UNIVERSITY OF CALIFORNIA SAN DIEGO

Securing the Standards: Bringing Cryptographic Security Proofs Closer To the Real World

A dissertation submitted in partial satisfaction of the
requirements for the degree Computer Science

in

Doctor of Philosophy

by

Hannah Elizabeth Davis

Committee in charge:

Professor Mihir Bellare, Chair
Professor Farinaz Koushanfar
Professor Daniele Micciancio
Professor Stefan Savage
Professor Deian Stefan

2023

The Dissertation of Hannah Elizabeth Davis is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2023

DEDICATION

In recognition of reading this manual before beginning to format the doctoral dissertation or master's thesis; for following the instructions written herein; for consulting with OGS Academic Affairs Advisers; and for not relying on other completed manuscripts, this manual is dedicated to all graduate students about to complete the doctoral dissertation or master's thesis.

In recognition that this is my one chance to use whichever justification, spacing, writing style, text size, and/or textfont that I want to while still keeping my headings and margins consistent.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Placeholder for acknowledgments.

VITA

2014	Bachelor of Mathematics, University of Minnesota, Twin Cities
2014	Bachelor of Computer Science, University of Minnesota, Twin Cities
2019–2021	Teaching Assistant, Department of Computer Science University of California, San Diego
2020	Master of Computer Science, University of California, San Diego
2018–2023	Research Assistant, University of California, San Diego
2023	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

“Power Dissipation in Fractal AC Circuits” Chen J., Rogers J., Anderson L., Andrews U., Brzoska A., Coffey A., Davis H., Fisher L., Hansalik M., Loew S., Teplyaev A. Journal of Physics A: Mathematical and Theoretical vol. 50, num. 32, 2017.

“Separate Your Domains: NIST PQC KEMs, Oracle Cloning and Read-Only Indifferentiability” Bellare M., Davis H., Günther F. Proceedings of 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT), 2020.

“Tighter Proofs for the SIGMA and TLS 1.3 Key Exchange Protocols” Davis H., Günther F. Proceedings of International Conference on Applied Cryptography and Network Security (ACNS), 2021.

“On the Concrete Security of TLS 1.3 PSK Mode” Davis H., Diemert D., Günther F., Jager T. Proceedings of 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT), 2022.

“Hardening Signature Schemes via Derive-then-Derandomize: Stronger Security Proofs for EdDSA’ Bellare M., Davis H., Di Z. Proceedings of 26th IACR International Conference on Practice and Theory of Public-Key Cryptography (PKC), 2023.

“Verifiable Distributed Aggregation Functions” Davis H., Patton C., Rosulek M., Schoppmann P. Proceedings of Privacy Enhancing Technology Symposium (PETS), 2023.

ABSTRACT OF THE DISSERTATION

Securing the Standards: Bringing Cryptographic Security Proofs Closer To the Real World

by

Hannah Elizabeth Davis

Computer Science in Doctor of Philosophy

University of California San Diego, 2023

Professor Mihir Bellare, Chair

Cryptographic standards published by organizations like NIST, ISO, and the IETF provide guidance for developers choosing and implementing cryptographic algorithms for their applications. In recent years, formal proofs of security have become an important part of validation for standardized algorithms; however, these proofs rely on abstractions which sometimes differ significantly from the schemes and protocols used in practice.

In this work, I will begin with a study of the ongoing NIST standardization process of post-quantum key-encapsulation mechanisms and highlight vulnerabilities in several (former) candidate algorithms which arise from a systematic mismatch between abstract primitives used in cryptographic models and their actual instantiation in implementations. I will then present a library of secure instantiation techniques and a way to extend schemes' existing proofs to their instantiations. Next, I will address the Transport Layer Security (TLS 1.3) Handshake Protocol and demonstrate by a concrete evaluation that prior work fails to prove practical security levels for many of the standardized parameter sets. I will then show tighter proofs that do justify these parameter sets and which additionally give the first fully justified abstraction of the TLS 1.3 key schedule in the random oracle model, and I will explain how certain parts of the TLS 1.3 design hinder the application of useful abstractions.

I will also explain how inaccurate portrayals of hash functions in the random oracle model impact the security analysis of the standardized EdDSA signature scheme and present an improved proof of security with better tightness and modularity. I conclude by introducing my work on the proposed standard for privacy-preserving measurement, including a new security model for Verifiable Distributed Aggregation Functions. Within this model, I discuss results for Prio3, an optimized version of the massively scalable, widely used Prio construction for private data collection, and Doplar, a new construction for private histogram generation.

Introduction

The cryptographic algorithms that protect our data in the Internet age are not, by and large, developed by cryptographers. Instead, many Internet applications rely on cryptographic standards for guidance. These standards documents are published by organizations like the National Institute of Standards and Technology (NIST) and the Internet Engineering Task Force (IETF) as authoritative references on how to implement secure cryptography. Standards ensure interoperability between Internet applications, and give developers a trusted source for their cryptographic needs. Standardized algorithms like the Transport Layer Security protocol (TLS 1.3) currently protect roughly to 85 percent of all Internet traffic [194].

Because standardized cryptography is often used at large scale, any vulnerabilities have significant consequences. Furthermore, adopting a new standard is a slow, expensive process, so updates and patches are relatively rare. Before standards are published, they therefore undergo a vigorous vetting process, complete with extensive public scrutiny. In recent years, this process often includes formal proofs of security among other validation methods. In this work, we provide new and improved proofs of security for several current and future cryptographic standards.

Security proofs establish bounds on the success probability of an adversary interacting with a target scheme in an abstract model that defines the attack surface. The exact limit on this probability depends both on the resources of the adversary and on the security of any underlying cryptographic primitives or mathematical assumptions. If a scheme's security is close to that of its components' security for all resource levels, we say that the bounds are "tight". Tight bounds can be used to help select parameter sizes for cryptographic components; other bounds may provide heuristic guarantees about a scheme's security. Once a scheme has a valid security proof, an attacker can only successfully attack it with high probability by violating the assumptions made by the proof or model, or by using enough resources to vacate the bounds.

We consider the existing proofs for current and future standards, and identify certain ways they do not rule out attacks: loose bounds and gaps between abstract threat models and implementations. Wherever possible, we seek to repair the existing proofs or leverage prior work in a modular way, rather than replace them entirely.

Hash functions, indifferentiability and the ROM.

One place where many proofs break down is in their treatment of hash functions. The random oracle model (ROM) of Bellare and Rogaway [38] is a powerful model in which hash functions are treated as publicly accessible random functions, often with infinite domains. Of course, such functions are unrealizable, and thus proofs in the ROM offer only heuristic evidence of security. However, the ROM is a commonly used assumption, relied on by security proofs for many standards [**?**, 183, **?**, 51, 74] and other widely-used cryptographic primitives, and there are few natural examples of schemes which are secure in the ROM but insecure in practice.

Not all hash functions can be suitably modeled as random oracles. The standardized hash functions are constructed by iterating an underlying compression function or random permutation, and it is essential to make sure that this underlying structure does not admit additional vulnerabilities. Maurer et al. developed the indifferentiability framework, which can be used to evaluate whether a particular construction can be used to securely instantiate a random oracle [156]. They proved a powerful composition theorem. If a scheme is proven secure (for most common definitions of security) in the random oracle model, and it is instantiated with an indifferentiable construction from some compression function, then the scheme is also secure when only the compression function is modeled as a random oracle.

Key encapsulation mechanisms.

We begin in Chapter 1 with a case study of the ongoing NIST standardization process for post-quantum key encapsulation mechanisms (KEMs). Of the initial, now-eliminated, candidates, we identify highly efficient key recovery attacks on three schemes. These attacks fall in a gap between a security model with three independent random oracles, and implementations which instantiate them using a single (indifferentiable) hash function. Because our attacks circumvent the candidates' security proofs rather contradicting them, they went unnoticed for more than a year of intense public scrutiny as proposed standards.

The failure of the attacked schemes was in a task we call oracle cloning: constructing multiple independent random oracles given access to a single RO. We highlight thirteen other candidate KEM schemes which do not approach oracle cloning with care and whose proofs also exhibit gaps, and ten schemes that performed oracle cloning well. We then collect a library of simple and secure oracle cloning techniques, including domain separation, and validate them in a new framework called read-only indifferentiability. Using these results, we extend the existing proofs of twelve of the thirteen questionable schemes to cover their oracle cloning methods, thus closing the gap. The thirteenth scheme was updated in a subsequent round of the standardization process to use one of our techniques [11].

Authenticated key exchange.

Over the next two chapters, we study the Transport Layer Security 1.3 Handshake Protocol [186]. This protocol is used establish secret, pseudorandom session keys for billions of Internet connections per day. As part of its standardization process, the handshake protocol received its first proof of security from Dowling et al. [93] in 2015. Although this proof provides heuristic evidence of security for the handshake protocol, we empirically demonstrate in Chapters 2 and 3 (for the full handshake and pre-shared key modes respectively) that its bounds are too loose to justify the standardized parameter sets for global usage scales.

The quadratic loss in the number of sessions in the Dowling bound is common to many contemporary proofs for authenticated key exchange protocols based on the Diffie–Hellman (DH) problem. The first fully tight bounds for this style of key exchange were given by Cohn-Gordon et al. [73] for a custom-designed key exchange protocol. The cost of this advancement was a change in assumption: the Cohn-Gordon proof relied on the interactive Strong DH assumption rather than more standard noninteractive DH assumptions.

In Chapter 2, we apply the Cohn-Gordon technique to the full TLS 1.3 handshake protocol and to the SIGMA key exchange protocol [138] and achieve a full justification of standardized parameter sets. We also justify the change of assumption in two ways: by evaluating the hardness of Strong DH in the generic group model, and by highlighting that the proof of Dowling et al. also assumes Strong DH implicitly. Diemert and Jager [88] gave a concurrent and independent analysis of the TLS 1.3 handshake with similar final bounds.

In Chapter 3, we build on the work of Chapter 2 and that of Diemert and Jager to tightly prove security for the pre-shared key modes of the TLS 1.3 handshake protocol. As an intermediate step, we establish the first justification of the TLS 1.3 key schedule in the indifferentiability framework. This approach is not only more rigorous than previous abstractions; it also simplifies the remaining proof and helps establish independence for the derived keys. However, we also highlight an obstacle in the poor domain separation of the key schedule that prevents an indifferentiability proof for one choice of mode and hash function (PSK-only mode with SHA384). Finally, we treat handshake encryption as a modular transform applied to a generic key exchange protocol and provide general results on the composition of such a transform.

EdDSA signatures. We address the EdDSA signature scheme [49] in Chapter 4. EdDSA is a tweaked variant of the Schnorr signature scheme [190] that hardens it against randomness reuse and certain side-channel attacks. It's a standardized signature algorithm for TLS 1.3, and is also used by many blockchain applications and encrypted messaging services, including WhatsApp and Signal.

Over the years, Schnorr signatures have received several proofs of security [184], including some recent tighter proofs from non-standard assumptions [32, **?**]. Ed25519, however, was first proven secure in 2020 by Brendel et al. [**?**]. Like the initial proofs of Schnorr signatures, their reduction is not tight and models its hash function as a random oracle. The latter quality presents a concern because Ed25519 uses SHA512, an MD-style hash function which is known to be differentiable from a random oracle [74] and subject to length-extension attacks.

We define a generic transform called Derive-then-Derandomize, that captures the hardening tweaks applied by Bernstein et al. for EdDSA. We prove that it works from standard assumptions. We then give a general lemma showing indifferentiability of **Shrink-MD**, a class of constructions that apply a shrinking output transform to an Merkle-Damgard-style hash function. The particular usage of SHA512 within Ed25519 falls within this class. Using these, we give a direct, fully tight reduction from EdDSA signatures to Schnorr signatures. Our proof enables tighter bounds for EdDSA that leverage both historic trust and recent analysis of Schnorr; it also captures the use of SHA512 as a hash function and includes length-extension attacks in its threat model.

Verifiable Distributed Aggregation Functions.

Finally in chapter 5, we make the first provable security contribution to an ongoing standardization process. The IETF's working group on privacy preserving measurement (PPM) [1], in their draft standard, defines a class of cryptographic primitives called "Verifiable Distributed Aggregation Functions (VDAFs)" [25]. VDAFs are a class of multi-party computation protocols that enable a collector, with the help of several third-party aggregators, to learn an aggregate statistic about a population of clients without compromising the privacy of individual client measurements. The Prio protocol by Corrigan-Gibbs and Boneh [75], an example of the VDAF paradigm has already been used at global scale as part of the Exposure Notification Private Analytics (ENPA) program during the Covid-19 pandemic [13].

We give the first provable security treatment for VDAFs, This includes a formal framework of syntax and game-based definitions capturing privacy, robustness, and correctness, and analysis of two constructions within this framework. The first is Prio3, a variant of Prio incorporating optimizations by Boneh et al. [58] and a candidate for standardization within the PPM draft. The second, called Doplar, we introduce as a way to reduce the round complexity of the Poplar system of Boneh et al. [59], itself a candidate for standardization. To achieve this improvement, Doplar requires slightly greater overall bandwidth and computation.

Chapter 1

Separate Your Domains

1.1 Introduction

Theoretical works giving, and proving secure, schemes in the random oracle (RO) model [39], often, for convenience, assume access to *multiple, independent* ROs. Implementations, however, like to implement them all via a *single* hash function like SHA256 that is assumed to be a RO.

The transition from one RO to many is, in principle, easy. One can use a method suggested by BR [39] and usually called "domain separation." For example to build three random oracles H_1, H_2, H_3 from a single one, H, define

$$H_1(x) = H(\langle 1 \rangle \| x),\ \ H_2(x) = H(\langle 2 \rangle \| x)\ \text{ and }\ H_3(x) = H(\langle 3 \rangle \| x)\ , \tag{1.1}$$

where $\langle i \rangle$ is the representation of integer i as a bit-string of some fixed length, say one byte. One might ask if there is justifying theory: a proof that the above "works," and a definition of what "works" means. A likely response is that it is obvious it works, and theory would be pedantic.

If it were merely a question of the specific domain-separation method of Equation (1.1), we'd be inclined to agree. But we have found some good reasons to revisit the question and look into theoretical foundations. They arise from the NIST Post-Quantum Cryptography (PQC) standardization process [176].

We analyzed the KEM submissions. We found attacks, breaking some of them, that arise from incorrect ways of turning one random oracle into many, indicating that the process is error-prone. We found other KEMs where methods other than Equation (1.1) were used and whether or not they work is unclear. In some submissions, instantiations for multiple ROs were left unspecified. In others, they differed between the specification and reference implementation.

Domain separation as per Equation (1.1) is a *method*, not a *goal*. We identify and name the underlying goal, calling it *oracle cloning*— given one RO, build many, independent ones. (More generally, given m ROs, build $n > m$ ROs.) We give a definition of what is an "oracle cloning method" and what it means for such a method to "work," in a framework we call read-only indifferentiability, a simple variant of classical indifferentiability [156]. We specify and study many oracle cloning methods, giving some general results to justify (prove read-only indifferentiability of) certain classes of them. The intent is not only to validate as many NIST PQC KEMs as possible (which we do) but to specify and validate methods that will be useful beyond that.

Below we begin by discussing the NIST PQC KEMs and our findings on them, and then turn to our theoretical treatment and results.

NIST PQC KEMs. In late 2016, NIST put out a call for post-quantum cryptographic algorithms [176]. In the first round they received 28 submissions targeting IND-CCA-secure KEMs, of which 17 remain in the second round [178].

Recall that in a KEM (Key Encapsulation Mechanism) KE, the encapsulation algorithm $\mathsf{KE.E}$ takes the public key pk (but no message) to return a symmetric key K and a ciphertext C^* encapsulating it, $(C^*, K) \leftarrow\$ \mathsf{KE.E}(pk)$. Given an IND-CCA KEM, one can easily build an IND-CCA PKE scheme by hybrid encryption [76], explaining the focus of standardization on the KEMs.

Most of the KEM submissions (23 in the first round, 15 in the second round) are constructed from a weak (OW-CPA, IND-CPA, ...) PKE scheme using either a method from Hofheinz, Hövelmanns and Kiltz (HHK) [119] or a related method from [87, 189, 132]. This results in a KEM KE_4, the subscript to indicate that it uses up to four ROs that we'll denote H_1, H_2, H_3, H_4. Results of [119, 87, 189, 132] imply that KE_4 is provably IND-CCA, *assuming the ROs* H_1, H_2, H_3, H_4 *are independent.*

Next, the step of interest for us, the oracle cloning: they build the multiple random oracles via a single RO H, replacing H_i with an oracle $\mathbf{F}[H](i, \cdot)$, where we refer to the construction $\mathbf{F}$ as a "cloning functor," and $\mathbf{F}[H]$ means that $\mathbf{F}$ gets oracle access to H. This turns KE_4 into a

KEM KE_1 that uses only a *single* RO H, allowing an implementation to instantiate the latter with a single NIST-recommended primitive like SHA3-512 or SHAKE256 [177]. (In some cases, KE_1 uses a number of ROs that is more than one but less than the number used by KE_4, which is still oracle cloning, but we'll ignore this for now.)

Often the oracle cloning method (cloning functor) is not specified in the submission document; we obtained it from the reference implementation. Our concern is the security of this method and the security of the final, single-RO-using KEM KE_1. (As above we assume the starting KE_4 is secure if its four ROs are independent.)

Oracle cloning in submissions. We surveyed the relevant (first- and second-round) NIST PQC KEM submissions, looking in particular at the reference code, to determine what choices of cloning functor $\mathbf{F}$ was made, and how it impacted security of KE_1. Based on our findings, we classify the submissions into groups as follows.

First is a group of *successfully attacked* submissions. We discover and specify attacks, enabled through erroneous RO cloning, on three (first-round) submissions: BIG QUAKE [24], DAGS [23] and Round2 [103]. (Throughout the paper, first-round submissions are in gray, second-round submissions in **bold**.) Our attacks on BIG QUAKE and Round2 recover the symmetric key K from the ciphertext C^* and public key. Our attack on DAGS succeeds in partial key recovery, recovering 192 bits of the symmetric key. These attacks are very fast, taking at most about the same time as taken by the (secret-key equipped, prescribed) decryption algorithm to recover the key. None of our attacks needs access to a decryption oracle, meaning we violate much more than IND-CCA.

Next is submissions with *questionable oracle cloning*. We put just one in this group, namely **NewHope** [11]. Here we do not have proof of security in the ROM for the final instantiated scheme KE_1. We do show that the cloning methods used here do not achieve our formal notion of rd-indiff security, but this does not result in an attack on KE_1, so we do not have a practical attack either. We recommend changes in the cloning methods that permit proofs.

Next is a group of ten submissions that use *ad-hoc oracle cloning* methods —as opposed, say, to conventional domain separation as per Equation (1.1)— but for which our results (to be discussed below) are able to prove security of the final single-RO scheme. In this group are

BIKE [14], KCL [200], LAC [154], Lizard [71], LOCKER [15], Odd Manhattan [182], **ROLLO-II** [157], **Round5** [19], **SABER** [79] and Titanium [197]. Still, the security of these oracle cloning methods remains brittle and prone to vulnerabilities under slight changes.

A final group of twelve submissions *did well*, employing something like Equation (1.1). In particular our results can prove these methods secure. In this group are **Classic McEliece** [46], **CRYSTALS-Kyber** [18], EMBLEM [193], **FrodoKEM** [170], **HQC** [159], LIMA [196], NTRU-HRSS-KEM [121], **NTRU Prime** [47], **NTS-KEM** [10], **RQC** [158], **SIKE** [130] and **ThreeBears** [116].

This classification omits 14 KEM schemes that do not fit the above framework. (For example they do not target IND-CCA KEMs, do not use HHK-style transforms, or do not use multiple random oracles.)

Lessons and response. We see that oracle cloning is error-prone, and that it is sometimes done in ad-hoc ways whose validity is not clear. We suggest that oracle cloning not be left to implementations. Rather, scheme designers should give proof-validated oracle cloning methods for their schemes. To enable this, we initiate a theoretical treatment of oracle cloning. We formalize oracle cloning methods, define what it means for one to be secure, and specify a library of proven-secure methods from which designers can draw. We are able to justify the oracle cloning methods of many of the unbroken NIST PQC KEMs. The framework of read-only indifferentiability we introduce and use for this purpose may be of independent interest.

The NIST PQC KEMs we break are first-round candidates, not second-round ones, and in some cases other attacks on the same candidates exist, so one may say the breaks are no longer interesting. We suggest reasons they are. Their value is illustrative, showing not only that errors in oracle cloning occur in practice, but that they can be devastating for security. In particular, the extensive and long review process for the first-round NIST PQC submissions seems to have missed these simple attacks, perhaps due to lack of recognition of the importance of good oracle cloning.

Indifferentiability background. Let SS, ES be sets of functions. (We will call them the starting and ending function spaces, respectively.) A functor $\mathbf{F}\colon \mathsf{SS} \to \mathsf{ES}$ is a deterministic algorithm that, given as oracle a function $s \in \mathsf{SS}$, defines a function $\mathbf{F}[s] \in \mathsf{ES}$. Indifferentiability of $\mathbf{F}$ is a way of defining what it means for $\mathbf{F}[s]$ to emulate e when s, e are randomly chosen from SS, ES, respectively. It permits a "composition theorem" saying that if $\mathbf{F}$ is indifferentiable then use of e in a scheme can be securely replaced by use of $\mathbf{F}[s]$.

Maurer, Renner and Holenstein (MRH) [156] gave the first definition of indifferentiability and corresponding composition theorem. However, Ristenpart, Shacham and Shrimpton (RSS) [187] pointed out a limitation, namely that it only applies to single-stage games. MRH-indiff fails to guarantee security in multi-stage games, a setting that includes many goals of interest including security under related-key attack, deterministic public-key encryption and encryption of key-dependent messages. Variants of MRH-indiff [74, 187, 86, 164] tried to address this, with limited success.

Rd-indiff. Indifferentiability is the natural way to treat oracle cloning. A cloning of one function into n functions ($n = 4$ above) can be captured as a functor (we call it a cloning functor) $\mathbf{F}$ that takes the single RO s and for each $i \in [1..n]$ defines a function $\mathbf{F}[s](i, \cdot)$ that is meant to emulate a RO. We will specify many oracle cloning methods in this way.

We define in Section 1.4 a variant of indifferentiability we call read-only indifferentiability (rd-indiff). The simulator —unlike for reset-indiff [187]— has access to a game-maintained state st, but —unlike MRH-indiff [156]— that state is read-only, meaning the simulator cannot alter it across invocations. Rd-indiff is a stronger requirement than MRH-indiff (if $\mathbf{F}$ is rd-indiff then it is MRH-indiff) but a weaker one than reset-indiff (if $\mathbf{F}$ is reset-indiff then it is rd-indiff). Despite the latter, rd-indiff, like reset-indiff, admits a composition theorem showing that an rd-indiff $\mathbf{F}$ may securely substitute a RO even in multi-stage games. (The proof of RSS [187] for reset-indiff extends to show this.) We do not use reset-indiff because some of our cloning functors do not meet it, but they do meet rd-indiff, and the composition benefit is preserved.

General results. In Section 1.4, we define *translating* functors. These are simply ones whose oracle queries are non-adaptive. (In more detail, a translating functor determines from its input W a list of queries, makes them to its oracle and, from the responses and W, determines its output.) We then define a condition on a translating functor $\mathbf{F}$ that we call *invertibility* and show that if $\mathbf{F}$ is an invertible translating functor then it is rd-indiff. This is done in two parts, Theorems 1 and 2, that differ in the degree of invertibility assumed. The first, assuming the greater degree of invertibility, allows a simpler proof with a simulator that does not need the read-only state allowed in rd-indiff. The second, assuming the lesser degree of invertibility, depends on a simulator that makes crucial use of the read-only state. It sets the latter to a key for a PRF that is then used to answer queries that fall outside the set of ones that can be trivially answered under the invertibility condition. This use of a computational primitive (a PRF) in the indifferentiability context may be novel and may seem odd, but it works.

We apply this framework to analyze particular, practical cloning functors, showing that these are translating and invertible, and then deducing their rd-indiff security. But the above-mentioned results are stronger and more general than we need for the application to oracle cloning. The intent is to enable further, future applications.

Analysis of oracle cloning methods. We formalize oracle cloning as the task of designing a functor (we call it a cloning functor) $\mathbf{F}$ that takes as oracle a function $s \in \mathsf{SS}$ in the starting space and returns a two-input function $e = \mathbf{F}[s] \in \mathsf{ES}$, where $e(i,\cdot)$ represents the i-th RO for $i \in [1..n]$. Section 1.5 presents the cloning functors corresponding to some popular and practical oracle cloning methods (in particular ones used in the NIST PQC KEMs), and shows that they are translating and invertible. Our above-mentioned results allow us to then deduce they are rd-indiff, which means they are safe to use in most applications, even ones involving multi-stage games. This gives formal justification for some common oracle cloning methods. We now discuss some specific cloning functors that we treat in this way.

The prefix (cloning) functor $\mathbf{F}_{\mathrm{pf}(\mathbf{p})}$ is parameterized by a fixed, public vector $\mathbf{p}$ such that no entry of $\mathbf{p}$ is a prefix of any other entry of $\mathbf{p}$. Receiving function s as an oracle, it defines function $e = \mathbf{F}_{\mathrm{pf}(\mathbf{p})}[s]$ by $e(i,X) = s(\mathbf{p}[i] \| X)$, where $\mathbf{p}[i]$ is the i^{th} element of vector $\mathbf{p}$. When $\mathbf{p}[i]$ is a fixed-length bitstring representing the integer i, this formalizes Equation (1.1).

Some NIST PQC submissions use a method we call output splitting. The simplest case is that we want $e(i,\cdot),\ldots,\varepsilon(n,\cdot)$ to all have the same output length L. We then define $e(i,X)$ as bits $(i-1)L+1$ through iL of the given function s applied to X. That is, receiving function s as an oracle, the splitting (cloning) functor $\mathbf{F}_{\mathrm{spl}}$ returns function $e = \mathbf{F}_{\mathrm{spl}}[s]$ defined by $e(i,X) = s(X)[(i-1)L+1..iL]$.

An interesting case, present in some NIST PQC submissions, is trivial cloning: just set $e(i,X) = s(X)$ for all X. We formalize this as the identity (cloning) functor $\mathbf{F}_{\mathrm{id}}$ defined by $\mathbf{F}_{\mathrm{id}}[s](i,X) = s(X)$. Clearly, this is not always secure. It can be secure, however, for usages that restrict queries in some way. One such restriction, used in several NIST PQC KEMs, is length differentiation: $e(i,\cdot)$ is queried only on inputs of some length l_i, where $l_1,\ldots,l_n$ are chosen to be distinct. We are able to treat this in our framework using the concept of working domains that we discuss next, but we warn that this method is brittle and prone to misuse.

Working domains. One could capture trivial cloning with length differentiation as a restriction on the domains of the ending functions, but this seems artificial and dangerous because the implementations do not enforce any such restriction; the functions there are defined on their full domains and it is, apparently, left up to applications to use the functions in a way that does not get them into trouble. The approach we take is to leave the functions defined on their full domains, but define and ask for security over a subdomain, which we called the working domain. A choice of working domain $\mathscr{W}$ accordingly parameterizes our definition of rd-indiff for a functor, and also the definition of invertibility of a translating functor. Our result says that the identity functor is rd-indiff for certain choices of working domains that include the length differentiation one.

Making the working domain explicit will, hopefully, force the application designer to think about, and specify, what it is, increasing the possibility of staying out of trouble. Working domains also provide flexibility and versatility under which different applications can make different choices of the domain.

Working domains not being present in prior indifferentiability formalizations, the comparisons, above, of rd-indiff with these prior formalizations assume the working domain is the full domain of the ending functions. Working domains alter the comparison picture; a cloning functor which is rd-indiff on a working domain may not be even MRH-indiff on its full domain.

Application to KEMs. The framework above is broad, staying in the land of ROs and not speaking of the usage of these ROs in any particular cryptographic primitive or scheme. As such, it can be applied to analyze RO instantiation in many primitives and schemes. In Section 1.6, we exemplify its application in the realm of KEMs as the target of the NIST PQC designs.

This may seem redundant, since an indifferentiability composition theorem says exactly that once indifferentiability of a functor has been shown, "all" uses of it are secure. However, prior indifferentiability frameworks do not consider working domains, so the known composition theorems apply only when the working domain is the full one. (Thus the reset-indiff composition theorem of [187] extends to rd-indiff so that we have security for applications whose security definitions are underlain by either single or multi-stage games, but only for full working domains.)

To give a composition theorem that is conscious of working domains, we must first ask what they are, or mean, in the application. We give a definition of the *working domain of a KEM* KE. This is the set of all points that the scheme algorithms query to the ending functions in usage, captured by a certain game we give. (Queries of the adversary may fall outside the working domain.) Then we give a working-domain-conscious composition theorem for KEMs (Theorem 3) that says the following. Say we are given an IND-CCA KEM KE whose oracles are drawn from a function space $\mathsf{KE.FS}$. Let $\mathbf{F}\colon \mathsf{SS} \to \mathsf{KE.FS}$ be a functor, and let $\overline{\mathsf{KE}}$ be the KEM obtained by implementing the oracles of the KE via $\mathbf{F}$. (So the oracles of this second KEM are drawn from the function space $\overline{\mathsf{KE}}.\mathsf{FS} = \mathsf{SS}$.) Let $\mathscr{W}$ be the working domain of KE, and assume $\mathbf{F}$ is rd-indiff over $\mathscr{W}$. Then $\overline{\mathsf{KE}}$ is also IND-CCA. Combining this with our rd-indiff results on particular cloning functors justifies not only conventional domain separation as an instantiation technique for KEMs, but also more broadly the instantiations in some NIST PQC submissions that do not use domain separation, yet whose cloning functors are rd-diff over the working domain of their KEMs. The most important example is the identity cloning functor used with length differentiation.

A key definitional element of our treatment that allows the above is, following [29], to embellish the *syntax* of a scheme (here a KEM KE) by having it name a function space $\mathsf{KE.FS}$ from which it wants its oracles drawn. Thus, the scheme specification must say how many ROs it wants, and of what domains and ranges. In contrast, in the formal version of the ROM in [39], there is a single, scheme-independent RO that has some fixed domain and range, for example mapping $\{0,1\}^*$ to $\{0,1\}$. This leaves a gap, between the object a scheme wants and what the model provides, that can lead to error. We suggest that, to reduce such errors, schemes specified in standards include a specification of their function space.

1.2 Oracle Cloning in NIST PQC Candidates

Notation. A KEM scheme KE specifies an encapsulation $\mathsf{KE.E}$ that, on input a public encryption key pk returns a session key K, and a ciphertext C^* encapsulating it, written $(C^*, K) \leftarrow\!\!\$\, \mathsf{KE.E}(pk)$. A PKE scheme PKE specifies an encryption algorithm $\mathsf{PKE.E}$ that, on input pk, message $M \in \{0,1\}^{\mathsf{PKE.ml}}$ and randomness R, deterministically returns ciphertext $C \leftarrow \mathsf{PKE.E}(pk, M; R)$. For neither primitive will we, in this section, be concerned with the key generation or decapsulation / decryption algorithm. We might write $\mathsf{KE}[X_1, X_2, \ldots]$ to indicate that the scheme has oracle access to functions $X_1, X_2, \ldots$, and correspondingly then write $\mathsf{KE.E}[X_1, X_2, \ldots]$, and similarly for PKE.

1.2.1 Design process

The literature [119, 87, 189, 132] provides many transforms that take a public-key encryption scheme PKE, assumed to meet some weaker-than-IND-CCA notion of security we denote $\mathrm{S}_{\mathsf{pke}}$ (for example, OW-CPA, OW-PCA or IND-CPA), and, with the aid of some number of random oracles, turn PKE into a KEM that is guaranteed (proven) to be IND-CCA *assuming the ROs are independent.* We'll refer to such transforms as *sound*. Many (most) KEMs submitted to the NIST Post-Quantum Cryptography standardization process were accordingly designed as follows:

(1) First, they specify a $\mathrm{S}_{\mathsf{pke}}$-secure public-key encryption scheme PKE.

(2) Second, they pick a sound transform $\mathbf{T}$ and obtain KEM $\mathsf{KE}_4[H_1, H_2, H_3, H_4] = \mathbf{T}[\mathsf{PKE}, H_2, H_3, H_4]$ (The notation is from [119]. The transforms use up to three random oracles that we are denoting H_2, H_3, H_4, reserving H_1 for possible use by the PKE scheme.) We refer to KE_4 (the subscript refers to its using 4 oracles) as the *base* KEM, and, as we will see, it differs across the transforms.

(3) Finally —the under-the-radar step that is our concern— the ROs $H_1, \ldots, H_4$ are constructed from cryptographic hash functions to yield what we call the *final* KEM KE_1. In more detail, the submissions make various choices of cryptographic hash functions $F_1, \ldots, F_m$ that we call the *base functions*, and, for $i = 1,2,3,4$, specify constructions $\mathbf{C}_i$ that, with oracle access to the base functions, define the H_i, which we write as $H_i \leftarrow \mathbf{C}_i[F_1,\ldots,F_m]$. We call this process oracle cloning, and we call H_i the *final functions.* (Common values of m are $1,2$.) The actual, submitted KEM KE_1 (the subscript because m is usually 1) uses the final functions, so that its encapsulation algorithm can be written as:

```
$\underline{\mathsf{KE}_1.\mathsf{E}[F_1,\ldots,F_m](pk)}$
For $i = 1,2,3,4$ do $H_i \leftarrow \mathbf{C}_i[F_1,\ldots,F_m]$
$(C^*,K) \leftarrow_{\$} \mathsf{KE}_4.\mathsf{E}[H_1,H_2,H_3,H_4](pk)$
Return $(C^*,K)$
```

The question now is whether the final KE_1 is secure. We will show that, for some submissions, it is not. This is true for the choices of base functions $F_1,\ldots,F_m$ made in the submission, but also if these are assumed to be ROs. It is true despite the soundness of the transform, meaning insecurity arises from poor oracle cloning, meaning choices of the constructions $\mathbf{C}_i$. We will then consider submissions for which we have not found an attack. In the latter analysis, we are willing to assume (as the submissions implicitly do) that $F_1,\ldots,F_m$ are ROs, and we then ask whether the final functions are "close" to independent ROs.

1.2.2 The base KEM

We need first to specify the base KE_4 (the result of the sound transform, from step (2) above). The NIST PQC submissions typically cite one of HHK [119], Dent [87], SXY [189] or JZCWM [132] for the sound transform they use, but our examinations show that the submissions have embellished, combined or modified the original transforms. The changes do *not* (to best of our knowledge) violate soundness (meaning the used transforms still yield an IND-CCA KE_4 if H_2,H_3,H_4 are independent ROs and PKE is $\mathsf{S}_{\mathrm{pke}}$-secure) but they make a succinct exposition challenging. We address this with a framework to unify the designs via a single, but parameterized, transform, capturing the submission transforms by different parameter choices.

