathtt{SPSK})$ and checks for

a suitable RO_{htk_C} query using the look-up table $\mathsf{DHEList}_{htk_C}[psk, d]$ maintained in RO_{htk_C} (see above for the definition). Reduction $\mathcal{B}$ queries $\mathsf{DDH}_a(Y, Z \cdot Y^{-\tau_u^i})$ for all $Z \in \mathsf{DHEList}_{htk_C}[psk, d]$, where τ_u^i is the randomizer used by the honest partner of π_v^j, which can be looked up from $\mathsf{KSRnd}[r_C, X]$ using π_u^i's nonce and key share. [6] If any one of these queries is answered positively, we have by the definition of DDH_a that $Z \cdot Y^{-\tau_u^i} = Y^a$, which implies that $Z = Y^{a+\tau_u^i} = X^{b+\tau_v^j}$ by definition of Y and X, which was computed by the honest partner π_u^i that has output the CH message received by π_v^j. This exactly is the Z value that π_v^j would have computed if we would have known the discrete logarithm of B. Hence, we have found the right Z value and only need to derandomize it to win the challenge. Therefore, we let $\mathcal{B}$ submit the value

$$Z \cdot Y^{-\tau_u^i} \cdot A^{-\tau_v^j} = Y^a \cdot A^{-\tau_v^j} = (g^a)^{b+\tau_v^j} \cdot (g^a)^{-\tau_v^j} = g^{ab}$$

as a solution to the strong Diffie–Hellman problem.

Observe that if $\mathsf{bad}_{\mathrm{DHE}}$ is set due to a query to RO_{htk_C} in Game 10, there is a random oracle query such that one of the above DDH_a queries will be answered positively. Thus, $\mathcal{B}$ will win if $\mathsf{bad}_{\mathrm{DDH}}$ is set. We do the same for htk_S with RO_{htk_S}.

If in the above process no query is answered positively, i.e., $\mathsf{bad}_{\mathrm{DDH}}$ will also not be set, then π_v^j samples the key $htk_C \xleftarrow{\$} \{0,1\}^{l+d}$ itself and logs the following information so that future RO queries can be answered appropriately:

$$\mathsf{RndList}_{htk_C}(psk, d = H(\mathtt{CH} \parallel \cdots \parallel \mathtt{SPSK})) := \left(\tau_u^i, \tau_v^j, (\mathtt{CH} \parallel \cdots \parallel \mathtt{SPSK}), \bot\right).$$

Again, we do the same for htk_S.

If psk is not corrupted, then $\mathsf{bad}_{\mathrm{DDH}}$ cannot possibly have been set and we do not need to worry about consistency with earlier random oracle queries. Therefore, we do not need to do the process described above and immediately sample htk_C and htk_S randomly as in Game 10. It logs the keys in table SKEYS under their respective session identifiers, which do not contain Z or any unknown values. In Game 10, we added entries to $\mathsf{PrgList}_{htk_C}$ and $\mathsf{PrgList}_{htk_S}$ in order to program future random oracle queries upon corruption. The reduction cannot do this here as it does not know Z; instead, it logs

$$\mathsf{RndList}_x[((u, v, pskid), d = H(\mathtt{CH} \parallel \cdots \parallel \mathtt{SPSK}))] := \left(\tau_u^i, \tau_v^j, (\mathtt{CH} \parallel \cdots \parallel \mathtt{SPSK}), \bot\right).$$

for $x \in \{htk_C, htk_S\}$. This will allow $\mathcal{B}$ to win if a later RevLongTermKey or random oracle query triggers $\mathsf{bad}_{\mathrm{DDH}}$.

3. To compute the ServerFinished message $\mathcal{B}$ proceeds exactly as in Step 2 except that it uses the random oracle RO_{fin_S} and context $\mathtt{CH} \parallel \cdots \parallel \mathtt{EE}$ through the

[6] Although this may cause several DDH queries in response to a single Send query, $\mathcal{B}$ is still "efficient" because it only checks random oracle queries whose context is d, and due to the lack of both nonce/group element and hash collisions d is unique to session π_u^i and its partner. Therefore each entry in $\mathsf{DHEList}_{htk_C}[psk, d]$ will be checked at most twice over the course of the entire reduction.

`EncryptedExtensions`. Also, the `ServerFinished` message is computed first by the server, so $\mathcal{B}$ does not check table SKEYS or TAGS for any entries. Reduction $\mathcal{B}$ also cannot log the inputs to random oracle query RO_{fin_S} in table TAGS (as done since game Game 6) because it does not know Z. Instead, it logs the derived value of fin_S in table TAGS and replaces Z in the index of TAGS by $\big(\tau_u^i, \tau_v^j, (\mathtt{CH} \parallel \cdots \parallel \mathtt{EE})\big)$. That is, if it computes fin_S for inputs psk, d_1, and d_2, it logs

$$\mathsf{TAGS}[fin_S, psk, (\tau_u^i, \tau_v^j, (\mathtt{CH} \parallel \cdots \parallel \mathtt{EE})), d_1, d_2] := fin_S.$$

That way, it is possible to identify Z without knowing it. For fin_S, we keep the same notation for the sets $\mathsf{DHEList}_x$, $\mathsf{RndList}_x$ and ROList_x numbered as the corresponding random oracle RO_x.

4. Reduction $\mathcal{B}$ proceeds exactly as for fin_S above, except that we again use different random oracles and the context $sid_{ats_C} = \mathtt{CH} \parallel \cdots \parallel \mathtt{SF} = sid_{ats_S} = sid_{ems}$, where sid_x denotes transcript contained in the session identifier which is prefixed by "x", and thus the hash $d = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \cdots \parallel \mathtt{SF})$. With respect to random oracles, we have RO_{ats_C} for ats_C, RO_{ats_S} for ats_S and RO_{ems} for ems, respectively. Reduction $\mathcal{B}$ logs the keys in table SKEYS under their respective session identifiers, which do not contain Z or any unknown values.

   After this is done, π_v^j outputs $(\mathtt{EE}, \mathtt{SF})$.

5. Upon receiving `CF`, $\mathcal{B}$ looks for a suitable entry for fin_C in TAGS. If there is a value fin_C consistent with π_v^j's view, $\mathcal{B}$ terminates the session as specified if `CF` does not match the looked-up value of fin_C. Otherwise, $\mathcal{B}$ continues to compute rms. To this end, $\mathcal{B}$ checks whether there is an entry in SKEYS that matches the stage-8 session identifier of π_v^j, if yes π_v^j simply copies that entry. If not, first observe that if there is no entry in SKEYS there is no honest stage-8 partner, which implies that psk needs to be corrupted as otherwise the game would have been aborted due to $\mathsf{bad}_{\mathrm{MAC}}$ introduced in Game 9. Therefore, the adversary also would be allowed to query RO_{rms} to compute rms. Thus, $\mathcal{B}$ needs to check whether the value for rms is already set. Here, we need to distinguish two cases. Namely, whether there is an honest contributive stage-3 partner or not.

   First note that as described in Step 1, $\mathcal{B}$ does not embed its challenge in `SKS` if there is no honest session output the `ClientHello` received, i.e., there is no honest contributive stage-3 partner. Therefore, here $\mathcal{B}$ can simply implement π_v^j as specified in Game 10.

   In case there is an honest contributive stage-3 partner, then $\mathcal{B}$ proceeds as described in Step 2 for oracle RO_{rms} and context hash $d = \mathsf{RO}_{\mathsf{Th}}(sid_{rms}) = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \cdots \parallel \mathtt{CF})$ to check whether the adversary already solved the SDH challenge for $\mathcal{B}$. Note that the stage-3 session identifier uniquely defines the DH key, thus if there is an honest partner and there is a respective RO_{rms} query, the adversary has to break SDH to submit the query.

Implementation of honest client sessions. Consider any client session π_u^i.

1. The reduction $\mathcal{B}$ proceeds exactly as in Game 10 until the session chooses its key share. Instead of choosing a fresh exponent as specified in Figure 6.2, it chooses a value $\tau_u^i \xleftarrow{\$} \mathbb{Z}_p$ uniformly at random and sets $X := A \cdot g^{\tau_u^i}$ as its key share in the `ClientKeyShare` message. Further, it logs τ_u^i in KSRnd indexed with (r_C, X). The rest is exactly as specified in Game 10. That is, it computes *ets* and *eems* and outputs $(\mathtt{CH}, \mathtt{CKS}, \mathtt{CPSK})$.
2. Upon receiving $(\mathtt{SH}, \mathtt{SKS}, \mathtt{SPSK})$, π_u^i checks whether there is an entry

$$\mathsf{SKEYS}[(\text{"}htk_C\text{"}, \mathtt{CH}, \ldots, \mathtt{SPSK})] \neq \bot.$$

   If this is the case, π_u^i knows that there is an honest stage-3 partner, and it copies all the keys stored under π_u^i's session identifier as defined in Game 10. If there is no suitable entry, $\mathcal{B}$ faces the problem that it already "committed" to not knowing the discrete logarithm of π_u^i's key share X by embedding A into it and thus we are not able to compute the DHE value. Since there is no entry in SKEYS for htk_C, we know that there is no honest stage-3 partner session by definition of SKEYS. That is, no honest server session computed $\mathtt{SKS}$ and thus it must have been chosen by the adversary. If the pre-shared key is corrupted, $\mathcal{B}$ needs to use the DDH_a oracle to check whether there already was a query issued to RO_x for $x \in \{htk_C, htk_S\}$. If this is not the case, π_u^i freshly samples random keys and remembers them for possible retroactive reprogramming of the random oracle. Concretely, we do the following for each random oracle RO_x for $x \in \{htk_C, htk_S\}$:

   First compute $d = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \ldots \parallel \mathtt{SPSK})$ and then query the DDH_a oracle for all $Z \in \mathsf{DHEList}_x[psk, d]$, where psk is the pre-shared key used by π_u^i, as

$$\mathrm{DDH}_a(Y, Z \cdot Y^{-\tau_u^i}) = 1 \iff Z = Y^a,$$

   where Y is the DH key share contained in $\mathtt{SPSK}$. See the server session implementation above for further explanation. If there is any of these queries is answered positively, let the respective key be $\mathsf{RO}_x(psk, Z, d)$. If there is no Z that results in a positive query, let $key \xleftarrow{\$} \{0,1\}^\lambda$ be sampled at random, and $\mathcal{B}$ logs the value for possible later reprogramming of the random oracle RO_x, i.e.,

$$\mathsf{RndList}_x[(psk, d = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \cdots \parallel \mathtt{SPSK}))] := \left(\tau_u^i, \bot, (\mathtt{CH} \parallel \cdots \parallel \mathtt{SPSK}), key\right).$$

   After that π_v^i either has copied the keys or chose them itself and will accept all of the stage keys among these keys.

   If the psk of π_u^i has not been corrupted, then no "right" query can have been made and the keys be sampled randomly. However, we still need to program future "right" RO queries after a corruption. Therefore set

$$\mathsf{RndList}_x[(psk, d = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \cdots \parallel \mathtt{SPSK}))] := \left(\tau_u^i, \bot, (\mathtt{CH} \parallel \cdots \parallel \mathtt{SPSK}), key\right).$$

   $\mathsf{PrgList}_x$ is not updated as in Game 10, because Z is unknown.

3. Upon receiving (EE, SF), similar to the previous step, π_v^j checks whether there is an entry in SKEYS and TAGS (to verify SF) corresponding to its stage-5 session identifier. If this is the case, it copies the keys from that list. In case there is none, we have that there is no honest stage-5 partner. Here, we need to distinguish the case whether there was an honest stage-3 partner before or not.

   Namely, the adversary could corrupt the *psk*, then change the EE output by an honest session and then compute a new SF message for the changed transcript. Hence, there is an honest stage-3 partner, but no stage-5 partner. In this case, $\mathcal{B}$ again applies the approach from above (see implementation of server session, Step 2) for the random oracles RO_x for $x \in \{ats_C, ats_S, ems\}$ and the context $d = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \cdots \parallel \mathtt{SF})$ checking whether the random oracles received already a correct query which set the keys ats_C, ats_S and ems. If this is the case and since there was a stage-3 partner, $\mathcal{B}$ has embedded the DH challenge in both the client and the server, this solves the strong Diffie–Hellman problem. When there is no such query the keys are chosen at random and all necessary information for possible retroactive programming of the random oracles RO_x is logged in the table $\mathsf{RndList}_x$. Please see above for details.

   However, if there is no honest stage-3 partner, SKS was chosen by the adversary. Hence, $\mathcal{B}$ needs to apply the procedure described in the previous step (Step 2) and use the oracle DDH_a to check the random oracles RO_x for $x \in \{ats_C, ats_S, ems\}$ whether they already set the keys. The important difference here is that a positive answer of the DDH_a oracle does not solve stDH, as SKS was chosen by the adversary. Note that $\mathcal{B}$ again needs to make sure that it gathers all the information needed to make retroactive programming of the random oracles possible by logging information in $\mathsf{RndList}_x$ as before.

4. π_u^i computes fin_C using the same process as above: if *psk* is corrupted, it checks for RO queries in $\mathsf{DHEList}_{fin_C}[psk, d]$ that could set $\mathsf{bad}_{\mathrm{DHE}}$ when π_u^i has an honest partner in stage 8 or fix the value of fin_C when no honest partner exists. It then solves SDH or sets fin_C accordingly. If no earlier RO query matches fin_C, then we sample fin_C randomly and log τ_u^i, fin_C, and the transcript in table $\mathsf{RndList}_{fin_C}$ under *psk* and the transcript hash d. If *psk* is uncorrupted, π_u^i immediately samples fin_C randomly and logs τ_u^i, fin_C, and the transcript in $\mathsf{RndList}_{fin_C}$ under index $((u, v, pskid), d)$.

   Next we compute *rms*. As π_u^i is not able to compute Z independent of there being a honest stage-3 partner or not, $\mathcal{B}$ need to apply the same procedure that was described before in Step 3, when there was no stage-5 partner for random oracle RO_{rms} and context $d = \mathsf{RO}_{\mathsf{Th}}(\mathtt{CH} \parallel \cdots \parallel \mathtt{CF})$. The only difference is that in case there was a stage-3 partner, a SDH solution is output when the DDH oracle returns true, and if there is no stage-3 partner, *rms* is only programmed. Then, π_u^i outputs CF.

Besides changing the implementation of the session oracles, we also need to adapt the random oracles RO_x for $x \in \{htk_C, \ldots, rms\}$ to make sure (1) $\mathcal{B}$ programs the random

oracle retroactively if the random oracle receives the right query and (2) to check whether the adversary computed Z for $\mathcal{A}$ for honestly partnered sessions.

Implementation of random oracle RO_x. If RO_x receives a query that was already answered, it answers consistently. However, if there is a new query of the form (psk, Z, d), it appends Z to the set $\mathsf{DHEList}_k[psk, d]$. If $\mathsf{RndList}_k[psk, d] \neq \bot$, then there already was a session using psk and context hash d trying to compute a key without knowing the correct DHE secret. Therefore, $\mathcal{B}$ uses the DDH_a oracle to check whether Z is that secret. Let $(\tau_u^i, \tau_v^j, ctxt, key)$ be the entry of $\mathsf{RndList}_k[psk, d]$, where τ_u^i and τ_v^j denote the randomness used by the client and the server to randomize the DDH challenge, respectively, $ctxt = \mathtt{CH} \parallel \mathtt{CKS} \parallel \mathtt{CPSK} \parallel \mathtt{SH} \parallel \mathtt{SKS} \parallel \mathtt{SPSK} \parallel \cdots$ denotes the context such that $d = \mathsf{RO}_{\mathsf{Th}}(ctxt)$ and key denotes the key chosen by the session since there was no random oracle fixing it. Using this information, it fetches $\mathtt{SKS} = Y$ and queries $\mathrm{DDH}_a(Y, Z \cdot Y^{-\tau_u^i})$. If this query is answered positively, $\mathcal{B}$ knows that the right DH value Z was queried. If $\tau_u^j = \bot$, i.e., the log in $\mathsf{RndList}_k$ was set by a client without an honestly partnered server, $\mathcal{B}$ needs to program the random oracle to be consistent. That is, $\mathsf{ROList}_k[psk, Z, d] := key$. Otherwise, $\mathcal{B}$ knows that the $\mathsf{PrgList}_x$ entry was set by an honestly partnered session, and thus Z is a randomized solution to the DDH challenge. Thus, $\mathcal{B}$ submits the solution $Z \cdot Y^{-\tau_u^i} \cdot A^{-\tau_v^j}$ to its SDH instance.

Unless $\mathcal{B}$ solved the SDH challenge, the oracle outputs $\mathsf{ROList}_x[psk, Z, d]$.

Implementation of corruption oracle RevLongTermKey. Finally, $\mathcal{B}$ needs to handle corruptions via the RevLongTermKey oracle. Since Game 8, the RevLongTermKey oracle upon input $(u, v, pskid)$ samples a fresh psk. It then uses lists $\mathsf{PrgList}_x$ to program all the random oracles RO_x for consistency with any sessions whose pre-shared key is now psk. Reduction $\mathcal{B}$ still does this, but in our reduction, the lists $\mathsf{PrgList}_x$ are no longer comprehensive. Some sessions fix the outputs of RO_x on some query without knowing the DHE input to that query. These sessions create log entries in $\mathsf{RndList}_x$, not $\mathsf{PrgList}_x$, and the entries have indices of the form $((u, v, pskid), d)$. $\mathcal{B}$ cannot use these entries to program past RO_x queries, but this is not necessary since any past RO_x query containing psk would set the bad_{psk} flag and cause the game to abort. $\mathcal{B}$ also cannot program future queries because we still do not know the DHE key. Instead, $\mathcal{B}$ just updates each matching entry in $\mathsf{PrgList}_x$ so that its index is (psk, d) instead of $((u, v, pskid), d)$. Future RO_x queries containing psk will then handle strong DH checking and programming for $\mathcal{B}$.

By the considerations above, we have that if $\mathsf{bad}_{\mathrm{DHE}}$ is set the $\mathcal{B}$ wins the SDH challenge. Then, by the considerations above

$$\Pr[\mathsf{Game}_9] \leq \Pr[\mathsf{Game}_{10}] + \Pr[\mathsf{bad}_{\mathrm{DHE}}] \leq \Pr[\mathsf{Game}_{10}] + \mathrm{Adv}^{\mathsf{SDH}}_{\mathbb{G},p}(\mathcal{B}).$$

Conclusion of Phase 3. We finally argue that the adversary's probability in determining the challenge bit b in Game 10 is at most $\frac{1}{2}$ if the Fresh predicate is true. First, recall that $\mathsf{Fresh} = \mathsf{true}$ implies no session can be tested and revealed in the same stage, and a tested session's partner may also be neither tested nor revealed in that stage. In the following, we refer to a session being "fresh" in a stage if this session does not violate the

conditions defined in the predicate Fresh in that stage. The Fresh predicate depends on the level of forward secrecy reached at the time of each Test query. First, if a session is tested in a non-forward secret stage, it remains only fresh if the *psk* was never corrupted. Second, if a session is tested in a weak forward secret 2 stage s, it remains fresh if the *psk* was never corrupted or if there is a contributive partner in stage s. Lastly, if a session is tested on a forward secret stage s, it remains fresh the *psk* was corrupted after forward secrecy was established for that stage (perhaps retroactively) or if there is a contributive partner.

Next, we argue for each level of forward secrecy that all tested keys in Game 10 which do not violate Fresh are uniformly and independently distributed from the view of the adversary. For the non-forward secret stages 1 (*ets*) and 2 (*eems*), the adversary cannot corrupt the *psk* of all sessions that it queried Test on stage 1 or 2. Since Game 8, we sample all session keys derived from uncorrupted pre-shared keys uniformly at random, or copy uniformly random keys from SKEYS. That is, the key returned by the Test query is a uniformly random key independent of the challenge bit b. Therefore, it cannot learn anything about either *ets* nor *eems* of any session with an uncorrupted key, and thus the response of a Test query will be a uniformly random string independent of the challenge bit b from the view of the adversary.

All other stages, i.e., stages 3–8, are weak forward secret 2 upon acceptance and become forward secret as soon as the session achieves explicit authentication. If the pre-shared key is never corrupted, we have by the same arguments given for the non-forward secret stages that the adversary receives a uniformly random key in response to the Test query independent of the challenge bit.

It remains to argue that the same is true if there is a contributive partner and the *psk* is corrupted. In this case, the adversary would need to make a random oracle query that triggers $\mathsf{bad}_{\mathrm{DHE}}$ introduced in Game 10 and would cause the game to abort. Without such a query the respective key is just a uniformly and independently distributed bitstring from the adversary's view. Hence, without losing the game, the adversary cannot learn anything about a weak forward secret 2 key, and thus it does not learn anything from the response of the Test query.

Since forward secret stages are weak forward secret 2 until explicit authentication is established, we only consider the case that a session that is tested on a weak forward secret 2 stage was corrupted after forward secrecy has been (retroactively) established. As we only establish forward secrecy after explicit authentication has been achieved, we can be sure due to ExplAuth never being violated that there is a partnered session for that stage. Hence, there also is a contributive partner and by the same arguments as given before the adversary would trigger $\mathsf{bad}_{\mathrm{DHE}}$ and lose the game before it can learn something about the session.

Overall, we have that the adversary $\mathcal{A}$ in Game 10 cannot gain any information on the challenge bit b without violating any of the predicates Sound, ExplAuth, or Fresh. Thus, the probability that Game 10 returns 1 is no greater than 1/2. Formally,

$$\Pr[\mathsf{Game}_{10}] \leq \frac{1}{2}.$$

Collecting all the terms, we get the final bound

$$\Pr[\mathrm{Exp}^{\mathsf{sMSKE}}_{\mathtt{TLS\text{-}PSK\text{-}DHE}}(\mathcal{A}) = 1] \leq \frac{2q^2_{\mathsf{Send}}}{2^{256} \cdot p} + \frac{(q_{\mathsf{RO}} + q_{\mathsf{Send}})^2}{2^\lambda} + \frac{q^2_{\mathsf{NewSecret}}}{2^\lambda} + \frac{(q_{\mathsf{RO}} + 6q_{\mathsf{Send}})^2}{2^\lambda} + \frac{q_{\mathsf{RO}} \cdot q_{\mathsf{NewSecret}}}{2^\lambda} + \frac{q_{\mathsf{Send}}}{2^\lambda} + \mathrm{Adv}^{\mathsf{SDH}}_{\mathbb{G},p}(\mathcal{B}) + \frac{1}{2}$$

which implies that

$$\mathrm{Adv}^{\mathsf{sMSKE}}_{\mathtt{TLS\text{-}PSK\text{-}DHE}}(\mathcal{A}) \leq \frac{2q^2_{\mathsf{Send}}}{2^{256} \cdot p} + \frac{(q_{\mathsf{RO}} + q_{\mathsf{Send}})^2}{2^\lambda} + \frac{q^2_{\mathsf{NewSecret}}}{2^\lambda} + \frac{(q_{\mathsf{RO}} + 6q_{\mathsf{Send}})^2}{2^\lambda} + \frac{q_{\mathsf{RO}} \cdot q_{\mathsf{NewSecret}}}{2^\lambda} + \frac{q_{\mathsf{Send}}}{2^\lambda} + \mathrm{Adv}^{\mathsf{SDH}}_{\mathbb{G},p}(\mathcal{B})$$

□

10.3.2 Tight Security Bound for the PSK-(EC)DHE Handshake

To deduce a tight security bound for the PSK-(EC)DHE handshake as defined in Figure 6.2, we proceed as in Section 9.3.2. In particular, this bound applies to the protocol that uses handshake traffic encryption and internal key (htk_C and htk_S), and only the hash function $\mathbf{H}$ is modeled as a random oracle RO_H. To deduce the this bound, we define the following intermediate protocols:

- KE is the protocol defined in Theorem 7.4, i.e., the TLS 1.3 PSK-(EC)DHE handshake protocol described in Figure 6.2 with $\mathbf{H} := \mathsf{RO}_H$ and **MAC**, **Extract**, and **Expand** defined from $\mathbf{H}$ as in Chapter 6 (with handshake encryption). This is the protocol, we give the final bound for.
- KE_1 is the protocol defined in Theorem 7.4 (as KE'), i.e., the PSK-(EC)DHE handshake protocol described on the right-hand side of Figure 7.2, with $\mathbf{H} := \mathsf{RO}_{\mathsf{Th}}$ and $\mathsf{TKDF}_x := \mathsf{RO}_x$, where $\mathsf{RO}_{\mathsf{Th}}$, RO_{binder}, …, RO_{rms} are random oracles (with handshake encryption).
- KE_2 is the protocol defined in Theorem 8.3, i.e., PSK-(EC)DHE handshake protocol described on the right-hand side of Figure 7.2, with $\mathbf{H} := \mathsf{RO}_{\mathsf{Th}}$ and $\mathsf{TKDF}_x := \mathsf{RO}_x$, where $\mathsf{RO}_{\mathsf{Th}}$, RO_{binder}, …, RO_{rms} are random oracles as KE_1, but without handshake encryption. This is the protocol `TLS-PSK-DHE`, we have proven tightly secure in Theorem 10.1.

Let $\mathcal{A}$ be any adversary against the sMSKE security of KE and let q_{RO} and q_{Send} be the number of queries issued by $\mathcal{A}$ to the random oracle RO (in total) and oracle q_{Send}, respectively, then Theorem 7.4 grants that we can construct an adversary $\mathcal{B}_1$ such that

$$\mathrm{Adv}^{\mathsf{sMSKE}}_{\mathsf{KE}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{sMSKE}}_{\mathsf{KE}_1}(\mathcal{B}_1) + \frac{2(12q_{\mathsf{Send}} + q_{\mathsf{RO}})^2}{2^\lambda} + \frac{2q^2_{\mathsf{RO}}}{2^\lambda} + \frac{8(q_{\mathsf{RO}} + 36q_{\mathsf{Send}})^2}{2^\lambda}.$$

Applying Theorem 8.3 yields that we can construct an adversary $\mathcal{B}_2$ from $\mathcal{B}_1$ such that

$$\mathrm{Adv}^{\mathsf{sMSKE}}_{\mathsf{KE}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{sMSKE}}_{\mathsf{KE}_2}(\mathcal{B}_2) + \frac{2(12q_{\mathsf{Send}} + q_{\mathsf{RO}})^2}{2^\lambda} + \frac{2q^2_{\mathsf{RO}}}{2^\lambda} + \frac{8(q_{\mathsf{RO}} + 36q_{\mathsf{Send}})^2}{2^\lambda}.$$

Finally, applying Theorem 10.1 yields that we can construct an adversary $\mathcal{B}$ such that

$$\begin{aligned}
\mathrm{Adv}^{\mathsf{sMSKE}}_{\mathsf{KE}}(\mathcal{A}) &\leq \frac{2q^2_{\mathsf{Send}}}{2^{256}\cdot p} + \frac{(q_{\mathsf{RO}}+q_{\mathsf{Send}})^2}{2^\lambda} + \frac{q^2_{\mathsf{NewSecret}}}{2^\lambda} + \frac{(q_{\mathsf{RO}}+6q_{\mathsf{Send}})^2}{2^\lambda} \\
&\quad + \frac{q_{\mathsf{RO}}\cdot q_{\mathsf{NewSecret}}}{2^\lambda} + \frac{q_{\mathsf{Send}}}{2^\lambda} + \mathrm{Adv}^{\mathsf{SDH}}_{\mathbb{G},p}(\mathcal{B}) + \frac{2(12q_{\mathsf{Send}}+q_{\mathsf{RO}})^2}{2^\lambda} \\
&\quad + \frac{2q^2_{\mathsf{RO}}}{2^\lambda} + \frac{8(q_{\mathsf{RO}}+36q_{\mathsf{Send}})^2}{2^\lambda} \\
&= \frac{2(12q_{\mathsf{Send}}+q_{\mathsf{RO}})^2 + 2q^2_{\mathsf{RO}} + 8(q_{\mathsf{RO}}+36q_{\mathsf{Send}})^2}{2^\lambda} + \frac{2q^2_{\mathsf{Send}}}{2^{256}\cdot p} + \mathrm{Adv}^{\mathsf{SDH}}_{\mathbb{G},p}(\mathcal{B}) \\
&\quad + \frac{(q_{\mathsf{RO}}+q_{\mathsf{Send}})^2 + q^2_{\mathsf{NewSecret}} + (q_{\mathsf{RO}}+6q_{\mathsf{Send}})^2 + q_{\mathsf{RO}}\cdot q_{\mathsf{NewSecret}} + q_{\mathsf{Send}}}{2^\lambda}
\end{aligned}$$

Formally, we get the following final result for the TLS 1.3 PSK-(EC)DHE handshake protocol.