Figure 1.1 (top) shows the encapsulation algorithm $\mathsf{KE}_4.\mathsf{E}$ of the KEM that our parameterized transform associates to PKE and H_1,H_2,H_3,H_4. The parameters are the variables X,Y,Z (they will be functions of other quantities in the algorithms), a boolean D, and an integer k^*.

```
Algorithm $\mathsf{KE}_4.\mathsf{E}[H_1,H_2,H_3,H_4](pk)$:
1 $M \leftarrow\!\!\$ \{0,1\}^{\mathsf{PKE.ml}}$ ; $R \leftarrow \varepsilon$
2 If $(\mathsf{D} = \mathsf{true})$ then $R \| K' \leftarrow H_2(X)$   // $|K'| = \mathsf{k}^*$
3 $C \leftarrow \mathsf{PKE.E}[H_1](pk, M; R)$
4 $C^* \leftarrow C \| Y$
5 $K \leftarrow H_4(Z)$ ; Return $(C^*, K)$
```

	D	k^*	X	Y	Z	Used in
$\mathbf{T}_1$	true	0	M	ε	M	LIMA, Odd Manhattan
$\mathbf{T}_2$	true	0	$pk \| M$	ε	$pk \| M$	**ThreeBears**
$\mathbf{T}_3$	true	0	M	ε	$M \| C$	**BIKE-1-CCA** **BIKE-3-CCA**, **LAC**
$\mathbf{T}_4$	true	0	$M \| pk$	ε	$M \| C$	**SIKE**
$\mathbf{T}_5$	true	0	M	$H_3(X)$	$M \| C$	**HQC**, **RQC**, **ROLLO-II**, LOCKER
$\mathbf{T}_6$	true	>0	$M \| H_3(pk)$	ε	$K' \| C$	**SABER**
$\mathbf{T}_7$	true	>0	$H_3(pk) \| H_3(M)$	ε	$K' \| H_3(C)$	**CRYSTALS-Kyber**
$\mathbf{T}_8$	true	0	M	$H_3(X)$	M	DAGS, NTRU-HRSS-KEM
$\mathbf{T}_9$	true	0	M	$H_3(X)$	$M \| C \| Y$	BIG QUAKE, EMBLEM, Lizard, Titanium
$\mathbf{T}_{10}$	true	>0	$H_4(M) \| H_4(pk)$	$H_3(X)$	$K' \| H_4(C \| Y)$	**NewHope**
$\mathbf{T}_{11}$	true	>0	$M \| pk$	$H_3(X)$	$K' \| C \| Y$	FrodoKEM, Round2 **Round5**
$\mathbf{T}_{12}$	true	>0	$pk \| M$	$H_3(X)$	$K' \| C$	KCL
$\mathbf{T}_{13}$	true	>0	$H_3(pk) \| M$	ε	$C \| K'$	**FrodoKEM**
$\mathbf{T}_{14}$	false	0	$\perp$	$H_3(M)$	$M \| C \| Y$	**Classic McEliece**
$\mathbf{T}_{15}$	true	0	M	ε	$R \| M$	**NTS-KEM**
$\mathbf{T}_{16}$	false	0	$\perp$	$H_3(M \| pk)$	$M \| C \| Y$	**Streamlined NTRU Prime**
$\mathbf{T}_{17}$	true	0	M	$H_3(M \| pk)$	$M \| C \| Y$	**NTRU LPRime**

Figure 1.1. Top: Encapsulation algorithm of the base KEM scheme produced by our parameterized transform. **Bottom:** Choices of parameters $X, Y, Z, \mathsf{D}, \mathsf{k}^*$ resulting in specific transforms used by the NIST PQC submissions. Second-round submissions are in **bold**, first-round submissions in gray. Submissions using different transforms in the two rounds appear twice.

When choices of these are made, one gets a fully-specified transform and corresponding base KEM KE_4. Each row in the table in the same Figure shows one such choice of parameters, resulting in 15 fully-specified transforms. The final column shows the submissions that use the transform.

The encapsulation algorithm at the top of Figure 1.1 takes input a public key pk and has oracle access to functions H_1, H_2, H_3, H_4. At line 1, it picks a random seed M of length the message length of the given PKE scheme. Boolean D being true (as it is except in two cases) means $\mathsf{PKE.E}$ is randomized. In that case, line 2 applies H_2 to X (the latter, determined as per the table, depends on M and possibly also on pk) and parses the output to get coins R for $\mathsf{PKE.E}$ and possibly (if the parameter $\mathsf{k}^* \neq 0$) an additional string K'. At line 3, a ciphertext C is produced by encrypting the seed M using $\mathsf{PKE.E}$ with public key pk and coins R. In some schemes, a second portion of the ciphertext, Y, often called the "confirmation", is derived from X or M, using H_3, as shown in the table, and line 4 then defines C^*. Finally, H_4 is used as a key derivation function to extract a symmetric key K from the parameter Z, which varies widely among transforms.

In total, 26 of the 39 NIST PQC submissions which target KEMs in either the first or second round use transforms which fall into our framework. The remaining schemes do not use more than one random oracle, construct KEMs without transforming PKE schemes, or target security definitions other than IND-CCA.

1.2.3 Submissions we break

We present attacks on `BIG QUAKE` [24], `DAGS` [23], and `Round2` [103]. These attacks succeed in full or partial recovery of the encapsulated KEM key from a ciphertext, and are extremely fast. We have implemented the attacks to verify them.

Although none of these schemes progressed to Round 2 of the competition without significant modification, to the best of our knowledge, none of the attacks we described were pointed out during the review process. Given the attacks' superficiality, this is surprising and suggests to us that more attention should be paid to oracle cloning methods and their vulnerabilities during review.

RANDOMNESS-BASED DECRYPTION. The PKE schemes used by `BIG QUAKE` and `Round2` have the property that given a ciphertext $C \leftarrow \mathsf{PKE.E}(pk, M; R)$ and also given the coins R, it is easy to recover M, even without knowledge of the secret key. We formalize this property, saying PKE allows randomness-based decryption, if there is an (efficient) algorithm $\mathsf{PKE.DecR}$ such that $\mathsf{PKE.DecR}(pk, \mathsf{PKE.E}(pk, M; R), R) = M$ for any public key pk, coins R and message m. This will be used in our attacks.

Attack on `BIG QUAKE`. The base KEM $\mathsf{KE}_1[H_1,H_2,H_3,H_4]$ is given by the transform $\mathbf{T}_9$ in the table of Figure 1.1. The final KEM $\mathsf{KE}_2[F]$ uses a single function F to instantiate the random oracles, which it does as follows. It sets $H_3 = H_4 = F$ and $H_2 = W[F] \circ F$ for a certain function W (the rejection sampling algorithm) whose details will not matter for us. The notation $W[F]$ meaning that W has oracle access to F. The following attack (explanations after the pseudocode) recovers the encapsulated KEM key K from ciphertext $C^* \leftarrow\!\!{\scriptstyle\$}\ \mathsf{KE}_1.\mathsf{E}[F](pk)$—

```
Adversary $\mathscr{A}[F](pk,C^*)$   // Input public key and ciphertext, oracle for $F$
1. $C \| Y \leftarrow C^*$   // Parse $C^*$ to get PKE ciphertext $C$ and $Y = H_3(M)$
2. $R \leftarrow W[F](Y)$   // Apply function $W[F]$ to $Y$ to recover coins $R$
3. $M \leftarrow \mathsf{PKE.DecR}(pk,C,R)$   // Use randomness-based decryption for $\mathsf{PKE}$
4. $K \leftarrow F(M)$ ; Return $K$
```

As per $\mathbf{T}_9$ we have $Y = H_3(M) = F(M)$. The coins for $\mathsf{PKE.E}$ are $R = H_2(M) = (W[F] \circ F)(M) = W[F](F(M)) = W[F](Y)$. Since Y is in the ciphertext, the coins R can be recovered as shown at line 2. The PKE scheme allows randomness-based decryption, so at line 3 we can recover the message M underlying C using algorithm $\mathsf{PKE.DecR}$. But $K = H_4(M) = F(M)$, so K can now be recovered as well. In conclusion, the specific cloning method chosen by `BIG QUAKE` leads to complete recovery of the encapsulated key from the ciphertext.

Attack on Round2. The base KEM $\mathsf{KE}_1[H_2,H_3,H_4]$ is given by the transform $\mathbf{T}_{11}$ in the table of Figure 1.1. The final KEM $\mathsf{KE}_2[F]$ uses a single base function F to instantiate the final functions, which it does as follows. It sets $H_4 = F$. The specification and reference implementation differ in how H_2, H_3 are defined: In the former, $H_2(x) = F(F(x)) \,\|\, F(x)$ and $H_3(x) = F(F(F(x)))$, while, in the latter, $H_2(x) = F(F(F(x))) \,\|\, F(x)$ and $H_3(x) = F(F(X))$. These differences arise from differences in the way the output of a certain function $W[F]$ is parsed.

Our attack is on the reference-implementation version of the scheme. We need to also know that the scheme sets k^* so that $R \| K' \leftarrow H_2(X)$ with $H_2(X) = F(F(F(X))) \| F(X)$ results in $R = F(F(F(X)))$. But $Y = H_3(X) = F(F(X))$, so $R = F(Y)$ can be recovered from the ciphertext. Again exploiting the fact that the PKE scheme allows randomness-based decryption, we obtain the following attack that recovers the encapsulated KEM key K from ciphertext $C^* \leftarrow\!\!{\scriptstyle\$}\ \mathsf{KE}_1.\mathsf{E}[F](pk)$—

```
Adversary $\mathscr{A}[F](pk,C^*)$   // Input public key and ciphertext, oracle for $F$
1. $C\|Y \leftarrow C^*$; $R \leftarrow F(Y)$
2. $M \leftarrow \mathsf{PKE.DecR}(pk,C,R)$ ; $K \leftarrow F(M)$ ; Return $K$
```

This attack exploits the difference between the way H_2, H_3 are defined across the specification and implementation, which may be a bug in the implementation with regard to the parsing of $W[F](x)$. However, the attack also exploits dependencies between H_2 and H_3, which ought not to exist when instantiating what are required to be distinct random oracles.

Round2 was incorporated into the second-round submission **Round5**, which specifies a different base function and cloning functor (the latter of which uses the secure method we call "output splitting") to instantiate oracles H_2 and H_3. This attack therefore does not apply to **Round5**.

Attack on DAGS. If x is a byte string we let $x[i]$ be its i-th byte, and if x is a bit string we let x_i be its i-th bit. We say that a function V is an extendable output function if it takes input a string x and an integer ℓ to return an ℓ-byte output, and $\ell_1 \leq \ell_2$ implies that $V(x,\ell_1)$ is a prefix of $V(x,\ell_2)$. If $v = v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8$ is a byte then let $Z(v) = 00 v_3 v_4 v_5 v_6 v_7 v_8$ be obtained by zeroing out the first two bits. If y is a string of ℓ bytes then let $Z'(y) = Z(y[1]) \| \cdots \| Z(y[\ell])$. Now let $V'(x,\ell) = Z'(V(x,\ell))$.

The base KEM $\mathsf{KE}_1[H_1,H_2,H_3,H_4]$ is given by the transform $\mathbf{T}_8$ in the table of Figure 1.1. The final KEM $\mathsf{KE}_2[V]$ uses an extendable output function V to instantiate the random oracles, which it does as follows. It sets $H_2(x) = V'(x,512)$ and $H_3(x) = V'(x,32)$. It sets $H_4(x) = V(x,64)$.

As per $\mathbf{T}_8$ we have $K = H_4(M)$ and $Y = H_3(M)$. Let L be the first 32 bytes of the 64-byte K. Then $Y = Z'(L)$. So Y reveals $32 \cdot 6 = 192$ bits of K. Since Y is in the ciphertext, this results in a partial encapsulated-key recovery attack. The attack reduces the effective length of K from $64 \cdot 8 = 512$ bits to $512 - 192 = 320$ bits, meaning 37.5% of the encapsulated key is recovered. Also $R = H_2(M)$, so Y, as part of the ciphertext, reveals 32 bytes of R, which does not seem desirable, even though it is not clear how to exploit it for an attack.

1.2.4 Submissions with unclear security

For the scheme **NewHope** [11], we can give neither an attack nor a proof of security. However, we can show that the final functions H_2, H_3, H_4 produced by the cloning functor $\mathbf{F}_{\texttt{NewHope}}$ with oracle access to a single extendable-output function V are differentiable from independent random oracles. The cloning functor $\mathbf{F}_{\texttt{NewHope}}$ sets $H_1(x) = V(x, 128)$ and $H_4 = V(x, 32)$. It computes H_2 and H_3 from V using the output splitting cloning functor. Concretely, KE_2 parses $V(x, 96)$ as $H_2(x) \| H_3(x)$, where H_2 has output length 64 bytes and H_3 has output length 32 bytes. Because V is an extendable-output function, $H_4(x)$ will be a prefix of $H_2(x)$ for any string x.

We do not know how to exploit this correlation to attack the IND-CCA security of the final KEM scheme $\mathsf{KE}_2[V]$, and we conjecture that, due to the structure of $\mathbf{T}_{10}$, no efficient attack exists. We can, however, attack the rd-indiff security of functor $\mathbf{F}_{\texttt{NewHope}}$, showing that that the security proof for the base KEM $\mathsf{KE}_1[H_2, H_3, H_4]$ does not naturally transfer to $\mathsf{KE}_2[V]$. Therefore, in order to generically extend the provable security results for KE_1 to KE_2, it seems advisable to instead apply appropriate oracle cloning methods.

1.2.5 Submissions with provable security but ambiguous specification

In their reference implementations, these submissions use cloning functors which we can and do validate via our framework, providing provable security in the random oracle model for the final KEM schemes. However, the submission documents do not clearly specify a secure cloning functor, meaning that variant implementations or adaptations may unknowingly introduce weaknesses. The schemes **BIKE** [14], KCL [200], LAC [154], Lizard [71], LOCKER [15], Odd Manhattan [182], **ROLLO-II** [157], **Round5** [19], **SABER** [79] and Titanium [197] fall into this group.

Length differentiation. Many of these schemes use the “identity" functor in their reference implementations, meaning that they set the final functions $H_1 = H_2 = H_3 = H_4 = F$ for a single base function F. If the scheme $\mathsf{KE}_1[H_1, H_2, H_3, H_4]$ never queries two different oracles on inputs of a single length, the domains of $H_1, \ldots, H_4$ are implicitly separated. Reference implementations typically enforce this separation by fixing the input length of every call to F. Our formalism calls this query restriction "length differentiation" and proves its security as an oracle cloning method. We also generalize it to all methods which prevent the scheme from querying any two distinct random oracles on a single input.

In the following, we discuss two schemes from the group, **ROLLO-II** and Lizard, where ambiguity about cloning methods between the specification and reference implementation jeopardizes the security of applications using these schemes. It will be important that, like BIG QUAKE and RoundTwo, the PKE schemes defined by **ROLLO-II** and Lizard allow randomness-based decryption.

The scheme **ROLLO-II** [157] defines its base KEM $\mathsf{KE}_1[H_1, H_2, H_3, H_4]$ using the $\mathbf{T}_5$ transform from Figure 1.1. The submission document states that H_1, H_2, H_3, and H_4 are “typically" instantiated with a single fixed-length hash function F, but does not describe the cloning functors used to do so. If the identity functor is used, so that $H_1 = H_2 = H_3 = H_4 = F$, (or more generally, any functor that sets $H_2 = H_3$), an attack is possible. In the transform $\mathbf{T}_5$, both H_2 and H_3 are queried on the same input M. Then $Y = H_3(M) = F(M) = H_2(M) = R$ leaks the PKE's random coins, so the following attack will allow total key recovery via the randomness-based decryption.

```
Adversary $\mathscr{A}[F](pk, C^*)$   // Input public key and ciphertext, oracle for $F$
1. $C \| Y \leftarrow C^*$ ; $M \leftarrow \mathsf{PKE.DecR}(pk, C, Y)$   // ($Y = R$ is the coins)
2. $K \leftarrow F(M \| C \| Y)$ ; Return $K$
```

In the reference implementation of **ROLLO-II**, however, H_2 is instantiated using a second, independent function V instead of F, which prevents the above attack. Although the random oracles H_1, H_3 and H_4 are instantiated using the identity functor, they are never queried on the same input thanks to length differentiation. As a result, the reference implementation of **ROLLO-II** is provably secure, though alternate implementations could be both compliant with the submission document and completely insecure. The relevant portions of both the specification and the reference implementation were originally found in the corresponding first-round submission (LOCKER).

Lizard [71] follows transform $\mathbf{T}_9$ to produce its base KEM $\mathsf{KE}_1[H_2, H_3, H_4]$. Its submission document suggests instantiation with a single function F as follows: it sets $H_3 = H_4 = F$, and it sets $H_2 = W \circ F$ for some postprocessing function W whose details are irrelevant here. Since, in $\mathbf{T}_9$, $Y = H_3(M) = F(M)$ and $R = H_2(M) = W \circ F(M) = W(Y)$, the randomness R will again be leaked through Y in the ciphertext, permitting a key-recovery attack using randomness-based decryption much like the others we have described. This attack is prevented in the reference implementation of Lizard, which instantiates H_3 and H_4 using an independent function G. The domains of H_3 and H_4 are separated by length differentiation. This allows us to prove the security of the final KEM $\mathsf{KE}_2[G,F]$, as defined by the reference implementation.

However, the length differentiation of H_3 and H_4 breaks down in the chosen-ciphertext-secure PKE variant specification of Lizard, which transforms KE_1. The PKE scheme, given a plaintext P, chooses a random message M, computes $R = H_2(M)$ and $Y = H_3(M)$ according to $\mathbf{T}_9$, but it computes $K = H_4(M)$, then includes the value $B = K \oplus P$ as part of the ciphertext C^*. Both the identity functor and the functor used by the KEM reference implementation set $H_3 = H_4$, so the following attack will extract the plaintext from any ciphertext–

```
Adversary $\mathscr{A}(pk, C^*)$   // Input public key and ciphertext
1. $C \| B \| Y \leftarrow C^*$   // Parse $C^*$ to get $Y$ and $B = P \oplus K$
2. $P \leftarrow Y \oplus B$ ; Return $P$   // $Y = H_3(M) = H_4(M) = K$ is the mask.
```

The reference implementation of the public-key encryption schemes prevents the attack by cloning H_3 and H_4 from G via a third cloning functor, this one using the output splitting method. Yet, the inconsistency in the choice of cloning functors between the specification and both implementations underlines that ad-hoc cloning functors may easily “get lost” in modifications or adaptations of a scheme.

1.2.6 Submissions with clear provable security

Here we place schemes which explicitly discuss their methods for domain separation and follow good practice in their implementations: **Classic McEliece** [46], **CRYSTALS-Kyber** [18], EMBLEM [193], **FrodoKEM** [170], **HQC** [159], LIMA [196], NTRU-HRSS-KEM [121], **NTRU Prime** [47], **NTS-KEM** [10], **RQC** [158], **SIKE** [130] and **ThreeBears** [116]. These schemes are careful to account for dependencies between random oracles that are considered to be independent in their security models. When choosing to clone multiple random oracles from a single primitive, the schemes in this group use padding bytes, deploy hash functions designed to accommodate domain separation, or restrictions on the length of the inputs which are codified in the specification. These explicit domain separation techniques can be cast in the formalism we develop in this work.

HQC and **RQC** are unique among the PQC KEM schemes in that their specifications warn that the identity functor admits key-recovery attacks. As protection, they recommend that H_2 and H_3 be instantiated with unrelated primitives.

SIGNATURES. Although the main focus of this paper is on domain separation in KEMs, we wish to note that these issues are not unique to KEMs. At least one digital signature scheme in the second round of the NIST PQC competition, **MQDSS** [70], models multiple hash functions as independent random oracles in its security proof, then clones them from the same primitive without explicit domain separation. We have not analyzed the NIST PQC digital signature schemes' security to see whether more subtle domain separation is present, or whether oracle collisions admit the same vulnerabilities to signature forgery as they do to session key recovery. This does, however, highlight that the problem of random oracle cloning is pervasive among more types of cryptographic schemes.

1.3 Preliminaries

BASIC NOTATION. By $[i..j]$ we abbreviate the set $\{i,\ldots,j\}$, for integers $i \leq j$. If $\mathbf{x}$ is a vector then $|\mathbf{x}|$ is its length (the number of its coordinates), $\mathbf{x}[i]$ is its i-th coordinate and $[\mathbf{x}] = \{\mathbf{x}[i] : i \in [1..|\mathbf{x}|]\}$ is the set of its coordinates. The empty vector is denoted $()$. If S is a set, then S^* is the set of vectors over S, meaning the set of vectors of any (finite) length with coordinates in S. Strings are identified with vectors over $\{0,1\}$, so that if $x \in \{0,1\}^*$ is a string then $|x|$ is its length, $x[i]$ is its i-th bit, and $x[i..j]$ is the substring from its i-th to its j-th bit (including), for $i \leq j$. The empty string is ε. If x,y are strings then we write $x \preceq y$ to indicate that x is a prefix of y. If S is a finite set then $|S|$ is its size (cardinality). A set $S \subseteq \{0,1\}^*$ is *length closed* if $\{0,1\}^{|x|} \subseteq S$ for all $x \in S$.

We let $y \leftarrow A[\mathrm{O}_1,\ldots](x_1,\ldots;r)$ denote executing algorithm A on inputs $x_1,\ldots$ and coins r, with access to oracles $\mathrm{O}_1,\ldots$, and letting y be the result. We let $y \leftarrow\!\!\$\, A[\mathrm{O}_1,\ldots](x_1,\ldots)$ be the resulting of picking r at random and letting $y \leftarrow A[\mathrm{O}_1,\ldots](x_1,\ldots;r)$. We let $\mathrm{OUT}(A[\mathrm{O}_1,\ldots](x_1,\ldots))$ denote the set of all possible outputs of algorithm A when invoked with inputs $x_1,\ldots$ and access to oracles $\mathrm{O}_1,\ldots$. Algorithms are randomized unless otherwise indicated. Running time is worst case. An adversary is an algorithm.

We use the code-based game-playing framework of [42]. A game G (see Figure 1.2 for an example) starts with an INIT procedure, followed by a non-negative number of additional procedures, and ends with a FIN procedure. Procedures are also called oracles. Execution of adversary $\mathscr{A}$ with game G consists of running $\mathscr{A}$ with oracle access to the game procedures, with the restrictions that $\mathscr{A}$'s first call must be to INIT, its last call must be to FIN, and it can call these two procedures at most once. The output of the execution is the output of FIN. We write $\Pr[\mathrm{G}(\mathscr{A})]$ to denote the probability that the execution of game G with adversary $\mathscr{A}$ results in the output being the boolean true. Note that our adversaries have no output. The role of what in other treatments is the adversary output is, for us, played by the query to FIN. We adopt the convention that the running time of an adversary is the worst-case time to execute the game with the adversary, so the time taken by game procedures (oracles) to respond to queries is included.

FUNCTIONS. As usual $g\colon \mathscr{D} \to \mathscr{R}$ indicates that g is a function taking inputs in the domain set $\mathscr{D}$ and returning outputs in the range set $\mathscr{R}$. We may denote these sets by $\mathsf{Dom}(g)$ and $\mathsf{Rng}(g)$, respectively.

We say that $g\colon \mathsf{Dom}(g) \to \mathsf{Rng}(g)$ has output length ℓ if $\mathsf{Rng}(g) = \{0,1\}^{\ell}$. We say that g is a single output-length (sol) function if there is some ℓ such that g has output length ℓ and also the set $\mathscr{D}$ is length closed. We let $\mathsf{SOL}(\mathscr{D},\ell)$ denote the set of all sol functions $g\colon \mathscr{D} \to \{0,1\}^{\ell}$.

We say g is an extendable output length (xol) function if the following are true: (1) $\mathsf{Rng}(g) = \{0,1\}^*$ (2) there is a length-closed set $\mathsf{Dom}_*(g)$ such that $\mathsf{Dom}(g) = \mathsf{Dom}_*(g) \times N$ (3) $|g(x,\ell)| = \ell$ for all $(x,\ell) \in \mathsf{Dom}(g)$, and (4) $g(x,\ell) \preceq g(x,\ell')$ whenever $\ell \leq \ell'$. We let $\mathsf{XOL}(\mathscr{D})$ denote the set of all xol functions $g\colon \mathscr{D} \to \{0,1\}^*$.

1.4 Read-only indifferentiability of translating functors

We define read-only indifferentiability (rd-indff) of functors. Then we define a class of functors called translating, and give general results about their rd-indiff security. Later we will apply this to analyze the security of cloning functors, but the treatment in this section is broader and, looking ahead to possible future applications, more general than we need for ours.

1.4.1 Functors and read-only indifferentiability

A random oracle, formally, is a function drawn at random from a certain space of functions. A construction (functor) is a mapping from one such space to another. We start with definitions for these.

Function spaces and functors. A function space FS is simply a set of functions, with the requirement that all functions in the set have the same domain $\mathsf{Dom}(\mathsf{FS})$ and the same range $\mathsf{Rng}(\mathsf{FS})$. Examples are $\mathrm{SOL}(\mathscr{D},\ell)$ and $\mathrm{XOL}(\mathscr{D})$. Now $f \leftarrow\!\!\$\, \mathsf{FS}$ means we pick a function uniformly at random from the set FS.

Sometimes (but not always) we want an extra condition called input independence. It asks that the values of f on different inputs are identically and independently distributed when $f \leftarrow\!\!\$\, \mathsf{FS}$. More formally, let $\mathscr{D}$ be a set and let Out be a function that associates to any $W \in \mathscr{D}$ a set $\mathrm{Out}(W)$. Let $\mathrm{Out}(\mathscr{D})$ be the union of the sets $\mathrm{Out}(W)$ as W ranges over $\mathscr{D}$. Let $\mathrm{FUNC}(\mathscr{D},\mathrm{Out})$ be the set of all functions $f\colon \mathscr{D} \to \mathrm{Out}(\mathscr{D})$ such that $f(W) \in \mathrm{Out}(W)$ for all $W \in \mathscr{D}$. We say that FS provides input independence if there exists such a Out such that $\mathsf{FS} = \mathrm{FUNC}(\mathsf{Dom}(\mathsf{FS}),\mathrm{Out})$. Put another way, there is a bijection between FS and the set S that is the cross product of the sets $\mathrm{Out}(W)$ as W ranges over $\mathsf{Dom}(\mathsf{FS})$. (Members of S are $|\mathsf{Dom}(\mathsf{FS})|$-vectors.) As an example the function space $\mathrm{SOL}(\mathscr{D},\ell)$ satisfies input independence, but $\mathrm{XOL}(\mathscr{D})$ does *not* satisfy input independence.

Let SS be a function space that we call the starting space. Let ES be another function space that we call the ending space. We imagine that we are given a function $s \in \mathsf{SS}$ and want to construct a function $e \in \mathsf{ES}$. We refer to the object doing this as a functor. Formally a *functor* is a deterministic algorithm $\mathbf{F}$ that, given as oracle a function $s \in \mathsf{SS}$, returns a function $\mathbf{F}[s] \in \mathsf{ES}$. We write $\mathbf{F}\colon \mathsf{SS} \to \mathsf{ES}$ to emphasize the starting and ending spaces of functor $\mathbf{F}$.

Rd-indiff. We want the ending function to "emulate" a random function from ES. Indifferentiability is a way of defining what this means. The original definition of MRH [156] has been followed by many variants [74, 187, 86, 164]. Here we give ours, called read-only indifferentiability, which implies composition not just for single-stage games, but even for multi-stage ones [187, 86, 164].

Let ES and SS be function spaces, and let $\mathbf{F}\colon \mathsf{SS} \to \mathsf{ES}$ be a functor. Our variant of

```
Game $\mathbf{G}^{\text{rd-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{ES},\mathscr{W},\mathsf{Sim}}$

INIT:
  $s \leftarrow\!\!\$\ \mathsf{SS}$
  $e_1 \leftarrow \mathbf{F}[s]$ ; $e_0 \leftarrow\!\!\$\ \mathsf{ES}$
  $b \leftarrow\!\!\$\ \{0,1\}$
  $st \leftarrow\!\!\$\ \mathsf{Sim.Setup}()$

PRIV($W$):
  If $W \in \mathscr{W}$ then return $e_b(W)$
  Else return $\perp$

PUB($U$):
  if $(b = 1)$ then return $s(U)$
  else return $\mathsf{Sim.Ev}[e_0](st, U)$

FIN($b'$):
  return $(b = b')$
```

Figure 1.2. Game defining read-only indifferentiability.

indifferentiability mandates a particular, strong simulator, which can read, but not write, its (game-maintained) state, so that this state is a static quantity. Formally a *read-only simulator* Sim for $\mathbf{F}$ specifies a *setup algorithm* $\mathsf{Sim.Setup}$ which outputs the state, and a deterministic *evaluation algorithm* $\mathsf{Sim.Ev}$ that, given as oracle a function $e \in \mathsf{ES}$, and given a string $st \in \mathsf{OUT}(\mathsf{Sim.Setup})$ (the read-only state), defines a function $\mathsf{Sim.Ev}[e](st, \cdot)\colon \mathsf{Dom}(\mathsf{SS}) \to \mathsf{Rng}(\mathsf{SS})$.

The intent is that $\mathsf{Sim.Ev}[e](st, \cdot)$ play the role of a starting function $s \in \mathsf{SS}$ satisfying $\mathbf{F}[s] = e$. To formalize this, consider the read-only indifferentiability game $\mathbf{G}^{\text{rd-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{ES},\mathscr{W},\mathsf{Sim}}$ of Figure 1.2, where $\mathscr{W} \subseteq \mathsf{Dom}(\mathsf{ES})$ is called the working domain. The adversary $\mathscr{A}$ playing this game is called a distinguisher. Its advantage is defined as

$$\mathbf{Adv}^{\text{rd-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{ES},\mathscr{W},\mathsf{Sim}}(\mathscr{A}) = 2 \cdot \Pr\left[\mathbf{G}^{\text{rd-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{ES},\mathscr{W},\mathsf{Sim}}(\mathscr{A})\right] - 1.$$

To explain, in the game, b is a challenge bit that the distinguisher is trying to determine. Function e_b is a random member of the ending space ES if $b = 0$ and is $\mathbf{F}[s](\cdot)$ if $b = 1$. The query W to oracle PRIV is required to be in $\mathsf{Dom}(\mathsf{ES})$. The oracle returns the value of e_b on W, but only if W is in the working domain, otherwise returning $\perp$. The query U to oracle PUB is required to be in $\mathsf{Dom}(\mathsf{SS})$. The oracle returns the value of s on U in the $b = 1$ case, but when $b = 0$, the simulator evaluation algorithm $\mathsf{Sim.Ev}$ must answer the query with access to an oracle for e_0. The distinguisher ends by calling FIN with its guess $b' \in \{0,1\}$ of b and the game returns true if $b' = b$ (the distinguisher's guess is correct) and false otherwise.

The working domain $\mathscr{W} \subseteq \mathsf{Dom}(\mathsf{ES})$, a parameter of the definition, is included as a way to allow the notion of read-only indifferentiability to provide results for oracle cloning methods like length differentiation whose security depends on domain restrictions.

The $\mathsf{Sim.Ev}$ algorithm is given direct access to e_0, rather than access to PRIV as in other definitions, to bypass the working domain restriction, meaning it may query e_0 at points in $\mathsf{Dom}(\mathsf{ES})$ that are outside the working domain.

All invocations of $\mathsf{Sim.Ev}[e_0]$ are given the same (static, game-maintained) state st as input, but $\mathsf{Sim.Ev}[e_0]$ cannot modify this state, which is why it is called read-only. Note INIT does not return st, meaning the state is not given to the distinguisher.

DISCUSSION. To compare rd-indiff to other indiff notions, we set $\mathscr{W} = \mathsf{Dom}(\mathsf{ES})$, because prior notions do not include working domains. Now, rd-indiff differs from prior indiff notions because it requires that the simulator state be just the immutable string chosen at the start of the game. In this regard, rd-indiff falls somewhere between the original MRH-indiff [156] and reset indiff [187] in the sense that our simulator is more restricted than in the first and less than in the second. A construction (functor) that is reset-indiff is thus rd-indiff, but not necessarily vice-versa, and a construct that is rd-indiff is MRH-indiff, but not necessarily vice-versa. Put another way, the class of rd-indff functors is larger than the class of reset-indiff ones, but smaller than the class of MRH-indiff ones. Now, RSS's proof [187] that reset-indiff implies security for multi-stage games extends to rd-indiff, so we get this for a potentially larger class of functors. This larger class includes some of the cloning functors we have described, which are not necessarily reset-indiff.

1.4.2 Translating functors

TRANSLATING FUNCTORS. We focus on a class of functors that we call translating. This class includes natural and existing oracle cloning methods, in particular all the effective methods used by NIST KEMs, and we will be able to prove general results for translating functors that can be applied to the cloning methods.

A translating functor $\mathbf{T}\colon \mathsf{SS} \to \mathsf{ES}$ is a functor that, with oracle access to s and on input $W \in \mathsf{Dom}(\mathsf{ES})$, non-adaptively calls s on a fixed number of inputs, and computes its output $\mathbf{T}[s](W)$ from the responses and W. Its operation can be split into three phases which do not share state: (1) a pre-processing phase which chooses the inputs to s based on W alone (2) the calls to s to obtain responses (3) a post-processing phase which uses W and the responses collected in phase 2 to compute the final output value $\mathbf{T}[s](W)$.

Proceeding to the definitions, let SS, ES be function spaces. A $(\mathsf{SS}, \mathsf{ES})$-*query translator* is a function (deterministic algorithm) $\mathsf{QT}\colon \mathrm{Dom}(\mathsf{ES}) \to \mathrm{Dom}(\mathsf{SS})^*$, meaning it takes a point W in the domain of the ending space and returns a vector of points in the domain of the starting space. This models the pre-processing. A $(\mathsf{SS}, \mathsf{ES})$-*answer translator* is a function (deterministic algorithm) $\mathsf{AT}\colon \mathrm{Dom}(\mathsf{ES}) \times \mathrm{Rng}(\mathsf{SS})^* \to \mathrm{Rng}(\mathsf{ES})$, meaning it takes the original W, and a vector of points in the range of the starting space, to return a point in the range of the ending space. This models the post-processing. To the pair $(\mathsf{QT}, \mathsf{AT})$, we associate the functor $\mathbf{TF}_{\mathsf{QT},\mathsf{AT}}\colon \mathsf{SS} \to \mathsf{ES}$, defined as follows:

```
Algorithm $\mathbf{TF}_{\mathsf{QT},\mathsf{AT}}[s](W)$    // Input $W \in \mathrm{Dom}(\mathsf{ES})$ and oracle $s \in \mathsf{SS}$
$\boldsymbol{U} \leftarrow \mathsf{QT}(W)$
For $j = 1, \ldots, |\boldsymbol{U}|$ do $\boldsymbol{V}[j] \leftarrow s(\boldsymbol{U}[j])$    // $\boldsymbol{U}[j] \in \mathsf{Dom}(\mathsf{SS})$
$Y \leftarrow \mathsf{AT}(W, \boldsymbol{V})$ ; Return $Y$
```

The above-mentioned calls of phase (2) are done in the second line of the code above, so that this implements a translating functor as we described. Formally we say that a functor $\mathbf{F}\colon \mathsf{SS} \to \mathsf{ES}$ is *translating* if there exists a $(\mathsf{SS}, \mathsf{ES})$-query translator QT and a $(\mathsf{SS}, \mathsf{ES})$-answer translator AT such that $\mathbf{F} = \mathbf{TF}_{\mathsf{QT},\mathsf{AT}}$.

Inverses. So far, query and answer translators may have just seemed an unduly complex way to say that a translating oracle construction is one that makes non-adaptive oracle queries. The purpose of making the query and answer translators explicit is to define *invertibility*, which determines rd-indiff security.

Let SS and ES be function spaces. Let QTI be a function (deterministic algorithm) that takes an input $U \in \mathrm{Dom}(\mathsf{SS})$ and returns a vector $\boldsymbol{W}$ over $\mathrm{Dom}(\mathsf{ES})$. We allow QTI to return the empty vector $()$, which is taken as an indication of failure to invert. Define the *support* of QTI, denoted $\mathbf{sup}(\mathsf{QTI})$, to be the set of all $U \in \mathrm{Dom}(\mathsf{SS})$ such that $\mathsf{QTI}(U) \neq ()$. Say that QTI has *full support* if $\mathbf{sup}(\mathsf{QTI}) = \mathrm{Dom}(\mathsf{SS})$, meaning there is no $U \in \mathrm{Dom}(\mathsf{SS})$ such that $\mathsf{QTI}(U) = ()$. Let ATI be a function (deterministic algorithm) that takes $U \in \mathrm{Dom}(\mathsf{SS})$ and a vector $\boldsymbol{Y}$ over $\mathrm{Rng}(\mathsf{ES})$ to return an output in $\mathrm{Rng}(\mathsf{SS})$. Given a function $e \in \mathsf{ES}$, we define the function $\mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}\colon \mathrm{Dom}(\mathsf{SS}) \to \mathrm{Rng}(\mathsf{SS})$ by

```
Function P[e]_{QTI,ATI}(U)    // $U \in \mathrm{Dom}(\mathsf{SS})$
$\boldsymbol{W} \leftarrow \mathsf{QTI}(U)$ ; $\boldsymbol{Y} \leftarrow e(\boldsymbol{W})$ ; $V \leftarrow \mathsf{ATI}(U, \boldsymbol{Y})$ ; Return $V$
```

Above, e is applied to a vector component-wise, meaning $e(\boldsymbol{W})$ is defined as the vector $(e(\boldsymbol{W}[1]), \ldots, e(\boldsymbol{W}[|\boldsymbol{W}|]))$.

We require that the function $\mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}$ belong to the starting space SS. Now let QT be a $(\mathsf{SS}, \mathsf{ES})$-query translator and AT a $(\mathsf{SS}, \mathsf{ES})$-answer translator. Let $\mathscr{W} \subseteq \mathrm{Dom}(\mathsf{ES})$ be a working domain. We say that $\mathsf{QTI}, \mathsf{ATI}$ *are inverses of* QT, AT *over* $\mathscr{W}$ if two conditions are true. The first is that for all $e \in \mathsf{ES}$ and all $W \in \mathscr{W}$ we have

$$\mathbf{TF}_{\mathsf{QT},\mathsf{AT}}[\mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}](W) = e(W)\ . \tag{1.2}$$

This equation needs some parsing. Fix a function $e \in \mathsf{ES}$ in the ending space. Then $s = \mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}$ is in SS. Recall that the functor $\mathbf{F} = \mathbf{TF}_{\mathsf{QT},\mathsf{AT}}$ takes a function s in the starting space as an oracle and defines a function $e' = \mathbf{F}[s]$ in the ending space. Equation (1.2) is asking that e' is identical to the original function e, on the working domain $\mathscr{W}$. The second condition (for invertibility) is that if $U \in \{\mathsf{QT}(W)[i] : W \in \mathscr{W}\}$ —that is, U is an entry of the vector $\boldsymbol{U}$ returned by QT on some input W— then $\mathsf{QTI}(U) \neq ()$. Note that if QTI has full support then this condition is already true, but otherwise it is an additional requirement.

We say that $(\mathsf{QT}, \mathsf{AT})$ is invertible over $\mathscr{W}$ if there exist $\mathsf{QTI}, \mathsf{ATI}$ such that $\mathsf{QTI}, \mathsf{ATI}$ are inverses of QT, AT over $\mathscr{W}$, and we say that a translating functor $\mathbf{TF}_{\mathsf{QT},\mathsf{AT}}$ is invertible over $\mathscr{W}$ if $(\mathsf{QT}, \mathsf{AT})$ is invertible over $\mathscr{W}$.

In the rd-indiff context, function $\mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}$ will be used by the simulator. Roughly, we try to set $\mathsf{Sim.Ev}[e](st, U) = \mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}(U)$. But we will only be able to successfully do this for $U \in \mathbf{sup}(\mathsf{QTI})$. The state st is used by $\mathsf{Sim.Ev}$ to provide replies when $U \notin \mathbf{sup}(\mathsf{QTI})$.

Equation (1.2) is a correctness condition. There is also a security metric. Consider the *translation indistinguishability* game $\mathbf{G}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}$ of Figure 1.3. Define the ti-advantage of adversary $\mathscr{B}$ via

$$\mathbf{Adv}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}(\mathscr{B}) = 2 \cdot \Pr\left[\mathbf{G}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}(\mathscr{B})\right] - 1.$$

```
Game $\mathbf{G}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}$
INIT:
  $b \leftarrow\!\!\$ \{0,1\}$ ; $e \leftarrow\!\!\$ \mathsf{ES}$
  $s_1 \leftarrow\!\!\$ \mathsf{SS}$ ; $s_0 \leftarrow \mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}$
PUB($U$): // $U \in \mathrm{Dom}(\mathsf{SS})$
  If $\mathsf{QTI}(U) = ()$ then return $\perp$
  return $s_b(U)$
FIN($b'$):
  return $(b = b')$
```

Figure 1.3. Game defining translation indistinguishability.

In reading the game, recall that $()$ is the empty vector, whose return by QTI represents an inversion error. TI-security is thus asking that if e is randomly chosen from the ending space, then the output of $\mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}$ on an input U is distributed like the output on U of a random function in the starting space, *but only as long as* $\mathsf{QTI}(U)$ *was non-empty.* We will see that the latter restriction creates some challenges in simulation whose resolution exploits using read-only state. We say that $(\mathsf{QTI}, \mathsf{ATI})$ provides perfect translation indistinguishability if $\mathbf{Adv}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}(\mathscr{B}) = 0$ for all $\mathscr{B}$, regardless of the running time of $\mathscr{B}$.

Additionally we of course ask that the functions $\mathsf{QT}, \mathsf{AT}, \mathsf{QTI}, \mathsf{ATI}$ all be efficiently computable. In an asymptotic setting, this means they are polynomial time. In our concrete setting, they show up in the running-time of the simulator or constructed adversaries. (The latter, as per our conventions, being the time for the execution of the adversary with the overlying game.)

1.4.3 Rd-indiff of translating functors

We now move on to showing that invertibility of a pair $(\mathsf{QT}, \mathsf{AT})$ implies rd-indifferentiability of the translating functor $\mathbf{TF}_{\mathsf{QT},\mathsf{AT}}$. We start with the case that QTI has full support.

Theorem 1. *Let* SS *and* ES *be function spaces. Let* $\mathscr{W}$ *be a subset of* $\mathrm{Dom}(\mathsf{ES})$*. Let* QT, AT *be* $(\mathsf{SS}, \mathsf{ES})$ *query and answer translators, respectively. Let* $\mathsf{QTI}, \mathsf{ATI}$ *be inverses of* QT, AT *over* $\mathscr{W}$*. Assume* QTI *has full support. Define read-only simulator* Sim *as per the top panel of Figure 1.4. Let* $\mathbf{F} = \mathbf{TF}_{\mathsf{QT},\mathsf{AT}}$*. Let* $\mathscr{A}$ *be any distinguisher. Then we construct a ti-adversary* $\mathscr{B}$ *such that*

$$\mathbf{Adv}^{\text{rd-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{ES},\mathscr{W},\mathsf{Sim}}(\mathscr{A}) \leq \mathbf{Adv}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}(\mathscr{B}) \, .$$

```
Algorithm $\mathsf{Sim.Setup}$:
    Return $\varepsilon$

Algorithm $\mathsf{Sim.Ev}[e](st, U)$:
    $\boldsymbol{W} \leftarrow \mathsf{QTI}(U)$ ; $\boldsymbol{Y} \leftarrow e(\boldsymbol{W})$ ; $V \leftarrow \mathsf{ATI}(U, \boldsymbol{Y})$
    Return $V$
```

```
Algorithm $\mathsf{Sim.Setup}$:
    $st \leftarrow\!\!\$\ \{0,1\}^{\mathsf{G.kl}}$
    Return $st$

Algorithm $\mathsf{Sim.Ev}[e](st, U)$:
    $\boldsymbol{W} \leftarrow \mathsf{QTI}(U)$
    If $\boldsymbol{W} = ()$ then return $\mathsf{G}_{st}[e](U)$
    $\boldsymbol{Y} \leftarrow e(\boldsymbol{W})$ ; $V \leftarrow \mathsf{ATI}(U, \boldsymbol{Y})$
    Return $V$
```

Figure 1.4. Simulators for Theorem 1 (top) and Theorem 2 (bottom).

```
Games $G_0$, $G_1$
INIT:
 1  $s \leftarrow\!\!\$\ \mathsf{SS}$  // Game $G_0$
 2  $e_0 \leftarrow\!\!\$\ \mathsf{ES}$ ; $s \leftarrow \mathrm{P}[e_0]_{\mathsf{QTI},\mathsf{ATI}}$  // Game $G_1$
PRIV($W$):
 3  If $W \in \mathscr{W}$ then return $\mathbf{F}[s](W)$
 4  Else return $\perp$
PUB($U$):
 5  return $s(U)$
FIN($b'$):
 6  return $(b' = 1)$

Game $G_2$
INIT:
 1  $e_0 \leftarrow\!\!\$\ \mathsf{ES}$
 2  $s \leftarrow \mathrm{P}[e_0]_{\mathsf{QTI},\mathsf{ATI}}$
PRIV($W$):
 3  If $W \in \mathscr{W}$ then return $e_0(W)$
 4  Else return $\perp$
PUB($U$):
 5  return $s(U)$
FIN($b'$):
 6  return $(b' = 1)$
```

```
Adversary $\mathscr{B}$:
 1  INIT()
 2  $\mathscr{A}$[INIT$'$, PUB$'$, PRIV$'$, FIN$'$]()
INIT$'$:
 3  Return
PUB$'$($U$):
 4  return PUB($U$)
PRIV$'$($W$):
 5  if $W \notin \mathscr{W}$ then return $\perp$
 6  $\boldsymbol{U} \leftarrow \mathsf{QT}(W)$
 7  For $j = 1, \ldots, |\boldsymbol{U}|$ do $\mathbf{V}[j] \leftarrow$ PUB($\boldsymbol{U}[j]$)
 8  return $\mathsf{AT}(W, \mathbf{V})$
FIN$'$($b'$):
 9  FIN($b'$)
```

Figure 1.5. Top: Games for proof of Theorem 1. Bottom: Adversary for proof of Theorem 1.

Let ℓ be the maximum output length of QT*. If* $\mathscr{A}$ *makes* $q_{\text{PRIV}}, q_{\text{PUB}}$ *queries to its* PRIV, PUB *oracles, respectively, then* $\mathscr{B}$ *makes* $\ell \cdot q_{\text{PRIV}} + q_{\text{PUB}}$ *queries to its* PUB *oracle. The running time of* $\mathscr{B}$ *is about that of* $\mathscr{A}$*.*

Proof of Theorem 1: Consider the games of Figure 1.5. In the left panel, line 1 is included only in G_0 and line 2 only in G_1, and this is the only way the games differ. Game G_0 is the real game, meaning the case $b = 1$ in game $\mathbf{G}^{\text{rd-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{ES},\mathscr{W},\mathsf{Sim}}$. In game G_2, oracle PRIV is switched to a random function e_0. From the description of the simulator in Figure 1.4 we see that

$$\mathsf{Sim.Ev}[e_0](\varepsilon, U) = \mathrm{P}[e_0]_{\mathsf{QTI},\mathsf{ATI}}(U)$$

for all $U \in \mathsf{Dom}(\mathsf{SS})$ and all $e_0 \in \mathsf{ES}$, so that oracle PUB in game G_2 is responding according to the simulator based on e_0. So game G_2 is the case $b = 0$ in game $\mathbf{G}^{\text{rd-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{ES},\mathscr{W},\mathsf{Sim}}$. Thus

$$\begin{aligned}\mathbf{Adv}^{\text{rd-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{ES},\mathscr{W},\mathsf{Sim}}(\mathscr{A}) &= \Pr[\mathrm{G}_0(\mathscr{A})] - \Pr[\mathrm{G}_2(\mathscr{A})] \\ &= (\Pr[\mathrm{G}_0(\mathscr{A})] - \Pr[\mathrm{G}_1(\mathscr{A})]) + (\Pr[\mathrm{G}_1(\mathscr{A})] - \Pr[\mathrm{G}_2(\mathscr{A})]) \,.\end{aligned}$$

We define translation-indistinguishability adversary $\mathscr{B}$ in Figure 1.5 so that

$$\Pr[\mathrm{G}_0(\mathscr{A})] - \Pr[\mathrm{G}_1(\mathscr{A})] \leq \mathbf{Adv}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}(\mathscr{B}) \,.$$

Adversary $\mathscr{B}$ is playing game $\mathbf{G}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}$. Using its PUB oracle, it presents the interface of G_0 and G_1 to $\mathscr{A}$. In order to simulate the PRIV oracle, $\mathscr{B}$ runs $\mathbf{TF}_{\mathsf{QT},\mathsf{AT}}[\text{PUB}]$. This is consistent with G_0 and G_1. If $b = 1$ in $\mathbf{G}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}$, then $\mathscr{B}$ perfectly simulates G_0 for $\mathscr{A}$. If $b = 1$, then $\mathscr{B}$ correctly simulates G_1 for $\mathscr{A}$. To complete the proof we claim that

$$\Pr[\mathrm{G}_1(\mathscr{A})] = \Pr[\mathrm{G}_2(\mathscr{A})] \,.$$

This is true by the correctness condition. The latter says that if $s \leftarrow \mathrm{P}[e_0]_{\mathsf{QTI},\mathsf{ATI}}$ then $\mathbf{F}[s]$ is just e_0 itself. So e_1 in game G_1 is the same as e_0 in game G_2, making their PRIV oracles identical. And their PUB oracles are identical by definition. ▮

The simulator in Theorem 1 is stateless, so when $\mathscr{W}$ is chosen to be $\mathsf{Dom}(\mathsf{ES})$ the theorem is establishing reset indifferentiability [187] of $\mathbf{F}$.

For translating functors where QTI does not have full support, we need an auxiliary primitive that we call a $(\mathsf{SS},\mathsf{ES})$-oracle aided PRF. Given an oracle for a function $e \in \mathsf{ES}$, an $(\mathsf{SS},\mathsf{ES})$-oracle aided PRF G defines a function $\mathsf{G}[e]\colon \{0,1\}^{\mathsf{G.kl}} \times \mathsf{Dom}(\mathsf{SS}) \to \mathsf{Rng}(\mathsf{SS})$. The first input is a key. For $\mathscr{C}$ an adversary, let $\mathbf{Adv}^{\mathrm{prf}}_{\mathsf{G},\mathsf{SS},\mathsf{ES}}(\mathscr{C}) = 2\Pr[\mathbf{G}^{\mathrm{prf}}_{\mathsf{G},\mathsf{SS},\mathsf{ES}}(\mathscr{C})] - 1$, where the game is

```
$\mathbf{G}^{\mathrm{prf}}_{\mathsf{G},\mathsf{SS},\mathsf{ES}}$

INIT():
  $b \leftarrow\!\!\$\ \{0,1\}$
  $e \leftarrow\!\!\$\ \mathsf{ES}$
  $st \leftarrow\!\!\$\ \{0,1\}^{\mathsf{G.kl}}$
  $s_1 \leftarrow \mathsf{G}[e](st,\cdot)$
  $s_0 \leftarrow\!\!\$\ \mathsf{SS}$

RO($W$):
  Return $e(W)$

FNO($U$):
  $V \leftarrow s_b(U)$
  Return $V$

FIN($b'$):
  Return $(b' = b)$
```

Figure 1.6. Game to define PRF security of $(\mathsf{SS},\mathsf{ES})$-oracle aided PRF G.

in Figure 4.1. The simulator uses its read-only state to store a key st for G, then using $\mathsf{G}(st,\cdot)$ to answer queries outside the support $\mathbf{sup}(\mathsf{QTI})$.

We introduce this primitive because it allows multiple instantiations. The simplest is that it is a PRF, which happens when it does not use its oracle. In that case the simulator is using a computational primitive (a PRF) in the indifferentiability context, which seems novel. Another instantiation prefixes st to the input and then invokes e to return the output. This works for certain choices of ES, but not always. Note G is used only by the simulator and plays no role in the functor.

Theorem 2. *Let* SS *and* ES *be function spaces, and assume they provide input independence. Let* $\mathscr{W}$ *be a subset of* $\mathsf{Dom}(\mathsf{ES})$*. Let* QT,AT *be* $(\mathsf{SS},\mathsf{ES})$ *query and answer translators, respectively. Let* $\mathsf{QTI},\mathsf{ATI}$ *be inverses of* QT,AT *over* $\mathscr{W}$*. Define read-only simulator* Sim *as per the bottom panel of Figure 1.4. Let* $\mathbf{F} = \mathbf{TF}_{\mathsf{QT},\mathsf{AT}}$*. Let* $\mathscr{A}$ *be any distinguisher. Then we construct a ti-adversary* $\mathscr{B}$ *and a prf-adversary* $\mathscr{C}$ *such that*

$$\mathbf{Adv}^{\mathrm{rd\text{-}indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{ES},\mathscr{W},\mathsf{Sim}}(\mathscr{A}) \leq \mathbf{Adv}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}(\mathscr{B}) + \mathbf{Adv}^{\mathrm{prf}}_{\mathsf{G},\mathsf{SS}}(\mathscr{C})\,.$$

Let ℓ *be the maximum output length of* QT *and* ℓ' *the maximum output length of* QTI*. If* $\mathscr{A}$ *makes* $q_{\mathrm{PRIV}}, q_{\mathrm{PUB}}$ *queries to its* PRIV, PUB *oracles, respectively, then* $\mathscr{B}$ *makes* $\ell \cdot q_{\mathrm{PRIV}} + q_{\mathrm{PUB}}$ *queries to its* PUB *oracle and* $\mathscr{C}$ *makes at most* $\ell \cdot \ell' \cdot q_{\mathrm{PRIV}} + q_{\mathrm{PUB}}$ *queries to its* RO *oracle and at most* $q_{\mathrm{PUB}} + \ell \cdot q_{\mathrm{PRIV}}$ *queries to its* FNO *oracle. The running times of* $\mathscr{B},\mathscr{C}$ *are about that of* $\mathscr{A}$*.*

Proof of Theorem 2: We will rely on the sequence of games in Figure 1.7. The first game G_0 is the real game, meaning the case $b = 1$ in game $\mathbf{G}^{\mathrm{rd\text{-}indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{ES},\mathscr{W},\mathsf{Sim}}$. Game G_1 differs from G_0

```
Games $G_0$, $G_1$
INIT:
  $s_1 \leftarrow\!\!\$\, \mathsf{SS}$
  $s_2 \leftarrow\!\!\$\, \mathsf{SS}$   // Game $G_1$
  $e_1 \leftarrow \mathbf{F}[s_1]$
PRIV($W$):
  If $W \in \mathscr{W}$ then return $e_1(W)$
  Else return $\perp$
PUB($U$):
  if $\mathsf{QTI}(U) = ()$ then
    return $s_2(U)$   // Game $G_1$
  return $s_1(U)$
FIN($b'$):
  return $(b' = 1)$
```

```
Game $G_2, G_3$
INIT:
  $e_0 \leftarrow\!\!\$\, \mathsf{ES}$
  $s_1 \leftarrow \mathrm{P}[e_0]_{\mathsf{QTI},\mathsf{ATI}}$
  $s_2 \leftarrow\!\!\$\, \mathsf{SS}$  // Game $G_2$
  $e_1 \leftarrow \mathbf{F}[s_1]$
  $st \leftarrow\!\!\$\, \mathsf{Sim.Setup}()$  // Game $G_3$
PRIV($W$):
  If $W \in \mathscr{W}$ then return $e_1(W)$
  Else return $\perp$
PUB($U$):
  if $\mathsf{QTI}(U) = ()$ then
    return $s_2(U)$  // Game $G_2$
    return $\mathsf{G}_{st}[e_0](U)$  // Game $G_3$
  return $s_1(U)$
FIN($b'$):
  return $(b' = 1)$
```

```
Game $G_4$
INIT:
  $e_0 \leftarrow\!\!\$\, \mathsf{ES}$
  $s_1 \leftarrow \mathrm{P}[e_0]_{\mathsf{QTI},\mathsf{ATI}}$
  $st \leftarrow\!\!\$\, \mathsf{Sim.Setup}()$
PRIV($W$):
  If $W \in \mathscr{W}$ then return $e_0(W)$
  Else return $\perp$
PUB($U$):
  if $\mathsf{QTI}(U) = ()$ then
    return $\mathsf{G}[e_0]_{st}(U)$
  return $s_1(U)$
FIN($b'$):
  return $(b' = 1)$
```

Figure 1.7. Games for proof of Theorem 2.

because it samples an additional function s_2 from the starting space. When an inversion error occurs in the PUB oracle, game G_1 answers using s_2 instead of s_1. Since the starting space SS provides input independence, both s_1 and s_2 are drawn from $\mathbf{FUNC}(\mathbf{Dom}(\mathsf{SS}), \mathbf{Out})$ for some $\mathbf{Out}$. Then on any input U, the outputs of s_1 and s_2 are identically and independently distributed. The adversary can therefore only tell that queries outside the support of QTI are not being answered by s_1 if the PUB oracle becomes inconsistent with the PRIV oracle. This happens only if the PRIV oracle, while computing $\mathbf{F}[s_1] = \mathbf{TF}_{\mathsf{QT},\mathsf{AT}}[s_1]$, queries s_1 on some point outside the support of QTI, which is impossible by the first condition in the definition of invertibility. Hence

$$\Pr[G_0(\mathscr{A})] = \Pr[G_1(\mathscr{A})].$$

```
Adversary $\mathscr{B}$:
$\textsc{init}()$
$\mathscr{A}[\textsc{init}', \textsc{pub}', \textsc{priv}', \textsc{fin}']()$

$\textsc{init}'$:
Return

$\textsc{pub}'(U)$:
if $T[U] \neq \bot$ then return $T[U]$
$W \leftarrow \textsc{pub}(U)$
if $W = \bot$ then
    $(i, X) \leftarrow U$
    $T[U] \leftarrow\!\!\$\, \mathrm{Out}(U)$
    $W \leftarrow T[U]$
return $W$

$\textsc{priv}'(W)$:
if $W \in \mathscr{W}$ then return $\mathbf{F}[\textsc{pub}](W)$
Else return $\bot$

$\textsc{fin}'(b')$:
$\textsc{fin}(b')$
```

```
Adversary $\mathscr{C}$:
$\textsc{init}()$
$\mathscr{A}[\textsc{init}', \textsc{pub}', \textsc{priv}', \textsc{fin}']()$

$\textsc{init}'$:
Return

$\textsc{pub}'(U)$:
if $\mathsf{QTI}(U) = ()$ then
    return $\textsc{Fno}(U)$
return $\mathrm{P}[\mathrm{RO}]_{\mathsf{QTI},\mathsf{ATI}}(U)$

$\textsc{priv}'(W)$:
If $W \in \mathscr{W}$ then
    return $\mathbf{F}[\textsc{pub}'](W)$
Else return $\bot$

$\textsc{fin}'(b')$:
$\textsc{fin}(b')$
```

Figure 1.8. Adversaries for proof of Theorem 2.

Between games G_1 and G_2, we draw a function e_0 from the ending space and replace s_1 with $\mathrm{P}_{\mathsf{QTI},\mathsf{ATI}}[e_0]$. We construct the translation-indistinguishability adversary $\mathscr{B}$ in Figure 1.7 so that

$$\Pr[\mathrm{G}_1(\mathscr{A})] - \Pr[\mathrm{G}_2(\mathscr{A})] \leq \mathbf{Adv}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}(\mathscr{B}).$$

This adversary simulates the interface of G_1 and G_2 for $\mathscr{A}$, using its $\textsc{pub}$ oracle to implement s_1 and check for inversion errors. It lazily samples s_2, which is consistent with G_1 and G_2 by the input independence of SS. Its $\textsc{priv}'$ oracle runs $\mathbf{F}[\textsc{pub}]$, which is consistent. When the challenge bit $b = 1$ in game $\mathbf{G}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}$, adversary $\mathscr{B}$ simulates game G_1 perfectly, and when $b = 0$ it perfectly simulates game G_2.

In game G_3, we replace s_2 with an $(\mathsf{SS}, \mathsf{ES})$-oracle-aided pseudorandom function G and sample a PRF key st in the $\textsc{init}$ oracle. We construct an adversary $\mathscr{C}$ in Figure 1.7 against the PRF-security of G. This adversary plays game $\mathbf{G}^{\mathrm{prf}}_{\mathsf{SS},\mathsf{ES},\mathsf{G}}$ and simulates the interface of games G_2 and G_3 for $\mathscr{A}$. It uses its RO oracle to simulate e_0, and it uses its $\textsc{Fno}$ oracle to answer $\textsc{pub}$ queries outside the support of QTI. When $b = 0$ in game $\mathbf{G}^{\mathrm{prf}}_{\mathsf{SS},\mathsf{ES},\mathsf{G}}$, the adversary perfectly simulates G_2

for $\mathscr{A}$, and when $b = 1$ it perfectly simulates G_3. Therefore

$$\Pr[\mathrm{G}_2(\mathscr{A})] - \Pr[\mathrm{G}_3(\mathscr{A})] \leq \mathbf{Adv}^{\mathrm{prf}}_{\mathsf{SS},\mathsf{ES},\mathsf{G}}(\mathscr{C}).$$

In Game G_4, we answer PRIV queries with e_0 directly, instead of with $\mathbf{F}[\mathrm{P}_{\mathsf{QTI},\mathsf{ATI}}[e_0]]$. By the correctness condition of invertibility, these two functions are identical, so

$$\Pr[\mathrm{G}_3(\mathscr{A})] = \Pr[\mathrm{G}_4(\mathscr{A})].$$

Looking at the pseudocode for simulator Sim in the bottom panel of Figure 1.4, we see that $\mathsf{Sim}.\mathrm{Ev}[e]$ first runs QTI on its input U. If $\mathsf{QTI}(U) = ()$, then it returns $\mathsf{G}_{st}[e](U)$. Otherwise, it runs $\mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}(U)$ and returns the output. This is identical to lines 6-8 of game G_4, so $\mathscr{A}$ wins G_4 if and only if it loses the ideal game (meaning the case $b = 0$), of the rd-indiff game $\mathbf{G}^{\mathrm{rd-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{ES},\mathscr{W},\mathsf{Sim}}$. Thus

$$\begin{aligned}
\mathbf{Adv}^{\text{reset-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{ES},\mathscr{W},\mathsf{Sim}}(\mathscr{A}) &= \Pr[\mathrm{G}_0(\mathscr{A})] - \Pr[\mathrm{G}_4(\mathscr{A})\\
&= \Pr[\mathrm{G}_1(\mathscr{A})] - \Pr[\mathrm{G}_3(\mathscr{A})]\\
&= (\Pr[\mathrm{G}_1(\mathscr{A})] - \Pr[\mathrm{G}_2(\mathscr{A})]) + (\Pr[\mathrm{G}_2(\mathscr{A})] - \Pr[\mathrm{G}_3(\mathscr{A}))]\\
&\leq \mathbf{Adv}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}(\mathscr{B}) + \mathbf{Adv}^{\mathrm{prf}}_{\mathsf{SS},\mathsf{ES},\mathsf{G}}(\mathscr{C}).
\end{aligned}$$

This completes the proof. ▌

1.5 Analysis of cloning functors

Section 1.4 defined the rd-indiff metric of security for functors and give a framework to prove rd-indiff of translating functors. We now apply this to derive security results about particular, practical cloning functors.

ARITY-n FUNCTION SPACES. The cloning functors apply to function spaces where a function specifies sub-functions, corresponding to the different random oracles we are trying to build. Formally, a function space FS is said to have arity n if its members are two-argument functions f

whose first argument is an integer $i \in [1..n]$. For $i \in [1..n]$ we let $f_i = f(i,\cdot)$ and $\mathsf{FS}_i = \{f_i : f \in \mathsf{FS}\}$, and refer to the latter as the i-th subspace of FS. We let $\mathsf{Dom}_i(\mathsf{FS})$ be the set of all X such that $(i,X) \in \mathrm{Dom}(\mathsf{FS})$.

We say that FS has sol subspaces if FS_i is a set of sol functions with domain $\mathsf{Dom}_i(\mathsf{FS})$, for all $i \in [1..n]$. More precisely, there must be integers $\mathsf{OL}_1(\mathsf{FS}),\ldots,\mathsf{OL}_n(\mathsf{FS})$ such that $\mathsf{FS}_i = \mathsf{SOL}(\mathsf{Dom}_i(\mathsf{FS}),\mathsf{OL}_i(\mathsf{FS}))$ for all $i \in [1..n]$. In this case, we let $\mathsf{Rng}_i(\mathsf{FS}) = \{0,1\}^{\mathsf{OL}_i(\mathsf{FS})}$. This is the most common case for practical uses of ROs.

To explain, access to n random oracles is modeled as access to a two-argument function f drawn at random from FS, written $f \leftarrow\$ \mathsf{FS}$. If FS has sol subspaces, then for each i, the function f_i is a sol function, with a certain domain and output length depending only on i. All such functions are included. This ensures input independence as we defined it earlier. Thus if $f \leftarrow\$ \mathsf{FS}$, then for each i and any distinct inputs to f_i, the outputs are independently distributed. Also functions $f_1,\ldots,f_n$ are independently distributed when $f \leftarrow\$ \mathsf{FS}$. Put another way, we can identify FS with $\mathsf{FS}_1 \times \cdots \times \mathsf{FS}_n$.

Domain-separating functors. We can now formalize the domain separation method by seeing it as defining a certain type of (translating) functor.

Let the ending space ES be an arity n function space. Let $\mathbf{F}\colon \mathsf{SS} \to \mathsf{ES}$ be a translating functor and QT,AT be its query and answer translations, respectively. Assume QT returns a vector of length 1 and that $\mathsf{AT}((i,X),\boldsymbol{V})$ simply returns $\boldsymbol{V}[1]$. We say that $\mathbf{F}$ is *domain separating* if the following is true: $\mathsf{QT}(i_1,X_1) \neq \mathsf{QT}(i_2,X_2)$ for any $(i_1,X_1),(i_2,X_2) \in \mathrm{Dom}(\mathsf{ES})$ that satisfy $i_1 \neq i_2$.

To explain, recall that the ending function is obtained as $e \leftarrow \mathbf{F}[s]$, and defines e_i for $i \in [1..n]$. Function e_i takes input X, lets $(u) \leftarrow \mathsf{QT}(i,X)$ and returns $s(u)$. The domain separation requirement is that if $(u_i) \leftarrow \mathsf{QT}(i,X_i)$ and $(u_j) \leftarrow \mathsf{QT}(j,X_j)$, then $i \neq j$ implies $u_i \neq u_j$, regardless of X_i,X_j. Thus if $i \neq j$ then the inputs to which s is applied are always different. The domain of s has been "separated" into disjoint subsets, one for each i.

Practical cloning functors. We show that many popular methods for oracle cloning in practice, including ones used in NIST KEM submissions, can be cast as translating functors.

In the following, the starting space $\mathsf{SS} = \mathsf{SOL}(\{0,1\}^*, \mathsf{OL}(\mathsf{SS}))$ is assumed to be a sol function space with domain $\{0,1\}^*$ and an output length denoted $\mathsf{OL}(\mathsf{SS})$. The ending space ES is an arity n function spaces that has sol subspaces.

Prefixing. Here we formalize the canonical method of domain separation. Prefixing is used in the following NIST PQC submissions: **`ClassicMcEliece`**, **`FrodoKEM`**, `LIMA`, **`NTRU Prime`**, `SIKE`, `QC-MDPC`, **`ThreeBears`**.

Let $\mathbf{p}$ be a vector of strings. We require that it be *prefix-free*, by which we mean that $i \neq j$ implies that $\mathbf{p}[i]$ is not a prefix of $\mathbf{p}[j]$. Entries of this vector will be used as prefixes to enforce domain separation. One example is that the entries of $\mathbf{p}$ are distinct strings all of the same length. Another is that a $\mathbf{p}[i] = \mathrm{E}(i)$ for some prefix-free code E like a Huffman code.

Assume $\mathsf{OL}_i(\mathsf{ES}) = \mathsf{OL}(\mathsf{SS})$ for all $i \in [1..n]$, meaning all ending functions have the same output length as the starting function. The functor $\mathbf{F}_{\mathrm{pf}(\mathbf{p})}\colon \mathsf{SS} \to \mathsf{ES}$ corresponding to $\mathbf{p}$ is defined by $\mathbf{F}_{\mathrm{pf}(\mathbf{p})}[s](i,X) = s(\mathbf{p}[i] \| X)$. To explain, recall that the ending function is obtained as $e \leftarrow \mathbf{F}_{\mathrm{pf}(\mathbf{p})}[s]$, and defines e_i for $i \in [1..n]$. Function e_i takes input X, prefixes $\mathbf{p}[i]$ to X to get a string X', applies the starting function s to X' to get Y, and returns Y as the value of $e_i(X)$.