Corollary 10.1. *Let* KE *be the TLS 1.3 PSK-(EC)DHE handshake protocol as specified in Figure 6.2. Let* $\mathbb{G}$ *be standardized group with order* p*. Let* $\lambda \in \mathbb{N}$ *be the output length in bits of the hash function* $\mathbf{H}$*. Further, let* $\mathbf{H}$ *be modeled as a random oracle* RO_H *and let* $\mathbf{MAC}$*,* $\mathbf{Extract}$*, and* $\mathbf{Expand}$ *defined from* $\mathbf{H}$ *as in Chapter 6. Then, for any adversary* $\mathcal{A}$*, we can construct an adversary* $\mathcal{B}$ *such that*

$$\begin{aligned}
\mathrm{Adv}^{\mathsf{sMSKE}}_{\mathsf{KE}}(\mathcal{A}) &\leq \frac{2(12q_{\mathsf{Send}}+q_{\mathsf{RO}})^2 + 2q^2_{\mathsf{RO}} + 8(q_{\mathsf{RO}}+36q_{\mathsf{Send}})^2}{2^\lambda} + \frac{2q^2_{\mathsf{Send}}}{2^{256}\cdot p} + \mathrm{Adv}^{\mathsf{SDH}}_{\mathbb{G},p}(\mathcal{B}) \\
&\quad + \frac{(q_{\mathsf{RO}}+q_{\mathsf{Send}})^2 + q^2_{\mathsf{NewSecret}} + (q_{\mathsf{RO}}+6q_{\mathsf{Send}})^2 + q_{\mathsf{RO}}\cdot q_{\mathsf{NewSecret}} + q_{\mathsf{Send}}}{2^\lambda}
\end{aligned}$$

where q_{Send} *is the number of* Send *and* q_{RO} *the (total) number of random oracle queries issued by adversary* $\mathcal{A}$*, respectively.*

10.4 Tight Security of the TLS 1.3 PSK-only Handshake

In this section, we briefly discuss how the tight security proof for the TLS 1.3 PSK-(EC)DHE handshake presented in Section 10.3.1 can be adapted to the PSK-only handshake. Then, we state the theorems for the abstracted TLS 1.3 PSK-only handshake as specified on the right-hand side in Figure 7.2 without handshake encryption and finally, for the TLS 1.3 PSK-only handshake as presented in Figure 6.2.

On adapting our PSK-(EC)DHE proof to PSK-only. The structure and the resulting tight security bound is in essence the same. However, there is of course the significant difference that the PSK-only mode does perform a Diffie–Hellman key exchange and therefore, there is no DH group and the messages `ClientHello` and `ServerHello` are different in that they do not contain a group element as key share (i.e., the messages `ClientKeyShare` and `ServerKeyShare` are not present). Consequently, we do not have a reduction to the SDH problem in Game 10. The main consequence of the absence of

the Diffie–Hellman values is that none of the keys can be forward secret. An adversary that gets hold of a user's pre-shared key can compute (resp. recompute) all previous and future session keys of that user.

Technically, the security proof for the TLS 1.3 PSK-only handshake would yield almost the same sequence of games as presented in Section 10.3.1. In particular, it would consist of the Games 0 to 9 (only excluding Game 10, which excluded the adversary guessing the DHE key). Games 2 to 9 remain almost identically ignoring everything that is related to the Diffie–Hellman values (e.g., in indexes). The only significant change is in Game 1, in which we exclude collision among the nonces and group elements chosen by honest sessions. In the PSK-only mode honest sessions do not sample a group element, and therefore Game 1 for the PSK-only mode only focuses on collisions among the nonces. Thus, the bound for Game 0 in the PSK-only mode is

$$\Pr[\mathsf{Game}_0] \leq \Pr[\mathsf{Game}_1] + \frac{2q_{\mathsf{Send}}^2}{2^{256}}.$$

We then obtain a similar result as presented in Section 10.3.1 for the abstracted TLS 1.3 PSK-only handshake as presented in the right-hand side of Figure 7.2 without handshake encryption.

Theorem 10.2. *Let TLS-PSK be the TLS 1.3 PSK-only handshake protocol (with optional 0-RTT) as specified on the right-hand side in Figure 7.2 without handshake encryption. Let λ be the output length in bits of* $\mathbf{H}$*, and let the pre-shared key space be* $\mathsf{KE.PSKS} = \{0,1\}^\lambda$*. Further, let* $\mathbf{H}$ *and* TKDF_x *for each* $x \in \{binder, \ldots, rms\}$ *be modeled as* 12 *independent random oracles* $\mathsf{RO}_{\mathsf{Th}}, \mathsf{RO}_{binder}, \ldots, \mathsf{RO}_{rms}$*. Then, for any adversary* $\mathcal{A}$ *it holds that*

$$\mathrm{Adv}^{\mathsf{sMSKE}}_{TLS\text{-}PSK}(\mathcal{A}) \leq \frac{2q_{\mathsf{Send}}^2}{2^{256}} + \frac{(q_{\mathsf{RO}} + q_{\mathsf{Send}})^2}{2^\lambda} + \frac{q_{\mathsf{NewSecret}}^2}{2^\lambda} + \frac{(q_{\mathsf{RO}} + 6q_{\mathsf{Send}})^2}{2^\lambda} + \frac{q_{\mathsf{RO}} \cdot q_{\mathsf{NewSecret}}}{2^\lambda} + \frac{q_{\mathsf{Send}}}{2^\lambda}$$

where q_{Send} *is the number of* Send *and* q_{RO} *the (total) number of random oracle queries issued by adversary* $\mathcal{A}$*, respectively.*

Following along the arguments of Section 10.3.2, we obtain the following final result for the TLS 1.3 PSK-only handshake as described in Figure 6.2.

Corollary 10.2. *Let* KE *be the TLS 1.3 PSK-only handshake protocol as specified in Figure 6.2. Let* $\lambda \in \mathbb{N}$ *be the output length in bits of the hash function* $\mathbf{H}$*. Further, let* $\mathbf{H}$ *be modeled as a random oracle* RO_H *and let* **MAC**, **Extract**, *and* **Expand** *defined from* $\mathbf{H}$ *as in Chapter 6. Then, for any adversary* $\mathcal{A}$ *it holds that*

$$\mathrm{Adv}^{\mathsf{sMSKE}}_{\mathsf{KE}}(\mathcal{A}) \leq \frac{2(12q_{\mathsf{Send}} + q_{\mathsf{RO}})^2 + 2q_{\mathsf{RO}}^2 + 8(q_{\mathsf{RO}} + 36q_{\mathsf{Send}})^2}{2^\lambda} + \frac{2q_{\mathsf{Send}}^2}{2^{256}} + \frac{(q_{\mathsf{RO}} + q_{\mathsf{Send}})^2 + q_{\mathsf{NewSecret}}^2 + (q_{\mathsf{RO}} + 6q_{\mathsf{Send}})^2 + q_{\mathsf{RO}} \cdot q_{\mathsf{NewSecret}} + q_{\mathsf{Send}}}{2^\lambda}$$

where q_{Send} *is the number of* Send *and* q_{RO} *the (total) number of random oracle queries issued by adversary* $\mathcal{A}$*, respectively.*

10.5 Discussion

In this chapter, we have proven a tight security bound for the TLS 1.3 PSK-only/PSK-(EC)DHE handshake protocol as described in Figure 6.2. The proof is in the random oracle model (Section 4.1.2). In the remainder, we discuss the result of this chapter and point to interesting open questions.

Differences between the full handshake analysis. The first thing we discuss is the difference between the full handshake analysis and the analysis for the PSK handshakes. We only focus on the PSK-(EC)DHE handshake since the PSK-(EC)DHE handshake also relies on a Diffie–Hellman key exchange. The general outline of the proofs are quite similar. Of course, due to the absence of digital signature, there is no reduction to the security of the signature scheme. Most significantly, in the PSK-(EC)DHE analysis presented in Section 10.3.1 we could split the proof up into three phases, ensuring that Sound cannot be violated, ExplAuth cannot be violated, and finally that the adversary can only guess the challenge bit b if Fresh remains valid. In the full handshake analysis Section 9.3, we could only argue about the latter two phases together in the end of the proof. The reason for this ultimately is the PSK. In the PSK handshake, the PSK obviously adds another secret to the key derivation. Therefore, an adversary that cannot guess the PSK of uncorrupted sessions will intuitively not be able to get any information about the keys exchanged, and thus also the MAC values, computed by honest sessions. Informally speaking in the PSK handshake analysis the final SDH reduction preventing the adversary from guessing the DHE key from corrupted sessions is merely to exclude one specific attack. Namely, that the adversary remains passive in an honest session or does not corrupt its PSK upon acceptance ensuring that forward secrecy is not violated, which then would cause Fresh to be violated (cf. Figure 5.7). Then, the adversary could corrupt the PSK of this session (after acceptance) and if it is able to predict the corresponding DHE key computed by that session, it can guess the challenge bit with certainty. In the full handshake analysis this is different. Due to the absence of the PSK the whole security of the keys, and MAC values, rely on the secrecy of the DHE keys. Thus, to even argue that an adversary cannot predict *Finished* messages of honest sessions, we need to exclude that the adversary is able to guess the DHE key. That is, the SDH reduction intuitively is already a necessary condition to argue that ExplAuth cannot be violated in the full handshake in contrast to the PSK handshakes.

Pre-shared keys. Secondly, we discuss the types of pre-shared keys we consider for our analysis. Recall that TLS 1.3 supports "resumption keys", i.e., keys that are derived from the resumption master secret *rms* exchanged in a previous run of the full handshake (cf. Section 6.5). Additionally, it supports "out-of-band" keys, which PSKs that are established externally. For the resumption keys it is reasonable to assume that they are uniform on $\{0,1\}^\lambda$, where λ is the output length of the TLS hash function, due to the way they are derived. However, for out-of-band keys that is not generally the case. First, they do not necessarily need to be of length λ (in bits). Second, they do not necessarily need to follow some (high-entropy) distribution. We note that the results of Chapter 7 regarding

the indifferentiability of the key schedule highly relies on the PSKs being of length λ. Nevertheless, it would be interesting how (if at all) our analyses would change if one includes the consideration of out-of-band PSKs. In general, we would expect that the results would generalize to any distribution on $\{0,1\}^\lambda$ that satisfy the length requirement, but do not have full entropy. This applies in particular to the bounds for Game 2 and 4, where we already remarked how they could be adapted based on the min-entropy of the PSK distribution. However, we remark that lower-entropy PSK distributions result in weaker bounds, due to the increased chance for collisions between PSKs as well as the adversary guessing a PSK.

Part III

On the Tightness of the TLS 1.3 Record Protocol

11

ON THE TIGHTNESS OF THE TLS 1.3 RECORD PROTOCOL

In this chapter, we briefly discuss the tightness of the TLS 1.3 record protocol. We do not give a formal treatment in this thesis, but we discuss the current state for completeness. In this context, we point to open questions for future work. We start with a short overview of the TLS 1.3 record protocol, which is specified in [Res18, Sect. 5]. Then, we discuss the steps we consider necessary to prove tight security for the TLS 1.3 record protocol.

TLS 1.3 Record Protocol. The record protocol is the part of TLS that handles the secure channel that TLS aims to provide. In particular, it takes the messages that a user wants to send, fragments this data into blocks (*records*), protects these records and transmits the protected data. The same goes for the reverse direction. That is, it takes the protected records, decrypts them, reassembles the fragments to the original messages and delivers these messages to a higher level (e.g., the handshake protocol or an application layer protocol) to be processed. Each of the records has a content-type (e.g., handshake data: `0x16`, application data: `0x17`, and alert data: `0x15`) this allows for multiple higher-level protocols to be multiplexed over the same *record layer* [Res18, Sect 5.1]. A higher-level protocol here is, for example, the handshake protocol, an application layer protocol, or the alert protocol [Res18, Sect. 6]. The record layer refers to the part of the record protocol that handles and protects the data streams. In particular, the record layer fragments data into so-called `TLSPlaintext` records (a definition of the struct is given in [Res18, Sect 5.1]), where each of these fragments can hold at most 2^{14} bytes. Since these might be fragmented over multiple records, there are certain rules. For example, handshake records (i.e., with content-type `0x16`) can be fragmented in a single record or split-up among multiple. However, if one handshake message is fragmented in multiple records, then it, for example, is not allowed to interleave with other content-types. Alert records (i.e., content-type `0x15`) are not allowed to be fragmented at all and application data (i.e., content-type `0x17`) might be fragmented arbitrarily.

In TLS 1.3, records might be protected [Res18, Sect. 5.2] using an authenticated encryption with associated data (AEAD) [Rog02] scheme. The possible schemes are defined through the cipher suites defined in [Res18, App. B.4]. These include AES128-GCM,

AES256-GCM, AES128-CCM, AES128-CCM8, which are defined in [McG08, MB12], and ChaCha20-Poly1305 defined in [NL18]. Each of these AEAD schemes is nonce-based [Rog02] and TLS 1.3 uses a special nonce randomization mechanism [Res18, Sect. 5.3] to derive the nonce for the record encryption. The nonce for the writing and reading of records is derived from a sequence number and the initialization vector (IV) that is part of the traffic keys derived in the handshake protocol. Recall that the traffic keys, such as the handshake traffic key *htk* or the application traffic key *atk*, consist of two components: a key k and an IV iv (cf. Section 6.4). The key k is the key for the record protection and the IV is used to randomize the sequence number to obtain the record nonce as follows. The record layer maintains two 64-bit sequence numbers initially set to 0. One sequence number is for reading (i.e., incoming) and the other one is for writing (i.e., outgoing) records. The respective sequence number is incremented after a record has been read or written. Using this sequence number of the record, the record layer derives a nonce for the AEAD scheme in two steps. First, each AEAD scheme specifies a range of nonce length that it supports ([N_MIN,N_MAX]; cf. [McG08]). In Section 6.4, we denoted the IV length in the context of the traffic key derivation as d, and d is always the larger value of 8 and N_MIN, i.e., $d = \max\{8, \texttt{N\_MIN}\}$. The nonce length for each of the supported schemes is 12 bytes (= N_MIN = N_MAX) [McG08, MB12, NL18]. Now, the sequence number is only 8 bytes and thus it is padded to the left with zeroes to d bytes, i.e., for a sequence number seq we have $N' := 0^{d-8} \,\|\, \texttt{seq}$. Second, the padded sequence number is "randomized" by the traffic key IV iv to obtain the nonce $N := N' \oplus iv = (0^{d-8} \| \texttt{seq}) \oplus iv$. Optionally, implementations could also artificially fill up the TLSPlaintext record length by 0 bytes to hide the length of record as the ciphertext might reveal the record length. For details, we refer to the work by Paterson, Ristenpart, and Shrimpton [PRS11], who discuss the notion of *length-hiding encryption*. Then, the record layer encrypts (resp. decrypts) the records using the corresponding key for the role (client/server) and the higher-level protocol using the record layer (e.g., handshake or application).

Composing the TLS 1.3 handshake with the record protocol. In general, key exchange protocols are only reasonable when used in combination with another protocol that uses the derived keys. However, it is not trivially clear that the composition of a key exchange protocol with a "partner protocol" still remains secure. To tame complexity of the security analysis it is always desirable to be as modular as possible. Therefore, the appealing option is to prove the two protocols secure on their own and apply some generic composition theorem to show that the composition remains secure. In case of the standard Bellare–Rogaway key exchange model [BR94], Brzuska, Fischlin, Warinschi, and Williams [BFWW11] were able to show that a protocol that is secure in the Bellare–Rogaway key exchange model can be securely composed with an arbitrary symmetric key protocol (e.g., a secure channel protocol).

Fischlin and Günther [FG14] transferred this theorem into the multi-stage setting, which also evolved over time with the MSKE model [DFGS15, FG17, DFGS16, Gün18]. The result states that an MSKE-secure protocol can be securely composed with an arbitrary symmetric key-protocol at a forward-secure, external and non-replayable stage.

On the tight security of TLS 1.3. Since, we already gave tight bounds for the handshake protocol in Chapters 9 and 10, a natural question that arises is whether the MSKE composition theorem can also be used to deduce the tight security of the "whole" cryptographic core of TLS 1.3 protocol, i.e., the composition of the handshake protocol and the record protocol. This depends on a number of aspects that we discuss in the remainder of this section. Since we already have tight security proof for the handshake protocol, the following questions remain to be answered:

1. Does the composition theorem apply to our variant of the MSKE model?
2. Is the composition theorem tightly secure?
3. Is the record protocol tightly secure?

Unfortunately, we are not able to give a definitive answer to any of these questions in this thesis, and as we discuss below, there are no affirmative answers at the moment. Nevertheless, we discuss what we know about these aspects and leave the formal treatment to future work.

Does the composition theorem apply to our variant of the MSKE model? Unfortunately, the (most recent version of the) MSKE composition theorem from [Gün18] does not *directly* apply to our variant of the MSKE model. This is mainly due to the advanced feature of *upgradeable authentication* that we adopted in our MSKE variant from [DFGS21]. Recall that this allows the authentication level of a stage in the MSKE to be upgraded upon acceptance of a later stage (cf. Section 5.2). More precisely, this means that if a stage (key) is unauthenticated upon acceptance, it might receive authentication retroactively by, e.g., the verification of an authentication message. As pointed out by [DFGS21] it is not obviously clear how to translate the notion of upgradable authentication (as well as replayability) to an arbitrary symmetric key protocol.

Therefore, we expect that the composition theorem still applies to our MSKE variant as long as only external, non-replayable stage-keys are used in the symmetric key protocol and we restrict the authentication to be fixed upon acceptance such that it does not change with acceptance of later stages. However, this still would need to be verified in a rigorous formal treatment.

Is the composition theorem tightly secure? First of all, let us state what compositional security means in the context of key exchange more precisely. For the composition of a key exchange protocol and an arbitrary symmetric key protocol, one defines a composed security experiment. This composed game consists of the key exchange experiment and the experiment that defines security for the symmetric key protocol. Intuitively, the key exchange experiment is run and whenever a key intended to be used externally is accepted it gets "registered" in the symmetric-key protocol allowing the adversary to interact with the symmetric-key experiment. The adversary now aims to break the security of the symmetric-key protocol. Thus, instead of choosing fresh keys for the symmetric-key protocol, the key exchange protocol is run. This also intuitively captures what we expect from the TLS 1.3 protocol, namely that it provides a secure

channel. Obviously, providing secure keys is a necessary condition for a secure channel, but the ultimate goal is to secure the communication of the peers.

The MSKE composition theorem [Gün18] reduces the compositional security of a MSKE protocol and an symmetric-key protocol to the security of the symmetric-key protocol, the MSKE security of the MSKE protocol, and the Match-security of the MSKE protocol (what we captured in this work by the Sound predicate in Chapter 5). As the theorem is stated in [Gün18] and its previous iterations [DFGS15, FG17, DFGS16], the reduction to the MSKE security is lossy and looses a factor that is linear in the number of sessions activated by the adversary. This stems from a hybrid argument that basically replaces every key registered in the composed experiment successively by a random key using the indistinguishability of the session key that MSKE security provides. Günther [Gün18] already conjectured that by leveraging the feature of multiple Test queries of the MSKE model that this hybrid argument might be avoidable, and thus improving the bound by this linear factor. In fact, Diemert and Jager [DJ21] revisited the proof and were able to confirm this conjecture.[1] Thus, the composition theorem for the MSKE can be proven tightly. This is a good indicator that tight compositional security is possible. However, it still is an open question whether it is possible to capture the advanced features (e.g., upgradeable authentication) of the most recent iterations of the MSKE models such as [DFGS21, DDGJ22b] and Chapter 5 in a similar composition theorem. Hence, it still remains unclear that such a theorem preserves tightness, even though we are confident that a similar technique should be applicable.

Is the record protocol tightly secure? Next, we turn to the question whether the record protocol can be proven tightly-secure. At the core of the record protocol are the AEAD schemes to protect the records in transition such that the record layer provides *confidentiality* and *integrity* [Res18, App. E.2]. That is, an adversary "[...] should not be able to determine the plain text contents of a given record." [Res18, App. E.2] and it "[...] should not be able to craft a new record which is different from an existing record which will be accepted by the receiver." [Res18, App. E.2]. Bellare and Tackmann [BT16] initiated the investigation of the multi-user security of AEAD schemes used in TLS 1.3 by analysing the security of AES-GCM in the multi-user setting using the nonce randomization mechasism of TLS 1.3 [Res18, Sect. 5.3]. However, they only give a bound that is non-tight and restrict the adversary to be passive (i.e., they only consider confidentiality). Hoang, Tessaro, and Thiruvengadam [HTT18] revisited the the work by [BT16] and show a tight bound for AES-GCM as used in TLS 1.3 without restricting the adversary to be passive. That is, they show full multi-user authenticated encryption (AE) security (i.e., confidentiality and integrity of the cipher text) for AES-GCM with nonce randomization. Degabriele, Govinden, Günther, and Paterson [DGGP21] extended the work of Hoang, Tessaro, and Thiruvengadam and Bellare and Tackmann by a treatment of the (tight) multi-user security of ChaCha20-Poly1305. Degabriele, Govinden, Gün-

[1] This result stems from joint work with Tibor Jager [DJ21]. While Tibor Jager and the author of thesis discussed all details of this paper together, this analysis was executed by the author of this thesis.

ther, and Paterson also claim to improve the tight bound given by Hoang, Tessaro, and Thiruvengadam for AES-GCM with nonce randomization.

Having tight multi-user secure building blocks is already a huge step forward for achieving tight security for the record layer. Naively one could already built a simple tightly-secure (multi-user) symmetric key protocol that is based on the multi-user security of the AEAD schemes to reflect the record layer encryption (cf. [DJ21]). With an appropriate (tight) composition theorem at hand and our tight analysis for the TLS handshake protocol, we could then prove tight composition-security for the TLS 1.3 handshake protocol when used with the supported AEAD schemes with nonce randomization. Unfortunately, the record protocol is much more than just plain AEAD encryption. Even though AEAD is the right tool to achieve confidentiality and integrity in the record protocol, the record protocol aims to provide a secure *channel*. This secure channel requires much more than only "authenticated encryption". Here, we also need to ensure that out-of-order delivery of messages are detected, in particular, replays of messages are mitigated (cf. [Res18, App. E.2] "Order protection/non-replayability"). Further, a secure channel needs means to terminate the channel in case errors occur (cf. [Res18, Sect. 6]). It needs to handle messages of arbitrary length in the form of fragmentation and in the case of TLS 1.3 it needs to support key updates. All of this is out of scope of the classical notion of AE(AD) [Rog02]. Here, messages are usually considered to be atomic (i.e., no fragmentation or data stream handling is considered), keys are chosen once upon setup of the scheme and are not updated, and the detection of out-of-order delivery and replays is also not incorporated in the standard definitions.

There has been work in the past years on various aspects of modern channel protocols. For example, on the protection against reordering, replay, or drops of messages (e.g., stateful authenticated encryption [BKN02], order-resilient channels [FGJ20, DK22]), considering fragment (stream-based channels [FGMP15]), key updated (multi-key channels [GM17]), error handling [BDPS12, BDPS14, BPS15], multiplexing of multiple data streams over one channel (partially-specified channels [PS18]), and bidirectional channels [MP17]. However, a model incorporating all features of TLS 1.3 remains elusive. We consider it an interesting open question to develop such a channel model that captures the record protocol as close as possible. Arguably the resulting model would be very complex. However, we are confident that if such a model can be formalized tight security would almost be an inherent consequence of the tight multi-user security of the AEAD building blocks.

We remark that there has been a (complete) analysis of the TLS 1.3 record layer (draft 18) [Del+17], which combines formal verification and a computational analysis. Their bound also is not tight (cf. [Del+17, Thm. 5]) as it looses a linear factor in the number of instances of the encryption scheme.

Conclusion. In this chapter, we informally discussed the current state of the tight security of the record protocol of TLS 1.3. At the moment, only AEAD schemes used at the core of the record protocol can be proven tightly-secure in the multi-user setting. It remains elusive whether the whole record protocol can be proven tightly-secure due to the lack of an appropriate model reflecting the features of the record protocol. Besides,

for our definition of MSKE-security a composition theorem remains an open question, as well, due to the advanced level of authentication we are considering. Transferring these generically to an arbitrary symmetric-key protocol is not obviously clear. Hence, the major open questions that remain for the tightness of the record layer protocol are to prove a tight composition theorem for our variant of MSKE protocols, to find an appropriate model to formalize the secure channel provided by the TLS 1.3 record layer protocol, and finally, to prove the TLS 1.3 record protocol tightly-secure in that model based on the tight multi-user security of the AEAD schemes used in the record protocol. With these open questions answered positively, one could deduce the tight security of the whole cryptographic core of TLS 1.3 by using our tight analysis of the TLS 1.3 handshake protocol.

Part IV

More Efficient Digital Signatures with Tight Multi-User Security

12

INTRODUCTION

Author's contribution. The contents of Chapters 12 to 15 are based on joint work with Kai Gellert, Tibor Jager, and Lin Lyu [DGJL21b]. Significant parts of the results presented in these chapters evolved through extensive mutual discussions. All authors contributed equally to these results. Due to joint and sometimes even indivisible individual contributions there are some parts contained in these chapters that were revised by the author of this thesis, but still almost in verbatim form from the original publication [DGJL21b] or its full version [DGJL21a]. This particularly holds for Chapters 13 and 14, in which we present our construction and its instantiations, respectively.

Contents

In the following chapters, we present a digital signature scheme that is tightly-secure in the sense of multi-user existential unforgeability with adaptive corruptions in the random oracle model. As we have seen in Chapter 9, this is exactly the security notion we required in our tight security proof for the TLS 1.3 full handshake. Since unfortunately none of the standardized signatures scheme satisfies this strong notion as already discussed before (cf. Sections 1.4 and 9.4), we present a new construction based on lossy identification schemes [AFLT12] that is the currently most efficient construction satisfying this notion (with respect to signature size) as discussed below. In the following, let us

elaborate why multi-user security is an important notion to look at for digital signature schemes, why proving tightness for these schemes is difficult, and why our construction separates from previous constructions satifying this notion.

Motivation for multi-user security. The commonly accepted standard notion for digital signature scheme is existential unforgeability under an chosen-message attack (EUF-CMA) as defined in Definition 4.6. However, this notion is a "single-user" notion, i.e., in the security model the adversary has access to a single public key and its challenge is to forge a signature with respect to this public key. Digital signature scheme are rarely used on their own, but rather as a building block in a larger cryptographic system, such as the TLS 1.3 handshake protocol presented in Chapter 6. In such a system there are usually multiple users involved such that the challenge an adversary is faced is a different one than that captured by the security model of EUF-CMA. Namely, instead of having access to a single public key, it might have access to many, and it is sufficient if the adversary is able to forge a signature with respect to *any* of these public keys to break the security that the digital signature scheme aims to provide for the higher-level system. That is, EUF-CMA-security does not really reflect the "real-world" requirements of an digital signature scheme. A stronger security notion that aims to capture this more precisely is existential unforgeability under an chosen-message attack in the multi-user setting with adaptive corruptions ($\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathsf{corr}}$) as defined in Definition 4.7. This notion generalizes EUF-CMA such that the adversary here has access to many public keys and its challenge is to forge a signature with respect to any of these public keys. Additionally, the adversary has the option to "corrupt" certain users by requesting their secret keys. The final forgery attempt then has to be for an uncorrupted public key to exclude trivial attacks. Unfortunately, EUF-CMA-security does not straightforwardly generalize to $\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathsf{corr}}$. Even though these notions are asymptotically equivalent, the reduction from $\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathsf{corr}}$-security to EUF-CMA-security has a concrete security loss that is linear in the number of users. This loss is induced by the straightforward reduction that essentially guesses one of the users in the $\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathsf{corr}}$ experiment that the adversary will output a forgery for and embeds the EUF-CMA "challenge" public key in this user. The aforementioned idea works, because the public key that the adversary outputs a forgery attempt is required to be uncorrupted over the whole execution of the security experiment. Therefore, the reduction never needs to hand the secret key corresponding to the EUF-CMA challenge public key to the adversary. The remaining key pairs, then simply can be simulated by the reduction such that it knows all the secret keys to answer potential corruptions.

Difficulty of tight multi-user-secure signatures. One can even prove that the security loss of the reduction outlined in the previous paragraph is unavoidable [BJLS16] for signature schemes satisfying certain properties. While many signature schemes satisfy these properties (e.g., having a unique secret key for every public key), there are some constructions that avoid this loss and thus circumvent this impossibility result. We discuss these constructions and compare it to the one presented in Chapter 13 in the next

paragraph, but first we discuss why achieving a tight reduction in the MU-EUF-CMA$^{\text{corr}}$ setting is a difficult task to achieve.