We claim that $\mathbf{F}_{\mathrm{pf}(\mathbf{p})}$ is a translating functor that is also a domain-separating functor as per the definitions above. To see this, define query translator $\mathsf{QT}_{\mathrm{pf}(\mathbf{p})}$ by $\mathsf{QT}_{\mathrm{pf}(\mathbf{p})}(i,X) = (\mathbf{p}[i] \| X)$, the 1-vector whose sole entry is $\mathbf{p}[i] \| X$. The answer translator $\mathsf{AT}_{\mathrm{pf}(\mathbf{p})}$, on input $(i,X), \boldsymbol{V}$, returns $\boldsymbol{V}[1]$, meaning it ignores i,X and returns the sole entry in its 1-vector $\boldsymbol{V}$.

We proceed to the inverses, which are defined as follows:

```
Algorithm $\mathsf{QTI}_{\mathrm{pf}(\mathbf{p})}(U)$
$\boldsymbol{W} \leftarrow ()$
For $i = 1, \ldots, n$ do
    If $\mathbf{p}[i] \preceq U$ then $\mathbf{p}[i] \| X \leftarrow U$ ; $\boldsymbol{W}[1] \leftarrow (i,X)$
Return $\boldsymbol{W}$
```

```
Algorithm $\mathsf{ATI}_{\mathrm{pf}(\mathbf{p})}(U, \boldsymbol{Y})$
If $\boldsymbol{Y} \neq ()$ then $V \leftarrow \boldsymbol{Y}[1]$
Else $V \leftarrow 0^{\mathsf{OL}(\mathsf{SS})}$
Return $V$
```

The working domain is the full one: $\mathscr{W} = \mathrm{Dom}(\mathsf{ES})$. We now verify Equation (1.2). Let $\mathsf{QT}, \mathsf{QTI}, \mathsf{AT}, \mathsf{ATI}$ be $\mathsf{QT}_{\mathrm{pf}(\mathbf{p})}, \mathsf{QTI}_{\mathrm{pf}(\mathbf{p})}, \mathsf{AT}_{\mathrm{pf}(\mathbf{p})}, \mathsf{ATI}_{\mathrm{pf}(\mathbf{p})}$, respectively. Then for all $W = (i,X) \in$

$\mathsf{Dom}(\mathsf{ES})$, we have:

$$\begin{aligned}\mathbf{TF}_{\mathsf{QT},\mathsf{AT}}[\mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}](W) &= \mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}(\mathbf{p}[i]\|X)\\ &= \mathsf{ATI}(\mathbf{p}[i]\|X,(e(i,X)))\\ &= e(i,X)\ .\end{aligned}$$

We observe that $(\mathsf{QTI}_{\mathrm{pf}(\mathbf{p})},\mathsf{ATI}_{\mathrm{pf}(\mathbf{p})})$ provides perfect translation indistinguishability. Since $\mathsf{QTI}_{\mathrm{pf}(\mathbf{p})}$ does not have full support, we can't use Theorem 1, but we can conclude rd-indiff via Theorem 2.

Identity. Many NIST PQC submissions simply let $e_i(X) = s(X)$, meaning the ending functions are identical to the starting one. This is captured by the identity functor $\mathbf{F}_{\mathrm{id}}\colon \mathsf{SS} \to \mathsf{ES}$, defined by $\mathbf{F}_{\mathrm{id}}[s](i,X) = s(X)$. This again assumes $\mathsf{OL}_i(\mathsf{ES}) = \mathsf{OL}(\mathsf{SS})$ for all $i \in [1..n]$, meaning all ending functions have the same output length as the starting function. This functor is translating, via $\mathsf{QT}_{\mathrm{id}}(i,X) = X$ and $\mathsf{AT}_{\mathrm{id}}((i,X),\boldsymbol{V}) = \boldsymbol{V}[1]$. It is however *not*, at least in general, domain separating.

Clearly, this functor is not, in general, rd-indiff. To make secure use of it nonetheless, applications can restrict the inputs to the ending functions to enforce a virtual domain separation, meaning, for $i \neq j$, the schemes never query e_i and e_j on the same input. One way to do this is length differentiation. Here, for $i \in [1..n]$, the inputs to which e_i is applied all have the same length l_i, and $l_1,\ldots,l_n$ are distinct. Length differentiation is used in the following NIST PQC submissions: **BIKE**,EMBLEM, **HQC**, **RQC**, **LAC**, LOCKER, **NTS-KEM**, **SABER**, Round2, **Round5**,Titanium. There are, of course, many other similar ways to enforce the virtual domain separation.

There are two ways one might capture this with regard to security. One is to restrict the domain $\mathsf{Dom}(\mathsf{ES})$ of the ending space. For example, for length differentiation, we would require that there exist distinct $l_1,\ldots,l_n$ such that for all $(i,X) \in \mathsf{Dom}(\mathsf{ES})$ we have $|X| = l_i$. For such an ending space, the identity functor would provide security. The approach we take is different. We don't restrict the domain of the ending space, but instead define security with respect to a subdomain, which we called the working domain, where the restriction is captured. This, we believe, is better suited for practice, for a few reasons. One is that a single implementation of the ending functions can be used securely in different applications that each have their own working domain. Another is that implementations of the ending functions do not appear to enforce any restrictions, leaving it up to applications to figure out how to securely use the functions. In this context, highlighting the working domain may help application designers think about what is the working domain in their application and make this explicit, which can reduce error.

But we warn that the identity functor approach is more prone to misuse and in the end more dangerous and brittle than some others.

As per the above, inverses can only be given for certain working domains. Let us say that $\mathscr{W} \subseteq \mathsf{Dom}(\mathsf{ES})$ separates domains if for all $(i_1, X_1), (i_2, X_2) \in \mathscr{W}$ satisfying $i_1 \neq i_2$, we have $X_1 \neq X_2$. Put another way, for any $(i, X) \in \mathscr{W}$ there is at most one j such that $X \in \mathsf{Dom}_j(\mathsf{ES})$. We assume an efficient inverter for $\mathscr{W}$. This is a deterministic algorithm $\mathrm{In}_{\mathscr{W}}$ that on input $X \in \{0,1\}^*$ returns the unique i such that $(i, X) \in \mathscr{W}$ if such an i exists, and otherwise returns $\perp$. (The uniqueness is by the assumption that $\mathscr{W}$ separates domains.)

As an example, for length differentiation, we pick some *distinct* integers $l_1, \ldots, l_n$ such that $\{0,1\}^{l_i} \subseteq \mathsf{Dom}_i(\mathsf{ES})$ for all $i \in [1..n]$. We then let $\mathscr{W} = \{(i, X) \in \mathsf{Dom}(\mathsf{ES}) : |X| = l_i\}$. This separates domains. Now we can define $\mathrm{In}_{\mathscr{W}}(X)$ to return the unique i such that $|X| = l_i$ if $|X| \in \{l_1, \ldots, l_n\}$, otherwise returning $\perp$.

The inverses are then defined using $\mathrm{In}_{\mathscr{W}}$, as follows, where $U \in \mathsf{Dom}(\mathsf{SS}) = \{0,1\}^*$:

```
Algorithm $\mathsf{QTI}_{\mathsf{id}}(U)$
$\boldsymbol{W} \leftarrow ()$ ; $i \leftarrow \mathrm{In}_{\mathscr{W}}(U)$
If $i \neq \perp$ then $\boldsymbol{W}[1] \leftarrow (i, U)$
Return $\boldsymbol{W}$
```

```
Algorithm $\mathsf{ATI}_{\mathsf{id}}(U, \boldsymbol{Y})$
If $\boldsymbol{Y} \neq ()$ then $V \leftarrow \boldsymbol{Y}[1]$
Else $V \leftarrow 0^{\mathsf{OL}(\mathsf{SS})}$
Return $V$
```

The correctness condition of Equation (1.2) over $\mathscr{W}$ is met, and since $\mathrm{In}_{\mathscr{W}}(X)$ never returns $\perp$ for $X \in \mathscr{W}$, the second condition of invertibility is also met. $(\mathsf{QTI}_{\mathsf{id}}, \mathsf{ATI}_{\mathsf{id}})$ provides perfect translation indistinguishability. Since $\mathsf{QTI}_{\mathsf{id}}$ does not have full support, we can't use Theorem 1, but we can conclude rd-indiff via Theorem 2.

OUTPUT-SPLITTING. We formalize another method that we call output splitting. It is used in the following NIST PQC submissions: **FrodoKEM**, NTRU-HRSS-KEM, Odd Manhattan,QC-MDPC, Round2, **Round5**.

Let $\ell_i = \mathsf{OL}_1(\mathsf{ES}) + \cdots + \mathsf{OL}_i(\mathsf{ES})$ for $i \in [1..n]$. Let $\ell = \mathsf{OL}(\mathsf{SS})$ be the output length of

```
Adversary $\mathscr{A}^{\text{INIT,PUB,PRIV,FIN}}$
INIT()
$y \leftarrow$ PUB(0) ; $d \leftarrow\!\$\, \{1,2\}$ ; $y_d \leftarrow$ PRIV($d$,0)
If ($y_d[1..256]) = y[1..256]$ then FIN(1) else FIN(0)
```

Figure 1.9. Adversary against the rd-indiff security of $\mathbf{F}_{\texttt{NewHope}}$.

the sol functions $s \in \mathsf{SS}$, and assume $\ell = \ell_n$. The output-splitting functor $\mathbf{F}_{\text{spl}} \colon \mathsf{SS} \to \mathsf{ES}$ is defined by $\mathbf{F}_{\text{spl}}[s](i,X) = s(X)[\ell_{i-1}+1..\ell_i]$. That is, if $e \leftarrow \mathbf{F}_{\text{spl}}[s]$, then $e_i(X)$ lets $Z \leftarrow s(X)$ and then returns bits $\ell_{i-1}+1$ through ℓ_i of Z. This functor is translating, via $\mathsf{QT}_{\text{spl}}(i,X) = X$ and $\mathsf{AT}_{\text{spl}}((i,X), \boldsymbol{V}) = \boldsymbol{V}[1][\ell_{i-1}+1..\ell_i]$. It is however *not* domain separating.

The inverses are defined as follows, where $U \in \mathsf{Dom}(\mathsf{SS}) = \{0,1\}^*$:

```
Algorithm $\mathsf{QTI}_{\text{spl}}(U)$
For $i = 1,\ldots,n$ do $\boldsymbol{W}[i] \leftarrow (i,U)$
Return $\boldsymbol{W}$
```

```
Algorithm $\mathsf{ATI}_{\text{spl}}(U, \boldsymbol{Y})$
$V \leftarrow \boldsymbol{Y}[1] \| \cdots \| \boldsymbol{Y}[n]$
Return $V$
```

The correctness condition of Equation (1.2) over $\mathscr{W} = \mathsf{ES}$ is met, and $(\mathsf{QTI}_{\text{spl}}, \mathsf{ATI}_{\text{spl}})$ provides perfect translation indistinguishability. Since $\mathsf{QTI}_{\text{spl}}$ has full support, we can conclude rd-indiff via Theorem 1.

Rd-indiff of `NewHope`. We next demonstrate how read-only indifferentiability can highlight subpar methods of oracle cloning, using the example of `NewHope` [11]. The base KEM KE_1 defined in the specification of `NewHope` relies on just two random oracles, G and H_4. (The base scheme defined by transform $\mathbf{T}_{10}$, which uses 3 random oracles H_2, H_3, and H_4, is equivalent to KE_1 and can be obtained by applying the output-splitting cloning functor to instantiate H_2 and H_3 with G. `NewHope`'s security proof explicitly claims this equivalence [11].)

The final KEM KE_2 instantiates these two functions through $\mathsf{SHAKE256}$ without explicit domain separation, setting $H_4(X) = \mathsf{SHAKE256}(X,32)$ and $G_(X) = \mathsf{SHAKE256}(X,96)$. For consistency with our results, which focus on sol function spaces, we model $\mathsf{SHAKE256}$ as a random member of a sol function space SS with some very large output length L, and assume that the adversary does not request more than L bits of output from $\mathsf{SHAKE256}$ in a single call. We let ES be the arity-2 sol function space defining sub-functions G and H_4. In this setting, the cloning functor $\mathbf{F}_{\texttt{NewHope}} : \mathsf{SS} \to \mathsf{ES}$ used by `NewHope` is defined by $\mathbf{F}_{\texttt{NewHope}}[s](1,X) = s(X)[1..256]$

and $\mathbf{F}_{\texttt{NewHope}}[s](2,X) = s(X)[1..768]$. We will show that this functor cannot achieve rd-indiff for the given oracle spaces and the working domain $\mathscr{W} = \{0,1\}^*$. In Figure 1.9, we give an adversary $\mathscr{A}$ which has high advantage in the rd-indiff game $\mathbf{G}^{\text{rd-indiff}}_{\mathbf{F}_{\texttt{NewHope}},\mathsf{SS},\mathsf{ES},\mathscr{W},\mathsf{Sim}}$ for any indifferentiability simulator Sim. When $b = 1$ in game $\mathbf{G}^{\text{rd-indiff}}_{\mathbf{F}_{\texttt{NewHope}},\mathsf{SS},\mathsf{ES},\mathscr{W},\mathsf{Sim}}$, we have that

$$y_d[1..256] = \mathbf{F}_{\texttt{NewHope}}[s](d,0)[1..256] = s(0)[1..256] = y[1..256],$$

so adversary $\mathscr{A}$ will always call FIN on the bit 1 and win. When $b = 0$ in game $\mathbf{G}^{\text{rd-indiff}}_{\mathbf{F}_{\texttt{NewHope}},\mathsf{SS},\mathsf{ES},\mathscr{W},\mathsf{Sim}}$, the two strings $y_1 = e_0(1,X)$ and $y_2 = e_0(2,X)$ will have different 256-bit prefixes, except with probability $\varepsilon = 2^{-256}$. Therefore, when $\mathscr{A}$ queries PUB(0), the simulator's response y can share the prefix of most one of the two strings y_1 and y_2. Its response must be independent of d, which is not chosen until after the query to PUB, so $\Pr[y[1..256] = y_d[1..256]] \leq 1/2 + \varepsilon$, regardless of the behavior of Sim. Hence, $\mathscr{A}$ breaks the indifferentiability of $\mathbf{Q}^{\texttt{NewHope}}$ with probability roughly $1/2$, rendering **`NewHope`**'s random oracle functor differentiable.

The implication of this result is that **`NewHope`**'s implementation differs noticeably from the model in which its security claims are set, even when $\mathsf{SHAKE256}$ is assumed to be a random oracle. This admits the possibility of hash function collisions and other sources of vulnerability that are not eliminated by the security proof. To claim provable security for **`NewHope`**'s implementation, further justification is required to argue that these potential collisions are rare or unexploitable. We do not claim that an attack on read-only indifferentiability implies an attack on the IND-CCA security of **`NewHope`**, but it does highlight a gap that needs to be addressed. Read-only indifferentiability constitutes a useful tool for detecting such gaps and measuring the strength of various oracle cloning methods.

1.6 Oracle Cloning in KEMs

Having shown rd-indiff of various practical cloning functors, we'd like to come back around and apply this to show IND-CCA security of KEMs (as the target primitive of the NIST PQC submissions) that use these functors. At one level, this may seem straightforward and unnecessary, for it is a special case of a general indifferentiability composition theorem, which says that once indifferentiability of a functor has been shown, "all" uses of it are secure. In particular, the composition theorems of [156, 187] for MRH-indefferentiability apply also to rd-indiff and guarantee security when the latter is measured via a single-stage game, which is true for IND-CCA KEMs. This, however, fails to account for working domains, which are not present in prior indifferentiability formulations; the existing composition results only guarantee security when the working domain is the full domain of the ending space. But this fails to be the case for some oracle cloning methods like length differentiation that are used in NIST PQC KEMs. We want a composition theorem that can allow us to conclude security of such usages.

For this, we first must ask what is the meaning or definition of the working domain in the context of the application, here IND-CCA KEMs. Below, we define this. Then we give a working-domain-conscious composition theorem for IND-CCA KEMs that allows us to draw the conclusions mentioned above. The starting point for this treatment is to enhance the syntax of KEMs to allow them to say precisely what types of ROs they want and use.

KEM SYNTAX. In the formal version of the ROM in [39], there is a single random oracle that has some fixed domain and range, for example mapping $\{0,1\}^*$ to $\{0,1\}$. Schemes, however, often want multiple random oracles, and also want their oracles to have particular domains and ranges that depend on the scheme. To capture this, we have the scheme syntax include a specification of the desired function space from which the random oracle is then drawn by games defining security. We suggest that schemes specified in standards include a specification of this space, to avoid errors.

Formally, a key-encapsulation mechanism (KEM) KE specifies the following. First is a function space $\mathsf{KE.FS}$. Now as usual there is a key-generation algorithm $\mathsf{KE.K}$ that, given access to an oracle $H \in \mathsf{KE.FS}$, returns a public encryption key and matching secret decryption key, $(pk, dk) \leftarrow\!\!\$\, \mathsf{KE.K}[H]$. Next there is an encapsulation algorithm $\mathsf{KE.E}$ that, given input pk, and given oracle H, returns a symmetric key $K \in \{0,1\}^{\mathsf{KE.kl}}$ and a ciphertext C encapsulating it, $(C, K) \leftarrow\!\!\$\, \mathsf{KE.E}[H](pk)$, where $\mathsf{KE.kl}$ is the symmetric-key length of KE. The randomness length of $\mathsf{KE.E}$ is denoted $\mathsf{KE.rl}$. Finally, there is a deterministic decapsulation algorithm $\mathsf{KE.D}$ that, given inputs dk, C, and given oracle H, returns $\mathsf{KE.D}[H](dk, C) \in \{0,1\}^{\mathsf{KE.kl}} \cup \{\bot\}$.

```
Game $\mathbf{G}^{\mathrm{ind-cca}}_{\mathsf{KE}}$

INIT:
  $H \leftarrow\$ \mathsf{KE.FS}$ ; $b \leftarrow\$ \{0,1\}$
  $(pk,dk) \leftarrow\$ \mathsf{KE.K}[\mathrm{RO}]$
  $(C^*,K_1^*) \leftarrow\$ \mathsf{KE.E}[\mathrm{RO}](pk)$
  $K_0^* \leftarrow\$ \{0,1\}^{\mathsf{KE.kl}}$
  return $pk, C^*, K_b^*$

DEC($C$):
  If $(C = C^*)$ then return $\bot$
  $K \leftarrow \mathsf{KE.D}[\mathrm{RO}](dk,C)$
  return $K$

RO($W$):
  return $H(W)$

FIN($b'$):
  return $(b = b')$
```

Figure 1.10. KEM security game for indistinguishability under chosen-ciphertext attacks.

Security definitions. We cast the standard security notion of indistinguishability under chosen-ciphertext attack (IND-CCA) for KEMs [76] in our extended syntax in Figure 1.10. Adversary $\mathscr{A}$ gets a challenge ciphertext C^* and a challenge key K_b^* that is either the key K_1^* underlying C^* or a random key K_0^*, and, to win, must determine b. Decapsulation oracle Dec allows it to decapsulate any non-challenge ciphertext of its choice. We let

$$\mathbf{Adv}^{\mathrm{ind-cca}}_{\mathsf{KE}}(\mathscr{A}) = 2\Pr[\mathbf{G}^{\mathrm{ind-cca}}_{\mathsf{KE}}] - 1$$

to be the ind-cca advantage of adversary $\mathscr{A}$.

Working domain of a KEM. Let KE be a KEM. Let $\mathscr{W} \subseteq \mathsf{Dom}(\mathsf{KE.FS})$ be a subset of $\mathsf{Dom}(\mathsf{KE.FS})$. Consider game $\mathbf{G}^{\mathrm{wdom}}_{\mathsf{KE},\mathscr{W}}$ in Figure 1.11. The intent is that, at the end of the game, the set $\mathscr{U}$ contains all queries made to RO by the scheme algorithms, while excluding ones made by the adversary $\mathscr{A}$ but not by scheme algorithms. Boolean flag sq controls when a query W to RO is to be put in $\mathscr{U}$ in accordance with this policy. (We do assume all queries to RO are in $\mathsf{Dom}(\mathsf{KE.FS})$.) The adversary wins if it can make the scheme algorithms query a point outside the working domain. Its wdom-advantage is $\mathbf{Adv}^{\mathrm{wdom}}_{\mathsf{KE},\mathscr{W}}(\mathscr{A}) = \Pr[\mathbf{G}^{\mathrm{wdom}}_{\mathsf{KE},\mathscr{W}}(\mathscr{A})]$. We say that $\mathscr{W}$ *is a working domain of* KE if $\mathbf{Adv}^{\mathrm{wdom}}_{\mathsf{KE},\mathscr{W}}(\mathscr{A}) = 0$ for all adversaries $\mathscr{A}$, regardless of the running time and number of oracle queries of $\mathscr{A}$.

The set $\mathsf{Dom}(\mathsf{KE.FS})$ is always a working domain of KE. The interesting case is when one can specify a subset of it that is a working domain.

Composition. Let KE be a given KEM that we assume is IND-CCA secure. Let $\mathbf{F}\colon \mathsf{SS} \to \mathsf{KE.FS}$ be a functor. We associate to them the KEM $\overline{\mathsf{KE}} = \mathbf{F}(\mathsf{KE})$ that is defined as follows. Its function space is $\overline{\mathsf{KE}}.\mathsf{FS} = \mathsf{SS}$, the starting space of the functor. The algorithms of $\overline{\mathsf{KE}}$, given an oracle for

```
Game $\mathbf{G}^{\mathrm{wdom}}_{\mathsf{KE},\mathscr{W}}$

INIT:
  $H \leftarrow\!\!\$\ \mathsf{KE.FS}$ ; $\mathsf{sq} \leftarrow \mathsf{true}$
  $(pk,dk) \leftarrow\!\!\$\ \mathsf{KE.K}[\mathrm{RO}]$ ; $(C,K) \leftarrow\!\!\$\ \mathsf{KE.E}[\mathrm{RO}](pk)$
  $\mathsf{sq} \leftarrow \mathsf{false}$ ; Return $pk,C,K$

DEC($C$):
  $\mathsf{sq} \leftarrow \mathsf{true}$ ; $K \leftarrow \mathsf{KE.D}[\mathrm{RO}](dk,C)$ ; $\mathsf{sq} \leftarrow \mathsf{false}$ ; Return $K$

RO($W$):
  If $\mathsf{sq}$ then $\mathscr{U} \leftarrow \mathscr{U} \cup \{W\}$
  return $H(W)$

FIN:
  return $(\mathscr{U} \not\subseteq \mathscr{W})$
```

Figure 1.11. Game to determine the working domain $\mathscr{W}$ of a KEM KE.

s, run the corresponding algorithm of KE with oracle $e = \mathbf{F}[s]$. Let $\mathscr{W}$ be a working domain for KE and assume $\mathbf{F}$ is rd-indiff over $\mathscr{W}$. Then Theorem 3, below, says that $\overline{\mathsf{KE}}$ is IND-CCA as well.

The application to NIST PQC KEMs is as follows. Let KE be a base KEM from one of the submissions, as discussed in Section 1.2, so that $\mathsf{KE.FS}$ is an arity-4 function space. We know (or are willing to assume) that KE is IND-CCA. Now, we want to instantiate the four oracles of KE by a single one, say drawn from the sol function space $\mathsf{SS} = \mathsf{SOL}(\{0,1\}^*, \ell)$ for some given value of ℓ like $\ell = 256$. We pick a cloning functor $\mathbf{F}\colon \mathsf{SS} \to \mathsf{KE.FS}$ that determines a function for the base KEM from one of the given functions. The example of interest is that this is the identity cloning functor, which is not rd-indiff over its full domain. Instantiating the oracles of KE, via the functor applied to an oracle of the starting space, yields the KEM $\overline{\mathsf{KE}}$. This is what, in Section 1.2, we called the final KEM, and the question is whether it is IND-CCA. Employing length differentiation corresponds to the base KEM having the corresponding working domain. From Section 1.5 we know that the identify functor is rd-indiff over this working domain. Now Theorem 3 says that the final KEM is IND-CCA.

Theorem 3. *Let* KE *be a KEM. Let* $\mathbf{F}\colon \mathsf{SS} \to \mathsf{KE.FS}$ *be a functor. Let* $\overline{\mathsf{KE}} = \mathbf{F}(\mathsf{KE})$ *be the KEM associated to them as above. Let* $\mathscr{W}$ *be a working domain for* KE*, and let* Sim *be a read-only simulator for* $\mathbf{F}$*. Let* $\mathscr{A}$ *be an ind-cca adversary. Then we construct adversaries* $\mathscr{B}$*, and* $\mathscr{D}$ *such*

```
Games $\mathrm{G}_0, \mathrm{G}_1$

INIT:
  $s \leftarrow\!\!\$\, \mathsf{SS}$ ; $e \leftarrow \mathbf{F}[s]$   // Game $\mathrm{G}_0$
  $st \leftarrow\!\!\$\, \mathsf{Sim.Setup}()$ ; $e \leftarrow\!\!\$\, \mathsf{KE.FS}$   // Game $\mathrm{G}_1$
  $b \leftarrow\!\!\$\, \{0,1\}$
  $(pk, dk) \leftarrow\!\!\$\, \mathsf{KE.K}[e]$
  $(C^*, K_1^*) \leftarrow\!\!\$\, \mathsf{KE.E}[e](pk)$
  $K_0^* \leftarrow\!\!\$\, \{0,1\}^{\mathsf{KE.kl}}$
  return $pk, C^*, K_b^*$

DEC($C$):
  If $(C = C^*)$ then return $\bot$
  $K \leftarrow \mathsf{KE.D}[e](dk, C)$
  return $K$

RO($U$):
  return $s(U)$   // Game $\mathrm{G}_0$
  return $\mathsf{Sim.Ev}[e](st, U)$   // Game $\mathrm{G}_1$

FIN($b'$):
  return $(b = b')$
```

```
Games $\mathrm{G}_2$, $\mathrm{G}_3$

INIT:
  $s \leftarrow\!\!\$\, \mathsf{SS}$ ; $e_1 \leftarrow \mathbf{F}[s]$
  $st \leftarrow\!\!\$\, \mathsf{Sim.Setup}()$ ; $e_0 \leftarrow\!\!\$\, \mathsf{KE.FS}$
  $c \leftarrow\!\!\$\, \{0,1\}$

PRIV($W$):
  If $W \notin \mathscr{W}$ then
    bad $\leftarrow$ true
    return $\bot$   // Game $\mathrm{G}_3$
  return $e_c(W)$

PUB($U$):
  if $(c = 1)$ then return $s(U)$
  else return $\mathsf{Sim.Ev}[e_0](st, U)$

FIN($c'$):
  return $(c = c')$
```

Figure 1.12. Games for the proof of Theorem 3.

that

$$\mathbf{Adv}^{\text{ind-cca}}_{\mathsf{KE}}(\mathscr{A}) \le \mathbf{Adv}^{\text{ind-cca}}_{\mathsf{KE}}(\mathscr{B}) + 2 \cdot \mathbf{Adv}^{\text{rd-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{KE.FS},\mathscr{W},\mathsf{Sim}}(\mathscr{D}) \,.$$

The running time of $\mathscr{D}$ is about that of $\mathscr{A}$. If $\mathscr{A}$ makes q queries to RO*, then the running time of $\mathscr{B}$ is about that of $\mathscr{A}$ plus q times the running time of* Sim.

Proof: Consider the games in Figure 1.12. We have

$$\begin{aligned}\mathbf{Adv}^{\text{ind-cca}}_{\mathsf{KE}}(\mathscr{A}) &= 2\Pr[\mathrm{G}_0(\mathscr{A})] - 1 \\ &= 2\Pr[\mathrm{G}_1(\mathscr{A})] - 1 + 2(\Pr[\mathrm{G}_0(\mathscr{A})] - \Pr[\mathrm{G}_1(\mathscr{A})]) \,.\end{aligned}$$

Let adversary $\mathscr{B}$ be as shown in Figure 1.13. Then

$$2\Pr[\mathrm{G}_1(\mathscr{A})] - 1 \le \mathbf{Adv}^{\text{ind-cca}}_{\mathsf{KE}}(\mathscr{B}) \,.$$

```
Adversary $\mathscr{D}$:
    $\mathscr{A}^{\text{INIT}',\text{DEC}',\text{RO}',\text{FIN}'}()$
INIT':
    $b \leftarrow\!\!\$ \{0,1\}$
    $(pk, dk) \leftarrow\!\!\$ \mathsf{KE.K}[\text{PRIV}]$
    $(C^*, K_1^*) \leftarrow\!\!\$ \mathsf{KE.E}[\text{PRIV}](pk)$
    $K_0^* \leftarrow\!\!\$ \{0,1\}^{\mathsf{KE.kl}}$
    return $pk, C^*, K_b^*$
DEC'(C):
    If $(C = C^*)$ then return $\perp$
    $K \leftarrow \mathsf{KE.D}[\text{PRIV}](dk, C)$
    return $K$
RO'(U):
    return $\text{PUB}(U)$
FIN'(b'):
    if $(b = b')$ then $\text{FIN}(1)$
    else $\text{FIN}(0)$
```

```
Adversary $\mathscr{B}$:
    $st \leftarrow\!\!\$ \mathsf{Sim.Setup}()$
    $\mathscr{A}^{\text{INIT}',\text{DEC}',\text{RO}',\text{FIN}'}()$
INIT':
    $(pk, C^*, K_b^*) \leftarrow \text{INIT}()$
    return $pk, C^*, K_b^*$
DEC'(C):
    return $\text{DEC}(C)$
RO'(W):
    return $\mathsf{Sim.Ev}[\text{RO}](st, W)$
FIN'(b'):
    $\text{FIN}(b')$
```

Figure 1.13. Adversaries for the proof of Theorem 3.

Game G_3 is game $\mathbf{G}^{\text{rd-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{KE.FS},\mathscr{W},\mathsf{Sim}}$. Game G_2 drops the working domain check at line 4. Let adversary $\mathscr{D}$ be as shown in Figure 1.13. Then

$$\Pr[\text{G}_0(\mathscr{A})] - \Pr[\text{G}_1(\mathscr{A})] \leq 2\Pr[\text{G}_2(\mathscr{D})] - 1 \,.$$

Games G_2, G_3 are identical-until-bad so by the Fundamental Lemma of Game Playing [42] we have

$$\begin{aligned} 2\Pr[\text{G}_2(\mathscr{D})] - 1 &= 2\Pr[\text{G}_3(\mathscr{D})] - 1 + 2(\Pr[\text{G}_2(\mathscr{D})] - \Pr[\text{G}_3(\mathscr{D})]) \\ &\leq 2\Pr[\text{G}_3(\mathscr{D})] - 1 + 2\Pr[\text{G}_2(\mathscr{D}) \text{ sets } \mathsf{bad}] \,. \end{aligned}$$

Now we have

$$2\Pr[\text{G}_3(\mathscr{D})] - 1 = \mathbf{Adv}^{\text{rd-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{KE.FS},\mathscr{W},\mathsf{Sim}}(\mathscr{D}) \,.$$

Adversary $\mathscr{D}$ invokes its PRIV oracle only on points queried by scheme algorithms, and, regardless of the challenge bit c, the function underlying PRIV is a member of $\mathsf{KE.FS}$. Because $\mathscr{W}$ is a working domain for KE, we have

$$\Pr[\mathrm{G}_2(\mathscr{D}) \text{ sets } \mathsf{bad}] = 0 \, .$$

This concludes the proof. ▌

Acknowledgments

We thank Dan Bernstein for comments and corrections. We thank the Eurocrypt 2020 reviewers for their comments.

Chapter 2

Tighter Bounds for the TLS 1.3 and SIGMA Key Exchange Protocols

2.1 Introduction

The Transport Layer Security (TLS) protocol [186] is responsible for securing billions of Internet connections every day. Usage statistics for Google Chrome[1] and Mozilla Firefox[2] report that 76–98% of all web page accesses are encrypted.At the heart of TLS is an authenticated key exchange (AKE) protocol, the so-called handshake protocol, responsible for providing the parties (client and server) with a shared, symmetric key that is fresh, private and authenticated. The ensuing record layer secures data using this key. The AKE protocol of TLS is based on the SIGMA ("SIGn-and-MAc") design of Krawczyk [138] for the Internet Key Exchange (IKE) protocol [117] of IPsec [136], which generically augments an unauthenticated, ephemeral Diffie–Hellman (DH) key exchange with authenticating signatures and MACs.

Naturally, the SIGMA AKE protocol and its incarnation in TLS have been the recipients of proofs of security. We contend that these largely justify the AKE protocols in principle, but not in practice, meaning not for the parameters in actual use and at the desired or expected level of security. Our work takes steps towards filling this gap.

2.1.1 Qualitative and Quantitative Bounds

Let us expand on this. The protocols KE we consider are built from a cyclic group $\mathbb{G}$ in which some DH problem P is assumed to be hard, a pseudorandom function PRF and unforgeable

[1] https://transparencyreport.google.com/https/

[2] https://telemetry.mozilla.org/

signature and MAC schemes S and M. The target for KE is session-key security with explicit authentication as originating from [40, 37, 67]. A proof of security has both a qualitative and quantitative dimension. Qualitatively, a proof of security for the AKE protocol KE says that KE meets its target definition assuming the building blocks meet theirs, where, in either case, meeting the definition means any poly-time adversary has negligible advantage in violating it.

The quantitative dimension associates to each adversary in the security game of KE a set of resources r, representing its runtime and attack surface (e.g., the number of users and executed protocol sessions the adversary has access to). It then relates the maximum advantage of any r-resource adversary in breaking KE's security to likewise advantage functions for the building blocks through an equation of the (simplified) form

$$\mathbf{Adv}_{\mathsf{KE}}(r) \leq f_{\mathbb{G}} \cdot \mathbf{Adv}_{\mathbb{G}}^{\mathsf{P}}(r_{\mathbb{G}}) + f_{\mathsf{S}} \cdot \mathbf{Adv}_{\mathsf{S}}^{\mathsf{EUF\text{-}CMA}}(r_{\mathsf{S}}) + \ldots,$$

deriving quantitative factors f_{X} and resources r_{X} for the advantage of each building block X.

Speaking asymptotically again, when f_{X} and r_{X} are polynomial functions in r, then $\mathbf{Adv}_{\mathsf{KE}}(r)$ is negligible whenever all building blocks' advantages are. Due to the complexity of key exchange models and the challenging task of combining the right components in a secure manner, key exchange analyses (including prior work on SIGMA [68] and TLS 1.3 [93, 147, 95, 101, 92]) indeed often remain abstract and consider only qualitative, asymptotic security bounds.