As already implicitly mentioned in the previous paragraph, the main challenge for a reduction to simulate the MU-EUF-CMA$^{\text{corr}}$ experiment is that it needs to solve usually some computational problem and at the same time it needs to know all secret keys to be able to respond to corruptions. However, key pairs of a digital signature scheme usually correspond to some computational problem that is related to the computational problem that we rely our security on. For example, in signature scheme instantiation based on the DDH problem presented in Section 14.1, we rely the security of the signature scheme on the DDH problem and public keys are of the form $(g^{x_0}, h^{x_0}, g^{x_1}, h^{x_1})$, where g is a generator of some cyclic group and h is a group element of that group. The secret key then is x_b, i.e., the discrete logarithm of one of the two subpairs of the public key. (x_{1-b} is discarded.) Now, recall that we discussed in Chapter 3 that an algorithm solving the DLOG problem in some cyclic group can ultimately be used to solve the DDH problem in the same group. This illustrates the challenge one is faced when aiming for a tight security proof for MU-EUF-CMA$^{\text{corr}}$. Namely, in this example the reduction to the DDH problem embeds its challenge into the public key and at the same time needs to be able to compute DLOGs to be able to simulate the MU-EUF-CMA$^{\text{corr}}$, because it needs to give out secret keys upon corruption. Additionally, it needs to use the adversary against the signature scheme to solve the DDH problem to result in a meanigful reduction. However, if it already can solve the DLOG problem, then it does not need the signature adversary to solve DDH. Hence, we are not able to related the security of the signature scheme to the success probability of solving the DDH problem.

In general, this means that for a reduction to some computational problem proving tight MU-EUF-CMA$^{\text{corr}}$-security that it needs to satisfy the following paradoxical properties:

1. It needs to know the secret key of all users to respond to the adaptive corruptions of public keys by the adversary. Here, it cannot rely on guessing a user that remains uncorrupted, because otherwise this would induce a loss.
2. The reduction needs to be able to extract a solution to a computational problem, while at the same time knowing the secret key to the corresponding instance of the signature scheme.

We address how our construction solves this paradox in detail at the beginning of Chapter 13.

Known constructions and our scheme. To the best of our knowledge, there are at the moment only a few constructions that are tightly multi-user secure with adaptive corruptions (MU-EUF-CMA$^{\text{corr}}$). We give an overview of the public key size, the signature size, the related assumption and the considered model in Table 12.1. The first scheme that achieved the notion of MU-EUF-CMA$^{\text{corr}}$-security was the construction by Bader et al. [Bad+15a]. It is in the standard model and relies on bilinear pairings. The signatures of this scheme are rather large (i.e., linear in the security parameter) such that this result

Table 12.1: Comparison of existing tightly-secure signature schemes in the multi-user setting with adaptive corruptions. $|\sigma|$ indicates the size of a signature and $|pk|$ the size of public keys, where $|\mathbb{G}|$ is the size of the binary representation of an element of the underlying group $\mathbb{G}$, $|p|$ is the size of the binary representation of an integer in the discrete interval $[0, p-1]$, where p is the order of $\mathbb{G}$, and λ is the security parameter. The column "Setting" indicates whether pairings / the Programmable Random Oracle (PRO) model / the Non-Programmable Random Oracle (NPRO) model is used. The column "sEUF" refers to whether the scheme is proven *strongly* existentially unforgeable.

Scheme	$\lvert\sigma\rvert$	$\lvert pk\rvert$	Loss	Assumption	Setting	sEUF
BHJKL [HJ12, Bad+15a]	$\mathcal{O}(\lambda)\lvert\mathbb{G}\rvert$	$\mathcal{O}(1)\lvert\mathbb{G}\rvert$	$\mathcal{O}(1)$	DLIN	Pairings	–
GJ [GJ18]	$2\lvert\mathbb{G}\rvert + 2\lambda + 4\lvert p\rvert$	$2\lvert\mathbb{G}\rvert$	$\mathcal{O}(1)$	DDH	PRO	–
HJKLPRS [Han+21]	$5\lvert\mathbb{G}\rvert$	$2\lvert\mathbb{G}\rvert$	$\mathcal{O}(1)$	SXDH	Pairings	–
Chapter 13	$3\lvert p\rvert$	$4\lvert\mathbb{G}\rvert$	$\mathcal{O}(1)$	Lossy ID/ DDH	NPRO	✓

can rather be seen as a possibility result in the standard model than an actual scheme of practical interest. They also gave an "almost-tight" variant of this scheme with a loss linear in the security parameter that has signature of constant size. However, this construction was later revisited by Han et al. [Han+21], who identified a gap in the proof, which they were not able to close such that they proposed a new variant of it. The variant still is in the standard model and uses bilinear pairings, but it is tightly-secure (with a constant loss). Due to the flaw in the proof of the second construction by Bader et al. [Bad+15a], we only include Han et al. [Han+21] in Table 12.1. The first practical scheme is due to Gjøsteen and Jager [GJ18], which is tightly-secure in the programmable random oracle model. Our scheme presented in Chapter 13 is the first generic construction that achieves tight $\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathsf{corr}}$-security in the *non*-programmable random oracle model and it can be instatiated under every computational problem that gives rise to a lossy identification scheme [AFLT12]. Further, our scheme is the first that achieves *strong* $\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathsf{corr}}$-security (i.e., $\mathsf{MU\text{-}sEUF\text{-}CMA}^{\mathsf{corr}}$) and the currently most efficient of these schemes. When instantiated with the DDH problem, signatures only consist of three $\mathbb{Z}_p$ elements, where p is the order of the underlying group. This improves the length of the signatures compared to the previously most efficient scheme as shown in Table 12.1.

Pan and Wagner [PW22] presented the first compact lattice-based signature scheme that is tightly $\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathsf{corr}}$-secure. Their construction builds upon the generic construction presented in this work originally published in [DGJL21b]. Pan and Wagner stress that their main intention for their work was to study the possibility of compact tight $\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathsf{corr}}$-secure based on lattices and thus they did not aim for efficiency. Their signatures consist of linearly in the security parameter many lattice vectors independent of the message length. However, they claim that the efficiency can be further improved. Nevertheless, we decided, mostly due to the different setting of lattices, to exclude their construction from our comparison. Also, we highlight that Pan and Wagner [PW22]

claim that they refined our generic construction [DGJL21b] such that it is more general. Finally, their result does not yield MU-sEUF-CMA$^{\text{corr}}$-security directly in contrast to our construction [DGJL21b]. However, they remark that it can be transformed tightly in a strongly secure one using a one-time signature [LM18] and a transformation [SPW07].

Strong unforgeability and its relevance for key exchange. As mentioned in the previous paragraph our construction is currently the most efficient construction that achieves tight MU-EUF-CMA$^{\text{corr}}$-security and is the first (and only) construction that is (directly) strongly unforgeable. Strong unforgeability (cf. Remark 4.2) basically captures that an adversary is not able to efficiently compute a new valid signature σ' for some message m given a valid signature $\sigma \neq \sigma'$ for message m. This is particularly useful for authenticated key exchange, when the partnering of two sessions is defined in the sense of "matching conversations" as in the famous Bellare–Rogaway model [BR94] for key exchange. Here, authentication is formalized by requiring that a session only accepts if there exists an honest partner such that the session and its partner agree on the conversation (i.e., the messages sent and received in the same order). Since a key exchange protocol usually includes sending a signature, an adversary that can compute new signatures from a give one, can efficiently break the above notion. Recall that in our security proof presented in Section 9.3, we additionally needed to rely on the `Finished` messages to guarantee integrity of the signatures sent to avoid strong unforgeability in our proof. Without the existence of additional integrity protection, we would need to require strong unforgeablity to exclude the above mentioned attack against authentication. We highlight that we explicitly did this to weaken the assumptions we need to require for our proof. Though, the more natural approach in our proof would have been to reduce to the strong unforgeability. To state another example, Gjøsteen and Jager [GJ18] even had to relax their notion of partnering and rely on the weaker notion by Li and Schäge [LS17] to overcome the lack of strong unforgeability of their signature scheme. With our scheme, we are confident that the partnering notion could be strengthen. This shows that the property of strong unforgeability in combination with the short signatures qualifies our scheme to be an ideal choice to instantiate tightly-secure authenticated key exchange with.

Outline of this Part. In Chapter 13, we present our generic construction for tightly MU-sEUF-CMA-secure digital signatures that is based on lossy identification scheme as defined in Section 4.5. Then, we present in Chapter 14 two instantiations for our genericconstruction one based on the DDH problem in Section 14.1 and another one based on the ϕ-hiding problem in Section 14.2. Finally, we briefly discuss our generic construction and point to possible future work in Chapter 15.

13

CONSTRUCTION

In this chapter, we present the construction of our digital signature scheme (originally presented in [DGJL21b]) and prove it strongly unforgeable in the multi-user setting with adaptive corruptions ($\mathsf{MU\text{-}sEUF\text{-}CMA}^{\mathsf{corr}}$). The security proof yields a tight security bound in the random oracle model (ROM).

Idea of the construction. Before we give the construction, let us first discuss its intuition. Our construction is based on sequential OR-proofs originally proposed by Abe, Ohkubo, and Suzuki [AOS02] in the context of group signature schemes. This is opposed to the "parallel" OR-proofs by Cramer, Damgård, and Schoenmakers [CDS94].[1] Informally, an OR-proof is an interactive proof system which works as follows. One party called the *prover* holds two statements x_0 and x_1 as well as a witness w_b with $b \in \{0,1\}$ for x_0 or x_1. This prover now wants to convince another party called the *verifier* holding only the statements x_0 and x_1 that it knows a witness for either of these statements. Ideally, this should hold without the verifier learning which of these statements is satified. In the construction, we transfer this idea to lossy identification schemes (LIDs) (Section 4.5). In the key generation, we generate two key pairs (pk_0, sk_0) and (pk_1, sk_1). Then, we choose a bit b and discard the secret key sk_{1-b}. That is, we now have a public key $pk = (pk_0, pk_1)$ and a secret key $sk = (b, sk_b)$. To sign a message $m \in \{0,1\}^*$, we then compute a sequential OR-proof proving that the signer knows a secret key for either of the two public keys (pk_0, pk_1). This works as follows. On a high level, the signer computes two transcript of the LID scheme, but since it only knows one of the secret keys, it computes one transcript honestly and simulates the other one. Since we want to hide which of the transcripts is honest, we apply the sequential OR-proof technique and make the challenge of the simulated transcript depend on the commitment of the honest transcript, and vice versa. This induces the sequential fashion of the proof as we first need to compute the honest commitment to simulate the other transcript, etc. The exact process is illustrated in Figure 13.1. Moreover, the challenges depend on the message as in the well-known Fiat–Shamir transform [FS87]. The signature then consists of the two commitments and responses of the computed transcripts.

[1] The distinction between sequential and parallel OR-proofs was used by Fischlin, Harasser, and Janson [FHJ20] and we adopt this to separate the two notions.

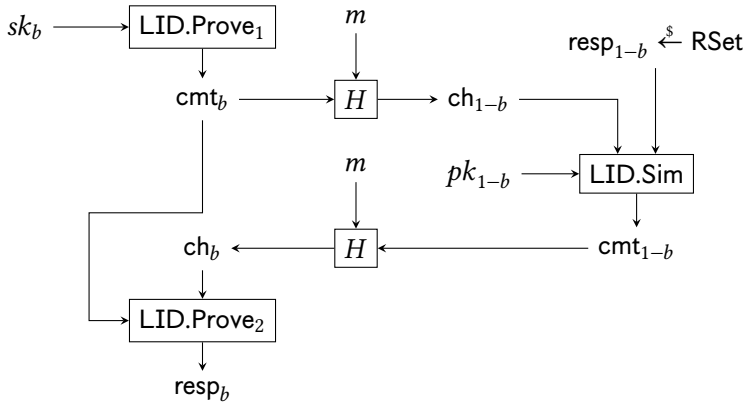

Figure 13.1: Signatures from sequential OR-proofs and lossy identification schemes.

Fischlin, Harasser, and Janson [FHJ20] proved the construction outlined above EUF-CMA-secure in the non-programmable ROM. We observed that by leveraging the commitment-recoverability of LID schemes introduced by Kiltz, Masny, and Pan [KMP16], we can slightly modify the construction such that signatures consist of a challenge of one of the transcripts and the two respective responses. This yields a signature size of only three $\mathbb{Z}_p$ elements when instantiating the signature scheme with DDH (as opposed to four group elements and two $\mathbb{Z}_p$ elements in the previous construction). Further, we observed that this construction can directly be proven strongly unforgeable and gives rise to tight multi-user security with adaptive corruptions while staying in the non-programmable ROM.

The OR-proof technique allows us here two solve the paradox discussed in Chapter 12 as follows. Since we always have two public keys (pk_0, pk_1) such that we only hold one secret key sk_b for either of the two, and the bit b is perfectly hidden from the adversary. The adversary cannot tell which is the real and which is the simulated component of the signature. Therefore, we have that the adversary outputs a forgery for the simulated component $(1-b)$ with probability $1/2$. Now, in the reduction, we can embed our challenge into the component $(1-b)$ (without the adversary detecting), and thus with probability $1/2$ the adversary will solve the challenge for us. This way we are able to answer corruption queries and solve the reduction's challenge at the same time without inducing a loss larger than $1/2$. The high-level idea was already used in [Bad+15a].

Construction. Let $\mathsf{LID} = (\mathsf{LID.Gen}, \mathsf{LID.LossyGen}, \mathsf{LID.Prove}, \mathsf{LID.Vrfy}, \mathsf{LID.Sim})$ be a LID scheme and let $H : \{0,1\}^* \to \mathsf{CSet}$ be a hash function mapping finite-length bit strings to the set of challenges CSet. We define the following digital signature scheme $\mathsf{Sig} = (\mathsf{Sig.Gen}, \mathsf{Sig.Sign}, \mathsf{Sig.Vrfy})$ from LID.

Key generation. The key generation algorithm Sig.Gen samples a bit $b \xleftarrow{\$} \{0,1\}$ and two independent key pairs $(pk_0, sk_0) \xleftarrow{\$} \mathsf{LID.Gen}$ and $(pk_1, sk_1) \xleftarrow{\$} \mathsf{LID.Gen}$. Then

it sets

$$pk := (pk_0, pk_1) \qquad \text{and} \qquad sk := (b, sk_b)$$

Note that the secret key consists only of sk_b and the other key sk_{1-b} is discarded.

Signing. The signing algorithm Sig.Sign takes as input $sk = (b, sk_b)$ and a message $m \in \{0,1\}^*$. Then it proceeds as follows.

1. It first computes $(\mathsf{cmt}_b, \mathsf{st}_b) \xleftarrow{\$} \mathsf{LID.Prove}_1(sk_b)$ and sets

   $$\mathsf{ch}_{1-b} := H(m, \mathsf{cmt}_b)$$

   Note that the ch_{1-b} is derived from cmt_b and m.

2. It generates the simulated transcript by choosing $\mathsf{resp}_{1-b} \xleftarrow{\$} \mathsf{RSet}$ and

   $$\mathsf{cmt}_{1-b} := \mathsf{LID.Sim}(pk_{1-b}, \mathsf{ch}_{1-b}, \mathsf{resp}_{1-b})$$

   using the simulator.

3. Finally, it computes

   $$\mathsf{ch}_b := H(m, \mathsf{cmt}_{1-b}) \quad \text{and} \quad \mathsf{resp}_b := \mathsf{LID.Prove}_2(sk_b, \mathsf{ch}_b, \mathsf{cmt}_b, \mathsf{st}_b)$$

   and outputs the signature $\sigma := (\mathsf{ch}_0, \mathsf{resp}_0, \mathsf{resp}_1)$. Note that ch_1 is not included in the signature.

Verification. The verification algorithm Sig.Vrfy takes as input a public key $pk = (pk_0, pk_1)$, a message $m \in \{0,1\}^*$, and a signature $\sigma = (\mathsf{ch}_0, \mathsf{resp}_0, \mathsf{resp}_1)$. It first recovers

$$\mathsf{cmt}_0 := \mathsf{LID.Sim}(pk_0, \mathsf{ch}_0, \mathsf{resp}_0)$$

From cmt_0 it can then compute

$$\mathsf{ch}_1 := H(m, \mathsf{cmt}_0)$$

and then recovers

$$\mathsf{cmt}_1 := \mathsf{LID.Sim}(pk_1, \mathsf{ch}_1, \mathsf{resp}_1)$$

Finally, the verification outputs 1 if and only if $\mathsf{ch}_0 = H(m, \mathsf{cmt}_1)$.

Proposition 13.1. *Let H be modeled as a random oracle. If scheme* LID *is commitment-recoverable (Definition 4.12) and perfectly complete (Definition 4.9), then the above signature scheme* Sig *is correct.*

Proof. Let $m \in \{0,1\}^*$ be an arbitrary message. Let (pk, sk) be any key pair such that $pk = (pk_0, pk_1)$ and $sk = (b, sk_b)$ for $(pk_i, sk_i) \xleftarrow{\$} \mathsf{LID.Gen}$ and $b \xleftarrow{\$} \{0,1\}$. Further, let $\sigma = (\mathsf{ch}_0, \mathsf{resp}_0, \mathsf{resp}_1)$ be any signature output by Sig.Sign. Next, we argue that Sig.Vrfy outputs 1 on input (pk, m, σ) with certainty. Without loss of generality fix $b = 0$, the analysis is easily adapted for the case $b = 1$. Since LID is commitment-recoverable, we have that $\mathsf{LID.Vrfy}(pk, \mathsf{cmt}, \mathsf{ch}, \mathsf{resp}) = 1$ if and only if $\mathsf{cmt} = \mathsf{LID.Sim}(pk, \mathsf{ch}, \mathsf{resp})$.

Further, by perfect completeness, we have that $\mathsf{cmt} = \mathsf{LID.Sim}(pk, \mathsf{ch}, \mathsf{resp})$ with certainty for $(pk, sk) \xleftarrow{\$} \mathsf{LID.Gen}$, $(\mathsf{cmt}, \mathsf{st}) \xleftarrow{\$} \mathsf{LID.Prove}_1(sk)$, $\mathsf{ch} \xleftarrow{\$} \mathsf{CSet}$, and $\mathsf{resp} := \mathsf{LID.Prove}_2(sk, \mathsf{cmt}, \mathsf{ch}, \mathsf{st})$. Now, under the assumption that H is modeled as a random oracle, we have that ch_0 is distributed uniformly at random on CSet and resp_0 is an honestly computed response because it was output by $\mathsf{Sig.Sign}$. By the considerations above, $\mathsf{LID.Sim}(pk_0, \mathsf{ch}_0, \mathsf{resp}_0)$ now outputs exactly the value cmt_0 output in Step 1 of $\mathsf{Sig.Sign}$ (with certainty). Further, as H is a deterministic algorithm, $\mathsf{ch}_1 = H(m, \mathsf{cmt}_0)$ is computed exactly as in Step 1 of $\mathsf{Sig.Sign}$ and therefore we have that $\mathsf{cmt}_1 = \mathsf{LID.Sim}(pk_1, \mathsf{ch}_1, \mathsf{resp}_1)$, as $\mathsf{LID.Sim}$ is deterministic. Finally, again because of the determinism of H, we have that $\mathsf{ch}_0 = H(m, \mathsf{cmt}_1)$. Hence, we have that σ is accepted with certainty and thus Sig is correct. For $b = 1$ the same argumentation applies in reverse order. □

Observe that $\mathsf{Sig.Vrfy}$ does not internally use $\mathsf{LID.Vrfy}$. Nevertheless, we can prove the following statement.

Proposition 13.2. *If* LID *is perfectly simulatable, then for all* $pk = (pk_0, pk_1)$, $m \in \{0,1\}^*$ *and* $\sigma = (\mathsf{ch}_0, \mathsf{resp}_0, \mathsf{resp}_1) \in \mathsf{CSet} \times \mathsf{RSet}^2$

$$\mathsf{Sig.Vrfy}(pk, m, \sigma) = 1 \implies \mathsf{LID.Vrfy}(pk_j, \mathsf{cmt}_j, \mathsf{ch}_j, \mathsf{resp}_j) = 1$$

for $j \in \{0,1\}$ *and* $\mathsf{cmt}_j, \mathsf{ch}_j, \mathsf{resp}_j$ *are defined as in the construction above.*

Proof. Since, we have $\mathsf{Sig.Vrfy}(pk, m, \sigma) = 1$ with $\sigma = (\mathsf{ch}_0, \mathsf{resp}_0, \mathsf{resp}_1)$ it holds by definition of $\mathsf{Sig.Vrfy}$ that there are transcripts $(\mathsf{cmt}_j, \mathsf{ch}_j, \mathsf{resp}_j)$ for $j \in \{0,1\}$ such that $\mathsf{cmt}_j = \mathsf{LID.Sim}(pk_j, \mathsf{ch}_j, \mathsf{resp}_j)$ and $\mathsf{ch}_j = H(m, \mathsf{cmt}_{1-j})$. Since LID is perfectly simulatable, it implies that $\mathsf{LID.Vrfy}(pk_j, \mathsf{cmt}_j, \mathsf{ch}_j, \mathsf{resp}_j) = 1$. □

Security of the construction. Next, we prove that the above construction is strongly unforgeable in multi-user setting with adaptive corruptions under the assumption that the hash function is modeled as a (non-programmable) random oracle and the scheme LID is commitment-recoverable, perfectly simulatable, ε_ℓ-lossy, ε_u-unique, has α-bit min-entropy and has an injective simulator (cf. Definitions 4.9 to 4.13). Formally, we prove the following theorem.

Theorem 13.1. *Let* $\mathsf{LID} = (\mathsf{LID.Gen}, \mathsf{LID.LossyGen}, \mathsf{LID.Prove}, \mathsf{LID.Vrfy}, \mathsf{LID.Sim})$ *be a LID scheme that is commitment-recoverable, perfectly simulatable,* ε_ℓ*-lossy,* ε_u*-unique, has* α*-bit min-entropy and has an injective simulator. Further, let* H *be modeled as a random oracle. Then, for each adversary* $\mathcal{A}$ *breaking the* $\mathsf{MU\text{-}sEUF\text{-}CMA}^{\mathsf{corr}}$ *security of the above signature scheme* Sig*, we can construct adversaries* $\mathcal{B}$ *and* $\mathcal{B}'$ *such that*

$$\begin{aligned}\mathrm{Adv}^{\mathsf{MU\text{-}sEUF\text{-}CMA}^{\mathsf{corr}}}_{\mathsf{Sig}}(\mathcal{A}) \leq \frac{2q_{\mathsf{Sign}}q_H}{2^\alpha} + \frac{2}{|\mathsf{CSet}|} + 2\mathrm{Adv}^{\mathsf{MU\text{-}IND\text{-}KEY}}_{\mathsf{LID}}(\mathcal{B}) + 2\epsilon_u \\ + 2\mathrm{Adv}^{\mathsf{MU\text{-}IND\text{-}KEY}}_{\mathsf{LID}}(\mathcal{B}') + 2 \cdot q_H^2 \cdot \varepsilon_\ell\end{aligned}$$

where q_{S} *is the number of signing queries and* q_H *is the number of hash queries.*

In the following, we prove the above theorem. On a high level, the proof works as follows. We proceed in a sequence of games [Sho04] and make a series of incremental changes to the MU-sEUF-CMA$^{\text{corr}}$ experiment. In a first step, we rule out repeating commitments in the signing queries, this ensures that every signing query makes fresh hash queries to compute the challenges. Then, we make sure that the two hash queries made in the final verification, i.e., when the validity of the forgery attempt is checked, already have been queried before. This is a preparation for one of the subsequent changes. Next, we exclude that the adversary uses implicit knowledge of the bit $b^{(i^*)}$ of the signature scheme instance i^* for which it submits a forgery attempt in the end. To this end, we exclude the case that the adversary reuses commitments in its forgery attempt that orignate from a signing query. Recall that we consider strong unforgeability. That is, the adversary potentially could transform a signature output by the signing oracle into a valid forgery by intutively switching the honest and simulated transcripts. Note that it has access to the transcripts by simply recomputing the verification itself. Previously we have ensured that the hash queries required for the verification of the forgery attempt are made before the forgery is submitted. Next, we ensure that the adversary can only win if the hash query for challenge $(1-b^{(i^*)})$ is made first. This implies that the adversary has no knowledge about $b^{(i^*)}$ and this is the prerequite for the final next step. In the final step, we replace all $(1 - b^{(i)})$ public keys by lossy public keys. As the bit $b^{(i^*)}$ is hidden from the adversary it cannot know which key is lossy, so intutively with probability $1/2$ it submits a forgery for the lossy key, which is due to the lossiness is unlikely to be valid. This concludes the proof.

Proof. In the sequel, we proceed in a sequence of games [Sho04] and let Game_δ denote the event that the experiment outputs 1 in Game δ.

GAME 0. This is the original security experiment $\mathrm{Exp}^{\text{MU-sEUF-CMA}^{\text{corr}}}_{\mathsf{Sig}}(\mathcal{A})$. In this experiment, adversary $\mathcal{A}$ is provided with oracles New, Sign, and Corrupt, as well as a hash oracle H since we are working in the random oracle model. In the following, it will be useful to specify the implementation of this game explicitly:

- The game initializes the counter of users $N := 0$, the set of corrupted users $\mathcal{Q}^{\text{corr}} := \emptyset$ and the table ROList of the random oracle. Then, it runs adversary $\mathcal{A}$ with access to the following oracles:
  1. Oracle New. When the adversary queries this oracle to create a new user, the game first increments $N := N + 1$ and initializes the chosen-message set $\mathcal{Q}^{(N)} := \emptyset$. Then, it runs Sig.Gen to compute a key pair $(pk^{(N)}, sk^{(N)})$. More precisely, the game samples a bit $b^{(N)} \xleftarrow{\$} \{0,1\}$ and two independent key pairs $(pk_0^{(N)}, sk_0^{(N)}) \xleftarrow{\$} \mathsf{LID.Gen}$ and $(pk_1^{(N)}, sk_1^{(N)}) \xleftarrow{\$} \mathsf{LID.Gen}$. Then, it sets $pk^{(N)} := (pk_0^{(N)}, pk_1^{(N)})$ and stores $(pk^{(N)}, b^{(N)}, sk_{b^{(N)}}^{(N)})$. Finally, it outputs $pk^{(N)}$. In the following proof, to simplify the notation, we use b instead of $b^{(i)}$ if i is clear from the context.
  2. Oracle $H(x)$. When the adversary or the simulation of the experiment make a hash oracle query for some $x \in \{0,1\}^*$, the game checks whether $y =$

$\mathsf{ROList}[x]$ for some $y \in \mathsf{CSet}$. If it exists, the game returns y. Otherwise the game selects $y \xleftarrow{\$} \mathsf{CSet}$, logs $y := \mathsf{ROList}[x]$ and returns y.

3. Oracle $\mathsf{Sign}(i, m)$. When the adversary queries the signing oracle with user i and message m, the game first sets $b := b^{(i)}$, then computes

$$(\mathsf{cmt}_b, \mathsf{st}_b) \xleftarrow{\$} \mathsf{LID.Prove}_1(sk_b^{(i)})$$

and sets $\mathsf{ch}_{1-b} := H(m, \mathsf{cmt}_b)$ by making a hash query. Then, the game chooses $\mathsf{resp}_{1-b} \xleftarrow{\$} \mathsf{RSet}$ and uses the simulator to compute $\mathsf{cmt}_{1-b} := \mathsf{LID.Sim}(pk_{1-b}^{(i)}, \mathsf{ch}_{1-b}, \mathsf{resp}_{1-b})$. Finally, the game queries oracle H to get $\mathsf{ch}_b := H(m, \mathsf{cmt}_{1-b})$ and then uses $\mathsf{LID.Prove}_2$ to compute

$$\mathsf{resp}_b := \mathsf{LID.Prove}_2(sk_b^{(i)}, \mathsf{ch}_b, \mathsf{cmt}_b, \mathsf{st}_b).$$

The game outputs signature $\sigma := (\mathsf{ch}_0, \mathsf{resp}_0, \mathsf{resp}_1)$ and logs the pair (m, σ) in set $Q^{(i)}$.

4. Oracle $\mathsf{Corrupt}(i)$. When the adversary $\mathcal{A}$ queries the $\mathsf{Corrupt}$ oracle for the secret key of user i, the game returns $sk^{(i)} := (b^{(i)}, sk_b^{(i)})$ to the adversary and logs i in the set Q^{corr}.