Standardized protocols like TLS in contrast have to define concrete choices for each cryptographic building block. This involves considering reasonable estimates for adversarial resources (like runtime t and number of key-exchange model queries q) and specific instances and parameters for the underlying components X. One would hope that key exchange proofs can provide guidance in making sound choices that result in the desired overall security level. Unfortunately, AKE security bounds regularly are highly non-tight, meaning that f_{X} and/or r_{X} for some components X are so large that reasonable stand-alone parameters for X yield vacuous key exchange advantages for practical parameters. While the asymptotic bound tells us that scaling up the parameters for X (say, the DDH problem [57]) will at some point result in a secure overall advantage, this causes efficiency concerns (e.g., doubling elliptic curve DH security

Adv. resources					SIGMA		TLS 1.3	
t	$\#U$	$\#S$	Curve	Target	CK [68]	Us (Thm. 5)	DFGS [92]	Us (Thm. 6)
2^{60}	2^{20}	2^{35}	`secp256r1`	2^{-68}	$\approx 2^{-61}$	$\approx 2^{-116}$	$\approx 2^{-64}$	$\approx 2^{-116}$
2^{60}	2^{30}	2^{55}	`secp256r1`	2^{-68}	$\approx 2^{-21}$	$\approx 2^{-106}$	$\approx 2^{-24}$	$\approx 2^{-106}$
2^{60}	2^{20}	2^{35}	`x25519`	2^{-68}	$\approx 2^{-57}$	$\approx 2^{-112}$	$\approx 2^{-60}$	$\approx 2^{-112}$
2^{60}	2^{30}	2^{55}	`x25519`	2^{-68}	$\approx 2^{-17}$	$\approx 2^{-102}$	$\approx 2^{-20}$	$\approx 2^{-102}$
2^{80}	2^{20}	2^{35}	`secp256r1`	2^{-48}	$\approx 2^{-21}$	$\approx 2^{-76}$	$\approx 2^{-24}$	$\approx 2^{-76}$
2^{80}	2^{30}	2^{55}	`secp256r1`	2^{-48}	1	$\approx 2^{-66}$	1	$\approx 2^{-66}$
2^{80}	2^{20}	2^{35}	`x25519`	2^{-48}	$\approx 2^{-17}$	$\approx 2^{-72}$	$\approx 2^{-20}$	$\approx 2^{-72}$
2^{80}	2^{30}	2^{55}	`x25519`	2^{-48}	1	$\approx 2^{-62}$	1	$\approx 2^{-62}$
2^{80}	2^{20}	2^{35}	`secp384r1`	2^{-112}	$\approx 2^{-149}$	$\approx 2^{-204}$	$\approx 2^{-152}$	$\approx 2^{-204}$
2^{80}	2^{30}	2^{55}	`secp384r1`	2^{-112}	$\approx 2^{-109}$	$\approx 2^{-194}$	$\approx 2^{-112}$	$\approx 2^{-194}$

Table 2.1. Exemplary concrete advantages of a key exchange adversary with given resources t (running time), $\#U$ (number of users), $\#S$ (number of sessions), in breaking the security of the SIGMA and TLS 1.3 protocols when instantiated with curve `secp256r1`, `secp384r1`, or `x25519`, based on the prior bounds by Canetti-Krawczyk [68] resp. Dowling et al. [92], and the bounds we establish (Theorem 5 and 6). Target indicates the maximal advantage $t/2^b$ tolerable when aiming for the respective curve's security level ($b = 128$ resp. 192 bits); entries in red-shaded cells miss that target. See Section 3.7 for full details and curves `secp521r1` and `x448`.

parameters means quadrupling the cost for group operations) and hence does not happen in practice.

We illustrate in Table 2.1 the effects of the non-tight bounds for SIGMA and TLS 1.3 when instantiating the protocols with NIST curves `secp256r1`, `secp384r1` [172], or curve `x25519` [150] and idealizing the protocols' other components (see Section 3.7 for full details). Following the curves' security, we aim at a security level of 128 bits, resp. 192 bits, meaning the ratio of an adversary's runtime to its advantage should be bounded by 2^{-128}, resp. 2^{-192}. When considering the advantage of key exchange adversaries running in time t, interacting in the security game with $\#U$ users and $\#S$ sessions, we can see that previous security bounds fail to meet the targeted security level for real-world–scale parameters ($\#U$ ranging in 2^{20}–2^{30} based on 2^{27} active certificates on the Internet[3], $\#S$ ranging in 2^{35}–2^{55} based on 2^{32} Internet users and 2^{33} daily Google searches[4]). In the security analysis by Canetti and Krawczyk [68] (CK) for SIGMA, the factor associated to the decisional Diffie–Hellman problem is $f_{\mathsf{DDH}}(t, \#U, \#S) = \#U \cdot \#S$, where $\#U$ and $\#S$ again are the number of users, resp. sessions, accessible by the adversary. The analysis by Dowling et al. [92] (DFGS) for TLS 1.3 reduces to the strong Diffie–Hellman problem [4]—via

[3] https://letsencrypt.org/stats/

[4] https://www.internetlivestats.com/

the PRF-ODH assumption [127, 63]—with factor $f_{\mathsf{stDH}}(t, \#U, \#S) = (\#S)^2$. In contrast, we reduce to the strong Diffie–Hellman problem with a constant factor for both SIGMA and TLS 1.3.

Let us discuss three data points from Table 2.1:

1. Already with medium-sized resources, investing time $t = 2^{60}$ and interacting with a million users ($\#U = 2^{20}$) and a few billion sessions ($\#S = 2^{35}$), the CK [68] and DFGS [92] advantage bounds for SIGMA and TLS 1.3 with curves `secp256r1` and `x25519` fall 6–11 bits below the target of 2^{-68} for 128-bit security.

2. When considering a more powerful, global-scale adversary ($t = 2^{80}$, $\#U = 2^{30}$, $\#S = 2^{55}$), both CK and DFGS bounds for `secp256r1`/`x25519` become fully vacuous; the upper bound on the probability of the adversary breaking the protocol is 1. We stress that `secp256r1` is the mandatory-to-implement curve for TLS 1.3; `secp256r1` and `x25519` together make up for 90% of the TLS 1.3 ECDHE handshakes reported through Firefox Telemetry.

3. Finally, and notably, even switching to the higher-security curve `secp384r1` helps only marginally in the latter case: the resulting advantage against SIGMA falls 3 bits short of the 192-bit security target of 2^{-112}, and the TLS advantage bound only barely meets that target.

For all curves and choices of parameters, our bounds do better.

2.1.2 Contributions

Most prior results in tightly secure key exchange (e.g., [21, 108]) apply only to bespoke protocols, carefully designed to allow for tighter proof techniques, at the cost of requiring more complex primitives which, in the end, eat up the gained practical efficiency. Recently, Cohn-Gordon et al. [73, ?] established a proof strategy for a simple and efficient DH key exchange with reasonable tightness loss (only linear in the number of users $\#U$), achieving implicit authentication through static DH keys through careful key derivation via a random oracle [39] with an optional explicit-authentication step.

Our work in contrast establishes tight security for standardized AKE protocols. We give tight reductions for the security of SIGMA and TLS 1.3 to the strong Diffie–Hellman problem [4], which in addition we prove is as hard as the discrete logarithm problem in the generic group model (GGM) [195, 155]. Instantiating our bounds shows that, with standardized real-world parameters, we achieve the intended security levels even when considering powerful, globally-scaled attackers.

Code-based security model and proofs.

For our proofs, we provide detailed proof steps and reductions using the code-based game-playing framework of Bellare and Rogaway [42]. Our security model is similar to the one applied by Cohn-Gordon et al. [73], but formalized also as a code-based game (in Section 3.3) and stronger in that it captures explicit authentication and regular ("perfect") forward secrecy (instead of only weak forward secrecy in [73]).

Tighter security proof of SIGMA(-I).

We establish fully quantitative security bounds for SIGMA and its identity-protecting variant SIGMA-I [138] in Sections 2.5 and 2.6. Our result is for BR-like [40] key exchange security and gives a tight reduction to the strong Diffie–Hellman problem [4] in the used DH group, and to the multi-user (mu) security of the employed pseudorandom function (PRF), signature scheme, and MAC scheme, adapting the techniques by Cohn-Gordon et al. [73] in the random oracle model [39]. The latter mu-security bounds are essentially equivalent to the corresponding bounds by CK [68]. Our improvement comes from shaving off a factor of $\#U \cdot \#S$ (number of users times number of sessions) on the DH problem advantage compared to CK. While we move to the interactive strong Diffie–Hellman problem (compared to the decisional DH (DDH) problem [57] used in [68]), we prove (in Appendix 2.4) that the strong DH problem, like DDH, is as hard as solving discrete logarithms in the generic group model [195, 155], reflecting the (only generic) algorithms known for solving discrete logarithms in elliptic curve groups.

Tighter security proof for the TLS 1.3 DH handshake.

We likewise establish fully quantitative security bounds for the key exchange of the recently standardized newest version of the Transport Layer Security protocol, TLS 1.3 [186], in Sections 2.7 and 2.8. The main quantitative improvement in our reduction is again a tight reduction to the strong DH problem, whereas prior bounds by DFGS [92] incurred a quadratic loss to the PRF-ODH assumption [127, 63], a loss which translates directly to strong DH [63].

While TLS 1.3 roughly follows the SIGMA-I design, its cascading key schedule impedes the precise technique of Cohn-Gordon et al. [73] and a direct application of our results on SIGMA-I, as no single function (to be modeled as a random oracle) binds the Diffie–Hellman values to the session context. We therefore have to carefully adapt the proof to accommodate the more complex key schedule and other core variations in TLS 1.3's key exchange, achieving conceptually similar tightness results as for SIGMA-I.

Evaluation.

In Section 3.7, we evaluate the concrete security implications of our improved bounds for SIGMA and TLS 1.3 for a wide range of real-world resource parameters and all five elliptic curves (`secp256r1`, `secp384r1`, `secp521r1`,`x25519`, `x448`) standardized for use in TLS 1.3 [186], a summary of which is displayed in Table 2.1. Leveraging our GGM bound for the strong Diffie–Hellman problem, we focus on the hardness of solving discrete logarithms in the respective elliptic curve groups, instantiating signatures based on ECDSA [172] resp. EdDSA [48]. We idealized the symmetric PRF, MAC, and hash function primitives (in two variants, with key and output sizes twice as large as the curve's security level, or fixed at 256 bits corresponding to the choice in most TLS 1.3 cipher suites).

We report that our tighter proofs indeed materialize for a wide range of real-world resource parameters (adversary runtime $t \in \{2^{40}, 2^{60}, 2^{80}\}$, number of users $\#U \in \{2^{20}, 2^{30}\}$, and number of sessions $\#S \in \{2^{35}, 2^{45}, 2^{55}\}$). The resulting attacker advantages meet the targeted security levels of all five curves. In comparison to the prior CK [68] SIGMA and DFGS [92] TLS 1.3 bounds, our results improve the obtained security across these real-world parameters by up to 85 bits for SIGMA and 92 bits for TLS 1.3, respectively.

2.1.3 Optimizations, Limitations, and Possible Extensions

SIGMA being a generic AKE design, the signature, PRF, and MAC schemes may be instantiated with primitives optimized for multi-user security. While we focus on standardized and deployed schemes in our evaluation without assuming tight mu-security, our SIGMA bound allows to directly leverage such optimization. For PRFs and MACs, efficient candidates exist (e.g., AMAC [29]). For signatures, tight mu-security is more challenging [22] and often involves computationally much more expensive constructions [21].

Like Cohn-Gordon et al. [73], our key exchange security model considers exposure of long-term secrets and session keys, but does not allow revealing internal session state or randomness (as in the (e)CK model [67, 149]). This is appropriate for protocols like TLS 1.3 not aiming to protect against such threats. The original SIGMA proof [68] did establish security in the CK model [67] allowing exposure of session state; in that sense our results are qualitatively weaker. In recent work, Jager et al. [126] give a tightly secure protocol which uses symmetric state encryption to protect against ephemeral state reveals. Establishing a tight security reduction for a SIGMA-style DH-based AKE protocol which can handle adaptive compromises of session state (including DH exponents) remains a challenging open problem.

In our proofs, we crucially rely on the ability to observe and program a random oracle used for key derivation in the AKE protocol, borrowing from [73]. Notably, the approach of Cohn-Gordon et al. is tailored to an AKE protocol achieving authenticity implicitly through mixing long-term DH keys into the key derivation. Our proofs can hence be seen as translating and adapting their technique to the setting of SIGMA and TLS 1.3, where an unauthenticated ephemeral DH exchange is explicitly authenticated through signatures and MACs, confirming that the generic SIGMA design as well as the standardized TLS 1.3 protocol bind enough context to their DH shares for this proof technique to work. Leveraging the random oracle model [39] is another qualitative difference compared to the original SIGMA proof [68] in the standard model. Interestingly, this distinction vanishes in comparison to the provable security results for the TLS 1.3 handshake protocol [93, 95, 101, 92] which employ the PRF-ODH assumption [127, 63], an interactive assumption which plausibly can only be instantiated in the random oracle model (from the strong DH assumption).

2.1.4 Concurrent Work

In concurrent and independent work, Diemert and Jager (DJ) [88] studied the tight security of the main TLS 1.3 handshake. Their work also tightly reduces the security of TLS 1.3 to the strong Diffie–Hellman problem by extending the technique of Cohn-Gordon et al. [73], and their bounds and ours are similarly tight. When instantiated with real-world parameters, both bounds are dominated by the same terms, as we will demonstrate in Section 3.7. Our proof differs from theirs in two key ways: We use an incomparable security model that is weaker in some ways and stronger in others, and we approximate the TLS 1.3 key schedule with fewer random oracles. We also contextualize our results quite differently than the DJ work, with a detailed numerical analysis that is enabled by our fully parameterized, concrete bounds. Uniquely to this work, we treat the more generic SIGMA-I protocol and justify our use of the strong DH problem with new bounds in the generic group model. Diemert and Jager [88] in turn study tight composition with the TLS record protocol.

The DJ analysis is carried out in the multi-stage key exchange model [100], proving security not only of the final session key, but also of intermediate handshake encryption keys and further secrets. While our proof does show security of these intermediate keys, we do not treat them as first-class keys accessible to the adversary through dedicated queries in the security model. Unlike either the DJ or Cohn-Gordon et al. works, our model addresses explicit authentication, which we prove via HMAC's unforgeability.

To tackle the challenge that TLS 1.3's key schedule does not bind DH values and session context in one function, DJ model the full cascading derivation of each intermediate key monolithically as an independent, programmable random oracle (cf. [88, Theorem 6]). We instead model the key schedule's inner HKDF [141] extraction and expansion functions as two individual random oracles, carefully connected via efficient look-up tables, yielding a slightly less extensive use of random oracles and compensating for the existence of shared computations in the derivation of multiple keys. This approach produces more compact bounds and allows our analysis to stay closer to the use of HKDF in TLS 1.3, where the output of one extraction call is used to derive multiple keys.

2.2 AKE Security Model

We provide our results in a game-based key exchange model formalized in Figure 2.1, at its core following the seminal work by Bellare and Rogaway [40] considering an active network adversary that controls all communication (initiating sessions and determining their next inputs through SEND queries) and is able to corrupt long-term secrets (REVLONGTERMKEY) as well as session keys (RevSessionKey). The adversary's goal is then to (a) distinguish the established shared *session key* in a "fresh" (not trivially compromised, captured through a Fresh predicate) session from a uniformly random key obtained through Test queries (breaking *key secrecy*), or (b) make a session accept without matching communication partner (breaking *explicit authentication*).

Following Cohn-Gordon et al. [73], we formalize our model in a real-or-random version (following Abdalla, Fouque, and Pointcheval [6] with added forward secrecy [5]) with *many* Test queries which all answer with a real or uniformly random session key based on the *same* random bit b. We focus on the security of the *main* session key established. While our proofs (for both SIGMA and TLS 1.3) establish security of the intermediate encryption and MAC keys, too, we do not treat them as first-class keys available to the adversary through Test and RevSessionKey queries. We expect that our results extend to a multi-stage key exchange (MSKE [100]) treatment and refer to the concurrent work by Diemert and Jager [88] for tight results for TLS 1.3 in a MSKE model.

In contrast to the work by Cohn-Gordon et al. [73] and Diemert and Jager [88], our model additionally captures explicit authentication through the $\mathsf{ExplicitAuth}$ predicate in Figure 2.1, ensuring sessions with non-corrupted peer accept with an honest partner session. We and [88] further treat protocols where the communication partner's identity of a session may be unknown at the outset and only learned during the protocol execution; this setting of "post-specified peers" [68] particularly applies to the SIGMA protocol family [138] as well as TLS 1.3 [186].

2.2.1 Key Exchange Protocols

We begin by formalizing the syntax of key exchange protocols. A key exchange protocol KE consists of three algorithms $(\mathsf{KGen}, \mathsf{Activate}, \mathsf{Run})$ and an associated key space $\mathsf{KE.KS}$ (where most commonly $\mathsf{KE.KS} = \{0,1\}^n$ for some $n \in \mathbb{N}$). The key generation algorithm $\mathsf{KGen}() \xrightarrow{\$} (pk, sk)$ generates new long-term public/secret key pairs. In the security model, we will associate key pairs to distinct *users* (or *parties*) with some identity $u \in \mathbb{N}$ running the protocol, and log the public long-term keys associated with each user identity in a list $peerpk$. (The adversary will be in control of initializing new users, identified by an increasing counter, and we assume it only references existing user identities.) The activation algorithm $\mathsf{Activate}(id, sk, peerid, peerpk, role) \xrightarrow{\$} (st', m')$

initiates a new session for a given user identity id (and associated long-term secret key sk) acting in a given role $role \in \{\mathsf{initiator}, \mathsf{responder}\}$ and aiming to communicate with some peer user identity $peerid$. Activate also takes as input the list $peerpk$ of all users' public keys; protocols may use this list to look up their own and their peers' public keys. We provide the entire list instead of just the user's and peers' public keys to accommodate protocols with post-specified peer. These protocols may leave $peerid$ unspecified at the time of session activation; when the peer identity is set at some later point, the list can be used to find the corresponding long-term key. Activation outputs a session state and (if $role = \mathsf{initiator}$) first protocol message m', and will be invoked in the security model to create a new session π_u^i at a user u (where the label i distinguishes different sessions of the same user). Finally, $\mathsf{Run}(id, sk, st, peerpk, m) \xrightarrow{\$} (st', m')$ delivers the next incoming key exchange message m to the session of user id with secret key sk and state st, resulting in an updated state st' and a response message m'. Like Activate, it relies on the list $peerpk$ to look up its own and its peer's long-term keys.

The state of each session in a key exchange protocol contains at least the following variables, beyond possibly further, protocol-specific information:

$peerid \in \mathbb{N}$. Reflects the (intended) partner identity of the session; in protocols with post-specified peers this is learned and set (once) by the session during the protocol execution.

$role \in \{\mathsf{initiator}, \mathsf{responder}\}$. The session's role, determined upon activation.

$status \in \{\mathsf{running}, \mathsf{accepted}, \mathsf{rejected}\}$. The session's status; initially $status = \mathsf{running}$, a session accepts when it switches to $status = \mathsf{accepted}$ (once).

$skey \in \mathsf{KE.KS}$. The derived session key (in the protocol-specific key space KE.KS), set upon acceptance.

sid. The session identifier used to define partnered session in the security model; initially unset, sid is determined (once) during protocol execution.

2.2.2 Key Exchange Security

We formalize our key exchange security game $G^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE},\mathcal{A}}$ in Figure 2.1, based on the concepts introduced above in Figure 2.1 and following the framework for code-based game playing by

Bellare and Rogaway [42]. After initializing the game, the adversary $\mathscr{A}$ is given access to queries NewUser (generating a new user's public/secret key pair), Send (controlling activation and message processing of sessions), RevSessionKey (revealing session keys), RevLongTermKey (corrupting user's long-term secrets), and Test (providing challenge real-or-random session keys), as well as a fin query to which it will submit its guess b' for the challenge bit b, ending the game.

The game $G^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE},\mathscr{A}}$ then (in fin) determines whether $\mathscr{A}$ was successful through the following three predicates, formalized in pseudocode in Figure 2.1:

Sound. The soundness predicate Sound checks that (a) no three session identifiers collide (hence the session identifier properly serves to identify two partnered sessions). Furthermore, it ensures that (b) accepted sessions with the same session identifier, agreeing partner identities, and distinct roles derive the same session key. The adversary breaks soundness if it violates either of these properties.

ExplicitAuth. The predicate ExplicitAuth captures explicit authentication in that it requires that for any session of some user *id* that accepted while its partner *peerid* was not corrupted (captured through logging relative acceptance time $\mathsf{t_{acc}}$ and long-term reveal time revltk_{peerid}) has (a) a partnered session run by the intended peer identity and in an opposite role, and (b) if that partnered session accepts, it will do so with peer identity *id*. The adversary breaks explicit authentication if this predicate evaluates to false.

Fresh. Finally, to capture key secrecy, we have to restrict the adversary to testing only so-called *fresh* sessions in order to exclude trivial attacks, which the freshness predicate Fresh ensures. A tested session is non-fresh, if (a) its session key has been revealed (in which case $\mathscr{A}$ knows the real key), (b) its partnered session (through *sid*) has been revealed or tested (in which case $\mathscr{A}$ knows the real key or may see two different random keys), or (c) its intended peer identity was compromised prior to accepting (in which case $\mathscr{A}$ may fully control the communication partner). If the adversary violates freshness, we invalidate its guess by overwriting $b' \leftarrow 0$.

We call two distinct sessions π^i_u and π^j_v *partnered* if $\pi^i_u.sid = \pi^j_v.sid$. We refer to sessions

$G^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE},\mathscr{A}}$

```
INIT:
time ← 0; users ← 0
$b \leftarrow\!\!\$\ \{0,1\}$

NEWUSER:
$\mathsf{users} \leftarrow \mathsf{users} + 1$
$(pk_{\mathsf{users}}, sk_{\mathsf{users}}) \leftarrow\!\!\$\ \mathsf{KGen}()$
$\mathsf{revltk}_{\mathsf{users}} \leftarrow \infty$
$peerpk[\mathsf{users}] \leftarrow pk_{\mathsf{users}}$
return $pk_{\mathsf{users}}$

SEND$(u,i,m)$:
if $\pi_u^i = \bot$ then
    $(peerid, role) \leftarrow m$
    $(\pi_u^i, m') \leftarrow\!\!\$\ \mathsf{Activate}(u, sk_u, peerid, peerpk, role)$
    $\pi_u^i.\mathsf{t_{acc}} \leftarrow 0$
else
    $(\pi_u^i, m') \leftarrow\!\!\$\ \mathsf{Run}(u, sk_u, \pi_u^i, peerpk, m)$
if $\pi_u^i.status = \mathsf{accepted}$ then
    $\mathsf{time} \leftarrow \mathsf{time} + 1$
    $\pi_u^i.\mathsf{t_{acc}} \leftarrow \mathsf{time}$
return $m'$

REVSESSIONKEY$(u,i)$:
if $\pi_u^i = \bot$ or $\pi_u^i.status \neq \mathsf{accepted}$ then
    return $\bot$
$\pi_u^i.\mathsf{revealed} \leftarrow \mathsf{true}$
return $\pi_u^i.skey$

REVLONGTERMKEY$(u)$:
$\mathsf{time} \leftarrow \mathsf{time} + 1$
$\mathsf{revltk}_u \leftarrow \mathsf{time}$
return $sk_u$

TEST$(u,i)$:
if $\pi_u^i = \bot$ or $\pi_u^i.status \neq \mathsf{accepted}$ or $\pi_u^i.\mathsf{tested}$ then
    return $\bot$
$\pi_u^i.\mathsf{tested} \leftarrow \mathsf{true}$
$T \leftarrow T \cup \{\pi_u^i\}$
$k_0 \leftarrow \pi_u^i.skey$
$k_1 \xleftarrow{\$} \mathsf{KE.KS}$
return $k_b$
```

```
FIN$(b')$:
if $\neg\mathsf{Sound}$ then
    return 1
if $\neg\mathsf{ExplicitAuth}$ then
    return 1
if $\neg\mathsf{Fresh}$ then
    $b' \leftarrow 0$
return $[[b = b']]$

Sound:
if $\exists$ distinct $\pi_u^i$, $\pi_v^j$, $\pi_w^k$ with $\pi_u^i.sid = \pi_v^j.sid = \pi_w^k.sid$ then    // no triple sid match
    return false
if $\exists \pi_u^i, \pi_v^j$ with
      $\pi_u^i.status = \pi_v^j.status = \mathsf{accepted}$
      and $\pi_u^i.sid = \pi_v^j.sid$
      and $\pi_u^i.peerid = v$ and $\pi_v^j.peerid = u$
      and $\pi_u^i.role \neq \pi_v^j.role$, but $\pi_u^i.skey \neq \pi_v^j.skey$ then
  // partnering implies same key
    return false
return true

ExplicitAuth:
return
      $\forall \pi_u^i : \pi_u^i.status = \mathsf{accepted}$
            and $\pi_u^i.\mathsf{t_{acc}} < \mathsf{revltk}_{\pi_u^i.peerid}$
  // all sessions accepting with a non-corrupted peer ...
          $\Longrightarrow \exists \pi_v^j : \pi_u^i.peerid = v$
              and $\pi_u^i.sid = \pi_v^j.sid$
              and $\pi_u^i.role \neq \pi_v^j.role$
  // ... have a partnered session ...
            and $(\pi_v^j.status = \mathsf{accepted} \Longrightarrow \pi_v^j.peerid = u)$
  // ... agreeing on the peerid (upon acceptance)

Fresh:
for each $\pi_u^i \in T$
    if $\pi_u^i.\mathsf{revealed}$ then
        return false    // tested session may not be revealed
    if $\exists \pi_v^j \neq \pi_u^i : \pi_v^j.sid = \pi_u^i.sid$
        and $(\pi_v^j.\mathsf{tested}$ or $\pi_v^j.\mathsf{revealed})$ then
        return false    // tested session's partnered session may not be tested or revealed
    if $\mathsf{revltk}_{\pi_u^i.peerid} < \pi_u^i.\mathsf{t_{acc}}$ then
        return false    // tested session's peer may not be corrupted prior to acceptance
return true
```

Figure 2.1. Key exchange security game.

generated by $\mathsf{Activate}$ (i.e., controlled by the game) as *honest* sessions to reflect that their behavior is determined honestly by the game and not the adversary. The long-term key of an honest session may still be corrupted, or its session key may be revealed without affecting this notion of "honesty".

Definition 1 (Key exchange security)**.** Let KE be a key exchange protocol and $G^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE},\mathscr{A}}$ be the key exchange security game defined in Figure 2.1. We define

$$\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}}(t, q_{\mathrm{N}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}) := 2 \cdot \max_{\mathscr{A}} \Pr\left[\mathrm{G}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE},\mathscr{A}} \Rightarrow 1\right] - 1,$$

where the maximum is taken over all adversaries, denoted $(t, q_{\mathrm{N}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}})$-$\mathsf{KE\text{-}SEC}$-*adversaries*, running in time at most t and making at most q_{N}, q_{S}, q_{RS}, q_{RL}, resp. q_{T} queries to their oracles NewUser, Send, RevSessionKey, RevLongTermKey, resp. Test.

2.2.3 Security Properties

Let us briefly revisit some core security properties captured in our key exchange security model.

First, we capture regular *key secrecy* of the main session key through Test queries, incorporating *forward secrecy* (sometimes called "perfect" forward secrecy) by allowing the adversary to corrupt any user as long as all tested sessions accept prior to corrupting their respective intended peer. This strengthens our model compared to that of Cohn-Gordon et al. [73] which only captures weak forward secrecy where the adversary has to be passive in sessions where it corrupts long-term secrets. Diemert and Jager [88] additionally treat the security of intermediate keys and further secrets beyond the main session key in a multi-stage approach [100], but without capturing explicit authentication.

Our model encodes *explicit authentication* (via $\mathsf{ExplicitAuth}$), a strengthening compared to the implicit-authentication model in [73].

Like [73, 88], our model captures *key-compromise impersonation* attacks by allowing the session owner's secret key of tested sessions to be corrupted at any point in time. Similarly, we do *not* capture *session-state or randomness reveals* [67, 149] or *post-compromise security* [72].

2.3 Assumptions, Building Blocks, and Multi-User Security

Before we continue to our main technical results, let us briefly introduce notation and discuss the multi-user security of the involved building blocks: strong Diffie–Hellman (including the GGM bound we prove), PRFs, digital signatures, MAC schemes, and hash functions.

2.3.1 Decisional and Strong Diffie–Hellman

The classical decisional Diffie–Hellman assumption [57] states that, when only observing the two Diffie–Hellman shares g^x, g^y, the resulting secret g^{xy} is indistinguishable from a random group element.

Definition 2 (Decisional Diffie–Hellman (DDH) assumption)**.** Let $\mathbb{G} = \langle g \rangle$ be a cyclic group of prime order p. We define

$$\mathbf{Adv}_{\mathbb{G}}^{\mathsf{DDH}}(t) := \max_{\mathscr{A}} \Big| \Pr[\mathscr{A}(\mathbb{G}, g, g^x, g^y, g^{xy}) \Rightarrow 1 \mid x, y \xleftarrow{\$} \mathbb{Z}_p] - \\ \Pr[\mathscr{A}(\mathbb{G}, g, g^x, g^y, g^z) \Rightarrow 1 \mid x, y, z \xleftarrow{\$} \mathbb{Z}_p] \Big|,$$

where the maximum is taken over all adversaries, denoted (t)-DDH-*adversaries* running in time at most t.

The strong Diffie–Hellman assumption, a weakening of the gap Diffie–Hellman assumption [180], states that solving the computational Diffie–Hellman problem given a restricted decisional Diffie–Hellman oracle is hard.

Definition 3 (Strong Diffie–Hellman assumption [180])**.** Let $\mathbb{G} = \langle g \rangle$ be a cyclic group of prime order p. Let $\mathsf{DDH}(X, Y, Z) := [[X^{\log_g(Y)} = Z]]$ be a decisional Diffie–Hellman oracle. We define

$$\mathbf{Adv}_{\mathbb{G}}^{\mathsf{stDH}}(t, q_{\mathrm{sDH}}) := \max_{\mathscr{A}} \Pr\left[\mathscr{A}^{\mathsf{DDH}(g^x, \cdot, \cdot)}(\mathbb{G}, g, g^x, g^y) = g^{xy} \;\middle|\; x, y \xleftarrow{\$} \mathbb{Z}_p\right],$$

where the maximum is taken over all adversaries, denoted (t, q_{sDH})-stDH-*adversaries* running in time at most t and making at most q_{sDH} queries to their DDH oracle.

The strong (or gap) Diffie–Hellman assumption has been deployed in numerous works to analyze practical key exchange designs, directly or through the PRF-ODH assumption [127, 63] it supports, including [127, 100, 93, 147, 95, 101, 92] as well as in the closely related works on practical tightness by Cohn-Gordon et al. [73] and Diemert and Jager [88]. To argue that it is reasonable to rely on the strong Diffie–Hellman assumption, we turn to the generic group model [195, 155]. Although some known algorithms for solving discrete logarithms in finite fields like index calculus fall outside the generic group model, the best known algorithms for elliptic curve groups are generic. Shoup [195] proved that, in the generic group model, any adversary computing at most t group operations in a group of prime order p has advantage at most $\mathscr{O}(t^2/p)$ in solving the discrete logarithm problem or the computational or decisional Diffie–Hellman problem in that group. We claim, and prove in Appendix 2.4, that any adversary in the generic group model making at most t group operations and DDH oracle queries, also has advantage at most $\mathscr{O}(t^2/p)$ in solving the strong Diffie–Hellman problem.

Theorem 4. *Let $\mathbb{G}$ be a group with prime order p. In the generic group model,* $\mathbf{Adv}^{\mathsf{stDH}}_{\mathbb{G}}(t,q) \leq 4t^2/p$.

2.3.2 Multi-User PRF Security

Let us recap the multi-user security notion for pseudorandom functions (PRFs) [31].

Definition 4 (Multi-user PRF security)**.** Let $\mathsf{PRF}\colon \{0,1\}^k \times \{0,1\}^m \to \{0,1\}^n$ be a function (for $k,n \in \mathbb{N}$ and $m \in \mathbb{N} \cup \{*\}$) and $\mathrm{G}^{\mathsf{mu\text{-}PRF}}_{\mathsf{PRF},\mathscr{A}}$ be the multi-user PRF security game defined as in Figure 2.2. We define

$$\mathbf{Adv}^{\mathsf{mu\text{-}PRF}}_{\mathsf{PRF}}(t,q_{\mathrm{NW}},q_{\mathrm{FN}},q_{\mathrm{FN/U}}) := 2 \cdot \max_{\mathscr{A}} \Pr\left[\mathrm{G}^{\mathsf{mu\text{-}PRF}}_{\mathsf{PRF},\mathscr{A}} \Rightarrow 1\right] - 1,$$

where the maximum is taken over all adversaries, denoted $(t,q_{\mathrm{NW}},q_{\mathrm{FN}},q_{\mathrm{FN/U}})$-$\mathsf{mu\text{-}PRF}$-*adversaries*, running in time at most t and making at most q_{NW} queries to their New oracle, at most q_{FN} total queries to their Fn oracle, and at most $q_{\mathrm{FN/U}}$ queries $\textsc{Fn}(i,\cdot)$ for any user i.

Generically, the multi-user security of PRFs reduces to single-user security (formally, $\mathrm{G}^{\mathsf{mu\text{-}PRF}}_{\mathsf{PRF},\mathscr{A}}$ with $\mathscr{A}$ restricted to $q_{\mathrm{NW}} = 1$ queries to New) with a factor in the number of users via a hybrid argument [31], i.e.,

$$\mathbf{Adv}^{\mathsf{mu\text{-}PRF}}_{\mathsf{PRF}}(t,q_{\mathrm{NW}},q_{\mathrm{FN}},q_{\mathrm{FN/U}}) \leq q_{\mathrm{NW}} \cdot \mathbf{Adv}^{\mathsf{mu\text{-}PRF}}_{\mathsf{PRF}}(t',1,q_{\mathrm{FN/U}},q_{\mathrm{FN/U}}),$$

```
$\underline{G^{\text{mu-PRF}}_{\text{PRF},\mathcal{A}}}$
INIT:
  $b \xleftarrow{\$} \{0,1\}$
  $u \leftarrow 0$

NEW:
  $u \leftarrow u+1$
  if $b=1$ then
    $K_u \leftarrow_{\$} \{0,1\}^k$
      $f_u := \mathsf{PRF}(K_u,\cdot)$
  else
    $f_u \xleftarrow{\$}$ FUNC

FN$(i,x)$:
  return $f_i(x)$

FIN$(b^*)$:
  return $[[b=b^*]]$
```

Figure 2.2. Multi-user PRF security of a pseudorandom function $\mathsf{PRF}\colon \{0,1\}^k \times \{0,1\}^m \to \{0,1\}^n$. FUNC is the space of all functions $\{0,1\}^m \to \{0,1\}^n$.

where $t \approx t'$. (Note that the total number q_{FN} of queries to the FN oracle across all users does not affect the reduction.) There exist simple and efficient constructions, like AMAC [29], that however achieve multi-user security tightly.