- When adversary $\mathcal{A}$ outputs a forgery attempt (i^*, m^*, σ^*), the game outputs 1 if and only if $\mathsf{Sig.Vrfy}(pk^{(i^*)}, m^*, \sigma^*) = 1$, $i^* \notin Q^{\mathsf{corr}}$, and $(m^*, \sigma^*) \notin Q^{(i^*)}$ hold. More precisely, for $\sigma^* = (\mathsf{ch}_0^*, \mathsf{resp}_0^*, \mathsf{resp}_1^*)$, the game recovers $\mathsf{cmt}_0^* := \mathsf{LID.Sim}(pk_0^{(i^*)}, \mathsf{ch}_0^*, \mathsf{resp}_0^*)$ and queries the hash oracle to get $\mathsf{ch}_1^* := H(m^*, \mathsf{cmt}_0^*)$. Then, it recovers $\mathsf{cmt}_1^* := \mathsf{LID.Sim}(pk_1^{(i^*)}, \mathsf{ch}_1^*, \mathsf{resp}_1^*)$ and queries the hash oracle to get $\mathsf{ch}^* := H(m^*, \mathsf{cmt}_1^*)$. Finally, the game outputs 1 if and only if $\mathsf{ch}_0^* = \mathsf{ch}^*$, $i^* \notin Q^{\mathsf{corr}}$ and $(m^*, \sigma^*) \notin Q^{(i^*)}$.

Then, by definition it holds that

$$\Pr[\mathsf{Game}_0] := \Pr[\mathrm{Exp}_{\mathsf{Sig}}^{\mathsf{MU\text{-}sEUF\text{-}CMA}^{\mathsf{corr}}}(\mathcal{A}) = 1].$$

GAME 1. In Game 1, we introduce a flag $\mathsf{bad}_{\mathsf{cmt}}$ and abort the game when it is set. The flag $\mathsf{bad}_{\mathsf{cmt}}$ is set if there exists a signing query $\mathsf{Sign}(i, m)$ such that at least one of the two hash queries $H(m, \mathsf{cmt}_{b^{(i)}})$ and $H(m, \mathsf{cmt}_{1-b^{(i)}})$ made in this signing query has been made *before*. To implement this change, we introduce a counter time (intialized with 0) that is incremented upon each oracle query. That is, every oracle executes the following operation as its first step: $\mathsf{time} := \mathsf{time} + 1$. Further, we augment the random oracle table ROList by a second entry time such that $\mathsf{ROList}[x] = (y, \mathsf{time})$, where time corresponds to the time when entry y was logged under x. In other words, time is the time when x was first queried to the random oracle. The default value for time when ROList is intialized is ∞. Then, Sign only needs to check whether it holds $t < \mathsf{time}$ or $t' < \mathsf{time}$ for $\mathsf{ROList}[m, \mathsf{cmt}_{b^{(i)}}] = (\cdot, t)$ and $\mathsf{ROList}[m, \mathsf{cmt}_{1-b^{(i)}}] = (\cdot, t')$, respectively. If so, $\mathsf{bad}_{\mathsf{cmt}}$ is set and the game is aborted.

Then, it holds (by the Difference Lemma [Sho04] or the Fundamental Lemma of game playing [BR06]) that

$$\Pr[\mathsf{Game}_0] \leq \Pr[\mathsf{Game}_1] + \Pr[\mathsf{bad}_{\mathsf{cmt}}].$$

In remains to analyze the probability that flag $\mathsf{bad}_{\mathsf{cmt}}$ is set. To this end, we divide the event that $\mathsf{bad}_{\mathsf{cmt}}$ is set into two events distinguishing whether $H(m, \mathsf{cmt}_{b^{(i)}})$ or $H(m, \mathsf{cmt}_{1-b^{(i)}})$ was the query causing $\mathsf{bad}_{\mathsf{cmt}}$ to be set.

1. There exists a signing query $\mathsf{Sign}(i, m)$ such that $H(m, \mathsf{cmt}_{b^{(i)}})$ has been made before.

   If this happens, then $\mathsf{cmt}_{b^{(i)}}$ is the output of $\mathsf{LID.Prove}_1(sk^{(i)})$ for any signing query. Since LID has α-bit min-entropy (cf. Definition 4.11), the probability that this happens is at most $q_{\mathsf{Sign}} q_H / 2^{\alpha}$ by the union bound, where q_{Sign} and q_H are the number of signing and random oracle queries issued by $\mathcal{A}$, respectively.

2. There exists a signing query $\mathsf{Sign}(i, m)$ such that $H(m, \mathsf{cmt}_{1-b^{(i)}})$ has been made before.

   Note that $\mathsf{cmt}_{1-b^{(i)}}$ is the output of

   $$\mathsf{LID.Sim}(pk^{(i)}_{1-b}, \mathsf{ch}_{1-b^{(i)}}, \mathsf{resp}_{1-b^{(i)}})$$

   where $\mathsf{ch}_{1-b^{(i)}} = H(m, \mathsf{cmt}_{b^{(i)}})$. Since LID.Sim is deterministic, we know that commitment $\mathsf{cmt}_{1-b^{(i)}}$ is determined by $pk^{(i)}_{1-b}, m, \mathsf{cmt}_{b^{(i)}}$ and $\mathsf{resp}_{1-b^{(i)}}$. Furthermore, since LID.Sim is injective with respect to challenges (cf. Definition 4.13), we know that the entropy of $\mathsf{cmt}_{1-b^{(i)}}$ in any fixed signing query is at least the entropy of $\mathsf{cmt}_{b^{(i)}}$ in that query. Thus, we obtain that the probability that this subevent happens is at most $q_{\mathsf{Sign}} q_H / 2^{\alpha}$, as well.

Hence, we have that $\Pr[\mathsf{bad}_{\mathsf{cmt}}] \leq \frac{2 q_{\mathsf{Sign}} q_H}{2^{\alpha}}$, where q_{Sign} and q_H are the number of signing and random oracle queries issued by $\mathcal{A}$, respectively, and α is the min-entropy of LID. Then, we have that

$$\Pr[\mathsf{Game}_0] \leq \Pr[\mathsf{Game}_1] + \frac{2 q_{\mathsf{Sign}} q_H}{2^{\alpha}}.$$

Observe that from Game 1 on, the hash queries $H(m, \mathsf{cmt}_{b^{(i)}})$ and $H(m, \mathsf{cmt}_{1-b^{(i)}})$ are *not* made before any signing query $\mathsf{Sign}(i, m)$ if the game is not aborted. This implies that each signing query uses independent and uniformly random $\mathsf{ch}_{1-b^{(i)}}$ and $\mathsf{ch}_{b^{(i)}}$, and they are not known to the adversary at that time.

GAME 2. In Game 2, we change the way the game checks the winning condition of the adversary. Precisely, in Game 1 checking the winning condition for a forgery attempt (i^*, m^*, σ^*) involves to issue two random oracle queries $H(m^*, \mathsf{cmt}^*_0)$ and $H(m^*, \mathsf{cmt}^*_1)$. We call the former a "0-query" and the latter a "1-query". Using the time log mechanism introduced in Game 1, we set a flag $\mathsf{bad}_{\text{not both}}$ if the forgery attempt is valid (i.e., is satifies the winning condition of Game 1), but either the 0-query or the 1-query to check the winning condition was not issued *before* the adversary submitted the forgery attempt

(i^*, m^*, σ^*). That is, it was issued for the first time during the verification. If $\mathsf{bad}_{\text{not both}}$ is set, then Game 2 outputs 0.

That is, it holds that

$$\Pr[\mathsf{Game}_1] \leq \Pr[\mathsf{Game}_2] + \Pr[\mathsf{bad}_{\text{not both}}].$$

In other words, a forgery attempt that is valid in Game 1 is also valid in Game 2 unless flag $\mathsf{bad}_{\text{not both}}$ is set. Now, let us analyze the probability that flag $\mathsf{bad}_{\text{not both}}$ is set for a forgery attempt that be considered valid in Game 1 and thus induces Game 2 to abort. Note that for an invalid forgery attempt both Game 1 and Game 2 output 0 independent of flag $\mathsf{bad}_{\text{not both}}$ being set, hence this is not relevant here. We split up the event of $\mathsf{bad}_{\text{not both}}$ being set such that it is set either because the 1-query has not been made or the 0-query has not been made:

- 1-query has not been made: Now, since the forgery attempt (i^*, m^*, σ^*) would be considered valid in Game 1, it satisfies the winning condition of Game 1, and in particular, it holds that $\mathsf{Sig.Vrfy}(pk^{(i^*)}, m^*, \sigma^*) = 1$. This implies that ch_0^* (chosen by the adversary) equals the 1-query hash result $\mathsf{ch}^* = H(m^*, \mathsf{cmt}_1^*)$, which is a random element in CSet. Since the 1-query has not been made at this time, the adversary has no knowledge about this value. Thus, it could only have guessed the right value of ch_0^* in the forgery attempt. That is, this event occurs with probability at most $1/|\mathsf{CSet}|$.
- 0-query has not been made: The result of the 0-query is $\mathsf{ch}_1^* = H(m^*, \mathsf{cmt}_0^*)$ and it is used to recover $\mathsf{cmt}_1^* = \mathsf{LID.Sim}(pk_1^{(i^*)}, \mathsf{ch}_1^*, \mathsf{resp}_1^*)$. Since the 0-query has not been made so far, the adversary has no knowledge about ch_1^* except that it is a random element in CSet. Together with the fact that algorithm $\mathsf{LID.Sim}$ is injective (cf. Definition 4.13), the adversary only knows that cmt_1^* is uniformly distributed over a set of size $|\mathsf{CSet}|$. To make the verification pass, the adversary would need to select ch_0^* which equals to $H(m^*, \mathsf{cmt}_1^*)$. However, there are $|\mathsf{CSet}|$ possible values for cmt_1^* so that this can happen with probability at most $1/|\mathsf{CSet}|$.

Overall, we deduce that

$$\Pr[\mathsf{bad}_{\text{not both}}] \leq \frac{2}{|\mathsf{CSet}|}$$

and therefore

$$\Pr[\mathsf{Game}_1] \leq \Pr[\mathsf{Game}_2] + \frac{2}{|\mathsf{CSet}|}.$$

Observe that from Game 2 on, now both the 0-query and 1-query were issued before the adversary submits its forgery attempt.

GAME 3. In Game 3, we add a flag $\mathsf{bad}_{\text{re-use}}$ and abort the game if it is set. The flag $\mathsf{bad}_{\text{re-use}}$ is again set when the game checks its winning condition and is set if the forgery attempt output by the adversary is valid with respect to Game 2, but the (first) 0-query and the (first) 1-query are made in a signing query $\mathsf{Sign}(i^*, m^*)$ and the pair $(\mathsf{cmt}_0^*, \mathsf{cmt}_1^*)$ (computed during the verification of the adversary's forgery attempt) equals

$(\mathsf{cmt}_0, \mathsf{cmt}_1)$, which are computed during the signing query. Note that if flag $\mathsf{bad}_{\text{re-use}}$ is set, then this implies that

$$\mathsf{ch}_j = H(m^*, \mathsf{cmt}_{1-j}) = H(m^*, \mathsf{cmt}^*_{1-j}) = \mathsf{ch}^*_j$$

for $j \in \{0, 1\}$. Now, since the corresponding forgery attempt is valid, it must hold $(m^*, \sigma^*) \notin Q^{(i^*)}$, i.e., the query $\mathsf{Sign}(i^*, m^*)$ that causes $\mathsf{bad}_{\text{re-use}}$ to be set output a different signature $\sigma \neq \sigma^*$. Thus, it must hold $(\mathsf{resp}^*_0, \mathsf{resp}^*_1) \neq (\mathsf{resp}_0, \mathsf{resp}_1)$. This can only occur in two possible ways:

- $\mathsf{bad}_{\text{re-use}} \wedge (\mathsf{resp}^*_{1-b^{(i^*)}} = \mathsf{resp}_{1-b^{(i^*)}}) \wedge (\mathsf{resp}^*_{b^{(i^*)}} \neq \mathsf{resp}_{b^{(i^*)}})$. Intuitively, this case implies that the adversary succesfully guessed the bit $b^{(i^*)}$ as it chose $\mathsf{resp}^*_0, \mathsf{resp}^*_1$ such that $\mathsf{resp}^*_{1-b^{(i^*)}}$ is equal and $\mathsf{resp}^*_{b^{(i^*)}}$ is unequal.[2] However, in Game 3 the bit $b^{(i^*)}$ is perfectly hidden from the adversary due to the following reasons:
  - The public key $pk^{(i^*)}$ is independent of $b^{(i^*)}$.
  - User i^* is not corrupted (or otherwise the forgery is invalid, anyway), so the bit $b^{(i^*)}$ is not leaked through corruptions.
  - The signature σ returned by oracle $\mathsf{Sign}(i^*, m^*)$ is independent of bit $b^{(i^*)}$. The reason for this is the following. Due to Game 1, each $\mathsf{Sign}(i^*, m^*)$ query uses uniformly random values $\mathsf{ch}_{1-b^{(i^*)}}$ and $\mathsf{ch}_{b^{(i^*)}}$. Thus, the signature essentially contains the two transcripts

    $$(\mathsf{cmt}_{b^{(i^*)}}, \mathsf{ch}_{b^{(i^*)}}, \mathsf{resp}_{b^{(i^*)}}) \quad \text{and} \quad (\mathsf{cmt}_{1-b^{(i^*)}}, \mathsf{ch}_{1-b^{(i^*)}}, \mathsf{resp}_{1-b^{(i^*)}})$$

    Note that the $b^{(i^*)}$ transcript is an "honestly generated" transcript and the $(1-b^{(i^*)})$ transcript is a "simulated" transcript with uniformly random $\mathsf{ch}_{1-b^{(i^*)}}$ and $\mathsf{resp}_{1-b^{(i^*)}}$. Due to the perfect simulatability of LID, we know that these two transcripts are identically distributed. Thus, adversary gains no information about $b^{(i^*)}$ through signatures.

  That is, this subcase can only occur if the adversary guesses $b^{(i^*)}$ and the probability for this case to occur is

  $$\frac{1}{2} \cdot \Pr[\mathsf{bad}_{\text{re-use}}].$$

- $\mathsf{bad}_{\text{re-use}} \wedge (\mathsf{resp}^*_{1-b^{(i^*)}} \neq \mathsf{resp}_{1-b^{(i^*)}})$. The overall goal is to analyze the probability of Game 3 aborting due to $\mathsf{bad}_{\text{re-use}}$, because $(\mathsf{resp}^*_{1-b^{(i^*)}} \neq \mathsf{resp}_{1-b^{(i^*)}})$ holds and thus rejecting a forgery attempt that is considered valid in Game 2. To analyze this subcase, we consider an intermediate game. Namely, we consider a game Game 3', which is identical to Game 3, but it uses lossy public keys $pk^{(i)}_{1-b} \xleftarrow{\$} \mathsf{LID.LossyGen}$ for every user i. That is, the New oracle always chooses the public key pk_{1-b} for a user as a lossy public key. Next, consider a forgery attempt (i^*, m^*, σ^*) that is valid in Game 2. This implies that σ^* is valid, and by Proposition 13.2 it holds that

  $$\mathsf{LID.Vrfy}(pk^{(i^*)}_{1-b}, \mathsf{cmt}^*_{1-b^{(i^*)}}, \mathsf{ch}^*_{1-b^{(i^*)}}, \mathsf{resp}^*_{1-b^{(i^*)}}) = 1.$$

[2] Note that we do not have a uniqueness property for LID with respect to normal keys. Introducing this non-standard property would potentially simplify this step.

Further, by definition the signing query $\mathsf{Sign}(i^*, m^*)$ also outputs valid signature σ for m^* for which is also holds by Proposition 13.2 that

$$\mathsf{LID.Vrfy}(pk_{1-b}^{(i^*)}, \mathsf{cmt}_{1-b^{(i^*)}}, \mathsf{ch}_{1-b^{(i^*)}}, \mathsf{resp}_{1-b^{(i^*)}}) = 1.$$

In the following, we denote by $\mathsf{bad}_{\text{re-use}}$ that the flag is set in Game 3 and by $\mathsf{bad}'_{\text{re-use}}$ that the flag is set in Game 3'. If $\mathsf{bad}'_{\text{re-use}}$ occurs, we have that $(\mathsf{cmt}_{1-b^{(i^*)}}, \mathsf{ch}_{1-b^{(i^*)}}) = (\mathsf{cmt}^*_{1-b^{(i^*)}}, \mathsf{ch}^*_{1-b^{(i^*)}})$, and further, $\mathsf{resp}_{1-b^{(i^*)}} \neq \mathsf{resp}^*_{1-b^{(i^*)}}$. Then, we can construct a straightforward reduction $\hat{\mathcal{B}}$ to the uniqueness property with respect to lossy keys. Then, we have that the probability that $\mathsf{bad}'_{\text{re-use}}$ is set and $(\mathsf{resp}^*_{1-b^{(i^*)}} \neq \mathsf{resp}_{1-b^{(i^*)}})$ holds is at most $\mathrm{Adv}^{\mathsf{unq}}_{\mathsf{LID}}(\hat{\mathcal{B}})$.

It remains to bound the difference between $\mathsf{bad}_{\text{re-use}} \wedge (\mathsf{resp}^*_{1-b^{(i^*)}} \neq \mathsf{resp}_{1-b^{(i^*)}})$ occuring in Game 3 and Game 3'. This can be bounded by a straightforward reduction to the multi-key-indistinguishability. To this end, we construct a reduction $\mathcal{B}$ that uses $\mathcal{A}$ as a subroutine as follows. The reduction has access to an oracle $\mathcal{O}$ that outputs a (fresh) public key pk that is either normal or lossy. It simulates Game 2 for $\mathcal{A}$, but in the simulation of the New oracle, it queries for every $pk_{1-b}^{(i)}$ its oracle $\mathcal{O}$ instead of running $\mathsf{LID.Gen}$. Note that Game 2 does not use $sk_{1-b}^{(i)}$ for all users i, so reduction $\mathcal{B}$ can simulate the game perfectly. Finally, if $\mathcal{A}$ outputs forgery and $\mathsf{bad}_{\text{re-use}} \wedge (\mathsf{resp}^*_{1-b^{(i^*)}} \neq \mathsf{resp}_{1-b^{(i^*)}})$, the reduction outputs 1. It is straightforward to see that it holds

$$\Pr[\mathsf{bad}_{\text{re-use}} \wedge (\mathsf{resp}^*_{1-b^{(i^*)}} \neq \mathsf{resp}_{1-b^{(i^*)}})] \leq \Pr[\mathcal{B}^{\mathsf{New}} = 1]$$

and

$$\Pr[\mathsf{bad}'_{\text{re-use}} \wedge (\mathsf{resp}^*_{1-b^{(i^*)}} \neq \mathsf{resp}_{1-b^{(i^*)}})] \leq \Pr[\mathcal{B}^{\mathsf{NewLoss}} = 1]$$

where New and $\mathsf{NewLoss}$ are as defined in Definition 4.9. This yields

$$\begin{aligned}\Pr[\mathsf{bad}_{\text{re-use}} \wedge (\mathsf{resp}^*_{1-b^{(i^*)}} \neq \mathsf{resp}_{1-b^{(i^*)}})] \leq \Pr[\mathsf{bad}'_{\text{re-use}} \wedge (\mathsf{resp}^*_{1-b^{(i^*)}} \neq \mathsf{resp}_{1-b^{(i^*)}})] \\ + \mathrm{Adv}^{\mathsf{MU\text{-}IND\text{-}KEY}}_{\mathsf{LID}}(\mathcal{B}).\end{aligned}$$

Overall we get for this subcase that the probability for it to occur is bounded from above by

$$\Pr[\mathsf{bad}_{\text{re-use}} \wedge (\mathsf{resp}^*_{1-b^{(i^*)}} \neq \mathsf{resp}_{1-b^{(i^*)}})] \leq \mathrm{Adv}^{\mathsf{MU\text{-}IND\text{-}KEY}}_{\mathsf{LID}}(\mathcal{B}) + \mathrm{Adv}^{\mathsf{unq}}_{\mathsf{LID}}(\hat{\mathcal{B}}).$$

The considerations above yield that

$$\Pr[\mathsf{bad}_{\text{re-use}}] \leq \frac{1}{2}\Pr[\mathsf{bad}_{\text{re-use}}] + \mathrm{Adv}^{\mathsf{MU\text{-}IND\text{-}KEY}}_{\mathsf{LID}}(\mathcal{B}) + \mathrm{Adv}^{\mathsf{unq}}_{\mathsf{LID}}(\hat{\mathcal{B}})$$

and equivalently

$$\Pr[\mathsf{bad}_{\text{re-use}}] \leq 2\mathrm{Adv}^{\mathsf{MU\text{-}IND\text{-}KEY}}_{\mathsf{LID}}(\mathcal{B}) + 2\mathrm{Adv}^{\mathsf{unq}}_{\mathsf{LID}}(\hat{\mathcal{B}}).$$

Hence, we have that

$$\begin{aligned}\Pr[\mathsf{Game}_2] &\leq \Pr[\mathsf{Game}_3] + \Pr[\mathsf{bad}_{\text{re-use}}] \\ &\leq \Pr[\mathsf{Game}_3] + 2\mathrm{Adv}_{\mathsf{LID}}^{\mathsf{MU\text{-}IND\text{-}KEY}}(\mathcal{B}) + 2\mathrm{Adv}_{\mathsf{LID}}^{\mathsf{unq}}(\hat{\mathcal{B}}).\end{aligned}$$

GAME 4. In Game 4, we further modify the winning condition. Namely, we introduce another flag $\mathsf{bad}_{\text{order}}$ that is set if a forgery attempt is considered valid in Game 3, and the first $b^{(i^*)}$-query is made before the $(1-b^{(i^*)})$-query. When $\mathsf{bad}_{\text{order}}$ is set the game outputs 0. Recall that we can again use the time log mechanism introduced in the previous games to check whether the flag has to be set.

We analyze $\mathsf{bad}_{\text{order}}$ and distinguish three different cases. Recall that due to Game 2, both the 0-query and 1-query corresponding to the forgery attempt already have been made when the adversary submits the forgery. Therefore, we distinguish the following three cases, when these queries are made. Let (i^*, m^*, σ^*) be the forgery attempt output by the adversary that satisfies the the winning condition of Game 3.

- Both the first 0-query and the first 1-query are issued in the context of one and the same query $\mathsf{Sign}(i^*, m^*)$.

  We have that the two hash queries

  $$\{H(m^*, \mathsf{cmt}_0^*), H(m^*, \mathsf{cmt}_1^*)\}$$

  computed during the final verification have the same input as the two hash queries $\{H(m^*, \mathsf{cmt}_0), H(m^*, \mathsf{cmt}_1)\}$ made by the signing query $\mathsf{Sign}(i^*, m^*)$. Due to Game 3, we have that it needs to hold $(\mathsf{cmt}_0^*, \mathsf{cmt}_1^*) = (\mathsf{cmt}_1, \mathsf{cmt}_0)$. Now, by definition Sign always issues $H(m^*, \mathsf{cmt}_{b^{(i^*)}})$ query before $H(m, \mathsf{cmt}_{1-b^{(i^*)}})$. Thus $\mathsf{bad}_{\text{order}}$ will not be for these 0-query and 1-query.
- Both the first 0-query and the first 1-query are made in one signing query $\mathsf{Sign}(i', m^*)$ for some $i' \neq i^*$.

  By definition of Sign the $b^{(i')}$-query is made first. Therefore, $\mathsf{bad}_{\text{order}}$ will be set if and only if $b^{(i')} = b^{(i^*)}$.
- The first 0-query and the first 1-query are not made in exactly one signing query. In other words, they lie in different signing queries or at least one of them is made by the adversary.

  Here, the adversary $\mathcal{A}$ actually has full control over which query is made first. Suppose that β-query is made first for some bit $\beta \in \{0, 1\}$ implicitly chosen by $\mathcal{A}$. That is, $\mathsf{bad}_{\text{order}}$ will be set if and only if $\beta = b^{(i^*)}$.

Using a similar line of arguments as in Game 3, we conclude that the bit $b^{(i^*)}$ needs to be perfectly hidden from the adversary. Therefore, we have

$$\Pr[\mathsf{bad}_{\text{order}}] \leq \frac{1}{2} \cdot \Pr[\mathsf{Game}_3]$$

because the event that the first $b^{(i^*)}$-query is made before the $(1 - b^{(i^*)})$-query solely depends on the choice of $b^{(i^*)}$, which is perfectly hidden from adversary, and thus this event is independent of the event the adversary outputs a valid forgery with respect to Game 3.

Overall, we have that

$$\Pr[\mathsf{Game}_3] \leq \Pr[\mathsf{Game}_4] + \Pr[\mathsf{bad}_{\text{order}}] \iff \Pr[\mathsf{Game}_3] \leq 2\Pr[\mathsf{Game}_4].$$

GAME 5. In Game 5, we change the generation of the key $pk_{1-b}^{(i)}$. Namely, the key generation in Game 5 is exactly as in Game 4 except that we choose lossy public keys $pk_{1-b}^{(i)} \xleftarrow{\$} \mathsf{LID.LossyGen}$ for every new user $i \in [N]$ the New oracle in Game 5.

Informally, adversary $\mathcal{A}$ can only detect this change if it is able to distinguish lossy from normal keys. Therefore, we bound the difference between Game 4 and Game 5 using a reduction $\mathcal{B}'$ to the multi-key-indistinguishability that is constructed as follows.

Construction of reduction $\mathcal{B}'$. The reduction $\mathcal{B}'$ receives no input and gets access to an oracle $\mathcal{O}$ that either outputs (fresh) normal public keys of LID or (fresh) lossy keys of LID. It uses adversary $\mathcal{A}$ as a subroutine and simulates Game 4 for $\mathcal{A}$. The only exception in the simulation compared to Game 4 is that it uses its oracle $\mathcal{O}$ in the simulation of the New oracle of the signature scheme Sig. Here, $\mathcal{O}$ is used to compute the public key $pk_{(1-b)}$ when a new user is created. In the end, when $\mathcal{A}$ outputs a forgery attempt, $\mathcal{B}'$ outputs 1 if and only if the forgery attempt is valid with respect to Game 4.

Analysis of reduction $\mathcal{B}'$. Note that $\mathcal{B}'$ only changes the implementation when it comes to the generation of the public keys $pk_{(1-b)}$. If oracle $\mathcal{O} = \mathsf{New}$ defined in Definition 4.9, then it perfectly simulates Game 4, as then alle public keys $pk_{(1-b)}$ are normal public keys. If oracle $\mathcal{O} = \mathsf{NewLoss}$ defined in Definition 4.9, then it perfectly simulates Game 5, as then alle public keys $pk_{(1-b)}$ are lossy public keys. Note that in Game 4, $sk_{(1-b)}$ is not used anyway, therefore $\mathcal{B}'$ does not need to simulate it.

Therefore, it holds

$$\Pr[\mathsf{Game}_4] \leq \Pr[\mathcal{B}'^{\mathsf{New}} = 1] \qquad \text{and} \qquad \Pr[\mathsf{Game}_5] \leq \Pr[\mathcal{B}'^{\mathsf{NewLoss}} = 1]$$

and

$$\Pr[\mathsf{Game}_4] \leq \Pr[\mathsf{Game}_5] + \mathrm{Adv}_{\mathsf{LID}}^{\mathsf{MU\text{-}IND\text{-}KEY}}(\mathcal{B}')$$

follows.

Finally, we claim that in Game 5 it holds

$$\Pr[\mathsf{Game}_5] \leq q_H^2 \cdot \varepsilon_\ell$$

where q_H are the number of queries issued by $\mathcal{A}$ to oracle H. Recall that the lossiness property defined in Definition 4.9 needs to hold for any adversary, even unbounded ones. Note that the multiplicative term q_H^2 does not break the tightness of our bound because ε_ℓ is a bound for all adversaries (informally speaking it is "statistically negligle"). Due to

the change of Game 4, we have ensured that the adversary has to ask an $(1-b^{(i^*)})$-query before any $b^{(i^*)}$-query. This forces the adversary to commit to a value $\mathsf{cmt}^*_{1-b^{(i^*)}}$ before the challenge $\mathsf{ch}^*_{1-b^{(i^*)}}$ is determined by a $b^{(i^*)}$-query. In Game 5, we have that for every user the public key pk_{1-b} is a lossy public key generated by LID.LossyGen. To win Game 5, the adversary has to output a valid signature σ^* that is associated with a transcript $(\mathsf{cmt}^*_{1-b^{(i^*)}}, \mathsf{ch}^*_{1-b^{(i^*)}}, \mathsf{resp}^*_{1-b^{(i^*)}})$ valid under $pk^{(i^*)}_{1-b^{(i^*)}}$. Since LID is ε_ℓ-lossy, we have that for lossy $pk^{(i^*)}_{1-b^{(i^*)}}$ and any commitment that only for an ε_ℓ-fraction of the challenges, there exists a response that yields a valid transcript. Since the adversary has at most q_H choices to choose $\mathsf{cmt}^*_{1-b^{(i^*)}}$ and for each of these choices at most q_H choices for $\mathsf{cmt}^*_{b^{(i^*)}}$ determining $\mathsf{ch}^*_{1-b^{(i^*)}} = H(m^*, \mathsf{cmt}^*_{b^{(i^*)}})$, we have that the probability that $(\mathsf{cmt}^*_{1-b^{(i^*)}}, \mathsf{ch}^*_{1-b^{(i^*)}}, \mathsf{resp}^*_{1-b^{(i^*)}})$ is valid under $pk^{(i^*)}_{1-b^{(i^*)}}$ is at most $q_H^2 \cdot \varepsilon_\ell$. Hence, the probability to win Game 5 is at most $q_H^2 \cdot \varepsilon_\ell$ as claimed before.