If we use a random oracle RO as a PRF with key length kl, then

$$\mathbf{Adv}^{\text{mu-PRF}}_{\text{RO}}(t, q_{\text{NW}}, q_{\text{FN}}, q_{\text{FN/U}}, q_{\text{RO}}) \leq \frac{q_{\text{NW}} \cdot q_{\text{RO}}}{2^{kl}}.$$

2.3.3 Multi-User Unforgeability with Adaptive Corruptions of Signatures and MACs

We recap the definition of digital signature schemes and message authentication codes (MACs) as well as the natural extension of classical *existential unforgeability under chosen-message attacks* [110] to the *multi-user* setting with *adaptive corruptions*. For signatures, this notion was considered by Bader et al. [21] and, without corruptions, by Menezes and Smart [161].

Definition 5 (Signature scheme)**.** A *signature scheme* $\mathsf{S} = (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vrfy})$ consists of three efficient algorithms defined as follows.

- $\mathsf{KGen}() \xrightarrow{\$} (pk, sk)$. This probabilistic algorithm generates a public verification key pk and a secret signing key sk.

- $\mathsf{Sign}(sk, m) \xrightarrow{\$} \sigma$. On input a signing key sk and a message m, this (possibly) probabilistic algorithm outputs a signature σ.

- $\mathsf{Vrfy}(vk, m, \sigma) \to d$. On input a verification key pk, a message m, and a signature σ, this deterministic algorithm outputs a decision bit $d \in \{0,1\}$ (where $d = 1$ indicates validity of the signature).

Definition 6 (Signature $\mathsf{mu\text{-}EUF\text{-}CMA}$ security)**.** Let S be a signature scheme and $\mathrm{G}^{\mathsf{mu\text{-}EUF\text{-}CMA}}_{\mathsf{S},\mathscr{A}}$ be the game for signature multi-user existential unforgeability under chosen-message attacks with adaptive corruptions defined as in Figure 2.3. We define

$$\mathbf{Adv}^{\mathsf{mu\text{-}EUF\text{-}CMA}}_{\mathsf{S}}(t, q_{\mathrm{Nw}}, q_{\mathrm{SG}}, q_{\mathrm{SG/U}}, q_{\mathrm{C}}) := \max_{\mathscr{A}} \Pr\left[\mathrm{G}^{\mathsf{mu\text{-}EUF\text{-}CMA}}_{\mathsf{S},\mathscr{A}} \Rightarrow 1\right],$$

where the maximum is taken over all adversaries, denoted $(t, q_{\mathrm{Nw}}, q_{\mathrm{SG}}, q_{\mathrm{SG/U}}, q_{\mathrm{C}})$-$\mathsf{mu\text{-}EUF\text{-}CMA}$-*adversaries*, running in time at most t and making at most q_{Nw}, q_{SG}, resp. q_{C} total queries to their New, Sign, resp. Corrupt oracle, and making at most $q_{\mathrm{SG/U}}$ queries $\textsc{Sign}(i, \cdot)$ for any user i.

Multi-user EUF-CMA security of signature schemes (with adaptive corruptions) can be reduced to classical, single-user EUF-CMA security (formally, $\mathrm{G}^{\mathsf{mu\text{-}EUF\text{-}CMA}}_{\mathsf{S},\mathscr{A}}$ with $\mathscr{A}$ restricted to $q_{\mathrm{Nw}} = 1$ queries to New) by a standard hybrid argument, losing a factor of number of users. Formally, this yields

$$\mathbf{Adv}^{\mathsf{mu\text{-}EUF\text{-}CMA}}_{\mathsf{S}}(t, q_{\mathrm{Nw}}, q_{\mathrm{SG}}, q_{\mathrm{SG/U}}, q_{\mathrm{C}}) \leq q_{\mathrm{Nw}} \cdot \mathbf{Adv}^{\mathsf{mu\text{-}EUF\text{-}CMA}}_{\mathsf{S}}(t', 1, q_{\mathrm{SG/U}}, q_{\mathrm{SG/U}}, 0),$$

where $t \approx t'$. (Note that the reduction is not affected by the total number of signature queries q_{SG} across all users.) In many cases, such loss is indeed unavoidable [22].

Definition 7 (MAC scheme)**.** A *MAC scheme* $\mathsf{M} = (\mathsf{KGen}, \mathsf{Tag}, \mathsf{Vrfy})$ consists of three efficient algorithms defined as follows.

- $\mathsf{KGen}() \xrightarrow{\$} K$. This probabilistic algorithm generates a key K.
- $\mathsf{Tag}(K, m) \xrightarrow{\$} \tau$. On input a key K and a message m, this (possibly) probabilistic algorithm outputs a message authentication code (MAC) τ.
- $\mathsf{Vrfy}(K, m, \tau) \to d$. On input a key K, a message m, and a MAC τ, this deterministic algorithm outputs a decision bit $d \in \{0, 1\}$ (where $d = 1$ indicates validity of the MAC).

Definition 8 (MAC $\mathsf{mu\text{-}EUF\text{-}CMA}$ security)**.** Let M be a MAC scheme and $\mathrm{G}^{\mathsf{mu\text{-}EUF\text{-}CMA}}_{\mathsf{M},\mathscr{A}}$ be the game for MAC multi-user existential unforgeability under chosen-message attacks with adaptive

```
$\mathrm{G}^{\textsf{mu-EUF-CMA}}_{\textsf{S},\mathscr{A}}$

INIT:
    $Q \leftarrow \emptyset$
    $\mathscr{C} \leftarrow \emptyset$
    $u \leftarrow 0$

CORRUPT($i$):
    $\mathscr{C} \leftarrow \mathscr{C} \cup \{i\}$
    return $sk_i$

NEW:
    $u \leftarrow u+1$
    $(pk_u, sk_u) \leftarrow\$ \textsf{KGen}()$
    return $pk_u$

SIGN($i,m$):
    $\sigma \leftarrow\$ \textsf{Sign}(sk_i, m)$
    $Q \leftarrow Q \cup \{(i,m)\}$
    return $\sigma$

FIN($i^*, m^*, \sigma^*$):
    $d^* \leftarrow \textsf{Vrfy}(pk_{i^*}, m^*, \sigma^*)$
    return $[[d^* = 1 \wedge i^* \notin \mathscr{C} \wedge (i^*, m^*) \notin Q]]$
```

```
$\mathrm{G}^{\textsf{mu-EUF-CMA}}_{\textsf{M},\mathscr{A}}$

INIT:
    $Q \leftarrow \emptyset$
    $\mathscr{C} \leftarrow \emptyset$
    $u \leftarrow 0$

CORRUPT($i$):
    $\mathscr{C} \leftarrow \mathscr{C} \cup \{i\}$
    return $K_i$

NEW:
    $u \leftarrow u+1$
    $K_u \leftarrow\$ \textsf{KGen}()$

TAG($i,m$):
    $\tau \leftarrow\$ \textsf{Tag}(K_i, m)$
    $Q \leftarrow Q \cup \{(i,m)\}$
    return $\tau$

VRFY($i,m,\tau$):
    $d \leftarrow \textsf{Vrfy}(K_i, m, \tau)$
    return $d$

FIN($i^*, m^*, \tau^*$):
    $d^* \leftarrow \textsf{Vrfy}(K_{i^*}, m^*, \tau^*)$
    return $[[d^* = 1 \wedge i^* \notin \mathscr{C} \wedge (i^*, m^*) \notin Q]]$
```

Figure 2.3. Multi-user existential unforgeability (mu-EUF-CMA) of signature schemes (top) and MAC schemes (bottom).

corruptions defined as in Figure 2.3. We define

$$\mathbf{Adv}^{\textsf{mu-EUF-CMA}}_{\textsf{M}}(t, q_{\mathrm{Nw}}, q_{\mathrm{TG}}, q_{\mathrm{TG/U}}, q_{\mathrm{VF}}, q_{\mathrm{VF/U}}, q_{\mathrm{C}}) := \max_{\mathscr{A}} \Pr\left[\mathrm{G}^{\textsf{mu-EUF-CMA}}_{\textsf{M},\mathscr{A}} \Rightarrow 1\right],$$

where the maximum is taken over all adversaries, denoted $(t, q_{\mathrm{Nw}}, q_{\mathrm{TG}}, q_{\mathrm{TG/U}}, q_{\mathrm{VF}}, q_{\mathrm{VF/U}}, q_{\mathrm{C}})$-mu-EUF-CMA-*adversaries*, running in time at most t and making at most q_{Nw}, q_{TG}, q_{VF}, resp. q_{C} queries to their NEW, SIGN, VRFY, resp. CORRUPT oracle, and making at most $q_{\mathrm{TG/U}}$ queries TAG$(i,\cdot)$, resp. $q_{\mathrm{VF/U}}$ queries VRFY$(i,\cdot)$ for any user i.

As for signature schemes, multi-user EUF-CMA security of MACs reduces to the single-user case ($q_{\mathrm{Nw}} = 1$) by a standard hybrid argument:

$$\begin{aligned}&\mathbf{Adv}^{\textsf{mu-EUF-CMA}}_{\textsf{M}}(t, q_{\mathrm{Nw}}, q_{\mathrm{TG}}, q_{\mathrm{TG/U}}, q_{\mathrm{VF}}, q_{\mathrm{VF/U}}, q_{\mathrm{C}})\\ &\leq q_{\mathrm{Nw}} \cdot \mathbf{Adv}^{\textsf{mu-EUF-CMA}}_{\textsf{M}}(t, 1, q_{\mathrm{TG/U}}, q_{\mathrm{TG/U}}, q_{\mathrm{VF/U}}, q_{\mathrm{VF/U}}, 0),\end{aligned}$$

where $t \approx t'$. (Note that the reduction is not affected by the total number of tagging and verification queries q_{TG} resp. q_{VF} across all users.)

Our multi-user definition of MACs provides a verification oracle, which is non-standard (and in general not equivalent to a definition with a single forgery attempts, as Bellare, Goldreich and Mityiagin [34] showed). For PRF-based MACs (which in particular includes HMAC used in TLS 1.3), it however is equivalent and the reduction from multi-query to single-query verification is tight [34].

In our key exchange reductions, we actually do not need to corrupt MAC keys, i.e., we achieve $q_{\mathrm{C}} = 0$. This in particular allows specific constructions like AMAC [29] achieving tight multi-user security (without corruptions).

If we use a random oracle RO as PRF-like MAC with key length kl and output length ol, then

$$\mathbf{Adv}^{\mathsf{mu\text{-}EUF\text{-}CMA}}_{\mathrm{RO}}(t, q_{\mathrm{Nw}}, q_{\mathrm{TG}}, q_{\mathrm{TG/U}}, q_{\mathrm{VF}}, q_{\mathrm{VF/U}}, q_{\mathrm{C}}, q_{\mathrm{RO}}) \leq \frac{q_{\mathrm{VF}}}{2^{ol}} + \frac{(q_{\mathrm{Nw}} - q_{\mathrm{C}}) \cdot q_{\mathrm{RO}}}{2^{kl}}.$$

2.3.4 Hash Function Collision Resistance

Finally, let us define collision resistance of hash functions.

Definition 9 (Hash function collision resistance)**.** Let $\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^{ol}$ for $ol \in \mathbb{N}$ be a function. For a given adversary $\mathscr{A}$ running in time at most t, we can consider

$$\mathbf{Adv}^{\mathsf{CR}}_{\mathsf{H}}(t) := \Pr\left[(m, m') \leftarrow_{\$} \mathscr{A} : m \neq m' \text{ and } \mathsf{H}(m) = \mathsf{H}(m')\right].$$

If we use a random oracle RO as hash function, then by the birthday bound

$$\mathbf{Adv}^{\mathsf{CR}}_{\mathrm{RO}}(t, q_{\mathrm{RO}}) \leq \frac{q_{\mathrm{RO}}^2}{2^{ol+1}} + \frac{1}{2^{ol}}.$$

2.4 Proof of the Strong Diffie–Hellman GGM Bound (Theorem 4)

Proof: We begin by giving a code-based game for the strong Diffie–Hellman problem in the generic group model. First, we establish some preliminaries, using the setting and notation of Bellare and Dai [32]. Let $\mathbb{G}$ be an arbitrary set of strings with prime order p, and let $E : \mathbb{Z}_p \to \mathbb{G}$

```
$\underline{\mathrm{G}_0}$
INIT():
    $p \leftarrow |\mathbb{G}|$; $E \leftarrow\!\!\$\ \mathsf{Bijections}(\mathbb{Z}_p, \mathbb{G})$
    $\mathbb{1} \leftarrow E(0)$; $g \leftarrow E(1)$
    $x, y \leftarrow\!\!\$\ \mathbb{Z}_p^*$; $X \leftarrow E(x)$; $Y \leftarrow E(y)$
    $GL \leftarrow \{\mathbb{1}, g, x, y\}$
    return $(\mathbb{1}, g, x, y)$
OP($A, B$, sgn ):
    if $A \notin GL$ or $B \notin GL$ then return $\perp$
    $c \leftarrow E^{-1}(A)$ sgn $E^{-1}(B) \mod p$
    $C \leftarrow E(c)$; $GL \leftarrow GL \cup \{C\}$
    return $C$
stDH($A, B$):
    if $A \notin GL$ or $B \notin GL$ then return $\perp$
    $z \leftarrow x \cdot E^{-1}(A) \mod p$
    $Z \leftarrow E(z)$
    return $[[Z = B]]$
FIN($Z$):
    if $Z \notin GL$ then return false
    $z \leftarrow x \cdot y \mod p$; return $[[Z = E(z)]]$
```

Figure 2.4. Game G_0 of the stDH proof.

be a bijection, called the encoding function. For any two strings $A, B \in \mathbb{G}$, we define the operation $A\,\mathrm{OP}_E\,B = E(E^{-1}(A) + E^{-1}(B) \mod p)$. The set $\mathbb{G}$ is a group with respect to this operation, and it is isomorphic to $\mathbb{Z}_p$. Therefore, $\mathbb{G}$ has the identity $E(0)$, and it is generated by $E(1)$.

In the generic group model, we wish for the adversary to compute group operations only through an oracle OP. We accomplish this by picking the encoding function E at random and keeping it secret; then providing oracle access to OP_E through OP. In this model, we can give a sequence of games bounding the advantage of any adversary $\mathscr{A}$ that makes t queries to the OP oracle and q queries to the stDH oracle.

Game 0. This first game formalizes the strong Diffie–Hellman problem in the generic group model. Note that for any $a \in \mathbb{Z}_p$, a is the discrete logarithm of the group element $E(a)$.

It follows that

$$\mathbf{Adv}_{\mathbb{G}}^{\mathsf{stDH}}(t, q_{\mathrm{sDH}}) = \Pr[\mathrm{G}_0 \Rightarrow 1].$$

Game 1. In Game G_1, we change the internal notation of the game. First, for clarity and without loss of generality, we assume the adversary queries its OP and stDH oracles only on valid inputs (meaning their inputs are valid group elements in GL). Instead of representing each element of $\mathbb{G}$ with an element of $\mathbb{Z}_p$, we use a vector over $\mathbb{Z}_p^3$. We define the basis vectors $\vec{e}_1 := (1,0,0)$, $\vec{e}_2 := (0,1,0)$, and $\vec{e}_3 := (0,0,1)$. We map $\mathbb{Z}_p^3$ to $\mathbb{Z}_p$ by taking the inner product with

```
G_1

INIT():
  $p \leftarrow |\mathbb{G}|$; $E \leftarrow\!\!\$\, \text{Bijections}(\mathbb{Z}_p, \mathbb{G})$
  $k \leftarrow 0$; $\mathbb{1} \leftarrow \mathrm{VE}(\vec{0})$; $g \leftarrow \mathrm{VE}(\vec{e_1})$
  $x, y \leftarrow\!\!\$\, \mathbb{Z}_p^*$; $\vec{x} \leftarrow \mathbb{1}, x, y$
  $X \leftarrow \mathrm{VE}(\vec{e_2})$; $Y \leftarrow \mathrm{VE}(\vec{e_3})$
  return $(\mathbb{1}, g, x, y)$

OP($A, B$, sgn ):
  $\vec{c} \leftarrow \mathrm{VE}^{-1}(A)$ sgn $\mathrm{VE}^{-1}(B)$  mod $p$
  $C \leftarrow \mathrm{VE}(\vec{c})$; return $C$

VE($\vec{t}$):
  if $TV[\vec{t}] \neq \perp$ then return $TV[\vec{t}]$
  $k \leftarrow k+1$; $\vec{t_k} \leftarrow \vec{t}$
  $v \leftarrow \langle \vec{t}, \vec{x} \rangle$; $C \leftarrow E(v)$; $GL \leftarrow GL \cup \{C\}$
  $TV[\vec{t}] \leftarrow C$; $TI[C] \leftarrow \vec{t}$
  return $TV[\vec{t}]$

stDH($A, B$):
  $\vec{a} \leftarrow \mathrm{VE}^{-1}(A)$; $\vec{b} \leftarrow \mathrm{VE}^{-1}(B)$
  return $[[\mathrm{VE}(x\vec{a}) = B]]$

FIN($Z$):
  return $[[\mathrm{VE}(x\vec{e_3}) = Z]]$

VE$^{-1}$($C$):
  return $TI[C]$
```

Figure 2.5. Game G_1 of the stDH proof.

the vector $(1, x, y)$. (Effectively, we are representing each element of $\mathbb{Z}_p$ as a linear combination modulo p of 1, x, and y.) We cache the map from $\mathbb{Z}_p^3$ to $\mathbb{G}$ induced by this transformation in a table TV and its inverse map in a table TI.

Although one element of $\mathbb{G}$ may now have multiple representations, the bilinearity of the inner product ensures that the view of the adversary is not changed, and $\Pr[\mathrm{G}_1] = \Pr[\mathrm{G}_0]$.

Game 2. Next, we replace the random encoding function E with a lazily sampled encoding represented by table TV for the forward direction and TI for the backward direction. Because we want our encoding to be one-to-one, we sample from the set $\mathbb{G} \setminus GL$. This assigns a unique element of $\mathbb{G}$ to each vector $\vec{t}$. However, as we've noted, each integer in $\mathbb{Z}_p$ has multiple representations in $\mathbb{Z}_p^3$. If two representations of the same integer are submitted to the encoding algorithm VE, we set a bad flag and program the encoding table to maintain consistency.

We also change the format of the check in the stDH oracle. Since $\mathrm{VE}(x\vec{a}) = B = \mathrm{VE}(\vec{b})$ if and only if $\langle x\vec{a}, \vec{x} \rangle = \langle \vec{b}, \vec{x} \rangle$, we return true if the latter condition holds and false otherwise. These two conditions are equivalent, so $\Pr[\mathrm{G}_2] = \Pr[\mathrm{G}_1]$.

Game 3. In this game, we stop programming the encoding table after the bad flag is set. Let F_1 denote the event that G_3 sets the bad flag at any point. By the fundamental lemma of game playing, $\Pr[\mathrm{G}_2] \leq \Pr[\mathrm{G}_3 \text{ and } \overline{F_1} + \Pr[F_1]$.

Game 4. We remove the now-redundant bad flag, but the FIN oracle now returns true if at any point in game G_3 the bad flag would have been set (i.e. if event F_1 occurs). Otherwise, all oracles behave exactly as they did in G_3. It follows that $\Pr[\mathrm{G}_3 \text{ and } \overline{F_1}] + \Pr[F_1] \leq \Pr[\mathrm{G}_4]$.

Additionally, in the stDH oracle, we separate out checking for trivial queries: if the adversary computed $A = g^a$ and $B = X^a$ for an integer a of their choosing. If this is so, then $\vec{a} = a\vec{e}_1$ and $\vec{b} = a\vec{e}_2$, so $\langle x\vec{a}, \vec{x}\rangle = xa = \langle \vec{b}, \vec{x}\rangle$, so may return true. If the query is nontrivial but should still return true according to our previous condition, we set a $\mathsf{bad}[2]$ flag. This does not change the oracle's response to any query, so the above bound still holds.

Game 5. In Game G_5, we no longer return true in the stDH oracle after the $\mathsf{bad}[2]$ flag is set. This makes the second check redundant and has the effect that the stDH oracle's behavior is no longer dependent on the value of either x or $\vec{x}$. Let event F_2 denote the event that G_5 sets the $\mathsf{bad}[2]$ flag. By the fundamental lemma of game playing, $\Pr[\mathrm{G}_4] \leq \Pr[\mathrm{G}_5 \text{ and } \overline{F_2}] + \Pr[F_2]$.

Game 6. In Game G_6, we remove the redundant check and bad flag from the stDH oracle, and in the FIN oracle we return true whenever the $\mathsf{bad}[2]$ flag would have been set in G_5. Otherwise all oracles behave precisely as they did in G_5. It follows that $\Pr[\mathrm{G}_5 \text{ and } \overline{F_2}] + \Pr[F_2] \leq \Pr[\mathrm{G}_6]$. We also move the initialization of variables x, y, and $\vec{x}$ from INIT to FIN. Since these variables are not used by any oracle but FIN, this does not change the view of the adversary.

At this point, we can collect the bounds from each gamehop to see that

$$\mathbf{Adv}_{\mathbb{G}}^{\mathsf{stDH}}(t, q_{\mathrm{sDH}}) \leq \Pr[\mathrm{G}_6].$$

Therefore we analyze the advantage of an adversary in game G_6.

We can separately analyze each condition of FIN. We know that x and y are sampled independently of the $t+4$ entries of TV. For each index $i \in [1 \ldots t+4]$, let F_i be the bivariate linear polynomial over $\mathbb{Z}_p$ whose coefficients are given by the vector $\vec{t_i}$. Then for any pair of vectors $(\vec{t_i}, \vec{t_j})$, the

$\boxed{\mathrm{G}_2}$, G_3

```
stDH$(A,B)$:
  $\vec{a} \leftarrow \mathrm{VE}^{-1}(A)$; $\vec{b} \leftarrow \mathrm{VE}^{-1}(B)$
  if $\langle x\vec{a},\vec{x}\rangle = \langle \vec{b},\vec{x}\rangle$ then return true
  return false

$\mathrm{VE}(\vec{t})$:
  if $TV[\vec{t}] \neq \perp$ then return $TV[\vec{t}]$
  $C \leftarrow \mathbb{G} \setminus GL$
  if $(\exists \vec{s} : TV[\vec{s}] \neq \perp$ and $\langle \vec{t},\vec{x}\rangle = \langle \vec{s},\vec{x}\rangle)$
      then bad $\leftarrow$ true; $\boxed{C \leftarrow TV[\vec{s}]}$
  $k \leftarrow k+1$; $\vec{t}_k \leftarrow \vec{t}$
  $GL \leftarrow GL \cup \{C\}$
  $TV[\vec{t}] \leftarrow C$; $TI[C] \leftarrow \vec{t}$
  return $TV[\vec{t}]$
```

G_6

```
INIT():
  $p \leftarrow |\mathbb{G}|$;
  $k \leftarrow 0$;$\mathbb{1} \leftarrow \mathrm{VE}(\vec{0})$; $g \leftarrow \mathrm{VE}(\vec{e}_1)$
  $X \leftarrow \mathrm{VE}(\vec{e}_2)$; $Y \leftarrow \mathrm{VE}(\vec{e}_3)$
  return $(\mathbb{1},g,x,y)$

FIN$(Z)$:
  $x,y \leftarrow\!\!\$\ \mathbb{Z}_p^*$; $\vec{x} \leftarrow (\mathbb{1},x,y)$
  if $\exists i,j : 1 \leq i < j \leq k$ and $\langle \vec{t}_i - \vec{t}_j,\vec{x}\rangle = 0)$
      then return true
  if $\exists i,j : 1 \leq i < j \leq k$
  and $\langle x\vec{t}_i - \vec{t}_j,\vec{x}\rangle = 0$ or $\langle x\vec{t}_j - \vec{t}_i,\vec{x}\rangle 0$
      then return true
  return $[[\mathrm{VE}(x\vec{e}_3) = Z]]$
```

$\boxed{\mathrm{G}_4}$, G_5

```
stDH$(A,B)$:
  $\vec{a} \leftarrow \mathrm{VE}^{-1}(A)$; $\vec{b} \leftarrow \mathrm{VE}^{-1}(B)$; $a \leftarrow \vec{a}[1]$
  if $\vec{a} = a\vec{e}_1$ and $\vec{b} = a\vec{e}_2$ then return true
  if $\langle x\vec{a},\vec{x}\rangle = \langle \vec{b},\vec{x}\rangle$ then bad[2] $\leftarrow$ true ; $\boxed{\text{return true}}$
  return false

$\mathrm{VE}(\vec{t})$:
  if $TV[\vec{t}] \neq \perp$ then return $TV[\vec{t}]$
  $C \leftarrow \mathbb{G} \setminus GL$
  $k \leftarrow k+1$; $\vec{t}_k \leftarrow \vec{t}$
  $GL \leftarrow GL \cup \{C\}$
  $TV[\vec{t}] \leftarrow C$; $TI[C] \leftarrow \vec{t}$
  return $TV[\vec{t}]$

FIN$(Z)$:
  if $\exists i,j : 1 \leq i < j \leq k$ and $\langle \vec{t}_i - \vec{t}_j,\vec{x}\rangle = 0$
      then return true
  return $[[\mathrm{VE}(x\vec{e}_3) = Z]]$

stDH$(A,B)$:
  $\vec{a} \leftarrow \mathrm{VE}^{-1}(A)$; $\vec{b} \leftarrow \mathrm{VE}^{-1}(B)$;$a \leftarrow \vec{a}[1]$
  if $(\vec{a} = a\vec{e}_1$ and $\vec{b} = a\vec{e}_2)$ then return true
  return 0
```

Figure 2.6. Top left: Games G_2 (changes highlighted in gray) and G_3 (changes highlighted in frames) of the strong Diffie–Hellman proof. Top right: Games G_4 and G_5. Bottom: Game G_6 (changes highlighted in gray) of the strong Diffie–Hellman proof.

condition $\langle \vec{t_i} - \vec{t_j} \rangle = 0$ holds only if $(1,x,y)$ is a root of $F_i - F_j$. Using Lemma 1 of [195] and a union bound over all pairs, the probability of this event is at most $(t+4)^2/p$.

For the second condition; we see that for any $(\vec{t_i}, \vec{t_j})$, it is true that $\langle x\vec{t_i} - \vec{t_j} \rangle = 0$ only if $(1,x,y)$ is a root of $XF_i - F_j$, which is a bivariate quadratic polynomial over $\mathbb{Z}_p$. Again Using Lemma 1 and a union bound, this occurs with probability at most $2(t+4)^2/p$.

If neither event occurs, then the adversary wins only if $[\mathrm{VE}(x\vec{e_3}) = Z]$. Because the second condition failed, we know that $(x\vec{e_3})$ is not an entry in table TV. Therefore the response to $\mathrm{VE}(x\vec{e_3})$ will be sampled uniformly at random, and it will equal Z with probability $1/p$. Then by the union bound, $\Pr[\mathrm{G}_6] \leq (3(t+4)^2+1)/p$. Collecting the bounds gives the theorem statement for all $t > 25$. ▮

2.5 The SIGMA Protocol

The SIGMA family of key exchange protocols introduced by Krawczyk [138, 139] describes several variants for building authenticated Diffie–Hellman key exchange using the "SIGn-and-MAc" approach. Its design has been adopted in several Internet security protocols, including, e.g., the Internet Key Exchange protocol [117, 135] as part of the IPsec Internet security protocol [136] and the newest version 1.3 of the Transport Layer Security (TLS) protocol [186].

Beyond the basic SIGMA design, we are particularly interested in the SIGMA-I variant which forms the basis of the TLS 1.3 key exchange and aims at hiding the protocol participants' identities as additional feature. We here present an augmented version of the basic SIGMA/SIGMA-I protocols which includes explicit exchange of session-identifying random numbers (nonces) to be closer to SIGMA(-like) protocols in practice, somewhat following the "full-fledged" SIGMA variant [139, Appendix B]. We illustrate these protocol flows in Figure 2.7. and Figure 2.8 formalizes both as key exchange protocols according to the syntax of Section 2.2.1.

The SIGMA and SIGMA-I protocols make use of a signature scheme $\mathsf{S} = (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Vrfy})$, a MAC scheme $\mathsf{M} = (\mathsf{KGen}, \mathsf{Tag}, \mathsf{Vrfy})$, a pseudorandom function PRF, and a function RO which we model as a random oracle. The parties' long-term secret keys consist of one signing key, i.e., $\mathsf{KE.KGen} = \mathsf{S.KGen}$. The protocols consists of three messages exchanged and accordingly two

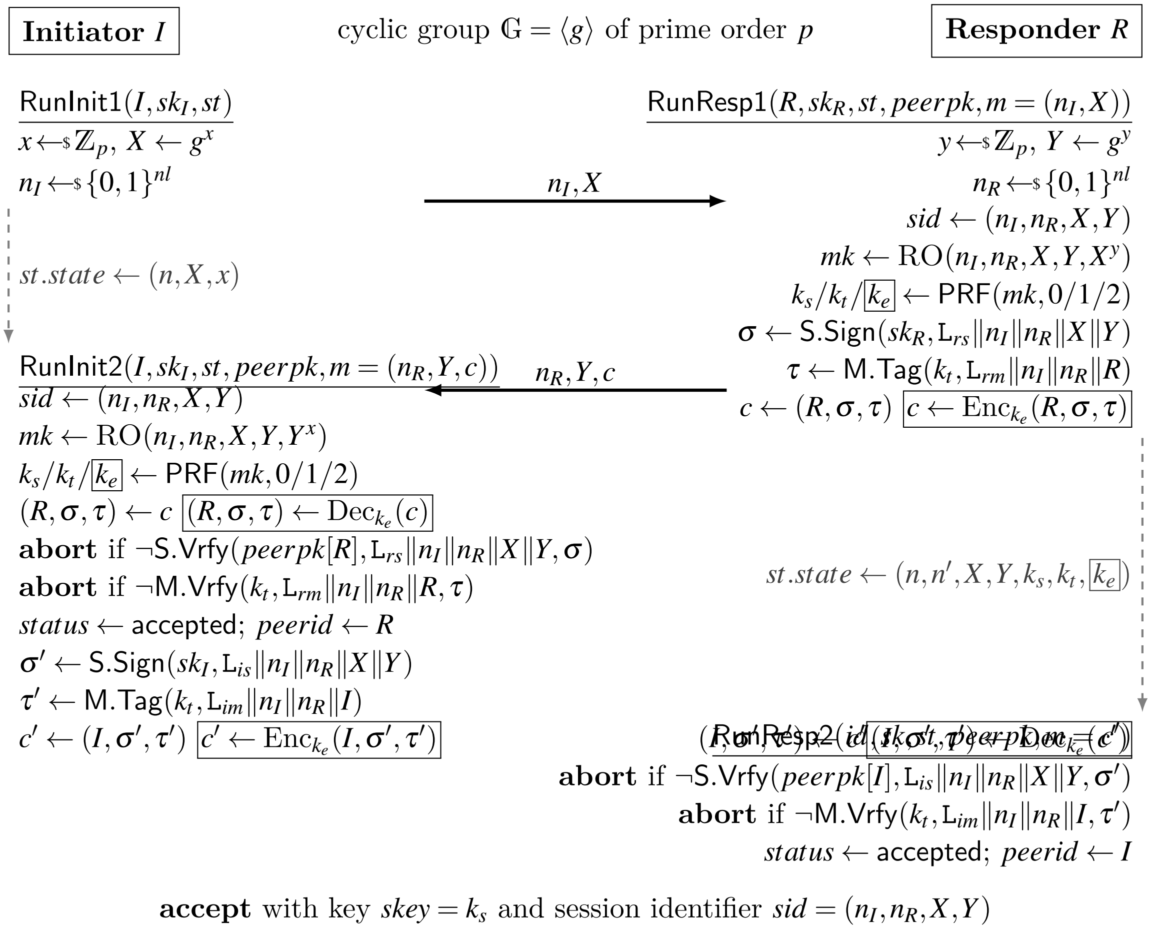

Initiator I
cyclic group $\mathbb{G} = \langle g \rangle$ of prime order p
Responder R
$\mathsf{RunInit1}(I, sk_I, st)$
$x \leftarrow_\$ \mathbb{Z}_p$, $X \leftarrow g^x$
$n_I \leftarrow_\$ \{0,1\}^{nl}$
n_I, X
$st.state \leftarrow (n, X, x)$
$\mathsf{RunResp1}(R, sk_R, st, peerpk, m = (n_I, X))$
$y \leftarrow_\$ \mathbb{Z}_p$, $Y \leftarrow g^y$
$n_R \leftarrow_\$ \{0,1\}^{nl}$
$sid \leftarrow (n_I, n_R, X, Y)$
$mk \leftarrow \mathrm{RO}(n_I, n_R, X, Y, X^y)$
$k_s/k_t/\boxed{k_e} \leftarrow \mathsf{PRF}(mk, 0/1/2)$
$\sigma \leftarrow \mathsf{S.Sign}(sk_R, \mathsf{L}_{rs} \| n_I \| n_R \| X \| Y)$
$\tau \leftarrow \mathsf{M.Tag}(k_t, \mathsf{L}_{rm} \| n_I \| n_R \| R)$
$c \leftarrow (R, \sigma, \tau)$ $\boxed{c \leftarrow \mathrm{Enc}_{k_e}(R, \sigma, \tau)}$
n_R, Y, c
$\mathsf{RunInit2}(I, sk_I, st, peerpk, m = (n_R, Y, c))$
$sid \leftarrow (n_I, n_R, X, Y)$
$mk \leftarrow \mathrm{RO}(n_I, n_R, X, Y, Y^x)$
$k_s/k_t/\boxed{k_e} \leftarrow \mathsf{PRF}(mk, 0/1/2)$
$(R, \sigma, \tau) \leftarrow c$ $\boxed{(R, \sigma, \tau) \leftarrow \mathrm{Dec}_{k_e}(c)}$
abort if $\neg\mathsf{S.Vrfy}(peerpk[R], \mathsf{L}_{rs} \| n_I \| n_R \| X \| Y, \sigma)$
abort if $\neg\mathsf{M.Vrfy}(k_t, \mathsf{L}_{rm} \| n_I \| n_R \| R, \tau)$
$status \leftarrow \mathsf{accepted}$; $peerid \leftarrow R$
$\sigma' \leftarrow \mathsf{S.Sign}(sk_I, \mathsf{L}_{is} \| n_I \| n_R \| X \| Y)$
$\tau' \leftarrow \mathsf{M.Tag}(k_t, \mathsf{L}_{im} \| n_I \| n_R \| I)$
$c' \leftarrow (I, \sigma', \tau')$ $\boxed{c' \leftarrow \mathrm{Enc}_{k_e}(I, \sigma', \tau')}$
$st.state \leftarrow (n, n', X, Y, k_s, k_t, \boxed{k_e})$
$\mathsf{RunResp2}(R, sk_R, st, peerpk, m = (c'))$
$(I, \sigma', \tau') \leftarrow c'$ $\boxed{(I, \sigma', \tau') \leftarrow \mathrm{Dec}_{k_e}(c')}$
abort if $\neg\mathsf{S.Vrfy}(peerpk[I], \mathsf{L}_{is} \| n_I \| n_R \| X \| Y, \sigma')$
abort if $\neg\mathsf{M.Vrfy}(k_t, \mathsf{L}_{im} \| n_I \| n_R \| I, \tau')$
$status \leftarrow \mathsf{accepted}$; $peerid \leftarrow I$
accept with key $skey = k_s$ and session identifier $sid = (n_I, n_R, X, Y)$

Figure 2.7. The SIGMA/SIGMA-I protocol flow diagram. $\boxed{\text{Boxed}}$ code is only performed in the SIGMA-I variant. Values L_x indicate label strings (distinct per x).

steps performed by both initiator and responder, which we describe in more detail now.

Initiator Step 1. The initiator picks a Diffie–Hellman exponent $x \overset{\$}{\leftarrow} \mathbb{Z}_p$ and a random nonce n_I of length nl and sends n_I and g^x.

Responder Step 1. The responder also picks a random DH exponent y and a random nonce n_R. It then derives a master key as $mk \leftarrow \mathrm{RO}(n_I, n_R, X, Y, X^y)$ from nonces, DH shares, and the joint DH secret $g^{xy} = (g^x)^y$. From mk, keys are derived via PRF with distinct labels: the session key k_s, the MAC key k_t, and (only in SIGMA-I) the encryption key k_e.

The responder computes a signature σ with sk_R over nonces and DH shares (and a unique label L_{rs}) and a MAC value τ under key k_t over the nonces and its identity R (and unique label L_{rm}). It sends n_I, g^y, as well as R, σ, and τ to the initiator. In SIGMA-I the last

```
$\mathsf{Activate}(id, sk, peerid, peerpk, role)$:
$st'.role \leftarrow role$
$st'.status \leftarrow$ running
if $role =$ initiator then
    $(st', m') \leftarrow \mathsf{RunInit1}(id, sk, st')$
else $m' \leftarrow \bot$
return $(st', m')$
```

```
$\mathsf{Run}(id, sk, st, peerpk, m)$:
if $st.status \neq$ running then
return $\bot$
if $st.role =$ initiator then
    $(st', m') \leftarrow \mathsf{RunInit2}(id, sk, st, peerpk, m)$
else if $st.sid = \bot$
    $(st', m') \leftarrow \mathsf{RunResp1}(id, sk, st, peerpk, m)$
else
    $(st', m') \leftarrow \mathsf{RunResp2}(id, sk, st, peerpk, m)$
return $(st', m')$
```

```
$\mathsf{RunInit1}(id, sk, st)$:
$n_I \xleftarrow{\$} \{0,1\}^{nl}$
$x \xleftarrow{\$} \mathbb{Z}_p$
$X \leftarrow g^x$
$st'.state \leftarrow (n_I, X, x)$
$m' \leftarrow (n_I, X)$
return $(st', m')$
```

```
$\mathsf{RunResp1}(id, sk, st, peerpk, m)$:
$(n_I, X) \leftarrow m$
$n_R \xleftarrow{\$} \{0,1\}^{nl}$
$y \xleftarrow{\$} \mathbb{Z}_p$
$Y \leftarrow g^y$
$st'.sid \leftarrow (n_I, n_R, X, Y)$
$\sigma \leftarrow \mathsf{S.Sign}(sk, \mathsf{L}_{rs} \| n_I \| n_R \| X \| Y)$
$mk \leftarrow \mathrm{RO}(n_I \| n_R \| X \| Y \| X^y)$
$k_s \leftarrow \mathsf{PRF}(mk, 0)$
$k_t \leftarrow \mathsf{PRF}(mk, 1)$
$\boxed{k_e \leftarrow \mathsf{PRF}(mk, 2)}$
$\tau \leftarrow \mathsf{M.Tag}(k_t, \mathsf{L}_{rm} \| n_I \| n_R \| id)$
$st'.state \leftarrow (n_I, n_R, X, Y, k_s, k_t)$
$\boxed{st'.state \leftarrow (n_I, n_R, X, Y, k_s, k_t, k_e)}$
$m' \leftarrow (n_R, Y, id, \sigma, \tau)$
$\boxed{m' \leftarrow (n_R, Y, \mathsf{Enc}(k_e, (id, \sigma, \tau)))}$
return $(st', m')$
```

```
$\mathsf{RunInit2}(id, sk, st, peerpk, m)$:
$(n_R, Y, peerid, \sigma, \tau) \leftarrow m$
$\boxed{(n_R, Y, c) \leftarrow m}$
$(n_I, X, x) \leftarrow st.state$
$st'.sid \leftarrow (n_I, n_R, X, Y)$
$mk \leftarrow \mathrm{RO}(n_I \| n_R \| X \| Y \| Y^x)$
$k_s \leftarrow \mathsf{PRF}(mk, 0)$
$k_t \leftarrow \mathsf{PRF}(mk, 1)$
$\boxed{k_e \leftarrow \mathsf{PRF}(mk, 2)}$
$\boxed{(peerid, \sigma, \tau) \leftarrow \mathsf{Dec}(k_e, c)}$
$st'.peerid \leftarrow peerid$
if $\mathsf{S.Vrfy}(peerpk[peerid], \mathsf{L}_{rs} \| n_I \| n_R \| X \| Y, \sigma)$
and $\mathsf{M.Vrfy}(k_t, \mathsf{L}_{rm} \| n_I \| n_R \| peerid, \tau)$ then
    $st'.status \leftarrow$ accepted
    $st'.skey \leftarrow k_s$
    $\sigma' \leftarrow \mathsf{S.Sign}(sk, \mathsf{L}_{is} \| n_I \| n_R \| X \| Y)$
    $\tau' \leftarrow \mathsf{M.Tag}(k_t, \mathsf{L}_{im} \| n_I \| n_R \| id)$
    $m' \leftarrow (id, \sigma', \tau')$
    $\boxed{m' \leftarrow \mathsf{Enc}(k_e, (id, \sigma', \tau'))}$
else
    $m' \leftarrow \bot$
    $st'.status \leftarrow$ rejected
return $(st', m')$
```

```
$\mathsf{RunResp2}(id, sk, st, peerpk, m)$:
$(n_I, n_R, X, Y, k_s, k_t) \leftarrow st.state$
$\boxed{(n_I, n_R, X, Y, k_s, k_t, k_e) \leftarrow st.state}$
$(peerid, \sigma', \tau') \leftarrow m$
$\boxed{(peerid, \sigma', \tau') \leftarrow \mathsf{Dec}(k_e, m)}$
$st'.peerid \leftarrow peerid$
if $\mathsf{S.Vrfy}(peerpk[peerid], \mathsf{L}_{is} \| n_I \| n_R \| X \| Y, \sigma')$
and $\mathsf{M.Vrfy}(k_t, \mathsf{L}_{im} \| n_I \| n_R \| peerid, \tau')$ then
    $st'.status \leftarrow$ accepted
    $st'.skey \leftarrow k_s$
else $st'.status \leftarrow$ rejected
$m' \leftarrow \varepsilon$
return $(st', m')$
```

Figure 2.8. The formalized SIGMA/SIGMA-I key exchange protocols (cf. Section 2.2.1). $\boxed{\text{Boxed}}$ code is only performed in the SIGMA-I variant.

three elements are encrypted using k_e to conceal the responder's identity against passive adversaries.

Initiator Step 2. The initiator also computes mk and keys k_s, k_t, and (in SIGMA-I, used to decrypt the second message part) k_e. It ensures both the received signature σ and MAC τ verify, and aborts otherwise.

It computes its own signature σ' under sk_I on nonces and DH shares (with a different label L_{is}) and a MAC τ' under k_t over the nonces and its identity I (with yet another label L_{im}). It sends I, σ', and τ' to the responder (in SIGMA-I encrypted under k_e) and accepts with session key k_s using the nonces and DH shares (n_I, n_R, X, Y) as session identifier.

Responder Step 2. The responder finally checks the initiator's signature σ' and MAC τ' (aborting if either fails) and then accepts with session key $skey = k_s$ and session identifier $sid = (n_I, n_R, X, Y)$.

2.6 Tighter Security Proof for SIGMA-I

We now come to our first main result, a tighter security proof for the SIGMA-I protocol. Note that by omitting message encryption our proof similarly applies to the basic SIGMA protocol.

Theorem 5. *Let the SIGMA-I protocol be as specified in Figure 2.8 based on a group $\mathbb{G}$ of prime order p, a PRF* PRF*, a signature scheme* S*, and a MAC* M*, and let* RO *in the protocol be modeled as a random oracle. For any $(t, q_{\mathrm{N}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}})$-*$\mathsf{KE\text{-}SEC}$*-adversary against SIGMA-I making at most q_{RO} queries to* RO*, we give algorithms $\mathscr{B}_1$, $\mathscr{B}_2$, $\mathscr{B}_3$, and $\mathscr{B}_4$ in the proof, with running times $t_{\mathscr{B}_1} \approx t + 2q_{\mathrm{RO}} \log_2 p$ and $t_{\mathscr{B}_i} \approx\ t$ (for $i = 2, \ldots, 4$) close to that of $\mathscr{A}$, such that*

$$\begin{aligned}&\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{SIGMA\text{-}I}}(t, q_{\mathrm{N}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}})\\ &\quad\leq \frac{3q_{\mathrm{S}}^2}{2^{nl+1} \cdot p} + \mathbf{Adv}^{\mathsf{stDH}}_{\mathbb{G}}(t_{\mathscr{B}_1}, q_{\mathrm{RO}}) + \mathbf{Adv}^{\mathsf{mu\text{-}PRF}}_{\mathsf{PRF}}(t_{\mathscr{B}_2}, q_{\mathrm{S}}, 3q_{\mathrm{S}}, 3)\\ &\qquad+ \mathbf{Adv}^{\mathsf{mu\text{-}EUF\text{-}CMA}}_{\mathsf{S}}(t_{\mathscr{B}_3}, q_{\mathrm{N}}, q_{\mathrm{S}}, q_{\mathrm{S}}, q_{\mathrm{RL}}) + \mathbf{Adv}^{\mathsf{mu\text{-}EUF\text{-}CMA}}_{\mathsf{M}}(t_{\mathscr{B}_4}, q_{\mathrm{S}}, q_{\mathrm{S}}, 1, q_{\mathrm{S}}, 1, 0).\end{aligned}$$

Here, nl is the nonce length in SIGMA-I and $\mathbb{G}$ is the used Diffie–Hellman group of prime order p.

In terms of multi-user security for the employed primitives, multi-user PRF and MAC security can be obtained tightly, e.g., via the efficient AMAC construction [29], and multi-user signature security can be generically reduced to single-user security of any signature scheme with a loss in the number of users, here parties (not sessions) in the key exchange game.

Proof:

Our proof of key exchange security for SIGMA-I proceeds via a sequence of code-based games [42]. For the first half, the proof conceptually follows the strategy put forward by Cohn-Gordon et al. [73].

Game 0. The initial game, G_0, is the key exchange security game played by $\mathscr{A}$ (cf. Figure 2.1), using the KGen, $\mathsf{Activate}$, and Run routines of SIGMA-I defined in Figure 2.8. Therefore,

$$\Pr[G_0 \Rightarrow 1] = \Pr[G^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE},\mathscr{A}} \Rightarrow 1].$$

Game 1. Between G_0 and G_1 (Figure 2.9), we make internal changes to the record-keeping of the game, namely we track the nonces and group elements chosen and received by honest sessions. Whenever two honest sessions pick the same nonce or group element, we set a flag $\mathsf{bad}[C]$. Whenever an honest responder session picks a nonce and group element that has already been received by an initiator session, we set a flag $\mathsf{bad}[O]$. This change is unobservable by the adversary, hence

$$\Pr[G_0 \Rightarrow 1] = \Pr[G_1 \Rightarrow 1].$$

Game 2. In Game G_2 (Figure 2.9), we abort whenever nonces and group elements collide among honest sessions (i.e., the $\mathsf{bad}[C]$ flag is set), or whenever an honest responder session chooses a nonce and group element already submitted by the adversary to an initiator (i.e.,

```
$G_1$, $\boxed{G_2}$

RunInit1($id, sk, st$):
    $n_I \xleftarrow{\$} \{0,1\}^{nl}$
    $x \xleftarrow{\$} \mathbb{Z}_p$
    $X \leftarrow g^x$
    if $(n_I, X) \in N$ then $\mathsf{bad}[C] \leftarrow \mathsf{true}$ $\boxed{\text{; abort}}$
    $N \leftarrow N \cup \{(n_I, X)\}$
    $st'.state \leftarrow (n_I, X, x)$
    $m' \leftarrow (n_I, X)$
    return $(st', m')$

RunInit2($id, sk, st, peerpk, m$):
    $(n_R, Y, c) \leftarrow m$
    $Recv \leftarrow Recv \cup \{(n_R, Y)\}$
    $(n_I, X, x) \leftarrow st.state$
    ...

RunResp1($id, sk, st, peerpk, m$):
    $(n_I, X) \leftarrow m$
    $n_R \xleftarrow{\$} \{0,1\}^{nl}$
    $y \xleftarrow{\$} \mathbb{Z}_p$
    $Y \leftarrow g^x$
    if $(n_R, Y) \in Recv$ then $\mathsf{bad}[O] \leftarrow \mathsf{true}$ $\boxed{\text{; abort}}$
    if $(n_R, Y) \in N$ then $\mathsf{bad}[C] \leftarrow \mathsf{true}$ $\boxed{\text{; abort}}$
    $N \leftarrow N \cup \{(n_R, Y)\}$
    $st'.sid \leftarrow (n_I, n_R, X, Y)$
    ...

RO($m$):
    if $H[m] = \perp$ then $H[m] \xleftarrow{\$} \{0,1\}^{kl}$
    return $H[m]$
```

Figure 2.9. Games G_1 (changes highlighted in gray) and G_2 (changes highlighted in $\boxed{\text{frames}}$) of the SIGMA-I proof; with the explicit (lazy-sampled) random oracle RO.

the $\mathsf{bad}[O]$ flag is set). By the identical-until-bad lemma [42],

$$\Pr[G_1 \Rightarrow 1] - \Pr[G_2 \Rightarrow 1] \leq \Pr[\mathsf{bad}[C] \text{ or } \mathsf{bad}[O] \leftarrow \mathsf{true} \text{ in } G_1].$$

In all of the calls to RunInit1 and RunResp1, up to q_S pairs of nonces and group elements are chosen uniformly at random. By the birthday bound, the probability of a collision between two of these pairs setting the $\mathsf{bad}[C]$ flag is at most $\frac{q_S^2}{2^{nl+1} \cdot p}$ (where nl is the nonce length and p the order of the Diffie–Hellman group). There are at most q_S pairs received by initiator sessions, so the probability that a responder session randomly chooses one of these pairs is at most $\frac{q_S}{2^{nl} \cdot p}$; then by the union bound we have that $\Pr[\mathsf{bad}[O] \leftarrow \mathsf{true} \text{ in } G_1] \leq \frac{q_S^2}{2^{nl} \cdot p}$. Since each of RunInit1 and RunResp1 is called at most once per Send query, if an adversary makes q_S queries to its Send oracle, then

$$\Pr[G_1 \Rightarrow 1] - \Pr[G_2 \Rightarrow 1] \leq \frac{3q_S^2}{2^{nl+1} \cdot p}.$$

In all subsequent games, we are now sure that each honest session chooses a unique nonce and group element. Since the session identifier $sid = (n_I, n_R, X, Y)$ contains exactly one initiator and one responder nonce, this furthermore implies that when two honest sessions are partnered, they must have different roles.

Game 3. In Game G_3 (Figure 2.10), we remove the now superfluous collision flags $\mathsf{bad}[C]$ and $\mathsf{bad}[O]$ and add additional bookkeeping. All honest initiator sessions now log their outgoing messages in an internal table Sent. Honest responder sessions use this table to check if the message they received was sent by an honest initiator session. If so, they log their keys k_t, k_e, and k_s in a second internal table, S, indexed by their session identifier. These changes are unobservable by the adversary, so

$$\Pr[G_2 \Rightarrow 1] = \Pr[G_3 \Rightarrow 1].$$

Game 4. In Game G_4 (Figure 2.10), we require that initiator sessions whose key material has already been computed by an honest partner session simply copy their partners' key material. When an honest initiator session π_u^i with nonce n and group element X receives a message $m \leftarrow (n_R, Y, c)$, it sets its session identifier $sid \leftarrow (n_I, n_R, X, Y)$. It then checks if $S[sid] \neq \perp$ (which is only the case if π_u^i has an honest partner), and if so uses the stored key material $k_s, k_t, k_e \leftarrow S[st'.sid]$. Recall that both partnered sessions agree on the DH shares X and Y as components of sid. They hence also agree on the shared DH secret $Z = g^{xy}$ and thus on the master key derived as $\mathrm{RO}(n_I \| n_R \| X \| Y \| Z)$ as well as the derived key k_s, k_t, and k_e. For the adversary $\mathscr{A}$ it is hence unobservable if initiators with honest partner actually compute their keys themselves or copy their partners' key material in Game G_4, so

$$\Pr[G_3 \Rightarrow 1] = \Pr[G_4 \Rightarrow 1].$$

Game 5. In Game G_5 (Figure 2.11), all honest sessions sample their master keys uniformly at random (as long as the random oracle has not been been queried on the corresponding input) and program the random oracle to that value (through RO's internal table $H[n_I \| n_R \| X \| Y \| Y^x] \leftarrow mk$). This is equivalent to RO performing the same checks and uniform sampling, and hence undetectable for $\mathscr{A}$:

$$\Pr[G_4 \Rightarrow 1] = \Pr[G_5 \Rightarrow 1].$$

```
$\mathrm{G}_3$, $\boxed{\mathrm{G}_4}$

$\mathsf{RunInit1}(id, sk, st)$:
$n_I \xleftarrow{\$} \{0,1\}^{nl}$
$x \xleftarrow{\$} \mathbb{Z}_p$
$X \leftarrow g^x$
if $(n_I, X) \in N$ then abort
$N \leftarrow N \cup \{(n_I, X)\}$
$st'.state \leftarrow (n_I, X, x)$
$m' \leftarrow (n_I, X)$
$\mathsf{Sent} \leftarrow \mathsf{Sent} \cup m'$
return $(st', m')$

$\mathsf{RunInit2}(id, sk, st, peerpk, m)$:
$(n_R, Y, c) \leftarrow m$
$Recv \leftarrow Recv \cup \{(n_R, Y)\}$
$(n_I, X, x) \leftarrow st.state$
$st'.sid \leftarrow (n_I, n_R, X, Y)$
if $S[st'.sid] \neq \bot$ then
    $mk \leftarrow \mathrm{RO}(n_I \| n_R \| X \| Y \| Y^x)$
    $k_s \leftarrow \mathsf{PRF}(mk, 0)$
    $k_t \leftarrow \mathsf{PRF}(mk, 1)$
    $k_e \leftarrow \mathsf{PRF}(mk, 2)$
    $\boxed{k_s, k_t, k_e \leftarrow S[st'.sid]}$
else
    $mk \leftarrow \mathrm{RO}(n_I \| n_R \| X \| Y \| Y^x)$
    $k_s \leftarrow \mathsf{PRF}(mk, 0)$
    $k_t \leftarrow \mathsf{PRF}(mk, 1)$
    $k_e \leftarrow \mathsf{PRF}(mk, 2)$
$(peerid, \sigma, \tau) \leftarrow \mathsf{Dec}(k_e, c)$
$st'.peerid \leftarrow peerid$
...

$\mathsf{RunResp1}(id, sk, st, peerpk, m)$:
$(n_I, X) \leftarrow m$
$n_R \xleftarrow{\$} \{0,1\}^{nl}$
$y \xleftarrow{\$} \mathbb{Z}_p$
$Y \leftarrow g^x$
if $(n_R, Y) \in Recv$ then abort
if $(n_R, Y) \in N$ then abort
$N \leftarrow N \cup \{(n_R, Y)\}$
$st'.sid \leftarrow (n_I, n_R, X, Y)$
$\sigma \leftarrow \mathsf{S.Sign}(sk, \mathsf{L}_{rs} \| n_I \| n_R \| X \| Y)$
$mk \leftarrow \mathrm{RO}(n_I \| n_R \| X \| Y \| X^y)$
$k_s \leftarrow \mathsf{PRF}(mk, 0)$
$k_t \leftarrow \mathsf{PRF}(mk, 1)$
$k_e \leftarrow \mathsf{PRF}(mk, 2)$
if $m \in \mathsf{Sent}$ then
    $S[st'.sid] \leftarrow (k_s, k_t, k_e)$
$\tau \leftarrow \mathsf{M.Tag}(k_t, \mathsf{L}_{rm} \| n_I \| n_R \| id)$
$st'.state \leftarrow (n_I, n_R, X, Y, k_s, k_t)$
$m' \leftarrow (n_R, Y, \mathsf{Enc}(k_e, (id, \sigma, \tau)))$
return $(st', m')$
```

Figure 2.10. Games G_3 (changes highlighted in gray) and G_4 (changes highlighted in $\boxed{\text{frames}}$) of the SIGMA-I proof.

Game 6. In Game G_6 (Figure 2.11), responder sessions whose first message came from an honest initiator stop programming the random oracle on their uniformly chosen master key mk. This is undetectable for adversary $\mathscr{A}$ unless it makes a query $\mathrm{RO}(n_I \| n_R \| X \| Y \| Z)$, where $sid = (n_I, n_R, X, Y)$ is the session identifier shared by two honest partnered sessions, and Z is the Diffie–Hellman secret corresponding to the pair (X, Y). We call this event F, and bound the probability of F by giving a reduction $\mathscr{B}_1$ (specified in Figure 2.12) to the strong Diffie–Hellman assumption in the DH group $\mathbb{G}$. The reduction makes at most as many queries to its stDH oracle as $\mathscr{A}$ makes to its RO oracle, as follows.

Given its strong DH challenge $(A = g^a, B = g^b)$ and having access to an oracle $\mathsf{stDH}_a(U,V)$ which outputs 1 if $U^a = V$ and 0 otherwise, $\mathscr{B}_1$ simulates G_6 for an adversary $\mathscr{A}$ as follows. In all honest initiator sessions, $\mathscr{B}_1$ embeds its challenge into the sent DH value as $X \leftarrow A \cdot g^r$, where $r \in \mathbb{Z}_p$ is sampled uniformly at random for each session. Furthermore, in all responder sessions receiving their first message from an honest initiator, $\mathscr{B}_1$ embeds its challenge as $Y \leftarrow B \cdot g^{r'}$, where $r' \in \mathbb{Z}_p$ is sampled uniformly at random for each session.

Let us first observe that if event F occurs, then the value Z in the random oracle query $\mathrm{RO}(n_I \| n_R \| X \| Y \| Z)$ will equal $g^{(a+r)(b+r')}$ for some $r, r' \in \mathbb{Z}_p$ chosen by $\mathscr{B}_1$, and consequently

$$Z \cdot Y^{-r} = g^{(a+r)(b+r')-(b+r')\cdot r} = g^{a(b+r')} = Y^a.$$

This equality can be tested for by $\mathscr{B}_1$ by calling its stDH_a oracle on the pair $(Y, Z \cdot Y^{-r})$. We let $\mathscr{B}_1$ do so whenever $\mathscr{A}$ queries RO on some value $(n_I \| n_R \| X \| Y \| Z)$ where $(n_I, X = A \cdot g^r)$ was output by an honest initiator session and $(n_R, Y = g^{(b+r')})$ was output by a responder session with an honest initiator; the responder stores (n_I, n_R, X, Y) in a look-up table Q so this can be checked efficiently. If $\mathsf{stDH}_a(Y, Z \cdot Y^{-r}) = 1$ on such occasion, i.e., event F occurs, $\mathscr{B}_1$ stops with output $Z \cdot Y^{-r} \cdot A^{-r'} = g^{(a+r)(b+r')} \cdot g^{-(b+r')\cdot r} \cdot g^{-ar'} = g^{ab}$ and wins. Therefore,

$$\Pr[F] \leq \mathbf{Adv}_{\mathrm{G}}^{\mathrm{stDH}}(\mathscr{B}_1).$$

One subtlety in this step is ensuring that $\mathscr{B}_1$ can correctly simulate answers to RevSessionKey queries to any initiator or responder session. We do so by accordingly programming the random oracle on the sampled master key, where needed. First of all observe that responder sessions without honest initiator keep picking their own Y share and compute mk regularly. Initiator and responder sessions with honest partner have the challenge embedded and sample an independent master key which is not programmed to the random oracle. However, $\mathscr{B}_1$ stops and wins (as described above) if $\mathscr{A}$ ever queries the random oracle on the correct DH secret; i.e., $\mathscr{A}$ never sees the (inconsistent) random oracle output for these master keys. The interesting case is when an initiator session (which always embeds the challenge in its DH share as $X = A \cdot g^r$) obtains a message (n_R, Y, c) *not* originating from an honest responder: Here, Y may well have been picked by the adversary who could furthermore have corrupted the initiator's peer and hence make the initiator accept—with a master key it cannot compute itself.

We therefore let $\mathscr{B}_1$ attempt to copy the adversary's master key, if it has been computed. The RO oracle logs all queries it receives by their putative session id (n_I, n_R, X, Y) in a look-up table H', so $\mathscr{B}_1$ can efficiently access all Z such that (n_I, n_R, X, Y, Z) has been queried to RO. Since the DH secret corresponding to the pair (X, Y) equals Y^{a+r}, if Z is this DH secret, then

$$Z \cdot Y^{-r} = Y^{(a+r)-r} = Y^{a}.$$

The reduction can check this equality using its stDH_a oracle and in that case use the response to $\mathrm{RO}(n_I, n_R, X, Y, Z)$ as mk. Otherwise, $\mathscr{B}_1$ samples mk at random and stores it in the table Q (Line 48 of Figure 2.12), indicating it should be programmed in the random oracle later if queried on a matching Z value (Line 75). This ensures all responses to RevSessionKey queries are consistent with $\mathscr{A}$'s queries to the random oracle RO.

Observe that, in all this, $\mathscr{B}_1$ calls its stDH oracle at most once for each entry $H[n_I \| n_R \| X \| Y \| Z] = mk$ in the RO table H. In RO, stDH is called (once) only for entries that were not present when $\mathsf{Q}[(n_I, n_R, X, Y)]$ was set, then H' is set. In $\mathsf{RunInit2}$ and $\mathsf{RunResp1}$, stDH is called only for matching H' entries established prior to setting Q. Therefore, if stDH is called in RO for an entry, it was not called in either $\mathsf{RunInit2}$ or $\mathsf{RunResp1}$. If stDH is called on an entry in $\mathsf{RunResp1}$, then the responder session is partnered, so its partner will copy its keys in $\mathsf{RunInit2}$ and not call stDH. Furthermore, due to uniqueness of nonces and DH shares (by Game G_2), no $\mathsf{RunInit2}$ or $\mathsf{RunResp1}$ call makes stDH be invoked twice for the same H' entry.

Since the total time to iterate through the for loops over all Run and RO queries is at most $O(q_{\mathrm{RO}})$, the running time of $\mathscr{B}_1$ is roughly that of $\mathscr{A}$, plus the time needed to compute the arguments of the stDH queries. Each of these arguments requires one group operation and one exponentiation. (All other operations performed by $\mathscr{B}_1$ add only a small constant amount of time per Send query, which is dominated by the runtime of $\mathscr{A}$.) The exponentiation can be computed using $2\log_2 p$ group operations using the square-and-multiply (or double-and-add) algorithm, so

G_5, $\boxed{\mathrm{G}_6}$

```
RunInit2($id, sk, st, peerpk, m$):
$(n_R, Y, c) \leftarrow m$
$Recv \leftarrow Recv \cup \{(n_R, Y)\}$
$(n_I, X, x) \leftarrow st.state$
$st'.sid \leftarrow (n_I, n_R, X, Y)$
if $S[st'.sid] \neq \bot$ then
    $k_s, k_t, k_e \leftarrow S[st'.sid]$
else
    $mk \xleftarrow{\$} \{0,1\}^{kl}$
    if $H[n_I \| n_R \| X \| Y \| Y^x] \neq \bot$
        $mk \leftarrow H[n_I \| n_R \| X \| Y \| Y^x]$
    $H[n_I \| n_R \| X \| Y \| Y^x] \leftarrow mk$
    $k_s \leftarrow \mathsf{PRF}(mk, 0)$
    $k_t \leftarrow \mathsf{PRF}(mk, 1)$
    $k_e \leftarrow \mathsf{PRF}(mk, 2)$
$(peerid, \sigma, \tau) \leftarrow \mathsf{Dec}(k_e, c)$
$st'.peerid \leftarrow peerid$
if $\mathsf{S.Vrfy}(peerpk[peerid], \mathsf{L}_{rs} \| n_I \| n_R \| X \| Y, \sigma)$
and $\mathsf{M.Vrfy}(k_t, \mathsf{L}_{rm} \| n_I \| n_R \| peerid)$ then
    $st'.status \leftarrow \mathsf{accepted}$
    $st'.skey \leftarrow k_s$
    $\sigma' \leftarrow \mathsf{S.Sign}(sk, \mathsf{L}_{is} \| n_I \| n_R \| X \| Y)$
    $\tau' \leftarrow \mathsf{M.Tag}(k_t, \mathsf{L}_{im} \| n_I \| n_R \| id)$
    $m' \leftarrow \mathsf{Enc}(k_e, (id, \sigma', \tau'))$
else
    $m' \leftarrow \bot$ ; $st'.status \leftarrow \mathsf{rejected}$
return $(st', m')$
```

```
RunResp1($id, sk, st, peerpk, m$):
$(n_I, X) \leftarrow m$
$n_R \xleftarrow{\$} \{0,1\}^{nl}$
$y \xleftarrow{\$} \mathbb{Z}_p$
$Y \leftarrow g^x$
if $(n_R, Y) \in Recv$ then abort
if $(n_R, Y) \in N$ then abort
$N \leftarrow N \cup \{(n_R, Y)\}$
$st'.sid \leftarrow (n_I, n_R, X, Y)$
$\sigma \leftarrow \mathsf{S.Sign}(sk, \mathsf{L}_{rs} \| n_I \| n_R \| X \| Y)$
$mk \xleftarrow{\$} \{0,1\}^{kl}$
$\boxed{\text{if } m \notin \mathsf{Sent} \text{ then}}$
    if $H[n_I \| n_R \| X \| Y \| X^y] \neq \bot$
        $mk \leftarrow H[n_I \| n_R \| X \| Y \| X^y]$
    $H[n_I \| n_R \| X \| Y \| X^y] \leftarrow mk$
$k_s \leftarrow \mathsf{PRF}(mk, 0)$
$k_t \leftarrow \mathsf{PRF}(mk, 1)$
$k_e \leftarrow \mathsf{PRF}(mk, 2)$
if $m \in \mathsf{Sent}$ then
    $S[st'.sid] \leftarrow (k_s, k_t, k_e)$
$\tau \leftarrow \mathsf{M.Tag}(k_t, \mathsf{L}_{rm} \| n_I \| n_R \| id)$
$st'.state \leftarrow (n_I, n_R, X, Y, k_s, k_t)$
$m' \leftarrow (n_R, Y, id, \sigma, \tau)$
return $(st', m')$
```

Figure 2.11. Games G_5 (changes highlighted in gray) and G_6 (changes highlighted in $\boxed{\text{frames}}$) of the SIGMA-I proof.

$t_{\mathscr{B}_1} \approx t + 2q_{\mathrm{RO}} \log_2 p$. The runtime t of $\mathscr{A}$ already includes the computation of $2q_{\mathrm{S}} \log_2 p$ group operations, so this is a significant but not prohibitive increase in runtime.

Having $\mathscr{B}_1$ perfectly simulate Game G_5 for $\mathscr{A}$ up to the point when F happens, and G_6 and G_5 differing only when F happens, we have

$$\Pr[\mathrm{G}_5 \Rightarrow 1] = \Pr[\mathrm{G}_6 \Rightarrow 1] + \Pr[F] \leq \Pr[\mathrm{G}_6 \Rightarrow 1] + \mathbf{Adv}^{\mathsf{stDH}}_{\mathbb{G}}()(t_{\mathscr{B}_1}, q_{\mathrm{RO}}),$$

and $t_{\mathscr{B}_1} \approx t + 2q_{\mathrm{RO}} \log_2 p$.