Final bound. Collecting all the terms implies

$$\begin{aligned}\mathrm{Adv}^{\mathsf{MU\text{-}sEUF\text{-}CMA}^{\mathsf{corr}}}_{\mathsf{Sig}}(\mathcal{A}) &= \Pr[\mathrm{Exp}^{\mathsf{MU\text{-}sEUF\text{-}CMA}^{\mathsf{corr}}}_{\mathsf{Sig}}(\mathcal{A}) = 1] \\ &\leq \frac{2q_{\mathsf{Sign}}q_H}{2^\alpha} + \frac{2}{|\mathsf{CSet}|} + 2\mathrm{Adv}^{\mathsf{MU\text{-}IND\text{-}KEY}}_{\mathsf{LID}}(\mathcal{B}) + 2\mathrm{Adv}^{\mathsf{unq}}_{\mathsf{LID}}(\hat{\mathcal{B}}) \\ &\quad + 2\mathrm{Adv}^{\mathsf{MU\text{-}IND\text{-}KEY}}_{\mathsf{LID}}(\mathcal{B}') + 2q_H^2 \cdot \varepsilon_\ell\end{aligned}$$

and the theorem follows.

□

14

INSTANTIATIONS

In this chapter, we present two instantiations of our generic construction presented in Chapter 13. One is based on the DDH problem and the other one is based on the ϕ-hiding assumption [ABP13, CMS99, KOS10]. The constructions described in this chapter are derived from [KW03, AFLT12, AFLT16, ABP13] and are well-known. The purpose of this section is to argue and justify that these constructions indeed satisfy all properties required for our signature scheme.

14.1 Instantiation based on Decisional Diffie–Hellman

The well-known DDH-based LID scheme uses a standard Σ-protocol [Dam10] to prove equality of discrete logarithms by Chaum, Evertse, and van de Graaf [CEv88] (cf. Figure 14.1) as foundation, which was used by Katz and Wang [KW03] to build tightly-secure signatures (in the single-user setting without corruptions).

Alternative characterization of DDH. In Definition 3.4, we defined the DDH problem in a cyclic group $\mathbb{G}$ with prime-order p and a generator $g \in \mathbb{G}$ such that given g^a, g^b, and g^c for $a, b \xleftarrow{\$} \mathbb{Z}_p$ it is to decide whether $c = ab \mod p$ or $c \xleftarrow{\$} \mathbb{Z}_p$. Alternatively, one can characterize the DDH problem not only with respect to a generator g, but also with respect to a group element $h \xleftarrow{\$} \mathbb{G}$. Then, the DDH problem is defined with respect to

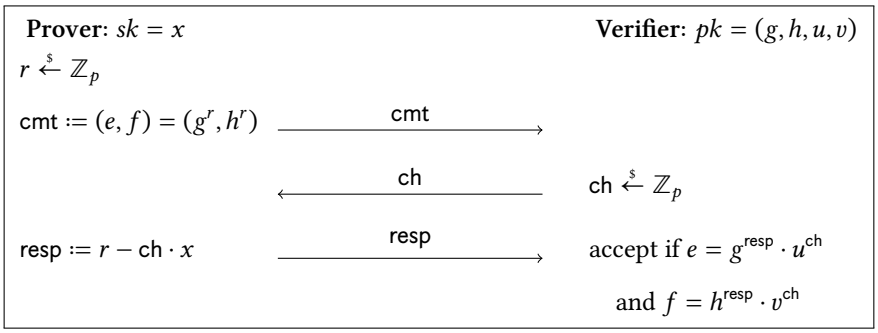

Figure 14.1: The DDH-based identification scheme [CEv88].

(g, h) as given $(u, v) \in \mathbb{G}^2$ it is to decide whether $\mathrm{dlog}_g u = \mathrm{dlog}_h v$. Note that these two characterizations are equivalent. If one sets $h = g^b$ and $u = g^a$, then $v = g^{ab}$ if and only if $\mathrm{dlog}_g u = \mathrm{dlog}_h v$. For the protocol shown in Figure 14.1, the latter characterization is more convenient so we use this one in this chapter.

14.1.1 A DDH-based LID Scheme

Let $(\mathbb{G}, g, p)$ be a cyclic group of prime order p and generator g and let $h \in \mathbb{G}$. We define the lossy identification scheme $\mathsf{LID} = (\mathsf{LID.Gen}, \mathsf{LID.LossyGen}, \mathsf{LID.Prove}, \mathsf{LID.Vrfy}, \mathsf{LID.Sim})$ based on the protocol presented in Figure 14.1 as follows:

Key generation. The algorithm LID.Gen chooses a value $x \xleftarrow{\$} \mathbb{Z}_p$ uniformly at random. It sets $pk := (g, h, u, v) = (g, h, g^x, h^x)$ and $sk := x$, and outputs (pk, sk).

Lossy key generation. The algorithm LID.LossyGen chooses two group elements $(u, v) \xleftarrow{\$} \mathbb{G}^2$ uniformly and independently at random. It outputs $pk := (g, h, u, v)$.

Proving. The algorithm LID.Prove is split up into the following two algorithms:

1. The algorithm $\mathsf{LID.Prove}_1$ takes as input a secret key $sk = x$, chooses a random value $r \xleftarrow{\$} \mathbb{Z}_p$, and computes a commitment $\mathsf{cmt} := (e, f) = (g^r, h^r)$, where g, h are the value of the pk corresponding to sk. It outputs $(\mathsf{cmt}, \mathsf{st})$ with $\mathsf{st} := r$.
2. The algorithm $\mathsf{LID.Prove}_2$ takes as input a secret key $sk = x$, a commitment $\mathsf{cmt} = (e, f)$, a challenge $\mathsf{ch} \in \mathbb{Z}_p$, a state $\mathsf{st} = r$, and outputs a response $\mathsf{resp} := r - \mathsf{ch} \cdot x$.

Verification. The verification algorithm LID.Vrfy takes as input a public key $pk = (g, h, u, v)$, a commitment $\mathsf{cmt} = (e, f)$, a challenge $\mathsf{ch} \in \mathbb{Z}_p$, and a response $\mathsf{resp} \in \mathbb{Z}_p$. It outputs 1 if and only if $e = g^{\mathsf{resp}} \cdot u^{\mathsf{ch}}$ and $f = h^{\mathsf{resp}} \cdot v^{\mathsf{ch}}$.

Simulation. The simulation algorithm LID.Sim takes as input a public key $pk = (g, h, u, v)$, a challenge $\mathsf{ch} \in \mathbb{Z}_p$, and a response $\mathsf{resp} \in \mathbb{Z}_p$. It outputs a commitment $\mathsf{cmt} = (e, f) = (g^{\mathsf{resp}} \cdot u^{\mathsf{ch}}, h^{\mathsf{resp}} \cdot v^{\mathsf{ch}})$.

Remark 14.1. Note that an honest public key generated with LID.Gen is of the form $pk = (g, h, u, v)$ such that (u, v) is a valid DDH tuple, whereas a lossy public key generated with LID.LossyGen is of the form $pk = (g, h, u, v)$ such that (u, v) is not a DDH tuple with (high probability).

Theorem 14.1. *The scheme* LID *defined above is lossy with*

$$\rho = 1, \quad \varepsilon_s = 0, \quad \varepsilon_\ell \leq 1/p,$$

and from any adversary $\mathcal{A}$ we can construct an adversary $\mathcal{B}$ such that

$$\mathrm{Adv}^{\mathsf{MU\text{-}IND\text{-}KEY}}_{\mathsf{LID}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{DDH}}_{\mathbb{G},g}(\mathcal{B}).$$

Furthermore, LID *is perfectly unique with respect to lossy keys (i.e., for any adversary* $\mathcal{A}$ *it holds* $\mathrm{Adv}^{\mathsf{unq}}_{\mathsf{LID}}(\mathcal{A}) = 0$*),* LID *has* α*-bit min-entropy with* $\alpha = \log_2(p)$*,* LID *is commitment-recoverable, and* LID *has an injective simulator.*

The proof of this theorem is rather standard and implicitly contained in the aforementioned prior works. For completeness, we provide a sketch below.

Proof. To show that LID is lossy, we need to show that it satisfies all properties presented in Section 4.5.

Completeness of normal keys. We claim that the above scheme is perfectly complete. To prove this, we show that for any honest transcript it holds that $\mathsf{LID.Vrfy}(pk, \mathsf{cmt}, \mathsf{ch}, \mathsf{resp}) = 1$. Let $(pk, sk) \xleftarrow{\$} \mathsf{LID.Gen}$ be an (honest) key pair and let $(\mathsf{cmt}, \mathsf{ch}, \mathsf{resp})$ be an honest transcript, that is, $\mathsf{ch} \xleftarrow{\$} \mathsf{CSet}$, $(\mathsf{cmt}, \mathsf{st}) \xleftarrow{\$} \mathsf{LID.Prove}_1(sk)$ and $\mathsf{resp} := \mathsf{LID.Prove}_2(sk, \mathsf{cmt}, \mathsf{ch}, \mathsf{st})$. By definition of the scheme, we have $pk = (g, h, u, v)$ with (u, v) such that $\mathrm{dlog}_g u = \mathrm{dlog}_h v$ and $sk = x$ and $\mathsf{cmt} = (e, f) = (g^r, h^r)$ and $\mathsf{resp} = r - \mathsf{ch} \cdot x$. Further, $\mathsf{LID.Vrfy}(pk, \mathsf{cmt}, \mathsf{ch}, \mathsf{resp}) = 1$ if and only if $e = g^{\mathsf{resp}} \cdot u^{\mathsf{ch}}$ and $f = h^{\mathsf{resp}} \cdot v^{\mathsf{ch}}$. Observe that

$$g^{\mathsf{resp}} \cdot u^{\mathsf{ch}} = g^{r - \mathsf{ch} \cdot x} \cdot g^{\mathsf{ch} \cdot x} = g^r = e.$$

An analogous equation holds for f if g is replaced by h. Hence, LID.Vrfy outputs 1 for every honest transcript.

Simulatability of transcripts. We claim that the above scheme is perfectly simulatable. To show this, we need to argue that the two distributions

$$\left\{ (\mathsf{cmt}, \mathsf{ch}, \mathsf{resp}) : \begin{array}{l} (\mathsf{cmt}, \mathsf{st}) \xleftarrow{\$} \mathsf{LID.Prove}_1(sk) \\ \mathsf{ch} \xleftarrow{\$} \mathbb{Z}_p \\ \mathsf{resp} := \mathsf{LID.Prove}_2(sk, \mathsf{ch}, \mathsf{cmt}, \mathsf{st}) \end{array} \right\}$$

and

$$\left\{ (\mathsf{cmt}, \mathsf{ch}, \mathsf{resp}) : \begin{array}{l} \mathsf{ch} \xleftarrow{\$} \mathbb{Z}_p \\ \mathsf{resp} \xleftarrow{\$} \mathbb{Z}_p \\ \mathsf{cmt} := \mathsf{LID.Sim}(pk, \mathsf{ch}, \mathsf{resp}) \end{array} \right\}$$

are identical. Recall that we have $pk = (g, h, u, v)$ with (u, v) with $\mathrm{dlog}_g u = \mathrm{dlog}_h v$, $sk = x$, $\mathsf{cmt} = (e, f) = (g^r, h^r)$ with $\mathsf{st} = r \xleftarrow{\$} \mathbb{Z}_p$, and $\mathsf{resp} = r - \mathsf{ch} \cdot x$ for an honest transcript (i.e., in the former distribution). Thus, we have that $\mathsf{cmt} = (e, f)$ is uniformly distributed over $\mathbb{G}^2$. Consequently, since $r \xleftarrow{\$} \mathbb{Z}_p$ and $\mathsf{ch} \xleftarrow{\$} \mathbb{Z}_p$, we also have that the response resp is distributed uniformly and independently (of cmt and ch) over $\mathbb{Z}_p$.

We will now take a look at the latter distribution. Note that ch and resp are both uniformly random elements over $\mathbb{Z}_p$. It remains to show that cmt in the simulated transcript is distributed uniformly over $\mathbb{G}^2$.

Recall that $\mathsf{cmt} := \mathsf{LID.Sim}(pk, \mathsf{ch}, \mathsf{resp})$ is defined as $\mathsf{cmt} := (e, f) = (g^{\mathsf{resp}} \cdot u^{\mathsf{ch}}, h^{\mathsf{resp}} \cdot v^{\mathsf{ch}})$. Observe that $\log_g(e) = \mathsf{resp} + \mathsf{ch} \cdot x$ and $\log_g(f) = \log_g(h) \cdot (\mathsf{resp} + \mathsf{ch} \cdot x)$. Since

$\mathsf{ch} \xleftarrow{\$} \mathbb{Z}_p$ and $\mathsf{resp} \xleftarrow{\$} \mathbb{Z}_p$, we have that both $\mathrm{dlog}_g(e)$ and $\mathrm{dlog}_g(f)$ are distributed uniformly and independently (of ch and resp) over $\mathbb{Z}_p$ and thus (e, f) is distributed uniformly over $\mathbb{G}^2$. Note that e, f are not distributed independently of each other (as it is the case in the honest transcript).

Indistinguishability of keys. As already remarked above, honest keys contain a DDH tuple, whereas lossy keys contain a non-DDH tuple (i.e., random $(u, v) \in \mathbb{G}^2$. Therefore, we claim that for every adversary $\mathcal{A}$ trying to distinguish honest from lossy keys of LID, we can construct an adversary $\mathcal{B}$ such that

$$\mathrm{Adv}^{\mathsf{MU\text{-}IND\text{-}KEY}}_{\mathsf{LID}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{DDH}}_{\mathbb{G},g}(\mathcal{B}).$$

To prove this claim, we give a construction of $\mathcal{B}$ running $\mathcal{A}$ as a subroutine. The adversary $\mathcal{B}$ receives a tuple (g, h, u, v) such that (u, v) either it holds $\mathrm{dlog}_g u = \mathrm{dlog}_h v$ or not. Adversary $\mathcal{B}$ needs to simulate an oracle $\mathcal{O}$ for $\mathcal{A}$ that outputs a public key. Here, $\mathcal{O}$ outputs a fresh rerandomization (g, h, u', v') of the tuple (g, h, u, v).[1] When $\mathcal{A}$ halts and outputs a bit b, $\mathcal{B}$ halts and outputs b as well.

Observe that due to the random self-reducibility of DDH that $\mathcal{B}$ perfectly simulates oracle New as defined in Definition 4.9 if it holds $\mathrm{dlog}_g u = \mathrm{dlog}_h v$ for the tuple (g, h, u, v). Otherwise, it perfectly simulates $\mathsf{NewLoss}$ as defined in Definition 4.9. In conclusion, we have

$$\mathrm{Adv}^{\mathsf{MU\text{-}IND\text{-}KEY}}_{\mathsf{LID}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{DDH}}_{\mathbb{G},g}(\mathcal{B}).$$

Lossiness. We claim that the above scheme LID is $(1/p)$-lossy. To show this, we first recall a standard result showing the soundness of the protocol to "prove DDH tuples" by Chaum, Evertse, and van de Graaf presented above. Namely, we claim that if $\log_g(u) \neq \log_h(v)$ holds for the public key $pk = (g, h, u, v)$ (i.e., pk is a lossy key and $(u, v) \xleftarrow{\$} \mathbb{G}^2$), for any commitment cmt there can only be at most one challenge ch such that the transcript is valid. We prove this statement by contradiction.

Let pk be any lossy public key that can be output by $\mathsf{LID.LossyGen}$ and let $\mathsf{cmt} = (e, f)$ be any commitment. We show that there is a response resp for only *one* ch such that $\mathsf{LID.Vrfy}(pk, \mathsf{cmt}, \mathsf{ch}, \mathsf{resp}) = 1$. Suppose two responses resp_1 and resp_2 for two different challenge $\mathsf{ch}_1 \neq \mathsf{ch}_2$ such that $\mathsf{LID.Vrfy}(pk, \mathsf{cmt}, \mathsf{ch}_1, \mathsf{resp}_1) = 1$ and $\mathsf{LID.Vrfy}(pk, \mathsf{cmt}, \mathsf{ch}_2, \mathsf{resp}_2) = 1$ holds. This implies by the definition of $\mathsf{LID.Vrfy}$ that

$$e = g^{\mathsf{resp}_1} u^{\mathsf{ch}_1} = g^{\mathsf{resp}_2} u^{\mathsf{ch}_2} \qquad \text{and} \qquad f = h^{\mathsf{resp}_1} v^{\mathsf{ch}_1} = h^{\mathsf{resp}_2} v^{\mathsf{ch}_2}.$$

Equivalently, we get by using the assumption that $\mathsf{ch}_1 \neq \mathsf{ch}_2$:

$$\log_g(u) = \frac{(\mathsf{resp}_1 - \mathsf{resp}_2)}{\mathsf{ch}_2 - \mathsf{ch}_1} \qquad \text{and} \qquad \log_h(v) = \frac{(\mathsf{resp}_1 - \mathsf{resp}_2)}{\mathsf{ch}_2 - \mathsf{ch}_1}.$$

[1] It is well-known that the DDH problem (and the DH related problems in general) are *random self-reducible*. That is, there is an algorithm that on input (g, h, u, v) outputs (u', v') such that $\mathrm{dlog}_g u = \mathrm{dlog}_h v$ if and only if $\mathrm{dlog}_g u' = \mathrm{dlog}_h v'$ and (u', v') are distributed as a fresh DDH pair. For more details, we refer, e.g., to [BBM00, Lemma 5.2].

However, this is a contraction to the assumption that $\log_g(u) \neq \log_h(v)$.

Using this, we have that for every commitment, there can only be at most one challenge ch for which there exists a response that yields a valid transcript. As there is only one challenge for cmt, we have that the ratio of challenges for which there exists a response that yields a valid transcript is at most $1/p$.

Uniqueness with respect to lossy keys. Let $pk = (g, h, u, v)$ with $(u, v) \xleftarrow{\$} \mathbb{G}^2$ and $(\mathsf{cmt}, \mathsf{ch}, \mathsf{resp})$ with $\mathsf{LID.Vrfy}(pk, \mathsf{cmt}, \mathsf{ch}, \mathsf{resp}) = 1$. Suppose that there is a $\mathsf{resp}' \neq \mathsf{resp}$ such that $\mathsf{LID.Vrfy}(pk, \mathsf{cmt}, \mathsf{ch}, \mathsf{resp}') = 1$. In this case, we have for $\mathsf{cmt} = (e, f)$ that

$$e = g^{\mathsf{resp}} u^{\mathsf{ch}} = g^{\mathsf{resp}'} u^{\mathsf{ch}} \qquad \text{and} \qquad f = h^{\mathsf{resp}} v^{\mathsf{ch}} = h^{\mathsf{resp}'} v^{\mathsf{ch}}.$$

However, this implies that

$$g^{\mathsf{resp}} = g^{\mathsf{resp}'} \qquad \text{and} \qquad h^{\mathsf{resp}} = h^{\mathsf{resp}'},$$

which implies that $\mathsf{resp} = \mathsf{resp}'$, contradicting the initial assumption.

Min-entropy. For any secret key sk, the commitment $\mathsf{cmt} \xleftarrow{\$} \mathsf{LID.Prove}_1(sk)$ equals (g^r, h^r) for $r \xleftarrow{\$} \mathbb{Z}_p$, which is independent of sk. So the min-entropy of cmt is $\alpha = \log_2(p)$.

Commitment-recoverable. The verification algorithm of LID first recovers a commitment using the simulator and then compares the result with the commitment in the transcript. So LID is commitment-recoverable.

Injective simulator. For any normal public key $pk = (g, h, u, v)$, any response resp and any challenge $\mathsf{ch} \neq \mathsf{ch}'$, we have that

$$\mathsf{LID.Sim}(pk, \mathsf{ch}, \mathsf{resp}) = (g^{\mathsf{resp}} u^{\mathsf{ch}}, h^{\mathsf{resp}} v^{\mathsf{ch}}),$$
$$\mathsf{LID.Sim}(pk, \mathsf{ch}', \mathsf{resp}) = (g^{\mathsf{resp}} u^{\mathsf{ch}'}, h^{\mathsf{resp}} v^{\mathsf{ch}'}).$$

Thus, if the above two pairs are equal, we must have that $(u^{\mathsf{ch}}, v^{\mathsf{ch}}) = (u^{\mathsf{ch}'}, v^{\mathsf{ch}'})$. That implies $\mathsf{ch} = \mathsf{ch}'$.

□

14.1.2 Concrete instantiation

We can now use the DDH-based lossy identification scheme to describe an explicit instantiation of our signature scheme based on the DDH assumption. To this end, let $\mathbb{G}$ be a group of prime order p with generator g, let $h \xleftarrow{\$} \mathbb{G}$ be a random generator and let $H : \{0,1\}^* \to \mathbb{Z}_p$ be a hash function. We construct a digital signature scheme $\mathsf{Sig} = (\mathsf{Sig.Gen}, \mathsf{Sig.Sign}, \mathsf{Sig.Vrfy})$ as follows.

Key generation. The key generation $\mathsf{Sig.Gen}$ algorithm samples $x_0, x_1 \xleftarrow{\$} \mathbb{Z}_p$, $b \xleftarrow{\$} \{0,1\}$. Then it sets

$$pk := (u_0, v_0, u_1, v_1) = (g^{x_0}, h^{x_0}, g^{x_1}, h^{x_1}) \qquad \text{and} \qquad sk := (b, x_b).$$

Signing. The signing algorithm Sig.Sign takes as input $sk = (b, x_b)$ and a message $m \in \{0,1\}^*$. Then it proceeds as follows.

1. It first chooses a random value $r \xleftarrow{\$} \mathbb{Z}_p$, and sets $(e_b, f_b) := (g^r, h^r)$ and
$$\mathsf{ch}_{1-b} := H(m, e_b, f_b).$$
2. Then it samples a value $\mathsf{resp}_{1-b} \xleftarrow{\$} \mathbb{Z}_p$ and computes
$$e_{1-b} = g^{\mathsf{resp}_{1-b}} u_{1-b}^{\mathsf{ch}_{1-b}} \qquad \text{and} \qquad f_{1-b} = h^{\mathsf{resp}_{1-b}} v_{1-b}^{\mathsf{ch}_{1-b}}.$$
3. Finally, it computes
$$\mathsf{ch}_b := H(m, e_{1-b}, f_{1-b}) \qquad \text{and} \qquad \mathsf{resp}_b := r - \mathsf{ch}_b \cdot x_b$$
and outputs the signature $\sigma := (\mathsf{ch}_0, \mathsf{resp}_0, \mathsf{resp}_1) \in \mathbb{Z}_p^3$.

Verification. The verification algorithm takes as input a public key $pk := (u_0, v_0, u_1, v_1)$, a message $m \in \{0,1\}^*$, and a signature $\sigma = (\mathsf{ch}_0, \mathsf{resp}_0, \mathsf{resp}_1)$.

If first computes
$$e_0 = g^{\mathsf{resp}_0} u_0^{\mathsf{ch}_0} \qquad \text{and} \qquad f_0 = h^{\mathsf{resp}_0} v_0^{\mathsf{ch}_0}.$$
From (e_0, f_0) it is then able to compute
$$\mathsf{ch}_1 := H(m, e_0, f_0)$$
and then
$$e_1 = g^{\mathsf{resp}_1} \cdot u_1^{\mathsf{ch}_1} \qquad \text{and} \qquad f_1 = h^{\mathsf{resp}_1} \cdot v_1^{\mathsf{ch}_1}.$$
Finally, the algorithm outputs 1 if and only if
$$\mathsf{ch}_0 = H(m, e_1, f_1).$$

Note that public keys are $pk \in \mathbb{G}^4$, secret keys are $sk \in \{0,1\} \times \mathbb{Z}_p$, and signatures are $\sigma \in \mathbb{Z}_p^3$.

14.2 Instantiation from the ϕ-Hiding Assumption

The second instantiation we present is based on the Guillou–Quisquater (GQ) identification scheme [GQ90], which proves that an element $U = S^e \bmod N$ is an e-th residue (cf. Figure 14.2). Abdalla, Ben Hamouda, and Pointcheval [ABP13] describe a lossy version of the GQ scheme, based on the ϕ-hiding. We observe that we can build a lossy identification scheme on a weaker assumption, which is implied by ϕ-hiding [ABP13, CMS99, KOS10].

In order to achieve tightness in the multi-user setting, we will need a common setup, which is shared across all users. This setup consists of a public tuple (N, e) where $N = p \cdot q$ is the product of two large random primes and e is a uniformly random prime of length $\ell_e \leq \lambda/4$ that divides $p - 1$. The factors p and q need to remain secret, so we assume that (N, e) either was generated by a trusted party, or by running a secure multi-party computation protocol with multiple parties.

14.2.1 The Guillou–Quisquater LID Scheme.

We define the lossy identification scheme $\mathsf{LID} = (\mathsf{LID.Gen}, \mathsf{LID.LossyGen}, \mathsf{LID.Prove}, \mathsf{LID.Vrfy}, \mathsf{LID.Sim})$ based on the protocol presented in Figure 14.2 as follows:

Common setup. The common system parameters are a tuple (N, e) where $N = p \cdot q$ is the product of two distinct primes p, q of length $\lambda/2$ and e is random prime of length $\ell_e \leq \lambda/4$ such that e divides $p - 1$.

Note that the parameters (N, e) are always in "lossy mode", and not switched from an "injective" pair (N, e) where e is coprime to $\phi(N) = (p-1)(q-1)$ to "lossy" in the security proof, as common in other works.

Key generation. The algorithm $\mathsf{LID.Gen}$ samples $S \xleftarrow{\$} \mathbb{Z}_N^*$ and computes $U = S^e$. It sets $pk = (N, e, U)$ and $sk = (N, e, S)$, where (N, e) are from the common parameters.

Lossy key generation. The lossy key generation algorithm $\mathsf{LID.LossyGen}$ samples U uniformly at random from the e-th non-residues modulo N.[2]

Proving. The algorithm $\mathsf{LID.Prove}$ is split up into the following two algorithms:

1. The algorithm $\mathsf{LID.Prove}_1$ takes as input a secret key $sk = (N, e, S)$, chooses a random value $r \xleftarrow{\$} \mathbb{Z}_N^*$, and computes a commitment $\mathsf{cmt} := r^e \bmod N$. It outputs $(\mathsf{cmt}, \mathsf{st})$ with $\mathsf{st} := r$.
2. The algorithm $\mathsf{LID.Prove}_2$ takes as input a secret key $sk = (N, e, S)$, a commitment cmt, a challenge $\mathsf{ch} \in \{0, \ldots, 2^{\ell_e} - 1\}$, a state $\mathsf{st} = r$, and outputs a response $\mathsf{resp} := r \cdot S^{\mathsf{ch}} \bmod N$.

Verification. The verification algorithm $\mathsf{LID.Vrfy}$ takes as input a public key $pk = (N, e, U)$, a commitment cmt, a challenge ch, and a response resp. It outputs 1 if and only if $\mathsf{resp} \neq 0 \bmod N$ and $\mathsf{resp}^e = \mathsf{cmt} \cdot U^{\mathsf{ch}}$.

Simulation. The simulation algorithm $\mathsf{LID.Sim}$ takes as input a public key $pk = (N, e, U)$, a challenge ch, and a response resp. It outputs a commitment $\mathsf{cmt} = \mathsf{resp}^e / U^{\mathsf{ch}}$.

Before we can prove that the above scheme indeed satisfies the properties we require for LID schemes, we need to introduce the n-fold higher residuosity problem that we rely our multi-key-indistinguishability on. To this end, we first define the RSA modulus generation algorithm. The following definition is from [ABP13].

Definition 14.1. Let ℓ_N be a positive integer and let RSA_{ℓ_N} be the set of all tuples (N, p_1, p_2) such that $N = p_1 p_2$ is a ℓ_N-bit number and p_1, p_2 are two distinct primes in the set of $\ell_N/2$-bit primes $\mathbb{P}_{\ell_N/2}$. Let R be any relation on p_1 and p_2, define $\mathsf{RSA}_{\ell_N}[R] := \{(N, p_1, p_2) \in \mathsf{RSA}_{\ell_N} \mid R(p_1, p_2) = 1\}$.

[2] This is indeed efficiently possible as $U \xleftarrow{\$} \mathbb{Z}_N^*$ is a not an e-th residue with probability $1 - 1/e$ and we can efficiently check whether a given U is an e-th residue when the factorization of N is known [ABP13].

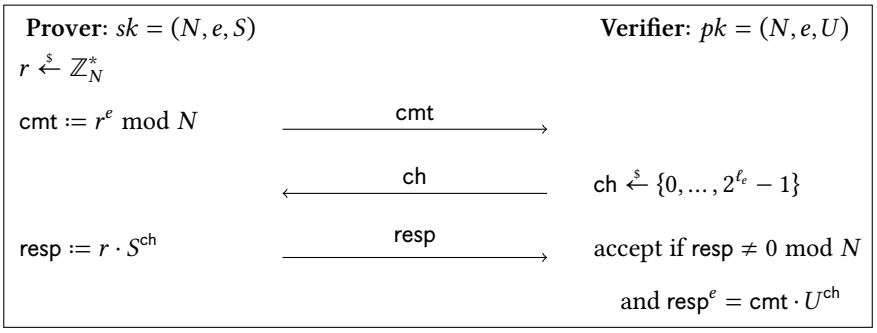

Figure 14.2: The Guillou–Quisquater identification scheme [GQ90].

We can use it to define the n-fold higher residuosity assumption as well as the ϕ-hiding assumption [ABP13, CMS99, KOS10].

Definition 14.2. Let e be a random prime of length $\ell_e \leq \ell_N/4$ and

$$(N, p_1, p_2) \xleftarrow{\$} \mathsf{RSA}_{\ell_N}[p_1 = 1 \bmod e]$$

and let $\mathsf{HR}_N[e] := \{g^e \bmod N \mid g \in \mathbb{Z}_N^*\}$ be the set of e-th residues modulo N. We define the advantage of any $\mathcal{A}$ in solving the *higher residuosity problem* as

$$\mathrm{Adv}^{n\text{-}\mathsf{HR}}(\mathcal{A}) := |\Pr[\mathcal{A}(N, e, y_1, \dots, y_n) = 1] - \Pr[\mathcal{A}(N, e, y'_1, \dots, y'_n) = 1]|,$$

where $y_1, \dots, y_n \xleftarrow{\$} \mathsf{HR}_N[e]$ and $y'_1, \dots, y'_n \xleftarrow{\$} \mathbb{Z}_N^* \setminus \mathsf{HR}_N[e]$.

Definition 14.3. Let $c \leq 1/4$ be a constant. For any adversary $\mathcal{A}$, define the advantage of $\mathcal{A}$ in solving the *ϕ-hiding problem* to be

$$\mathrm{Adv}^{\phi\mathrm{H}}(\mathcal{A}) := |\Pr[\mathcal{A}(N, e) = 1] - \Pr[\mathcal{A}(N', e) = 1]|,$$

where $e \xleftarrow{\$} \mathbb{P}_{c\ell_N}$, $(N, p_1, p_2) \xleftarrow{\$} \mathsf{RSA}_{\ell_N}[\gcd(e, \phi(N)) = 1]$ and $(N', p'_1, p'_2) \xleftarrow{\$} \mathsf{RSA}_{\ell_N}[p'_1 = 1 \bmod e]$.

For the LID scheme presented above, we have the following properties.

Theorem 14.2. *The scheme* LID *defined above is lossy with*

$$\rho = 1, \quad \varepsilon_s = 0, \quad \varepsilon_\ell \leq 1/2^{\ell_e},$$

and from any adversary $\mathcal{A}$ that issues n queries to distinguish normal from lossy keys, we can construct an adversary $\mathcal{B}$ such that

$$\mathrm{Adv}^{\mathsf{MU\text{-}IND\text{-}KEY}}_{\mathsf{LID}}(\mathcal{A}) \leq \mathrm{Adv}^{n\text{-}\mathsf{HR}}(\mathcal{B})$$

Furthermore, for any adversary $\mathcal{A}$ against the uniqueness, we can construct an adversary $\mathcal{B}'$ such that $\mathrm{Adv}^{\mathsf{unq}}_{\mathsf{LID}}(\mathcal{A}) \leq \mathrm{Adv}^{\phi H}(\mathcal{B}')$, LID *has α-bit min-entropy with $\alpha \geq \lambda - 2$,* LID *is commitment-recoverable, and* LID *has an injective simulator.*

The above theorem has been proven in [ABP13] for most of its statements. What is left is a proof for multi-key-indistinguishability and uniqueness, which we provide below.

Lemma 14.1. *For any adversary $\mathcal{A}$ against the multi-key-indistinguishability of* LID *in Figure 14.2, we can construct adversaries $\mathcal{B}$ and $\mathcal{B}'$ such that an adversary $\mathcal{B}$ such that*

$$\mathrm{Adv}^{\mathsf{MU\text{-}IND\text{-}KEY}}_{\mathsf{LID}}(\mathcal{A}) \leq \mathrm{Adv}^{n\text{-}\mathsf{HR}}(\mathcal{B})$$

Proof Sketch. The proof is a straightforward reduction. $\mathcal{B}$ receives $(N, e, y_1, \ldots, y_n)$ as input and defines the common parameters as (N, e) and it simulates $\mathcal{A}$'s oracle $\mathcal{O}$ as follows: Upon the i-th query, it outputs $pk_i := y_i$. When $\mathcal{A}$ outputs a bit b, then $\mathcal{B}$ outputs the same bit b. Note that this defines real keys if the y_i are e-th residues, and lossy keys if the y_i are e-th non-residues. Thus, $\mathcal{B}$ simulates New if the values y_i are e-th residues and $\mathsf{NewLoss}$ otherwise. □

Lemma 14.2. *For any adversary $\mathcal{A}$ against the uniqueness, we can construct an adversary $\mathcal{B}'$ such that* $\mathrm{Adv}^{\mathsf{unq}}_{\mathsf{LID}}(\mathcal{A}) \leq \mathrm{Adv}^{\phi H}(\mathcal{B}')$.

Proof Sketch. The proof is a straightforward reduction. $\mathcal{B}'$ gets a tuple (N, e), now it generates a lossy public key U as defined above and hands (N, e, U) to the adversary $\mathcal{A}$. When the adversary outputs $(\mathsf{cmt}, \mathsf{ch}, \mathsf{resp}, \mathsf{resp}')$, $\mathcal{B}'$ outputs 1 if and only if $\mathsf{resp}^e = (\mathsf{resp}')^e = \mathsf{cmt} \cdot U^{\mathsf{ch}} \wedge \mathsf{resp} \neq \mathsf{resp}'$. We denote the conjunction by win. Now, we conclude

$$\begin{aligned}\mathrm{Adv}^{\mathsf{unq}}_{\mathsf{LID}}(\mathcal{A}) &= \Pr[\mathsf{win} \text{ occurs when } \mathcal{A} \text{ gets } (N, e, U)\ ] \\ &\leq \Pr[\mathsf{win} \text{ occurs when } \mathcal{A} \text{ gets } (N', e, U)\ ] + \mathrm{Adv}^{\phi\mathrm{H}}(\mathcal{B}')\end{aligned}$$

where $e \xleftarrow{\$} \mathbb{P}_{c\ell_N}$, $(N, p_1, p_2) \xleftarrow{\$} \mathsf{RSA}_{\ell_N}[\gcd(e, \phi(N)) = 1]$ and $(N', p_1', p_2') \xleftarrow{\$} \mathsf{RSA}_{\ell_N}[p_1' = 1 \bmod e]$. It remains to analyze $\Pr[\mathsf{win} \text{ occurs when } \mathcal{A} \text{ gets } (N', e, U)\ ]$. We claim that win never occurs if the adversary gets a public key of the form (N', e, U). The reason is that e is coprime to $\phi(N')$. This implies that $x \mapsto x^e \bmod N'$ is a bijection over $\mathbb{Z}_N^*$. Consequently, for every cmt, ch, and U there is at most one resp such that $\mathsf{resp}^e = \mathsf{cmt} \cdot U^{\mathsf{ch}} \bmod N'$. Hence, the lemma follows. □

Finally, we can show that the n-fold higher residuosity assumption is tightly implied by the ϕ-hiding assumption [ABP13, CMS99, KOS10].

Lemma 14.3. *For any adversary $\mathcal{A}$ we can construct adversaries $\mathcal{B}$ and $\mathcal{B}'$ such that*

$$\mathrm{Adv}^{n\text{-}\mathsf{HR}}(\mathcal{A}) \leq \mathrm{Adv}^{\phi H}(\mathcal{B}) + \mathrm{Adv}^{\phi H}(\mathcal{B}') + n/e$$

Proof Sketch. First, we have that

$$\begin{aligned}\mathrm{Adv}^{n\text{-}\mathsf{HR}}(\mathcal{A}) &= |\Pr[\mathcal{A}(N', e, y_1, \ldots, y_n) = 1] - \Pr[\mathcal{A}(N', e, y_1', \ldots, y_n') = 1]| \\ &\leq |\Pr[\mathcal{A}(N', e, y_1, \ldots, y_n) = 1] - \Pr[\mathcal{A}(N, e, r_1, \ldots, r_n) = 1]| \\ &\quad + |\Pr[\mathcal{A}(N, e, r_1, \ldots, r_n) = 1] - \Pr[\mathcal{A}(N', e, r_1, \ldots, r_n) = 1]| \\ &\quad + |\Pr[\mathcal{A}(N', e, r_1, \ldots, r_n) = 1] - \Pr[\mathcal{A}(N', e, y_1', \ldots, y_n') = 1]|,\end{aligned}$$

where where $y_1, \ldots, y_n \xleftarrow{\$} \mathsf{HR}_N[e], y'_1, \ldots, y'_n \xleftarrow{\$} \mathbb{Z}_N^* \setminus \mathsf{HR}_N[e], r_1, \ldots, r_n \xleftarrow{\$} \mathbb{Z}_N^*, (N, p_1, p_2) \xleftarrow{\$} \mathsf{RSA}_{\ell_N}[\gcd(e, \phi(N)) = 1]$ and $(N', p'_1, p'_2) \xleftarrow{\$} \mathsf{RSA}_{\ell_N}[p'_1 = 1 \bmod e]$. Next, we claim that it holds

$$|\Pr[\mathcal{A}(N', e, y_1, \ldots, y_n) = 1] - \Pr[\mathcal{A}(N, e, r_1, \ldots, r_n) = 1]| \leq \mathrm{Adv}^{\phi\mathrm{H}}(\mathcal{B}).$$

We prove this claim by constructing a straightforward reduction $\mathcal{B}$. The reduction receives as input (N, e). It samples $x_1, \ldots, x_n \xleftarrow{\$} \mathbb{Z}_N$ uniformly random and then defines $y_i := x_i^e \bmod N$ for $i \in \{1, \ldots, n\}$. Then it runs $\mathcal{A}$ on input $(N, e, y_1, \ldots, y_n)$ and returns whatever $\mathcal{A}$ outputs. Note that if (N, e) is a "lossy" key, so that $e \mid \phi(N)$, then the y_i are random e-th residues. However, if $\gcd(e, \phi(N)) = 1$, then all $y_i =: r_i$ are uniformly random in $\mathbb{Z}_N^*$ independent of e, since the map $x \mapsto x^e \bmod N$ is a permutation.

Using an even simpler reduction that only choses $r_i \xleftarrow{\$} \mathbb{Z}_{N'}^*$, we can prove that

$$|\Pr[\mathcal{A}(N, e, r_1, \ldots, r_n) = 1] - \Pr[\mathcal{A}(N', e, r_1, \ldots, r_n) = 1]| \leq \mathrm{Adv}^{\phi\mathrm{H}}(\mathcal{B}').$$

Finally, it remains to analyze $|\Pr[\mathcal{A}(N', e, r_1, \ldots, r_n) = 1] - \Pr[\mathcal{A}(N', e, y'_1, \ldots, y'_n) = 1]|$. We claim that this is bounded from above by n/e. To see this, recall $r \xleftarrow{\$} \mathbb{Z}_N^*$ is an e-residue with probability $1/e$ (cf. [ABP13]). Thus, the distributions $\{(r_1, \ldots, r_n) : r_i \xleftarrow{\$} \mathbb{Z}_N^*\}$ and $\{(y'_1, \ldots, y'_n) : y'_i \xleftarrow{\$} \mathbb{Z}_N^* \setminus \mathsf{HR}_N[e]\}$ are statistically close with a statistical distance of n/e.

Overall, we have that $\mathrm{Adv}^{n\text{-}\mathsf{HR}}(\mathcal{A}) \leq \mathrm{Adv}^{\phi\mathrm{H}}(\mathcal{B}) + \mathrm{Adv}^{\phi\mathrm{H}}(\mathcal{B}') + n/e$. □

15

DISCUSSION

In Chapter 13, we presented a generic construction of a tightly MU-sEUF-CMA$^{\text{corr}}$-secure digital signature scheme in the non-programmable random oracle model. The construction is based on lossy identification schemes as defined in Section 4.5 and can be instantiated with every computational problem that gives rise to such a scheme. In Chapter 14, we presented two possible instantiations for our scheme, one based on DDH and the other one based on ϕ-hiding. The scheme is the currently most efficient scheme with tight MU-EUF-CMA$^{\text{corr}}$ security and the only one that is directly strongly unforgeable.[1] Therefore, it is an ideal candidate to be used in combination with an signature-based authenticated key exchange protocol that aims for tight security. In particular, it could be considered for future revisions of TLS if tight security is one goal of the protocol standard in the future.

Open questions. For future work, we consider it interesting to investigate more instantiations of our generic signature scheme construction and to analyse their efficiency. In particular, if our scheme should serve as an alternative to the currently standardized signature schemes for real-world protocols such as the TLS 1.3 handshake, the efficiency of our scheme has to be studied in more detail. Especially, it would be interesting to investigate how competive our construction instantiated under an appropriate assumption in comparison to the standardized signatures, such as RSA-PSS and (EC)DSA, is.

[1] Pan and Wagner [PW22] noted that their construction, which builds upon our construction, can be tightly transformed into a strong signature with a transform using a one-time signature, we however are able to directly prove strong security.

Conclusion

16

CONCLUSION

In this thesis, we gave major insights on the tight security of the TLS 1.3 protocol. To this end, we looked at the TLS 1.3 handshake protocol and the TLS 1.3 record protocol separately. Formally, our results focused mainly on the handshake protocol.

For the handshake protocol, we formally proved tight security bounds for all variants of the handshake protocol with a constant security loss in the multi-stage key exchange (MSKE) model, where prior to our work all (computational) analyses suffered from a quadratic loss in the number of TLS sessions. The proofs are in the random oracle model (ROM) and rely on the assumption that the TLS 1.3 hash function is modeled as a random oracle. In a first step, we showed that the TLS 1.3 key schedule, the key derivation procedure of TLS 1.3, can be abstracted to 12 independent random oracles using the indifferentiability framework relying only on the assumption that the TLS 1.3 hash function is modeled as a random oracle. This abstraction models every key and every MAC value derived in the TLS 1.3 handshake as its own (independent random) function, which reflects the natural intuition one would have when thinking about a key derivation function. Namely, providing "as-good-as-random" keys. With this abstraction at hand, we reduced the complexity of the TLS 1.3 protocol and abstracted the complex, interleaved construction of the key schedule from our perspective for the proof. In particular, this abstraction has the property that the Diffie–Hellman key and the corresponding TLS transcript are input to the same function call. This property allowed us to almost directly apply the technique to prove Diffie–Hellman-based key exchange protocols tightly-secure by Cohn-Gordon et al. [Coh+19]. Here, our abstraction was the crucial ingredient to enable the proof with reasonable assumptions. Finally, we showed that the MSKE-security of the TLS 1.3 full handshake reduces to the existential unforgeability under an adaptive chosen-message attack in the multi-user setting with adaptive corruptions ($\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathsf{corr}}$) of the signature scheme used for authentication and to the strong Diffie–Hellman problem (SDH) in the underlying group. For the PSK-(EC)DHE handshake, we reduced the MSKE security to the strong Diffie–Hellman problem (SDH) in the underlying group under the assumption that the TLS hash function is a random oracle. Both security proofs relied on the assumption that the TLS 1.3 hash function is a random oracle. For the PSK-only handshake, the assumption that TLS hash function is a random oracle was even sufficient to prove tight MSKE-security.

For the TLS 1.3 record protocol, we did not give a formal treatment, but discussed the current state of it with respect to tight security. We remarked that there currently is a lack of a suitable model to capture all features of the record protocol, thus we were not able to give a final answer for the tight security of the record protocol. Nevertheless, it has been shown in prior work that the authenticated encryption with associated data (AEAD) schemes, the main building block of the record protocol, supported by TLS 1.3 can be proven to be tightly multi-user-secure. This is already a significant step towards tightness of the whole record protocol.

Unfortunately, none of the signature schemes supported by the TLS 1.3 handshake is tightly-secure in the sense of $\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathsf{corr}}$-security. That is, the TLS 1.3 full handshake as it is today cannot be instantiated tightly due to the lack of the tight multi-user-security of the signature schemes. To propose an alternative to the standardized signatures, we presented a tightly $\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathsf{corr}}$-secure digital signature with short signatures. The construction is in the random oracle model and is generic. It can be instantiated under any computational problem that gives rise to a loss identification scheme and we present two instantiations in this work: one based on the decisional Diffie–Hellman problem (DDH) and one based on the ϕ-hiding problem. Our signature scheme is the currently most-efficient scheme with tight $\mathsf{MU\text{-}EUF\text{-}CMA}^{\mathsf{corr}}$-security with respect to signature size, as signatures when instantiated with DDH consist only of three $\mathbb{Z}_p$ elements, where p is the prime-order of the underlying Diffie–Hellman group. Moreover, our signature scheme is the first (and only) one that is (directly) tightly $\mathsf{MU\text{-}sEUF\text{-}CMA}^{\mathsf{corr}}$-secure. Therefore, our signature scheme is not only a good candidate to instantiate TLS 1.3 with tight security, but also to be used in any key exchange protocol using authentication based on digital signatures that aim for tight security. Thus, it is also of interest independently of TLS 1.3.

Overall, it remains to be said that with the insights gained in this thesis proving tight security of the overall TLS 1.3 seems to be possible to achieve under reasonably strong assumptions. Even though we did not fully show this formally in this thesis, we advanced the understanding of the tight security of TLS 1.3 protocol by giving formal results for a major part of the protocol via our tight analysis of the TLS 1.3 handshake. As a real-world protocol that was not designed to be tightly-secure, it is almost surprising that proving tight security for the TLS 1.3 handshake protocol is possible. For the remaining open questions discussed in the respective chapters, particularly regarding the TLS 1.3 record protocol, we are confident that these can be solved in further investigations. Finally, we consider investigating the tight security of cryptographic constructions and giving improved, tighter bounds to be valuable. This particularly holds for constructions deployed in the real world and allows for a theoretically-sound deployment of these constructions, in which security does not have to suffer in favor of efficiency.

BIBLIOGRAPHY

[ABR01] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. "The Oracle Diffie-Hellman Assumptions and an Analysis of DHIES". In: *Topics in Cryptology – CT-RSA 2001*. Ed. by David Naccache. Vol. 2020. Lecture Notes in Computer Science. Springer, Heidelberg, April 2001, pp. 143–158. DOI: 10.1007/3-540-45353-9_12 (cit. on pp. 10, 25, 165, 166).

[ABP13] Michel Abdalla, Fabrice Ben Hamouda, and David Pointcheval. "Tighter Reductions for Forward-Secure Signature Schemes". In: *PKC 2013: 16th International Conference on Theory and Practice of Public Key Cryptography*. Ed. by Kaoru Kurosawa and Goichiro Hanaoka. Vol. 7778. Lecture Notes in Computer Science. Springer, Heidelberg, February 2013, pp. 292–311. DOI: 10.1007/978-3-642-36362-7_19 (cit. on pp. 225, 230–234).

[AFLT12] Michel Abdalla, Pierre-Alain Fouque, Vadim Lyubashevsky, and Mehdi Tibouchi. "Tightly-Secure Signatures from Lossy Identification Schemes". In: *Advances in Cryptology – EUROCRYPT 2012*. Ed. by David Pointcheval and Thomas Johansson. Vol. 7237. Lecture Notes in Computer Science. Springer, Heidelberg, April 2012, pp. 572–590. DOI: 10.1007/978-3-642-29011-4_34 (cit. on pp. 11, 14, 33–35, 205, 208, 225).

[AFLT16] Michel Abdalla, Pierre-Alain Fouque, Vadim Lyubashevsky, and Mehdi Tibouchi. "Tightly Secure Signatures From Lossy Identification Schemes". In: *Journal of Cryptology* 29.3 (July 2016), pp. 597–631. DOI: 10.1007/s00145-015-9203-7 (cit. on pp. 33–35, 225).

[AOS02] Masayuki Abe, Miyako Ohkubo, and Koutarou Suzuki. "1-out-of-n Signatures from a Variety of Keys". In: *Advances in Cryptology – ASIACRYPT 2002*. Ed. by Yuliang Zheng. Vol. 2501. Lecture Notes in Computer Science. Springer, Heidelberg, December 2002, pp. 415–432. DOI: 10.1007/3-540-36178-2_26 (cit. on p. 211).

[Adr+15] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice". In: *ACM CCS 2015: 22nd Conference on Computer and Communications Security*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. ACM Press, October 2015, pp. 5–17. DOI: 10.1145/2810103.2813707 (cit. on p. 15).

[AlF+13] Nadhem J. AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. "On the Security of RC4 in TLS". In: *USENIX Security 2013: 22nd USENIX Security Symposium*. Ed. by Samuel T. King. USENIX Association, August 2013, pp. 305–320 (cit. on p. 15).

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols". In: *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2013, pp. 526–540. DOI: 10.1109/SP.2013.42 (cit. on p. 15).

[Arf+19] Ghada Arfaoui, Xavier Bultel, Pierre-Alain Fouque, Adina Nedelcu, and Cristina Onete. "The privacy of the TLS 1.3 protocol". In: *Proceedings on Privacy Enhancing Technologies* 2019.4 (October 2019), pp. 190–210. DOI: 10.2478/popets-2019-0065 (cit. on pp. 16, 129).

[AGJ19] Nimrod Aviram, Kai Gellert, and Tibor Jager. "Session Resumption Protocols and Efficient Forward Security for TLS 1.3 0-RTT". In: *Advances in Cryptology – EUROCRYPT 2019, Part II*. Ed. by Yuval Ishai and Vincent Rijmen. Vol. 11477. Lecture Notes in Computer Science. Springer, Heidelberg, May 2019, pp. 117–150. DOI: 10.1007/978-3-030-17656-3_5 (cit. on p. 16).

[AGJ21] Nimrod Aviram, Kai Gellert, and Tibor Jager. "Session Resumption Protocols and Efficient Forward Security for TLS 1.3 0-RTT". In: *Journal of Cryptology* 34.3 (July 2021), p. 20. DOI: 10.1007/s00145-021-09385-0 (cit. on p. 16).

[Avi+16] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. "DROWN: Breaking TLS Using SSLv2". In: *USENIX Security 2016: 25th USENIX Security Symposium*. Ed. by Thorsten Holz and Stefan Savage. USENIX Association, August 2016, pp. 689–706 (cit. on pp. 3, 15).

[Bad+15a] Christoph Bader, Dennis Hofheinz, Tibor Jager, Eike Kiltz, and Yong Li. "Tightly-Secure Authenticated Key Exchange". In: *TCC 2015: 12th Theory of Cryptography Conference, Part I*. Ed. by Yevgeniy Dodis and Jesper Buus Nielsen. Vol. 9014. Lecture Notes in Computer Science. Springer, Heidelberg, March 2015, pp. 629–658. DOI: 10.1007/978-3-662-46494-6_26 (cit. on pp. 11, 13, 14, 16, 33, 207, 208, 212).

[BJLS16] Christoph Bader, Tibor Jager, Yong Li, and Sven Schäge. "On the Impossibility of Tight Cryptographic Reductions". In: *Advances in Cryptology – EUROCRYPT 2016, Part II*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Vol. 9666. Lecture Notes in Computer Science. Springer, Heidelberg, May 2016, pp. 273–304. DOI: 10.1007/978-3-662-49896-5_10 (cit. on pp. 11, 14, 166, 206).

[Bad+15b] Christian Badertscher, Christian Matt, Ueli Maurer, Phillip Rogaway, and Björn Tackmann. "Augmented Secure Channels and the Goal of the TLS 1.3 Record Layer". In: *ProvSec 2015: 9th International Conference on Provable Security*. Ed. by Man Ho Au and Atsuko Miyaji. Vol. 9451. Lecture Notes in Computer Science. Springer, Heidelberg, November 2015, pp. 85–104. DOI: 10.1007/978-3-319-26059-4_5 (cit. on p. 8).

[BTPL15] R. Barnes, M. Thomson, A. Pironti, and A. Langley. "Deprecating Secure Sockets Layer Version 3.0". RFC 7568. IETF, June 2015. URL: http://tools.ietf.org/rfc/rfc7568.txt (cit. on p. 3).

[BPS15] Guy Barwell, Dan Page, and Martijn Stam. "Rogue Decryption Failures: Reconciling AE Robustness Notions". Cryptology ePrint Archive, Report 2015/895. https://eprint.iacr.org/2015/895. 2015 (cit. on p. 201).

[Bel06] Mihir Bellare. "New Proofs for NMAC and HMAC: Security without Collision-Resistance". In: *Advances in Cryptology – CRYPTO 2006*. Ed. by Cynthia Dwork. Vol. 4117. Lecture Notes in Computer Science. Springer, Heidelberg, August 2006, pp. 602–619. DOI: 10.1007/11818175_36 (cit. on p. 62).

[Bel15] Mihir Bellare. "New Proofs for NMAC and HMAC: Security without Collision Resistance". In: *Journal of Cryptology* 28.4 (October 2015), pp. 844–878. DOI: 10.1007/s00145-014-9185-x (cit. on p. 62).

[BBM00] Mihir Bellare, Alexandra Boldyreva, and Silvio Micali. "Public-Key Encryption in a Multi-user Setting: Security Proofs and Improvements". In: *Advances in Cryptology – EUROCRYPT 2000*. Ed. by Bart Preneel. Vol. 1807. Lecture Notes in Computer Science. Springer, Heidelberg, May 2000, pp. 259–274. DOI: 10.1007/3-540-45539-6_18 (cit. on p. 228).

[BCK96] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. "Keying Hash Functions for Message Authentication". In: *Advances in Cryptology – CRYPTO'96*. Ed. by Neal Koblitz. Vol. 1109. Lecture Notes in Computer Science. Springer, Heidelberg, August 1996, pp. 1–15. DOI: 10.1007/3-540-68697-5_1 (cit. on p. 62).

[BDG20] Mihir Bellare, Hannah Davis, and Felix Günther. "Separate Your Domains: NIST PQC KEMs, Oracle Cloning and Read-Only Indifferentiability". In: *Advances in Cryptology – EUROCRYPT 2020, Part II*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12106. Lecture Notes in Computer Science. Springer, Heidelberg, May 2020, pp. 3–32. DOI: 10.1007/978-3-030-45724-2_1 (cit. on pp. 9, 79, 82, 86, 87, 91, 92).

[BDJR97] Mihir Bellare, Anand Desai, Eric Jokipii, and Phillip Rogaway. "A Concrete Security Treatment of Symmetric Encryption". In: *38th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, October 1997, pp. 394–403. DOI: 10.1109/SFCS.1997.646128 (cit. on p. 85).

[BKN02] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. "Authenticated Encryption in SSH: Provably Fixing The SSH Binary Packet Protocol". In: *ACM CCS 2002: 9th Conference on Computer and Communications Security*. Ed. by Vijayalakshmi Atluri. ACM Press, November 2002, pp. 1–11. DOI: 10.1145/586110.586112 (cit. on p. 201).

[BR09a] Mihir Bellare and Thomas Ristenpart. "Simulation without the Artificial Abort: Simplified Proof and Improved Concrete Security for Waters' IBE Scheme". Cryptology ePrint Archive, Report 2009/084. https://eprint.iacr.org/2009/084. 2009 (cit. on p. 6).

[BR09b] Mihir Bellare and Thomas Ristenpart. "Simulation without the Artificial Abort: Simplified Proof and Improved Concrete Security for Waters' IBE Scheme". In: *Advances in Cryptology – EUROCRYPT 2009*. Ed. by Antoine Joux. Vol. 5479. Lecture Notes in Computer Science. Springer, Heidelberg, April 2009, pp. 407–424. DOI: 10.1007/978-3-642-01001-9_24 (cit. on p. 6).

[BR93] Mihir Bellare and Phillip Rogaway. "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols". In: *ACM CCS 93: 1st Conference on Computer and Communications Security*. Ed. by Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby. ACM Press, November 1993, pp. 62–73. DOI: 10.1145/168588.168596 (cit. on pp. 28, 29).

[BR94] Mihir Bellare and Phillip Rogaway. "Entity Authentication and Key Distribution". In: *Advances in Cryptology – CRYPTO'93*. Ed. by Douglas R. Stinson. Vol. 773. Lecture Notes in Computer Science. Springer, Heidelberg, August 1994, pp. 232–249. DOI: 10.1007/3-540-48329-2_21 (cit. on pp. 12, 15, 40, 44, 48, 59, 198, 209).

[BR04] Mihir Bellare and Phillip Rogaway. "Code-Based Game-Playing Proofs and the Security of Triple Encryption". Cryptology ePrint Archive, Report 2004/331. https://eprint.iacr.org/2004/331. 2004 (cit. on p. 40).

[BR06] Mihir Bellare and Phillip Rogaway. "The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs". In: *Advances in Cryptology – EUROCRYPT 2006*. Ed. by Serge Vaudenay. Vol. 4004. Lecture Notes in Computer Science. Springer, Heidelberg, May 2006, pp. 409–426. DOI: 10.1007/11761679_25 (cit. on pp. 40, 101, 217).

[BT16] Mihir Bellare and Björn Tackmann. "The Multi-user Security of Authenticated Encryption: AES-GCM in TLS 1.3". In: *Advances in Cryptology – CRYPTO 2016, Part I*. Ed. by Matthew Robshaw and Jonathan Katz. Vol. 9814. Lecture Notes in Computer Science. Springer, Heidelberg, August 2016, pp. 247–276. DOI: 10.1007/978-3-662-53018-4_10 (cit. on pp. 8, 9, 200).

[Ben+95] Josh Benaloh, Butler Lampson, Daniel Simon, Terence Spies, and Bennet Yee. "The Private Communication Technology Protocol". Internet-Draft draft-benaloh-pct-00. Work in Progress. Internet Engineering Task Force, November 1995. 40 pp. URL: https://datatracker.ietf.org/doc/draft-benaloh-pct/00/ (cit. on p. 2).

[Ber+11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "High-Speed High-Security Signatures". In: *Cryptographic Hardware and Embedded Systems – CHES 2011*. Ed. by Bart Preneel and Tsuyoshi Takagi. Vol. 6917. Lecture Notes in Computer Science. Springer, Heidelberg, September 2011, pp. 124–142. DOI: 10.1007/978-3-642-23951-9_9 (cit. on p. 11).

[Beu+15] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. "A Messy State of the Union: Taming the Composite State Machines of TLS". In: *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2015, pp. 535–552. DOI: 10.1109/SP.2015.39 (cit. on pp. 15, 16).

[BBK17] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. "Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate". In: *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2017, pp. 483–502. DOI: 10.1109/SP.2017.26 (cit. on p. 16).

[Bha+16] Karthikeyan Bhargavan, Christina Brzuska, Cédric Fournet, Matthew Green, Markulf Kohlweiss, and Santiago Zanella-Béguelin. "Downgrade Resilience in Key-Exchange Protocols". In: *2016 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2016, pp. 506–525. DOI: 10.1109/SP.2016.37 (cit. on pp. 16, 63).

[Bha+14a] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. "Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS". In: *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2014, pp. 98–113. DOI: 10.1109/SP.2014.14 (cit. on p. 16).

[Bha+13] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. "Implementing TLS with Verified Cryptographic Security". In: *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2013, pp. 445–459. DOI: 10.1109/SP.2013.37 (cit. on p. 15).

[Bha+14b] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Santiago Zanella Béguelin. "Proving the TLS Handshake Secure (As It Is)". In: *Advances in Cryptology – CRYPTO 2014, Part II*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8617. Lecture Notes in Computer Science. Springer, Heidelberg,

August 2014, pp. 235–255. DOI: 10.1007/978-3-662-44381-1_14 (cit. on p. 15).

[BL16a] Karthikeyan Bhargavan and Gaëtan Leurent. "On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN". In: *ACM CCS 2016: 23rd Conference on Computer and Communications Security*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. ACM Press, October 2016, pp. 456–467. DOI: 10.1145/2976749.2978423 (cit. on p. 15).

[BL16b] Karthikeyan Bhargavan and Gaëtan Leurent. "Transcript Collision Attacks: Breaking Authentication in TLS, IKE and SSH". In: *ISOC Network and Distributed System Security Symposium – NDSS 2016*. The Internet Society, February 2016 (cit. on pp. 3, 15).

[Ble98] Daniel Bleichenbacher. "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1". In: *Advances in Cryptology – CRYPTO'98*. Ed. by Hugo Krawczyk. Vol. 1462. Lecture Notes in Computer Science. Springer, Heidelberg, August 1998, pp. 1–12. DOI: 10.1007/BFb0055716 (cit. on p. 15).

[BSY18] Hanno Böck, Juraj Somorovsky, and Craig Young. "Return Of Bleichenbacher's Oracle Threat (ROBOT)". In: *USENIX Security 2018: 27th USENIX Security Symposium*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, August 2018, pp. 817–849 (cit. on p. 16).

[Bod14] Krzysztof Kotowicz Bodo Möller Thai Duong. "This POODLE bites: Exploiting the SSL 3.0 fallback". https://www.openssl.org/~bodo/ssl-poodle.pdf. September 2014 (cit. on pp. 3, 15).

[BDPS12] Alexandra Boldyreva, Jean Paul Degabriele, Kenneth G. Paterson, and Martijn Stam. "Security of Symmetric Encryption in the Presence of Ciphertext Fragmentation". In: *Advances in Cryptology – EUROCRYPT 2012*. Ed. by David Pointcheval and Thomas Johansson. Vol. 7237. Lecture Notes in Computer Science. Springer, Heidelberg, April 2012, pp. 682–699. DOI: 10.1007/978-3-642-29011-4_40 (cit. on p. 201).

[BDPS14] Alexandra Boldyreva, Jean Paul Degabriele, Kenneth G. Paterson, and Martijn Stam. "On Symmetric Encryption with Distinguishable Decryption Failures". In: *Fast Software Encryption – FSE 2013*. Ed. by Shiho Moriai. Vol. 8424. Lecture Notes in Computer Science. Springer, Heidelberg, March 2014, pp. 367–390. DOI: 10.1007/978-3-662-43933-3_19 (cit. on p. 201).

[Boy+13] Colin Boyd, Cas Cremers, Michele Feltz, Kenneth G. Paterson, Bertram Poettering, and Douglas Stebila. "ASICS: Authenticated Key Exchange Security Incorporating Certification Systems". In: *ESORICS 2013: 18th European Symposium on Research in Computer Security*. Ed. by Jason Cramp-

ton, Sushil Jajodia, and Keith Mayes. Vol. 8134. Lecture Notes in Computer Science. Springer, Heidelberg, September 2013, pp. 381–399. DOI: 10.1007/978-3-642-40203-6_22 (cit. on p. 43).

[BG20] Colin Boyd and Kai Gellert. "A Modern View on Forward Security". In: *The Computer Journal* 64.4 (August 2020), pp. 639–652. DOI: 10.1093/comjnl/bxaa104. URL: https://doi.org/10.1093/comjnl/bxaa104 (cit. on p. 44).

[Bra93] Stefan Brands. "An Efficient Off-Line Electronic Cash System Based On The Representation Problem". Tech. rep. CS-R9323. CWI, 1993 (cit. on p. 24).

[BFG19] Jacqueline Brendel, Marc Fischlin, and Felix Günther. "Breakdown Resilience of Key Exchange Protocols: NewHope, TLS 1.3, and Hybrids". In: *ESORICS 2019: 24th European Symposium on Research in Computer Security, Part II*. Ed. by Kazue Sako, Steve Schneider, and Peter Y. A. Ryan. Vol. 11736. Lecture Notes in Computer Science. Springer, Heidelberg, September 2019, pp. 521–541. DOI: 10.1007/978-3-030-29962-0_25 (cit. on p. 16).

[BFGJ17] Jacqueline Brendel, Marc Fischlin, Felix Günther, and Christian Janson. "PRF-ODH: Relations, Instantiations, and Impossibility Results". In: *Advances in Cryptology – CRYPTO 2017, Part III*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10403. Lecture Notes in Computer Science. Springer, Heidelberg, August 2017, pp. 651–681. DOI: 10.1007/978-3-319-63697-9_22 (cit. on pp. 166, 167).

[BFWW11] Christina Brzuska, Marc Fischlin, Bogdan Warinschi, and Stephen C. Williams. "Composability of Bellare-Rogaway key exchange protocols". In: *ACM CCS 2011: 18th Conference on Computer and Communications Security*. Ed. by Yan Chen, George Danezis, and Vitaly Shmatikov. ACM Press, October 2011, pp. 51–62. DOI: 10.1145/2046707.2046716 (cit. on pp. 59, 198).

[CMS99] Christian Cachin, Silvio Micali, and Markus Stadler. "Computationally Private Information Retrieval with Polylogarithmic Communication". In: *Advances in Cryptology – EUROCRYPT'99*. Ed. by Jacques Stern. Vol. 1592. Lecture Notes in Computer Science. Springer, Heidelberg, May 1999, pp. 402–414. DOI: 10.1007/3-540-48910-X_28 (cit. on pp. 225, 230, 232, 233).

[Can01] Ran Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *42nd Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, October 2001, pp. 136–145. DOI: 10.1109/SFCS.2001.959888 (cit. on p. 15).

[CK01] Ran Canetti and Hugo Krawczyk. "Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels". In: *Advances in Cryptology – EUROCRYPT 2001*. Ed. by Birgit Pfitzmann. Vol. 2045. Lecture Notes

in Computer Science. Springer, Heidelberg, May 2001, pp. 453–474. DOI: 10.1007/3-540-44987-6_28 (cit. on p. 12).

[CK02] Ran Canetti and Hugo Krawczyk. "Security Analysis of IKE's Signature-based Key-Exchange Protocol". In: *Advances in Cryptology – CRYPTO 2002*. Ed. by Moti Yung. Vol. 2442. Lecture Notes in Computer Science. https://eprint.iacr.org/2002/120/. Springer, Heidelberg, August 2002, pp. 143–161. DOI: 10.1007/3-540-45708-9_10 (cit. on p. 40).

[Cel+21] Sofía Celi, Armando Faz-Hernández, Nick Sullivan, Goutam Tamvada, Luke Valenta, Thom Wiggers, Bas Westerbaan, and Christopher A. Wood. "Implementing and Measuring KEMTLS". In: *Progress in Cryptology - LATINCRYPT 2021: 7th International Conference on Cryptology and Information Security in Latin America*. Ed. by Patrick Longa and Carla Ràfols. Vol. 12912. Lecture Notes in Computer Science. Bogotá, Colombia: Springer, Heidelberg, October 2021, pp. 88–107. DOI: 10.1007/978-3-030-88238-9_5 (cit. on p. 16).

[CHSW22] Sofía Celi, Jonathan Hoyland, Douglas Stebila, and Thom Wiggers. "A Tale of Two Models: Formal Verification of KEMTLS via Tamarin". In: *ESORICS 2022: 27th European Symposium on Research in Computer Security, Part III*. Ed. by Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng. Vol. 13556. Lecture Notes in Computer Science. Springer, Heidelberg, September 2022, pp. 63–83. DOI: 10.1007/978-3-031-17143-7_4 (cit. on p. 16).

[CEv88] David Chaum, Jan-Hendrik Evertse, and Jeroen van de Graaf. "An Improved Protocol for Demonstrating Possession of Discrete Logarithms and Some Generalizations". In: *Advances in Cryptology – EUROCRYPT'87*. Ed. by David Chaum and Wyn L. Price. Vol. 304. Lecture Notes in Computer Science. Springer, Heidelberg, April 1988, pp. 127–141. DOI: 10.1007/3-540-39118-5_13 (cit. on pp. 225, 228).

[Cod14] Codenomicon. "Heartbleed Bug". http://heartbleed.com/. April 2014 (cit. on pp. 3, 16).

[Coh+19] Katriel Cohn-Gordon, Cas Cremers, Kristian Gjøsteen, Håkon Jacobsen, and Tibor Jager. "Highly Efficient Key Exchange Protocols with Optimal Tightness". In: *Advances in Cryptology – CRYPTO 2019, Part III*. Ed. by Alexandra Boldyreva and Daniele Micciancio. Vol. 11694. Lecture Notes in Computer Science. Springer, Heidelberg, August 2019, pp. 767–797. DOI: 10.1007/978-3-030-26954-8_25 (cit. on pp. 13, 26, 135, 137, 138, 144, 169, 176, 239).

[Cor02] Jean-Sébastien Coron. "Optimal Security Proofs for PSS and Other Signature Schemes". In: *Advances in Cryptology – EUROCRYPT 2002*. Ed. by Lars R. Knudsen. Vol. 2332. Lecture Notes in Computer Science. Springer, Heidelberg, April 2002, pp. 272–287. DOI: 10.1007/3-540-46035-7_18 (cit. on p. 11).

[CDS94] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. "Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols". In: *Advances in Cryptology – CRYPTO'94*. Ed. by Yvo Desmedt. Vol. 839. Lecture Notes in Computer Science. Springer, Heidelberg, August 1994, pp. 174–187. DOI: 10.1007/3-540-48658-5_19 (cit. on p. 211).

[Cre+17] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. "A Comprehensive Symbolic Analysis of TLS 1.3". In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. ACM Press, October 2017, pp. 1773–1788. DOI: 10.1145/3133956.3134063 (cit. on p. 16).

[CHSv16] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication". In: *2016 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2016, pp. 470–485. DOI: 10.1109/SP.2016.35 (cit. on p. 16).

[Dam10] Ivan Damård. "On Σ-protocols". https://cs.au.dk/~ivan/Sigma.pdf. 2010 (cit. on p. 225).

[DDGJ22a] Hannah Davis, Denis Diemert, Felix Günther, and Tibor Jager. "On the Concrete Security of TLS 1.3 PSK Mode". Cryptology ePrint Archive, Report 2022/246. https://eprint.iacr.org/2022/246. 2022 (cit. on pp. 39, 77, 99, 101, 106, 108, 125, 126, 129, 135, 169).

[DDGJ22b] Hannah Davis, Denis Diemert, Felix Günther, and Tibor Jager. "On the Concrete Security of TLS 1.3 PSK Mode". In: *Advances in Cryptology – EUROCRYPT 2022, Part II*. Ed. by Orr Dunkelman and Stefan Dziembowski. Vol. 13276. Lecture Notes in Computer Science. Springer, Heidelberg, May 2022, pp. 876–906. DOI: 10.1007/978-3-031-07085-3_30 (cit. on pp. 10, 39, 40, 42, 45, 47, 52, 56, 77, 79, 82, 84, 85, 88, 92–94, 99, 101, 106, 125, 126, 129–131, 133, 135, 138, 169, 200).

[DG20] Hannah Davis and Felix Günther. "Tighter Proofs for the SIGMA and TLS 1.3 Key Exchange Protocols". Cryptology ePrint Archive, Report 2020/1029. https://eprint.iacr.org/2020/1029. 2020 (cit. on p. 165).

[DG21a] Hannah Davis and Felix Günther. "Tighter Proofs for the SIGMA and TLS 1.3 Key Exchange Protocols". In: *ACNS 21: 19th International Conference on Applied Cryptography and Network Security, Part II*. Ed. by Kazue Sako and Nils Ole Tippenhauer. Vol. 12727. Lecture Notes in Computer Science. Springer, Heidelberg, June 2021, pp. 448–479. DOI: 10.1007/978-3-030-78375-4_18 (cit. on pp. 10, 42, 56, 77–79, 135, 138, 144, 165, 169).

[dFW20] Cyprien de Saint Guilhem, Marc Fischlin, and Bogdan Warinschi. "Authentication in Key-Exchange: Definitions, Relations and Composition". In: *CSF 2020: IEEE 33st Computer Security Foundations Symposium*. Ed. by Limin Jia and Ralf Küsters. IEEE Computer Society Press, 2020, pp. 288–303. DOI: 10.1109/CSF49147.2020.00028 (cit. on pp. 42, 163).

[DGGP21] Jean Paul Degabriele, Jérôme Govinden, Felix Günther, and Kenneth G. Paterson. "The Security of ChaCha20-Poly1305 in the Multi-User Setting". In: *ACM CCS 2021: 28th Conference on Computer and Communications Security*. Ed. by Giovanni Vigna and Elaine Shi. ACM Press, November 2021, pp. 1981–2003. DOI: 10.1145/3460120.3484814 (cit. on pp. 8, 9, 200).

[DK22] Jean Paul Degabriele and Vukašin Karadžić. "Overloading the Nonce: Rugged PRPs, Nonce-Set AEAD, and Order-Resilient Channels". In: *Advances in Cryptology – CRYPTO 2022, Part IV*. Ed. by Yevgeniy Dodis and Thomas Shrimpton. Vol. 13510. Lecture Notes in Computer Science. Springer, Heidelberg, August 2022, pp. 264–295. DOI: 10.1007/978-3-031-15985-5_10 (cit. on p. 201).

[Del+17] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. "Implementing and Proving the TLS 1.3 Record Layer". In: *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2017, pp. 463–482. DOI: 10.1109/SP.2017.58 (cit. on pp. 8, 16, 201).

[DGJL21a] Denis Diemert, Kai Gellert, Tibor Jager, and Lin Lyu. "More Efficient Digital Signatures with Tight Multi-User Security". Cryptology ePrint Archive, Report 2021/235. https://eprint.iacr.org/2021/235. 2021 (cit. on p. 205).

[DGJL21b] Denis Diemert, Kai Gellert, Tibor Jager, and Lin Lyu. "More Efficient Digital Signatures with Tight Multi-user Security". In: *PKC 2021: 24th International Conference on Theory and Practice of Public Key Cryptography, Part II*. Ed. by Juan Garay. Vol. 12711. Lecture Notes in Computer Science. Springer, Heidelberg, May 2021, pp. 1–31. DOI: 10.1007/978-3-030-75248-4_1 (cit. on pp. 17, 33, 36, 205, 208, 209, 211).

[DJ21] Denis Diemert and Tibor Jager. "On the Tight Security of TLS 1.3: Theoretically Sound Cryptographic Parameters for Real-World Deployments". In: *Journal of Cryptology* 34.3 (July 2021), p. 30. DOI: 10.1007/s00145-021-09388-x (cit. on pp. 10, 11, 77–79, 82, 135, 138, 169, 200, 201).

[DA99] T. Dierks and C. Allen. "The TLS Protocol Version 1.0". RFC 2246. IETF, January 1999. URL: http://tools.ietf.org/rfc/rfc2246.txt (cit. on p. 3).

[DR06] T. Dierks and E. Rescorla. "The Transport Layer Security (TLS) Protocol Version 1.1". RFC 4346. IETF, April 2006. URL: http://tools.ietf.org/rfc/rfc4346.txt (cit. on p. 3).

[DH76] Whitfield Diffie and Martin E. Hellman. "New Directions in Cryptography". In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654 (cit. on pp. 9, 24).

[DRST13] Yevgeniy Dodis, Thomas Ristenpart, John Steinberger, and Stefano Tessaro. "To Hash or Not to Hash Again? (In)differentiability Results for H^2 and HMAC". Cryptology ePrint Archive, Report 2013/382. https://eprint.iacr.org/2013/382. 2013 (cit. on pp. 94–96).

[DRST12] Yevgeniy Dodis, Thomas Ristenpart, John P. Steinberger, and Stefano Tessaro. "To Hash or Not to Hash Again? (In)Differentiability Results for H^2 and HMAC". In: *Advances in Cryptology – CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Lecture Notes in Computer Science. Springer, Heidelberg, August 2012, pp. 348–366. DOI: 10.1007/978-3-642-32009-5_21 (cit. on pp. 90, 94).

[DFGS15] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. "A Cryptographic Analysis of the TLS 1.3 Handshake Protocol Candidates". In: *ACM CCS 2015: 22nd Conference on Computer and Communications Security*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. ACM Press, October 2015, pp. 1197–1210. DOI: 10.1145/2810103.2813653 (cit. on pp. 8, 10, 11, 40, 59, 198, 200).

[DFGS16] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. "A Cryptographic Analysis of the TLS 1.3 draft-10 Full and Pre-shared Key Handshake Protocol". Cryptology ePrint Archive, Report 2016/081. https://eprint.iacr.org/2016/081. 2016 (cit. on pp. 8, 40, 198, 200).

[DFGS21] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. "A Cryptographic Analysis of the TLS 1.3 Handshake Protocol". In: *Journal of Cryptology* 34.4 (October 2021), p. 37. DOI: 10.1007/s00145-021-09384-1 (cit. on pp. 8, 11, 39, 40, 42, 43, 52, 56, 63, 166, 167, 199, 200).

[DS15] Benjamin Dowling and Douglas Stebila. "Modelling Ciphersuite and Version Negotiation in the TLS Protocol". In: *ACISP 15: 20th Australasian Conference on Information Security and Privacy*. Ed. by Ernest Foo and Douglas Stebila. Vol. 9144. Lecture Notes in Computer Science. Springer, Heidelberg, June 2015, pp. 270–288. DOI: 10.1007/978-3-319-19962-7_16 (cit. on pp. 15, 63).

[DG21b] Nir Drucker and Shay Gueron. "Selfie: reflections on TLS 1.3 with PSK". In: *Journal of Cryptology* 34.3 (July 2021), p. 27. DOI: 10.1007/s00145-021-09387-y (cit. on pp. 16, 44).

[Duo11] Thai Duong. "BEAST". http://vnhacker.blogspot.com.au/2011/09/beast.html. September 2011 (cit. on pp. 3, 15).

[Dwo15] Morris Dworkin. "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions". NIST FIPS 202. August 2015. DOI: 10.6028/NIST.FIPS.202. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf (cit. on p. 28).

[EH06] D. Eastlake 3rd and T. Hansen. "US Secure Hash Algorithms (SHA and HMAC-SHA)". RFC 4634. IETF, July 2006. URL: http://tools.ietf.org/rfc/rfc4634.txt (cit. on p. 28).

[EH11] D. Eastlake 3rd and T. Hansen. "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)". RFC 6234. IETF, May 2011. URL: http://tools.ietf.org/rfc/rfc6234.txt (cit. on p. 28).

[EJ01] D. Eastlake 3rd and P. Jones. "US Secure Hash Algorithm 1 (SHA1)". RFC 3174. IETF, September 2001. URL: http://tools.ietf.org/rfc/rfc3174.txt (cit. on p. 28).

[Edd22] W. Eddy. "Transmission Control Protocol (TCP)". RFC 9293. IETF, August 2022. URL: http://tools.ietf.org/rfc/rfc9293.txt (cit. on p. 2).

[EH95] Dr. Taher Elgamal and Kipp E.B. Hickman. "The SSL Protocol". Internet-Draft draft-hickman-netscape-ssl-00. Work in Progress. Internet Engineering Task Force, April 1995. 31 pp. URL: https://datatracker.ietf.org/doc/draft-hickman-netscape-ssl/00/ (cit. on p. 2).

[ET05] P. Eronen and H. Tschofenig. "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)". RFC 4279. IETF, December 2005. URL: http://tools.ietf.org/rfc/rfc4279.txt (cit. on p. 15).

[FKP17] Manuel Fersch, Eike Kiltz, and Bertram Poettering. "On the One-Per-Message Unforgeability of (EC)DSA and Its Variants". In: *TCC 2017: 15th Theory of Cryptography Conference, Part II*. Ed. by Yael Kalai and Leonid Reyzin. Vol. 10678. Lecture Notes in Computer Science. Springer, Heidelberg, November 2017, pp. 519–534. DOI: 10.1007/978-3-319-70503-3_17 (cit. on p. 11).

[FS87] Amos Fiat and Adi Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *Advances in Cryptology – CRYPTO'86*. Ed. by Andrew M. Odlyzko. Vol. 263. Lecture Notes in Computer Science. Springer, Heidelberg, August 1987, pp. 186–194. DOI: 10.1007/3-540-47721-7_12 (cit. on p. 211).

[FG14] Marc Fischlin and Felix Günther. "Multi-Stage Key Exchange and the Case of Google's QUIC Protocol". In: *ACM CCS 2014: 21st Conference on Computer and Communications Security*. Ed. by Gail-Joon Ahn, Moti Yung, and Ninghui Li. ACM Press, November 2014, pp. 1193–1204. DOI: 10.1145/2660267.2660308 (cit. on pp. 10, 40, 198).

[FG17] Marc Fischlin and Felix Günther. "Replay Attacks on Zero Round-Trip Time: The Case of the TLS 1.3 Handshake Candidates". In: *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017*. IEEE, April 2017, pp. 60–75 (cit. on pp. 8, 40, 198, 200).

[FGJ20] Marc Fischlin, Felix Günther, and Christian Janson. "Robust Channels: Handling Unreliable Networks in the Record Layers of QUIC and DTLS 1.3". Cryptology ePrint Archive, Report 2020/718. https://eprint.iacr.org/2020/718. 2020 (cit. on p. 201).

[FGMP15] Marc Fischlin, Felix Günther, Giorgia Azzurra Marson, and Kenneth G. Paterson. "Data Is a Stream: Security of Stream-Based Channels". In: *Advances in Cryptology – CRYPTO 2015, Part II*. Ed. by Rosario Gennaro and Matthew J. B. Robshaw. Vol. 9216. Lecture Notes in Computer Science. Springer, Heidelberg, August 2015, pp. 545–564. DOI: 10.1007/978-3-662-48000-7_27 (cit. on p. 201).

[FGSW16] Marc Fischlin, Felix Günther, Benedikt Schmidt, and Bogdan Warinschi. "Key Confirmation in Key Exchange: A Formal Treatment and Implications for TLS 1.3". In: *2016 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2016, pp. 452–469. DOI: 10.1109/SP.2016.34 (cit. on pp. 16, 42, 163).

[FHJ20] Marc Fischlin, Patrick Harasser, and Christian Janson. "Signatures from Sequential-OR Proofs". In: *Advances in Cryptology – EUROCRYPT 2020, Part III*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12107. Lecture Notes in Computer Science. Springer, Heidelberg, May 2020, pp. 212–244. DOI: 10.1007/978-3-030-45727-3_8 (cit. on pp. 35, 211, 212).

[FKK11] A. Freier, P. Karlton, and P. Kocher. "The Secure Sockets Layer (SSL) Protocol Version 3.0". RFC 6101. IETF, August 2011. URL: http://tools.ietf.org/rfc/rfc6101.txt (cit. on p. 2).

[Gaj08] Sebastian Gajek. "A Universally Composable Framework for the Analysis of Browser-Based Security Protocols". In: *ProvSec 2008: 2nd International Conference on Provable Security*. Ed. by Joonsang Baek, Feng Bao, Kefei Chen, and Xuejia Lai. Vol. 5324. Lecture Notes in Computer Science. Springer, Heidelberg, October 2008, pp. 283–297 (cit. on p. 15).

[Gaj+08] Sebastian Gajek, Mark Manulis, Olivier Pereira, Ahmad-Reza Sadeghi, and Jörg Schwenk. "Universally Composable Security Analysis of TLS". In: *ProvSec 2008: 2nd International Conference on Provable Security*. Ed. by Joonsang Baek, Feng Bao, Kefei Chen, and Xuejia Lai. Vol. 5324. Lecture Notes in Computer Science. Springer, Heidelberg, October 2008, pp. 313–327 (cit. on p. 15).

[GKS13] Florian Giesen, Florian Kohlar, and Douglas Stebila. "On the security of TLS renegotiation". In: *ACM CCS 2013: 20th Conference on Computer and Communications Security*. Ed. by Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung. ACM Press, November 2013, pp. 387–398. DOI: 10.1145/2508859.2516694 (cit. on p. 15).

[Gil16] D. Gillmor. "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)". RFC 7919. IETF, August 2016. URL: http://tools.ietf.org/rfc/rfc7919.txt (cit. on p. 108).

[GJ18] Kristian Gjøsteen and Tibor Jager. "Practical and Tightly-Secure Digital Signatures and Authenticated Key Exchange". In: *Advances in Cryptology – CRYPTO 2018, Part II*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10992. Lecture Notes in Computer Science. Springer, Heidelberg, August 2018, pp. 95–125. DOI: 10.1007/978-3-319-96881-0_4 (cit. on pp. 11–14, 17, 137, 208, 209).

[GM82] Shafi Goldwasser and Silvio Micali. "Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information". In: *14th Annual ACM Symposium on Theory of Computing*. ACM Press, May 1982, pp. 365–377. DOI: 10.1145/800070.802212 (cit. on p. 4).

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. "A Digital Signature Scheme Secure Against Adaptive Chosen-message Attacks". In: *SIAM Journal on Computing* 17.2 (April 1988), pp. 281–308 (cit. on p. 31).

[GQ90] Louis C. Guillou and Jean-Jacques Quisquater. "A "Paradoxical" Indentity-Based Signature Scheme Resulting from Zero-Knowledge". In: *Advances in Cryptology – CRYPTO'88*. Ed. by Shafi Goldwasser. Vol. 403. Lecture Notes in Computer Science. Springer, Heidelberg, August 1990, pp. 216–231. DOI: 10.1007/0-387-34799-2_16 (cit. on pp. 230, 232).

[Gün18] Felix Günther. "Modeling Advanced Security Aspects of Key Exchange and Secure Channel Protocols". http://tuprints.ulb.tu-darmstadt.de/7162/. PhD thesis. Darmstadt, Germany: Technische Universität Darmstadt, 2018 (cit. on pp. 10, 11, 40, 59, 198–200).

[GM17] Felix Günther and Sogol Mazaheri. "A Formal Treatment of Multi-key Channels". In: *Advances in Cryptology – CRYPTO 2017, Part III*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10403. Lecture Notes in Computer Science. Springer, Heidelberg, August 2017, pp. 587–618. DOI: 10.1007/978-3-319-63697-9_20 (cit. on pp. 8, 201).

[GRTW21] Felix Günther, Simon Rastikian, Patrick Towa, and Thom Wiggers. "KEMTLS with Delayed Forward Identity Protection in (Almost) a Single Round Trip". Cryptology ePrint Archive, Report 2021/725. https://eprint.iacr.org/2021/725. 2021 (cit. on p. 16).

[GRTW22] Felix Günther, Simon Rastikian, Patrick Towa, and Thom Wiggers. "KEMTLS with Delayed Forward Identity Protection in (Almost) a Single Round Trip". In: *ACNS 22: 20th International Conference on Applied Cryptography and Network Security*. Ed. by Giuseppe Ateniese and Daniele Venturi. Vol. 13269. Lecture Notes in Computer Science. Springer, Heidelberg, June 2022, pp. 253–272. DOI: 10.1007/978-3-031-09234-3_13 (cit. on p. 16).

[Han+21] Shuai Han, Tibor Jager, Eike Kiltz, Shengli Liu, Jiaxin Pan, Doreen Riepel, and Sven Schäge. "Authenticated Key Exchange and Signatures with Tight Security in the Standard Model". In: *Advances in Cryptology – CRYPTO 2021, Part IV*. Ed. by Tal Malkin and Chris Peikert. Vol. 12828. Lecture Notes in Computer Science. Virtual Event: Springer, Heidelberg, August 2021, pp. 670–700. DOI: 10.1007/978-3-030-84259-8_23 (cit. on pp. 11, 16, 208).

[HTT18] Viet Tung Hoang, Stefano Tessaro, and Aishwarya Thiruvengadam. "The Multi-user Security of GCM, Revisited: Tight Bounds for Nonce Randomization". In: *ACM CCS 2018: 25th Conference on Computer and Communications Security*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM Press, October 2018, pp. 1429–1440. DOI: 10.1145/3243734.3243816 (cit. on pp. 8, 9, 200, 201).

[HJ12] Dennis Hofheinz and Tibor Jager. "Tightly Secure Signatures and Public-Key Encryption". In: *Advances in Cryptology – CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Lecture Notes in Computer Science. Springer, Heidelberg, August 2012, pp. 590–607. DOI: 10.1007/978-3-642-32009-5_35 (cit. on pp. 16, 208).

[IT21] J. Iyengar and M. Thomson. "QUIC: A UDP-Based Multiplexed and Secure Transport". RFC 9000. IETF, May 2021. URL: http://tools.ietf.org/rfc/rfc9000.txt (cit. on p. 39).

[JKM18] Tibor Jager, Saqib A. Kakvi, and Alexander May. "On the Security of the PKCS#1 v1.5 Signature Scheme". In: *ACM CCS 2018: 25th Conference on Computer and Communications Security*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM Press, October 2018, pp. 1195–1208. DOI: 10.1145/3243734.3243798 (cit. on p. 11).

[JKRS21] Tibor Jager, Eike Kiltz, Doreen Riepel, and Sven Schäge. "Tightly-Secure Authenticated Key Exchange, Revisited". In: *Advances in Cryptology – EUROCRYPT 2021, Part I*. Ed. by Anne Canteaut and François-Xavier Standaert. Vol. 12696. Lecture Notes in Computer Science. Springer, Heidelberg, October 2021, pp. 117–146. DOI: 10.1007/978-3-030-77870-5_5 (cit. on p. 11).

[JKSS12] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. "On the Security of TLS-DHE in the Standard Model". In: *Advances in Cryptology – CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Lecture Notes in Computer Science. Springer, Heidelberg, August 2012, pp. 273–293. DOI: 10.1007/978-3-642-32009-5_17 (cit. on pp. 15, 63, 166, 167).

[JSS15] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. "On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption". In: *ACM CCS 2015: 22nd Conference on Computer and Communications Security*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. ACM Press, October 2015, pp. 1185–1196. DOI: 10.1145/2810103.2813657 (cit. on p. 15).

[JMV01] Don Johnson, Alfred Menezes, and Scott A. Vanstone. "The Elliptic Curve Digital Signature Algorithm (ECDSA)". In: *International Journal for Information Security* 1.1 (2001), pp. 36–63. DOI: 10.1007/s102070100002. URL: https://doi.org/10.1007/s102070100002 (cit. on p. 11).

[JK02] Jakob Jonsson and Burton S. Kaliski Jr. "On the Security of RSA Encryption in TLS". In: *Advances in Cryptology – CRYPTO 2002*. Ed. by Moti Yung. Vol. 2442. Lecture Notes in Computer Science. Springer, Heidelberg, August 2002, pp. 127–142. DOI: 10.1007/3-540-45708-9_9 (cit. on p. 15).

[JL17] S. Josefsson and I. Liusvaara. "Edwards-Curve Digital Signature Algorithm (EdDSA)". RFC 8032. IETF, January 2017. URL: http://tools.ietf.org/rfc/rfc8032.txt (cit. on p. 11).

[Kak19] Saqib A. Kakvi. "On the Security of RSA-PSS in the Wild". In: *Proceedings of the 5th ACM Workshop on Security Standardisation Research Workshop, London, UK, November 11, 2019*. Ed. by Maryam Mehrnezhad, Thyla van der Merwe, and Feng Hao. ACM, 2019, pp. 23–34. DOI: 10.1145/3338500.3360333. URL: https://doi.org/10.1145/3338500.3360333 (cit. on p. 11).

[Kal98] B. Kaliski. "PKCS #1: RSA Encryption Version 1.5". RFC 2313. IETF, March 1998. URL: http://tools.ietf.org/rfc/rfc2313.txt (cit. on p. 11).

[KL21] Jonathan Katz and Yehuda Lindell. "Introduction to Modern Cryptography". 3rd ed. CRC Press, 2021 (cit. on p. 27).

[KW03] Jonathan Katz and Nan Wang. "Efficiency Improvements for Signature Schemes with Tight Security Reductions". In: *ACM CCS 2003: 10th Conference on Computer and Communications Security*. Ed. by Sushil Jajodia, Vijayalakshmi Atluri, and Trent Jaeger. ACM Press, October 2003, pp. 155–164. DOI: 10.1145/948109.948132 (cit. on p. 225).

[KMP16] Eike Kiltz, Daniel Masny, and Jiaxin Pan. "Optimal Security Proofs for Signatures from Identification Schemes". In: *Advances in Cryptology – CRYPTO 2016, Part II*. Ed. by Matthew Robshaw and Jonathan Katz. Vol. 9815. Lecture Notes in Computer Science. Springer, Heidelberg, August 2016, pp. 33–61. DOI: 10.1007/978-3-662-53008-5_2 (cit. on pp. 33, 36, 212).

[KOS10] Eike Kiltz, Adam O'Neill, and Adam Smith. "Instantiability of RSA-OAEP under Chosen-Plaintext Attack". In: *Advances in Cryptology – CRYPTO 2010*. Ed. by Tal Rabin. Vol. 6223. Lecture Notes in Computer Science. Springer, Heidelberg, August 2010, pp. 295–313. DOI: 10.1007/978-3-642-14623-7_16 (cit. on pp. 225, 230, 232, 233).

[Kle08] J. Klensin. "Simple Mail Transfer Protocol". RFC 5321. IETF, October 2008. URL: http://tools.ietf.org/rfc/rfc5321.txt (cit. on p. 1).

[KSS13] Florian Kohlar, Sven Schäge, and Jörg Schwenk. "On the Security of TLS-DH and TLS-RSA in the Standard Model". Cryptology ePrint Archive, Report 2013/367. https://eprint.iacr.org/2013/367. 2013 (cit. on p. 15).

[Koh+14] Markulf Kohlweiss, Ueli Maurer, Cristina Onete, Bjoern Tackmann, and Daniele Venturi. "(De-)Constructing TLS". Cryptology ePrint Archive, Report 2014/020. https://eprint.iacr.org/2014/020. 2014 (cit. on p. 8).

[KN09] J. Korhonen and U. Nilsson. "Service Selection for Mobile IPv4". RFC 5446. IETF, February 2009. URL: http://tools.ietf.org/rfc/rfc5446.txt (cit. on pp. 3, 15).

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. "HMAC: Keyed-Hashing for Message Authentication". RFC 2104. IETF, February 1997. URL: http://tools.ietf.org/rfc/rfc2104.txt (cit. on p. 62).

[KE10] H. Krawczyk and P. Eronen. "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)". RFC 5869. IETF, May 2010. URL: http://tools.ietf.org/rfc/rfc5869.txt (cit. on p. 62).

[Kra10a] Hugo Krawczyk. "Cryptographic Extraction and Key Derivation: The HKDF Scheme". Cryptology ePrint Archive, Report 2010/264. https://eprint.iacr.org/2010/264. 2010 (cit. on p. 62).

[Kra10b] Hugo Krawczyk. "Cryptographic Extraction and Key Derivation: The HKDF Scheme". In: *Advances in Cryptology – CRYPTO 2010*. Ed. by Tal Rabin. Vol. 6223. Lecture Notes in Computer Science. Springer, Heidelberg, August 2010, pp. 631–648. DOI: 10.1007/978-3-642-14623-7_34 (cit. on p. 62).

[Kra16] Hugo Krawczyk. "A Unilateral-to-Mutual Authentication Compiler for Key Exchange (with Applications to Client Authentication in TLS 1.3)". In: *ACM CCS 2016: 23rd Conference on Computer and Communications Security*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. ACM Press, October 2016, pp. 1438–1450. DOI: 10.1145/2976749.2978325 (cit. on p. 16).

[KPW13] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. "On the Security of the TLS Protocol: A Systematic Analysis". In: *Advances in Cryptology – CRYPTO 2013, Part I*. Ed. by Ran Canetti and Juan A. Garay. Vol. 8042. Lecture Notes in Computer Science. Springer, Heidelberg, August 2013, pp. 429–448. DOI: 10.1007/978-3-642-40041-4_24 (cit. on pp. 15, 63, 166, 167).

[KW16] Hugo Krawczyk and Hoeteck Wee. "The OPTLS Protocol and TLS 1.3". In: *2016 IEEE European Symposium on Security and Privacy*. IEEE, March 2016, pp. 81–96. DOI: 10.1109/EuroSP.2016.18 (cit. on p. 8).

[LHT16] A. Langley, M. Hamburg, and S. Turner. "Elliptic Curves for Security". RFC 7748. IETF, January 2016. URL: http://tools.ietf.org/rfc/rfc7748.txt (cit. on p. 108).

[LP19] Roman Langrehr and Jiaxin Pan. "Tightly Secure Hierarchical Identity-Based Encryption". In: *PKC 2019: 22nd International Conference on Theory and Practice of Public Key Cryptography, Part I*. Ed. by Dongdai Lin and Kazue Sako. Vol. 11442. Lecture Notes in Computer Science. Springer, Heidelberg, April 2019, pp. 436–465. DOI: 10.1007/978-3-030-17253-4_15 (cit. on p. 17).

[LW05] Arjen K. Lenstra and Benne de Weger. "On the Possibility of Constructing Meaningful Hash Collisions for Public Keys". In: *ACISP 05: 10th Australasian Conference on Information Security and Privacy*. Ed. by Colin Boyd and Juan Manuel González Nieto. Vol. 3574. Lecture Notes in Computer Science. Springer, Heidelberg, July 2005, pp. 267–279 (cit. on p. 15).

[Li+16] Xinyu Li, Jing Xu, Zhenfeng Zhang, Dengguo Feng, and Honggang Hu. "Multiple Handshakes Security of TLS 1.3 Candidates". In: *2016 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2016, pp. 486–505. DOI: 10.1109/SP.2016.36 (cit. on p. 8).

[LS17] Yong Li and Sven Schäge. "No-Match Attacks and Robust Partnering Definitions: Defining Trivial Attacks for Security Protocols is Not Trivial". In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. ACM Press, October 2017, pp. 1343–1360. DOI: 10.1145/3133956.3134006 (cit. on p. 209).

[Li+14] Yong Li, Sven Schäge, Zheng Yang, Florian Kohlar, and Jörg Schwenk. "On the Security of the Pre-shared Key Ciphersuites of TLS". In: *PKC 2014: 17th International Conference on Theory and Practice of Public Key Cryptography*. Ed. by Hugo Krawczyk. Vol. 8383. Lecture Notes in Computer Science. Springer, Heidelberg, March 2014, pp. 669–684. DOI: 10.1007/978-3-642-54631-0_38 (cit. on p. 15).

[LM18] Vadim Lyubashevsky and Daniele Micciancio. "Asymptotically Efficient Lattice-Based Digital Signatures". In: *Journal of Cryptology* 31.3 (July 2018), pp. 774–797. DOI: 10.1007/s00145-017-9270-z (cit. on p. 209).

[Man15] Itsik Mantin. "Attacking SSL when using RC4: Breaking SSL with a 13-year-old RC4 Weakness". In: *Black Hat Asia*. https://www.imperva.com/docs/HII_Attacking_SSL_when_using_RC4.pdf. March 2015 (cit. on p. 15).

[MP17] Giorgia Azzurra Marson and Bertram Poettering. "Security Notions for Bidirectional Channels". In: *IACR Transactions on Symmetric Cryptology* 2017.1 (2017), pp. 405–426. DOI: 10.13154/tosc.v2017.i1.405-426 (cit. on p. 201).

[Mau94] Ueli M. Maurer. "Towards the Equivalence of Breaking the Diffie-Hellman Protocol and Computing Discrete Algorithms". In: *Advances in Cryptology – CRYPTO'94*. Ed. by Yvo Desmedt. Vol. 839. Lecture Notes in Computer Science. Springer, Heidelberg, August 1994, pp. 271–281. DOI: 10.1007/3-540-48658-5_26 (cit. on p. 24).

[Mau05] Ueli M. Maurer. "Abstract Models of Computation in Cryptography (Invited Paper)". In: *10th IMA International Conference on Cryptography and Coding*. Ed. by Nigel P. Smart. Vol. 3796. Lecture Notes in Computer Science. Springer, Heidelberg, December 2005, pp. 1–12 (cit. on p. 165).

[MRH04] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. "Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology". In: *TCC 2004: 1st Theory of Cryptography Conference*. Ed. by Moni Naor. Vol. 2951. Lecture Notes in Computer Science. Springer, Heidelberg, February 2004, pp. 21–39. DOI: 10.1007/978-3-540-24638-1_2 (cit. on pp. 9, 78, 80, 82–86).

[MW96] Ueli M. Maurer and Stefan Wolf. "Diffie-Hellman Oracles". In: *Advances in Cryptology – CRYPTO'96*. Ed. by Neal Koblitz. Vol. 1109. Lecture Notes in Computer Science. Springer, Heidelberg, August 1996, pp. 268–282. DOI: 10.1007/3-540-68697-5_21 (cit. on p. 24).

[McG08] D. McGrew. "An Interface and Algorithms for Authenticated Encryption". RFC 5116. IETF, January 2008. URL: http://tools.ietf.org/rfc/rfc5116.txt (cit. on pp. 69, 107, 198).

[MB12] D. McGrew and D. Bailey. "AES-CCM Cipher Suites for Transport Layer Security (TLS)". RFC 6655. IETF, July 2012. URL: http://tools.ietf.org/rfc/rfc6655.txt (cit. on pp. 69, 107, 198).

[ML21] A. Melnikov and B. Leiba. "Internet Message Access Protocol (IMAP) - Version 4rev2". RFC 9051. IETF, August 2021. URL: http://tools.ietf.org/rfc/rfc9051.txt (cit. on p. 1).

[Mey+14] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. "Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks". In: *USENIX Security 2014: 23rd USENIX Security Symposium*. Ed. by Kevin Fu and Jaeyeon Jung. USENIX Association, August 2014, pp. 733–748 (cit. on p. 15).

[MF21a] Arno Mittelbach and Marc Fischlin. "The Theory of Hash Functions and Random Oracles - An Approach to Modern Cryptography". Information Security and Cryptography. Springer, 2021. DOI: 10.1007/978-3-030-63287-8. URL: https://doi.org/10.1007/978-3-030-63287-8 (cit. on p. 82).

[MF21b] K. Moriarty and S. Farrell. "Deprecating TLS 1.0 and TLS 1.1". RFC 8996. IETF, March 2021. URL: http://tools.ietf.org/rfc/rfc8996.txt (cit. on p. 3).

[MKJR16] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. "PKCS #1: RSA Cryptography Specifications Version 2.2". RFC 8017. IETF, November 2016. URL: http://tools.ietf.org/rfc/rfc8017.txt (cit. on p. 11).

[MSW08] Paul Morrissey, Nigel P. Smart, and Bogdan Warinschi. "A Modular Security Analysis of the TLS Handshake Protocol". In: *Advances in Cryptology – ASIACRYPT 2008*. Ed. by Josef Pieprzyk. Vol. 5350. Lecture Notes in Computer Science. Springer, Heidelberg, December 2008, pp. 55–73. DOI: 10.1007/978-3-540-89255-7_5 (cit. on p. 15).

[MR96] J. Myers and M. Rose. "Post Office Protocol - Version 3". RFC 1939. IETF, May 1996. URL: http://tools.ietf.org/rfc/rfc1939.txt (cit. on p. 1).

[Nat13] National Institute of Standards and Technology. "FIPS PUB 186-4: Digital Signature Standard (DSS)". 2013 (cit. on pp. 7, 108).

[NL18] Y. Nir and A. Langley. "ChaCha20 and Poly1305 for IETF Protocols". RFC 8439. IETF, June 2018. URL: http://tools.ietf.org/rfc/rfc8439.txt (cit. on pp. 69, 107, 198).

[NT99] Noam Nisan and Amnon Ta-Shma. "Extracting Randomness: A Survey and New Constructions". In: *Journal of Computer and System Sciences* 58.1 (1999), pp. 148–173. DOI: 10.1006/jcss.1997.1546. URL: https://doi.org/10.1006/jcss.1997.1546 (cit. on p. 62).

[NZ96] Noam Nisan and David Zuckerman. "Randomness is Linear in Space". In: *Journal of Computer and System Sciences* 52.1 (1996), pp. 43–52. DOI: 10.1006/jcss.1996.0004. URL: https://doi.org/10.1006/jcss.1996.0004 (cit. on p. 62).

[OP01] Tatsuaki Okamoto and David Pointcheval. "The Gap-Problems: A New Class of Problems for the Security of Cryptographic Schemes". In: *PKC 2001: 4th International Workshop on Theory and Practice in Public Key Cryptography*. Ed. by Kwangjo Kim. Vol. 1992. Lecture Notes in Computer Science. Springer, Heidelberg, February 2001, pp. 104–118. DOI: 10.1007/3-540-44586-2_8 (cit. on p. 25).

[Opp16] Rolf Oppliger. "SSL and TLS: Theory and Practice, Second Edition". Artech House, 2016 (cit. on pp. 2, 3).

[PW22] Jiaxin Pan and Benedikt Wagner. "Lattice-Based Signatures with Tight Adaptive Corruptions and More". In: *PKC 2022, Part II*. Ed. by Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe. Vol. 13178. LNCS. Springer, 2022, pp. 347–378. DOI: 10.1007/978-3-030-97131-1_12. URL: https://doi.org/10.1007/978-3-030-97131-1%5C_12 (cit. on pp. 17, 208, 235).

[PRS11] Kenneth G. Paterson, Thomas Ristenpart, and Thomas Shrimpton. "Tag Size Does Matter: Attacks and Proofs for the TLS Record Protocol". In: *Advances in Cryptology – ASIACRYPT 2011*. Ed. by Dong Hoon Lee and Xiaoyun Wang. Vol. 7073. Lecture Notes in Computer Science. Springer, Heidelberg, December 2011, pp. 372–389. DOI: 10.1007/978-3-642-25385-0_20 (cit. on pp. 15, 198).

[PS18] Christopher Patton and Thomas Shrimpton. "Partially Specified Channels: The TLS 1.3 Record Layer without Elision". In: *ACM CCS 2018: 25th Conference on Computer and Communications Security*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM Press, October 2018, pp. 1415–1428. DOI: 10.1145/3243734.3243789 (cit. on pp. 8, 201).

[PHG13] Angelo Prado, Neal Harris, and Yoel Gluck. "SSL, Gone in 30 Seconds: A BREACH Beyond CRIME". In: *Black Hat USA 2013*. https://www.blackhat.com/us-13/archives.html#Prado. August 2013 (cit. on p. 16).

[Res18] E. Rescorla. "The Transport Layer Security (TLS) Protocol Version 1.3". RFC 8446. IETF, August 2018. URL: http://tools.ietf.org/rfc/rfc8446.txt (cit. on pp. 1, 3, 8, 9, 39, 61–65, 68–70, 72–74, 79, 80, 101, 105–107, 109–113, 115, 116, 120, 121, 123, 127, 197, 198, 200, 201).

[RM12] E. Rescorla and N. Modadugu. "Datagram Transport Layer Security Version 1.2". RFC 6347. IETF, January 2012. URL: http://tools.ietf.org/rfc/rfc6347.txt (cit. on p. 123).

[RTFK22] E. Rescorla, H. Tschofenig, T. Fossati, and A. Kraus. "Connection Identifier for DTLS 1.2". RFC 9146. IETF, March 2022. URL: http://tools.ietf.org/rfc/rfc9146.txt (cit. on p. 123).

[RSS11] Thomas Ristenpart, Hovav Shacham, and Thomas Shrimpton. "Careful with Composition: Limitations of the Indifferentiability Framework". In: *Advances in Cryptology – EUROCRYPT 2011*. Ed. by Kenneth G. Paterson. Vol. 6632. Lecture Notes in Computer Science. Springer, Heidelberg, May 2011, pp. 487–506. DOI: 10.1007/978-3-642-20465-4_27 (cit. on pp. 85, 86).

[Riv92] R. Rivest. "The MD5 Message-Digest Algorithm". RFC 1321. IETF, April 1992. URL: http://tools.ietf.org/rfc/rfc1321.txt (cit. on pp. 3, 28).

[RD12] Juliano Rizzo and Thai Duong. "The CRIME attack". Presented at ekoparty '12. http://goo.gl/mlw1X1. 2012 (cit. on pp. 3, 16).

[Rog02] Phillip Rogaway. "Authenticated-Encryption With Associated-Data". In: *ACM CCS 2002: 9th Conference on Computer and Communications Security*. Ed. by Vijayalakshmi Atluri. ACM Press, November 2002, pp. 98–107. DOI: 10.1145/586110.586125 (cit. on pp. 197, 198, 201).

[Rog06a] Phillip Rogaway. "Formalizing Human Ignorance". In: *Progress in Cryptology - VIETCRYPT 06: 1st International Conference on Cryptology in Vietnam*. Ed. by Phong Q. Nguyen. Vol. 4341. Lecture Notes in Computer Science. Springer, Heidelberg, September 2006, pp. 211–228 (cit. on p. 28).

[Rog06b] Phillip Rogaway. "Formalizing Human Ignorance: Collision-Resistant Hashing without the Keys". Cryptology ePrint Archive, Report 2006/281. https://eprint.iacr.org/2006/281. 2006 (cit. on p. 28).

[SSW20] Peter Schwabe, Douglas Stebila, and Thom Wiggers. "Post-Quantum TLS Without Handshake Signatures". In: *ACM CCS 2020: 27th Conference on Computer and Communications Security*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. ACM Press, November 2020, pp. 1461–1480. DOI: 10.1145/3372297.3423350 (cit. on pp. 16, 40, 42, 45).

[SSW21] Peter Schwabe, Douglas Stebila, and Thom Wiggers. "More Efficient Post-quantum KEMTLS with Pre-distributed Public Keys". In: *ESORICS 2021: 26th European Symposium on Research in Computer Security, Part I*. Ed. by Elisa Bertino, Haya Shulman, and Michael Waidner. Vol. 12972. Lecture Notes in Computer Science. Springer, Heidelberg, October 2021, pp. 3–22. DOI: 10.1007/978-3-030-88418-5_1 (cit. on p. 16).

[Sho97] Victor Shoup. "Lower Bounds for Discrete Logarithms and Related Problems". In: *Advances in Cryptology – EUROCRYPT'97*. Ed. by Walter Fumy. Vol. 1233. Lecture Notes in Computer Science. Springer, Heidelberg, May 1997, pp. 256–266. DOI: 10.1007/3-540-69053-0_18 (cit. on p. 165).

[Sho04] Victor Shoup. "Sequences of games: a tool for taming complexity in security proofs". Cryptology ePrint Archive, Report 2004/332. https://eprint.iacr.org/2004/332. 2004 (cit. on pp. 98, 101, 141, 215, 217).

[SPW07] Ron Steinfeld, Josef Pieprzyk, and Huaxiong Wang. "How to Strengthen Any Weakly Unforgeable Signature into a Strongly Unforgeable Signature". In: *Topics in Cryptology – CT-RSA 2007*. Ed. by Masayuki Abe. Vol. 4377. Lecture Notes in Computer Science. Springer, Heidelberg, February 2007, pp. 357–371. DOI: 10.1007/11967668_23 (cit. on p. 209).

[SLW07] Marc Stevens, Arjen K. Lenstra, and Benne de Weger. "Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities". In: *Advances in Cryptology – EUROCRYPT 2007*. Ed. by Moni Naor. Vol. 4515. Lecture Notes in Computer Science. Springer, Heidelberg, May 2007, pp. 1–22. DOI: 10.1007/978-3-540-72540-4_1 (cit. on p. 15).

[Ste+09] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen K. Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. "Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate". In: *Advances in Cryptology – CRYPTO 2009*. Ed. by Shai Halevi. Vol. 5677. Lecture Notes in Computer Science. Springer, Heidelberg, August 2009, pp. 55–69. DOI: 10.1007/978-3-642-03356-8_4 (cit. on p. 15).

[TC11] S. Turner and L. Chen. "Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms". RFC 6151. IETF, March 2011. URL: http://tools.ietf.org/rfc/rfc6151.txt (cit. on p. 3).

[TP11] S. Turner and T. Polk. "Prohibiting Secure Sockets Layer (SSL) Version 2.0". RFC 6176. IETF, March 2011. URL: http://tools.ietf.org/rfc/rfc6176.txt (cit. on p. 3).

[VG16] Mathy Vanhoef and Tom Van Goethem. "HEIST: HTTP Encrypted Information can be Stolen Through TCP-windows". In: *Black Hat USA*. https://tom.vg/papers/heist_blackhat2016.pdf. August 2016 (cit. on p. 16).

[VP15] Mathy Vanhoef and Frank Piessens. "All Your Biases Belong to Us: Breaking RC4 in WPA-TKIP and TLS". In: *USENIX Security 2015: 24th USENIX Security Symposium*. Ed. by Jaeyeon Jung and Thorsten Holz. USENIX Association, August 2015, pp. 97–112 (cit. on p. 15).

[Vau02] Serge Vaudenay. "Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS..." In: *Advances in Cryptology – EUROCRYPT 2002*. Ed. by Lars R. Knudsen. Vol. 2332. Lecture Notes in Computer Science. Springer, Heidelberg, April 2002, pp. 534–546. DOI: 10.1007/3-540-46035-7_35 (cit. on p. 3).

[WS96] David Wagner and Bruce Schneier. “Analysis of the SSL 3.0 Protocol”. In: *Second USENIX Workshop on Electronic Commerce*. http://www.usenix.org/publications/library/proceedings/ec96/index.html. November 1996 (cit. on pp. 2, 15).

[WY05] Xiaoyun Wang and Hongbo Yu. “How to Break MD5 and Other Hash Functions”. In: *Advances in Cryptology – EUROCRYPT 2005*. Ed. by Ronald Cramer. Vol. 3494. Lecture Notes in Computer Science. Springer, Heidelberg, May 2005, pp. 19–35. DOI: 10.1007/11426639_2 (cit. on p. 15).