Game 7. In Game G_7 (Figure 2.13), responder oracles responding to honest messages samples

```
$\mathscr{B}_1(A,B)^{\mathsf{stDH}_a(\cdot,\cdot)}$

$\mathsf{RunInit1}(id, sk, st)$:
$n_I \xleftarrow{\$} \{0,1\}^{nl}$
$r \xleftarrow{\$} \mathbb{Z}_p$
$X \leftarrow A \cdot g^r$
if $(n_I, X) \in N$ then abort
$N \leftarrow N \cup \{(n_I, X)\}$
$st'.state \leftarrow (n_I, X, r)$
$m' \leftarrow (n_I, X)$
$\mathsf{Sent}[m'] \leftarrow x$
return $(st', m')$

$\mathsf{RunInit2}(id, sk, st, peerpk, m)$:
$(n_R, Y, c) \leftarrow m$
$Recv \leftarrow Recv \cup \{(n_R, Y)\}$
$(n_I, X, r) \leftarrow st.state$
$st'.sid \leftarrow (n_I, n_R, X, Y)$
if $S[st'.sid] \neq \bot$ then
    $k_s, k_t, k_e \leftarrow S[st'.sid]$
else
    $mk \xleftarrow{\$} \{0,1\}^{kl}$
    for each $Z \in H'[n_I \| n_R \| X \| Y]$
        if $\mathsf{stDH}_a(Y, Z \cdot Y^{-r}) = 1$ then
            $mk \leftarrow H[n_I \| n_R \| X \| Y \| Z]$
    $\mathsf{Q}[st'.sid] \leftarrow (r, \bot, mk)$
    $k_s \leftarrow \mathsf{PRF}(mk, 0)$
    $k_t \leftarrow \mathsf{PRF}(mk, 1)$
    $k_e \leftarrow \mathsf{PRF}(mk, 2)$
$(peerid, \sigma, \tau) \leftarrow \mathsf{Dec}(k_e, c)$
$st'.peerid \leftarrow peerid$
if $\mathsf{S.Vrfy}(\mathsf{L}_{rs} \| n_I \| n_R \| X \| Y, \sigma, pk_{peerid})$
and $\mathsf{M.Vrfy}(k_t, \mathsf{L}_{rm} \| n_I \| n_R \| peerid)$ then
    $st'.status \leftarrow \mathsf{accepted}$
    $st'.skey \leftarrow k_s$
    $\sigma' \leftarrow \mathsf{S.Sign}(sk, \mathsf{L}_{is} \| n_I \| n_R \| R \| W)$
    $\tau' \leftarrow \mathsf{M.Tag}(k_t, \mathsf{L}_{im} \| n_I \| n_R \| id)$
    $m' \leftarrow \mathsf{Enc}(k_e, (id, \sigma', \tau'))$
else
    $m' \leftarrow \bot$
    $st'.status \leftarrow \mathsf{rejected}$
return $(st', m')$
```

```
$\mathsf{RunResp1}(id, sk, st, peerpk, m)$:
$(n_I, X) \leftarrow m$
$n_R \xleftarrow{\$} \{0,1\}^{nl}$
$r' \xleftarrow{\$} \mathbb{Z}_p$
$mk \xleftarrow{\$} \{0,1\}^{kl}$
if $m \in \mathsf{Sent}$ then
    $r \leftarrow \mathsf{Sent}[m]$
    $Y \leftarrow B \cdot g^{r'}$
    $st'.sid \leftarrow (n_I, n_R, X, Y)$
    for each $Z \in H'[n_I \| n_R \| X \| Y]$
        if $\mathsf{stDH}_a(Y, Z \cdot Y^{-r}) = 1$ then
            $\textsc{Finalize}(Z \cdot Y^{-r} \cdot A^{-r'})$
    $\mathsf{Q}[st'.sid] \leftarrow (r, r', mk)$
else
    $Y \leftarrow g^{r'}$
    $st'.sid \leftarrow (n_I, n_R, X, Y)$
    if $H[n_I \| n_R \| X \| Y \| X^y] \neq \bot$
        $mk \leftarrow H[n_I \| n_R \| X \| Y \| X^{r'}]$
    $H[n_I \| n_R \| X \| Y \| X^y] \leftarrow mk$
if $(n_R, Y) \in Recv$ then abort
if $(n_R, Y) \in N$ then abort
$N \leftarrow N \cup \{(n_R, Y)\}$
$\sigma \leftarrow \mathsf{S.Sign}(sk, \mathsf{L}_{rs} \| n_I \| n_R \| X \| Y)$
$k_s \leftarrow \mathsf{PRF}(mk, 0)$
$k_t \leftarrow \mathsf{PRF}(mk, 1)$
$k_e \leftarrow \mathsf{PRF}(mk, 2)$
if $m \in \mathsf{Sent}$ then
    $S[st'.sid] \leftarrow (k_s, k_t, k_e)$
$\tau \leftarrow \mathsf{M.Tag}(k_t, \mathsf{L}_{rm} \| n_I \| n_R \| id)$
$st'.state \leftarrow (n_I, n_R, X, Y, k_s, k_t, k_e)$
$m' \leftarrow (n_R, Y, \mathsf{Enc}(k_e, (id, \sigma, \tau)))$
return $(st', m')$

$\mathrm{RO}(m)$:
if $H[m] = \bot$ then
    $H[m] \xleftarrow{\$} \{0,1\}^{kl}$
    parse $n_I \| n_R \| X \| Y \| Z \leftarrow m$
    $H'[n_I \| n_R \| X \| Y] \leftarrow H'[n_I \| n_R \| X \| Y] \cup \{Z\}$
    if $\mathsf{Q}[(n_I, n_R, X, Y)] \neq \bot$ then
        $(r, r', mk) \leftarrow \mathsf{Q}[n_I, n_R, X, Y]$
        if $\mathsf{stDH}_a(Y, Z \cdot Y^{-r}) = 1$ then
            if $r' = \bot$ then $H[m] \leftarrow mk$
            else $\textsc{Finalize}(Z \cdot Y^{-r} \cdot A^{-r'})$
return $H[m]$
```

Figure 2.12. Reduction $\mathscr{B}_1$ to the strong Diffie–Hellman assumption of the SIGMA-I proof. Sections highlighted in gray have been significantly altered compared to Game G_6.

$\underline{\mathrm{G}_7}$

```
RunResp1$(id, sk, st, peerpk, m)$:
$(n_I, X) \leftarrow m$
$n_R \xleftarrow{\$} \{0,1\}^{nl}$
$y \xleftarrow{\$} \mathbb{Z}_p$
$Y \leftarrow g^y$
if $(n_R, Y) \in Recv$ then abort
if $(n_R, Y) \in N$ then abort
$N \leftarrow N \cup \{(n_R, Y)\}$
$st'.sid \leftarrow (n_I, n_R, X, Y)$
$\sigma \leftarrow \mathsf{S.Sign}(sk, \mathsf{L}_{rs}\|n_I\|n_R\|X\|Y)$
$mk \xleftarrow{\$} \{0,1\}^{kl}$
if $m \notin \mathsf{Sent}$ then
    if $H[Y^x\|n_I\|n_R\|X\|Y] \neq \perp$
        $mk \leftarrow H[Y^x\|n_I\|n_R\|X\|Y]$
$k_s \leftarrow \mathsf{PRF}(mk, 0)$
$k_t \leftarrow \mathsf{PRF}(mk, 1)$
$k_e \leftarrow \mathsf{PRF}(mk, 2)$
if $m \in \mathsf{Sent}$
    $k_s \xleftarrow{\$} \{0,1\}^{kl}$
    $k_t \xleftarrow{\$} \{0,1\}^{kl}$
    $k_e \xleftarrow{\$} \{0,1\}^{kl}$
    $S[st'.sid] \leftarrow (k_s, k_t, k_e)$
$\tau \leftarrow \mathsf{M.Tag}(k_t, \mathsf{L}_{rm}\|n_I\|n_R\|id)$
$st'.state \leftarrow (n_I, n_R, X, Y, k_s, k_t)$
$m' \leftarrow (n_R, Y, \mathsf{Enc}(k_e, (id, \sigma, \tau))$
return $(st', m')$
```

$\underline{\mathscr{B}_2^{\mathrm{FN}(\cdot,\cdot)}}$

```
RunResp1$(id, sk, st, peerpk, m)$:
$(n_I, X) \leftarrow m$
$n_R \xleftarrow{\$} \{0,1\}^{nl}$
$y \xleftarrow{\$} \mathbb{Z}_p$
$Y \leftarrow g^y$
if $(n_R, Y) \in Recv$ then abort
if $(n_R, Y) \in N$ then abort
$N \leftarrow N \cup \{(n_R, Y)\}$
$st'.sid \leftarrow (n_I, n_R, X, Y)$
$\sigma \leftarrow \mathsf{S.Sign}(sk, \mathsf{L}_{rs}\|n_I\|n_R\|X\|Y)$
$mk \xleftarrow{\$} \{0,1\}^{kl}$
if $m \notin \mathsf{Sent}$ then
    if $H[n_I\|n_R\|X\|Y\|X^y] \neq \perp$
        $mk \leftarrow H[n_I\|n_R\|X\|Y\|X^y]$
$k_s \leftarrow \mathsf{PRF}(mk, 0)$
$k_t \leftarrow \mathsf{PRF}(mk, 1)$
$k_e \leftarrow \mathsf{PRF}(mk, 2)$.
if $m \in \mathsf{Sent}$
    $\mathrm{NEW}()$; $i{+}{+}$
    $k_s \leftarrow \mathrm{FN}(i, 0)$
    $k_t \leftarrow \mathrm{FN}(i, 1)$
    $k_e \leftarrow \mathrm{FN}(i, 2)$
    $S[st'.sid] \leftarrow (k_s, k_t, k_e)$
$\tau \leftarrow \mathsf{M.Tag}(k_t, \mathsf{L}_{rm}\|n_I\|n_R\|id)$
$st'.state \leftarrow (n_I, n_R, X, Y, k_s, k_t)$
$m' \leftarrow (n_R, Y, \mathsf{Enc}(k_e, (id, \sigma, \tau))$
return $(st', m')$
```

Figure 2.13. Game G_7 and reduction $\mathscr{B}_2$ to PRF security of the SIGMA-I proof. Changes from G_6 resp. compared to G_7 highlighted in gray .

session, MAC, and encryption keys k_s, k_t, and k_e randomly instead of computing them through a PRF. (Initiator oracles partnered with an honest responder will continue to copy those, now randomly sampled keys.)

Since the PRF key mk in this case is sampled independently of the random oracle and the rest of the game, this reduces straightforwardly to the multi-user security of the PRF via the reduction $\mathscr{B}_2$ we give in Figure 2.13. The adversary $\mathscr{B}_2$ makes one NEW and two FUNC queries for each RunResp1 query, or three FUNC queries in SIGMA-I. Notably, it makes at most three FUNC queries per user, and no CORRUPT queries because mk is never revealed to the adversary. Outside of the oracle calls, its running time exactly equals that of $\mathscr{A}$ in Game G_6, as their pseudocode is identical, so $t_{\mathscr{B}_2} \approx t$. Using its FN oracle of the PRF game, $\mathscr{B}_2$ perfectly simulates G_6 if the oracle gives real-PRF answers and G_7 if it returns uniformly random values. Therefore,

$$\Pr[\mathrm{G}_6 \Rightarrow 1] \leq \Pr[\mathrm{G}_7 \Rightarrow 1] + \mathbf{Adv}^{\text{mu-PRF}}_{\text{PRF}}()(t_{\mathscr{B}_2}, q_{\mathrm{S}}, 3q_{\mathrm{S}}, 3, 0).$$

Observe that from now on, session and MAC keys of responder oracles that received honest initiator's messages are chosen independently at random, and that initiator oracles with matching *sid* will copy those keys. Notably, this is the case even for sessions whose (own or peer's) long-term secret have been revealed to the adversary. We will use these properties in the following to argue authentication of sessions as well as forward security of the session keys.

Our final game hops are concerned with the explicit authentication performed through signatures and MACs in the SIGMA-I protocol, and as such extend those proof steps for implicit authentication of the main protocols in [73].

Game 8. In Game G_8 (Figure 2.14), we log all messages for which signatures are generated by an honest session, and set a bad flag $\mathsf{bad}[S]$ if the adversary submits a valid signature under an uncorrupted signing key for a message which was not produced by an honest session. This internal bookkeeping does not affect the adversary's advantage, so

$$\Pr[\mathrm{G}_7 \Rightarrow 1] = \Pr[\mathrm{G}_8 \Rightarrow 1].$$

Game 9. In Game G_9 (Figure 2.14), we abort if the $\mathsf{bad}[S]$ flag is set. By the identical-until-bad lemma, the difference in advantage between G_8 and G_9 is bounded by the probability that this event occurs, which we reduce via an algorithm $\mathscr{B}_3$ to the multi-user security of the digital signature scheme S.

In the reduction, $\mathscr{B}_3$ obtains all long-term public keys from the multi-user signature game and uses its signing oracles for any honest signature to be produced. It therefore makes q_{N} queries to NEW and one Sign query for each call to RunResp1 or RunInit2, for at most q_{S} such queries. It relays REVLONGTERMKEY queries as corruptions in its multi-user game, making q_{RL} corruption

```
$\mathrm{G}_8$, $\boxed{\mathrm{G}_9}$

$\mathsf{RunInit2}(id, sk, st, peerpk, m)$:
  ...
  if $\mathsf{S.Vrfy}(peerpk[peerid], \mathsf{L}_{rs}\|n_I\|n_R\|X\|Y, \sigma)$
  and $\mathsf{M.Vrfy}(k_t, \mathsf{L}_{rm}\|n_I\|n_R\|peerid, \tau)$ then
    if $\mathsf{revltk}_{peerid} = \infty$ and
        $(peerid, \mathsf{L}_{rs}\|n_I\|n_R\|X\|Y) \notin \mathsf{Q}_S$ then
      $\mathsf{bad}[S] \leftarrow \mathsf{true}$ $\boxed{\text{; abort}}$
    $st'.status \leftarrow \mathsf{accepted}$
    $st'.skey \leftarrow k_s$
    $\sigma' \leftarrow \mathsf{S.Sign}(sk, \mathsf{L}_{is}\|n_I\|n_R\|X\|Y)$
    $\mathsf{Q}_S \leftarrow \mathsf{Q}_S \cup \{(id, \mathsf{L}_{is}\|n_I\|n_R\|X\|Y)\}$
    $\tau' \leftarrow \mathsf{M.Tag}(k_t, \mathsf{L}_{im}\|n_I\|n_R\|id)$
  ...

$\mathsf{RunResp1}(id, sk, st, peerpk, m)$:
  ...
  $\sigma \leftarrow \mathsf{S.Sign}(sk, \mathsf{L}_{rs}\|n_I\|n_R\|X\|Y)$
  $\mathsf{Q}_S \leftarrow \mathsf{Q}_S \cup \{(id, \mathsf{L}_{rs}\|n_I\|n_R\|X\|Y)\}$
  ...
  $\tau \leftarrow \mathsf{M.Tag}(k_t, \mathsf{L}_{rm}\|n_I\|n_R\|id)$
  ...

$\mathsf{RunResp2}(id, sk, st, peerpk, m)$:
  ...
  if $\mathsf{S.Vrfy}(peerpk[peerid], \mathsf{L}_{is}\|n_I\|n_R\|X\|Y, \sigma')$
  and $\mathsf{M.Vrfy}(k_t, \mathsf{L}_{im}\|n_I\|n_R\|peerid, \tau')$ then
    if $\mathsf{revltk}_{peerid} = \infty$ and
        $(peerid, \mathsf{L}_{rs}\|n_I\|n_R\|X\|Y) \notin \mathsf{Q}_S$ then
      $\mathsf{bad}[S] \leftarrow \mathsf{true}$ $\boxed{\text{; abort}}$
    $st'.status \leftarrow \mathsf{accepted}$
    $st'.skey \leftarrow k_s$
  else $st'.status \leftarrow \mathsf{rejected}$
  return $(st', m')$
```

```
$\mathrm{G}_{10}$, $\boxed{\mathrm{G}_{11}}$

$\mathsf{RunInit2}(id, sk, st, peerpk, m)$:
  ...
  if $\mathsf{S.Vrfy}(peerpk[peerid], \mathsf{L}_{rs}\|n_I\|n_R\|X\|Y, \sigma)$
  and $\mathsf{M.Vrfy}(k_t, \mathsf{L}_{rm}\|n_I\|n_R\|peerid)$ then
    if $\mathsf{revltk}_{peerid} = \infty$ and
        $(peerid, \mathsf{L}_{rs}\|n_I\|n_R\|X\|Y) \notin \mathsf{Q}_S$ then
      abort
    if $S[st'.sid] \neq \perp$ and
        $(st'.sid, \mathsf{L}_{rm}\|n_I\|n_R\|peerid) \notin \mathsf{Q}_M$ then
      $\mathsf{bad}[M] \leftarrow \mathsf{true}$ $\boxed{\text{; abort}}$
    $st'.status \leftarrow \mathsf{accepted}$
    $st'.skey \leftarrow k_s$
    $\sigma' \leftarrow \mathsf{S.Sign}(sk, \mathsf{L}_{is}\|n_I\|n_R\|X\|Y)$
    $\mathsf{Q}_S \leftarrow \mathsf{Q}_S \cup \{(id, \mathsf{L}_{is}\|n_I\|n_R\|X\|Y)\}$
    $\tau' \leftarrow \mathsf{M.Tag}(k_t, \mathsf{L}_{im}\|n_I\|n_R\|id)$
    if $S[st'.sid] \neq \perp$ then
      $\mathsf{Q}_M \leftarrow \mathsf{Q}_M \cup \{(st'.sid, \mathsf{L}_{im}\|n_I\|n_R\|id)\}$
  ...

$\mathsf{RunResp1}(id, sk, st, peerpk, m)$:
  ...
  $\sigma \leftarrow \mathsf{S.Sign}(sk, \mathsf{L}_{rs}\|n_I\|n_R\|X\|Y)$
  $\mathsf{Q}_S \leftarrow \mathsf{Q}_S \cup \{(id, \mathsf{L}_{rs}\|n_I\|n_R\|X\|Y)\}$
  ...
  $\tau \leftarrow \mathsf{M.Tag}(k_t, \mathsf{L}_{rm}\|n_I\|n_R\|id)$
  if $S[st'.sid] \neq \perp$ then
    $\mathsf{Q}_M \leftarrow \mathsf{Q}_M \cup \{(st'.sid, \mathsf{L}_{rm}\|n_I\|n_R\|id)\}$
  ...

$\mathsf{RunResp2}(id, sk, st, peerpk, m)$:
  ...
  if $\mathsf{S.Vrfy}(peerpk[peerid], \mathsf{L}_{is}\|n_I\|n_R\|X\|Y, \sigma')$
  and $\mathsf{M.Vrfy}(k_t, \mathsf{L}_{im}\|n_I\|n_R\|peerid, \tau')$ then
    if $\mathsf{revltk}_{peerid} = \infty$ and
          $(peerid, \mathsf{L}_{is}\|n_I\|n_R\|X\|Y) \notin \mathsf{Q}_S$ then
      abort
    if $S[st'.sid] \neq \perp$ and
        $(st'.sid, (peerid, \mathsf{L}_{im}\|n_I\|n_R\|peerid) \notin \mathsf{Q}_M$ then
      $\mathsf{bad}[M] \leftarrow \mathsf{true}$ $\boxed{\text{; abort}}$
    $st'.status \leftarrow \mathsf{accepted}$
    $st'.skey \leftarrow k_s$
  else $st'.status \leftarrow \mathsf{rejected}$
  return $(st', m')$
```

Figure 2.14. Games G_8, G_9, G_{10}, and G_{11} of the SIGMA-I proof. Changes in G_8 and G_{10} are highlighted in gray, changes in G_9 and G_{11} are highlighted in $\boxed{\text{frames}}$.

queries in total. When $\mathsf{bad}[S]$ is triggered, $\mathscr{B}_3$ submits the triggering message and signature under the targeted (uncorrupted) public key as its forgery. As the triggering message was not signed before under the corresponding secret key (and hence not queried to the signing oracle by $\mathscr{B}_3$), the forgery is valid and $\mathscr{B}_3$ wins if $\mathsf{bad}[S]$ is set. It follows that

$$\Pr[\mathrm{G}_8 \Rightarrow 1] \leq \Pr[\mathrm{G}_9 \Rightarrow 1] + \mathbf{Adv}_{\mathsf{S}}^{\text{mu-EUF-CMA}}(\mathscr{B}_3)(t_{\mathscr{B}_3}, q_{\mathrm{N}}, q_{\mathrm{S}}, q_{\mathrm{S}}, q_{\mathrm{RL}}).$$

Except for the replacement of key generation, signatures, corruptions with oracle queries, the pseudocode of $\mathscr{B}_3$ is identical to that of $\mathscr{A}$ in game G_8, so $t_{\mathscr{B}_3} \approx t$.

Game 10. In Game G_{10} (Figure 2.14), we remove the now redundant $\mathsf{bad}[S]$ flag again, and log all MAC tags generated by honest sessions with honest partners in a list Q_M (using, as before, the table S to determine whether a session has an honest partner). We set a flag $\mathsf{bad}[M]$ if a session with an honest partner receives a valid MAC tag which was not computed by any honest oracle. This bookkeeping is similar to the changes from G_7 to G_8, but noting MAC tags instead of signatures. As before, the bookkeeping itself does not affect the adversary's advantage:

$$\Pr[\mathrm{G}_9 \Rightarrow 1] = \Pr[\mathrm{G}_{10} \Rightarrow 1].$$

Game 11. In Game G_{11} (Figure 2.14), we abort if the $\mathsf{bad}[M]$ flag is set to true. Again applying the identical-until-bad lemma, we need to bound the probability of $\mathsf{bad}[M]$ being set in G_{10}, which we do via the following reduction $\mathscr{B}_4$ to the multi-user EUF-CMA security of the MAC scheme M.

The reduction $\mathscr{B}_4$ simulates G_{10} truthfully, except that for any session with honest origin partner (i.e., session with state st where $S[st.sid] \neq \perp$), $\mathscr{B}_4$ does not compute k_t itself, but instead assigns an incremented user identifier i to this session's sid and computes any calls to Tag or Vrfy using its corresponding oracles for user i. There is at most one query to NEWUSER, and one each to Tag and Vrfy for each of $\mathscr{A}$'s queries to SEND. Hence $\mathscr{B}_4$ makes at most q_{S} queries to each of these three oracles, and at most one query to Tag and Vrfy per user in the mu-EUF-CMA game. When $\mathsf{bad}[M]$ is triggered, $\mathscr{B}_4$ submits the triggering message and MAC tag under user identifier i as its forgery. In the simulation, sessions will share a user identifier i if and only if they are partnered and would share keys in Game G_{10}. These keys are furthermore unique to one initiator and one responder session only, so consistency is maintained. Furthermore, k_t cannot be exposed (by RevLongTermKey or RevSessionKey) to adversary $\mathscr{A}$, hence implicitly replacing it with the MAC game's oracles is sound, and $\mathscr{B}_4$ makes no Corrupt queries. Except for oracle replacements, the pseudocode of $\mathscr{B}_4$ is identical to that of $\mathscr{A}$ in G_{10}, so $t_{\mathscr{B}_4} \approx t$.

If $\mathsf{bad}[M]$ is triggered, then $S[st'.sid] \neq \perp$, so $st'.sid$ corresponds to some user identifier i in the multi-user EUF-CMA game. Additionally, a tag τ for message m was verified under identity i, and $(st'.sid, m)$ was not logged in Q_M. Since $\mathscr{B}_4$ logs $(st'.sid, m)$ every time it calls its Tag oracle on the pair (i, m), this call cannot have occurred. Then τ is a valid forgery on m, which $\mathscr{B}_4$ will output for user i to win the EUF-CMA game. Thus,

$$\Pr[\mathrm{G}_{10} \Rightarrow 1] \leq \Pr[\mathrm{G}_{11} \Rightarrow 1] + \mathbf{Adv}_{\mathsf{M}}^{\text{mu-EUF-CMA}}()(t_{\mathscr{B}_4}, q_{\mathrm{S}}, q_{\mathrm{S}}, 1, q_{\mathrm{S}}, 1, 0).$$

We can now consider the final advantage of an adversary playing Game G_{11}. Adversary $\mathscr{A}$ has a non-zero advantage if in the final oracle query $\textsc{fin}(b')$

1. Sound is false,
2. $\mathsf{ExplicitAuth}$ is false, or
3. Fresh is true and $b' = b$.[5]

Soundness.

The flag Sound is set if (1) three honest sessions hold the same session identifier, or if (2) two partnered sessions hold different session keys.

For (1): No three honest sessions can share the same session identifiers, as this would require a collision in either the contained initiator or responder nonce, which is excluded by Game G_2.

[5] If Fresh is false, $b = b' = 0$ happens with probability $\frac{1}{2}$, so $\mathscr{A}$'s advantage is 0.

For (2): The session identifier includes both nonces n and n_R and DH shares X and Y, which together determine the derived master key $mk = \mathrm{RO}(n_I \| n_R \| X \| Y \| Z)$ (where Z is the DH secret from X and Y) and thus the session key. Agreement on the session identifier hence implies deriving the same session key.

Hence, in Game G_{11}, Sound is always true.

Explicit authentication.

The predicate $\mathsf{ExplicitAuth}$ requires that for any session π_u^i accepting with a non-compromised peer v, there exists a partnered session π_v^j of user v with opposite role which, if it accepts, has u set as its peer.

The session π_u^i, prior to accepting, obtained a valid signature on $\pi_u^i.sid$ and a label corresponding to a role $r \neq \pi_u^i.role$. Due to Game G_9, this signature must have been issued by an honest session π_v^j (since v was not compromised at this point). All honest sessions sign their own sid and a label corresponding to their own role, so $\pi_u^i.sid = \pi_v^j.sid$ and $\pi_u^i.role = r \neq \pi_v^j.role$ are satisfied.

Furthermore, when π_v^j accepts, it must have received a valid MAC tag τ on a label identifying an opposite-role session and that session's user identity, as well as their shared nonces. Due to Game G_{11}, this MAC value must have been computed by an honest session holding the same nonces, as π_v^j has an honest partner session and therefore $S[\pi_v^j.sid] \neq \perp$. Furthermore, by Game G_2, nonces do not collide and hence that session must have been π_u^i, thus computing the MAC on user identity u, which π_v^j accordingly sets as peer identity.

Therefore $\mathsf{ExplicitAuth}$ is always true in G_{11}. Note that we did not require that the long-term key of user u was uncorrupted, and we allow the adversary to continue interacting with sessions after compromise; hence covering key compromise impersonation attacks.

Guessing the challenge bit.

Finally, we have to consider $\mathscr{A}$'s chance of guessing the challenge bit b, which it may only learn through TEST queries such that all tested sessions are fresh (i.e., Fresh is true).

The Fresh predicate being true ensures that all tested sessions (those in T) accepted prior to their respective partner being corrupt. Then, as $\mathsf{ExplicitAuth}$ is true, we have that for each tested session there exists an honest session with the same *sid* and different roles. This session, by Fresh, was not tested or revealed. Being partnered, the first message (n_I, X) between these two honest sessions was not tampered with, so in the responder session, whether it was the tested session or its partner, the master and session keys are sampled uniformly at random (due to Games G_6 and G_7). Since the initiator session holds the same *sid*, it copied the responder's random session key (due to Game G_4). This random session key was not revealed in either of the two sessions (by Fresh), and hence from $\mathscr{A}$'s perspective is a uniformly random and independent value. In all TEST oracle responses, k_0 and k_1 are hence identically distributed and so G_{11} is fully independent of b. It follows that the adversary $\mathscr{A}$ has no better than a $\frac{1}{2}$ probability of choosing b' equal to b, so

$$\Pr[\mathrm{G}_{11} \Rightarrow 1] = \frac{1}{2},$$

which concludes the proof. ▮

2.7 The TLS 1.3 Handshake Protocol

The Transport Layer Security (TLS) protocol in version 1.3 [186] bases its key exchange design (the so-called handshake protocol) on a variant of SIGMA-I. Following the core SIGMA design, the TLS 1.3 main handshake is an ephemeral Diffie–Hellman key exchange, authenticated through a combination of signing and MAC-ing the (full, hashed) communication transcript.[6] Additionally, and similar to SIGMA-I, beyond establishing the main (application traffic) session key, handshake traffic keys are derived and used to encrypt part of the handshake.

Beyond additional protocol features like negotiating the cryptographic algorithms to be used, communicating further information in extensions, etc.—which we do not capture here—, TLS 1.3 however deviates in two core cryptographic aspects from the more simplistic and abstract SIGMA(-I) design: it hashes the communication transcript when deriving keys and computing signatures and MACs, and it uses a significantly more complicated key schedule. In this section we revisit the TLS 1.3 handshake and discuss the careful technical changes and additional

[6] TLS 1.3 also specifies an abbreviated resumption-style handshake based on pre-shared keys; we focus on the main DH-based handshake in this work.

assumptions needed to translate our tight security results for SIGMA-I to TLS 1.3's main key exchange mode.

2.7.1 Protocol Description

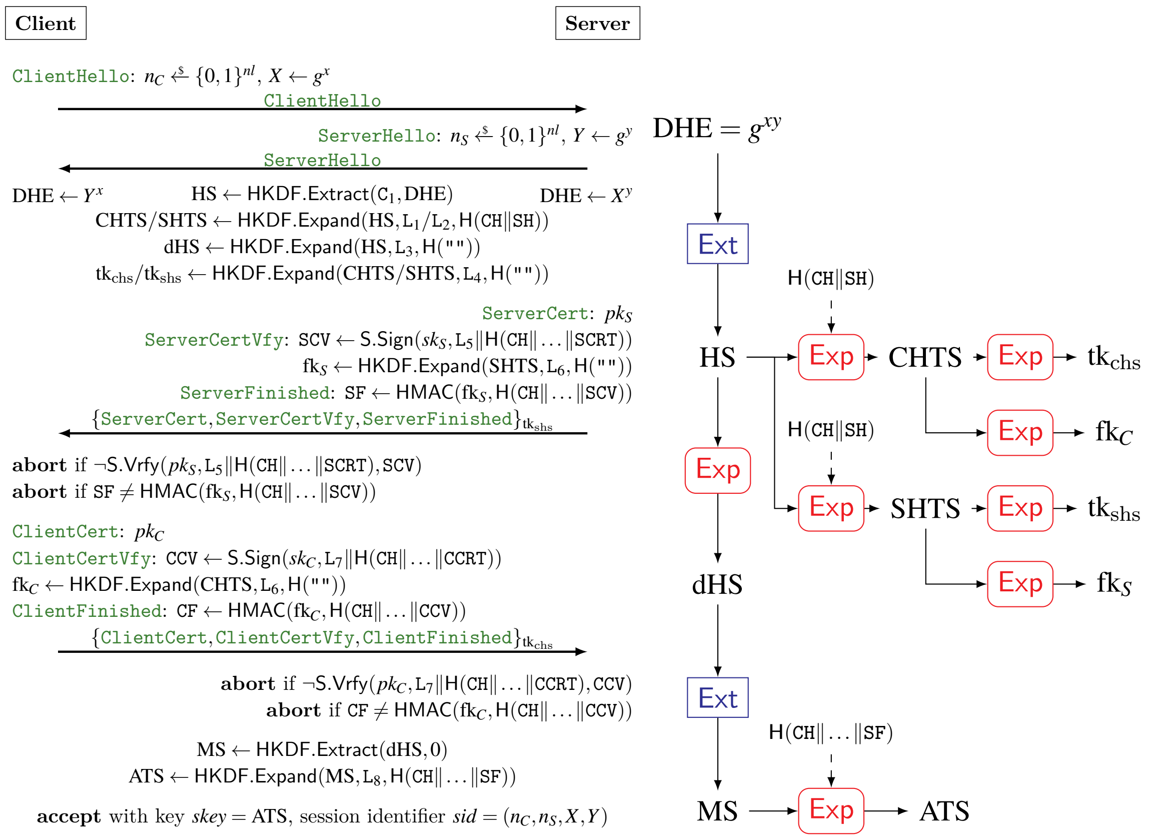
We focus on a slightly simplified version of the handshake encompassing all essential cryptographic aspects for our tightness results. In particular, we only consider mutual authentication and security of the main application traffic keys (see [93, 95, 101, 92] for full computational, multi-stage key exchange analyses of the different modes with varying authentication) and accordingly leave out some computations and additional messages. To ease linking back to the underlying SIGMA-I structure, we describe the protocol in the following referencing back to the latter (cf. Section 2.5). We illustrate the handshake protocol and its accompanying key schedule in Figure 2.15, the latter deriving keys in the extract-then-expand paradigm of the HKDF key derivation function [141].[7]

In the TLS 1.3 handshake, the client acts as initiator and the server as responder. Within `Hello` messages, both send nonce values n_C resp. n_S together with ephemeral Diffie–Hellman shares g^x resp. g^y. Based on these values, both parties extract a handshake secret HS from the shared DH value $\mathsf{DHE} = g^{xy}$ using $\mathsf{HKDF.Extract}$ with a constant salt input.[8] In a second step, client and server derive their respective handshake traffic keys $\mathsf{tk}_{\mathrm{chs}}$, $\mathsf{tk}_{\mathrm{shs}}$ and MAC keys fk_C, fk_S through two levels of $\mathsf{HKDF.Expand}$ steps from the handshake secret HS, including in the first level distinct labels and the hashed communication transcript $\mathsf{H}(\mathtt{CH}\|\mathtt{SH})$ so far as context information.

The handshake traffic keys are then used to encrypt the remaining handshake messages. First the server, then the client send their certificate (carrying their identity and public key), a signature over the hashed transcript up to including their certificate ($\mathsf{H}(\mathtt{CH}\|\ldots\|\mathtt{SCRT})$, resp. $\mathsf{H}(\mathtt{CH}\|\ldots\|\mathtt{CCRT})$), as well as a MAC over the (hashed) transcript up to incl. their signatures ($\mathsf{H}(\mathtt{CH}\|\ldots\|\mathtt{SCV})$, resp. $\mathsf{H}(\mathtt{CH}\|\ldots\|\mathtt{CCV})$). Note the similarity to SIGMA-I here: each party signs

[7] We follow the standard HKDF syntax: $\mathsf{HKDF.Extract}(\mathit{XTS}, \mathit{SKM})$ on input salt XTS and source key material SKM outputs a pseudorandom key PRK. $\mathsf{HKDF.Expand}(\mathit{PRK}, \mathit{CTXinfo})$ on input a pseudorandom key PRK and context information $\mathit{CTXinfo}$ outputs pseudorandom key material KM.

[8] This salt input becomes relevant for pre-shared key handshakes, but in the full handshake takes the constant value $\mathtt{C}_1 = \mathsf{Expand}(\mathsf{Extract}(0,0), \texttt{"derived"}, \mathsf{H}(\texttt{""}))$.


Client
Server
ClientHello: $n_C \xleftarrow{\$} \{0,1\}^{nl}$, $X \leftarrow g^x$
ClientHello
ServerHello: $n_S \xleftarrow{\$} \{0,1\}^{nl}$, $Y \leftarrow g^y$
ServerHello
$\mathrm{DHE} \leftarrow Y^x$
$\mathsf{HS} \leftarrow \mathsf{HKDF.Extract}(\mathsf{C}_1, \mathrm{DHE})$
$\mathrm{DHE} \leftarrow X^y$
$\mathsf{CHTS}/\mathsf{SHTS} \leftarrow \mathsf{HKDF.Expand}(\mathsf{HS}, \mathsf{L}_1/\mathsf{L}_2, \mathsf{H}(\mathtt{CH}\|\mathtt{SH}))$
$\mathsf{dHS} \leftarrow \mathsf{HKDF.Expand}(\mathsf{HS}, \mathsf{L}_3, \mathsf{H}(\texttt{""}))$
$\mathsf{tk_{chs}}/\mathsf{tk_{shs}} \leftarrow \mathsf{HKDF.Expand}(\mathsf{CHTS}/\mathsf{SHTS}, \mathsf{L}_4, \mathsf{H}(\texttt{""}))$
ServerCert: pk_S
ServerCertVfy: $\mathtt{SCV} \leftarrow \mathsf{S.Sign}(sk_S, \mathsf{L}_5\|\mathsf{H}(\mathtt{CH}\|\dots\|\mathtt{SCRT}))$
$\mathsf{fk}_S \leftarrow \mathsf{HKDF.Expand}(\mathsf{SHTS}, \mathsf{L}_6, \mathsf{H}(\texttt{""}))$
ServerFinished: $\mathtt{SF} \leftarrow \mathsf{HMAC}(\mathsf{fk}_S, \mathsf{H}(\mathtt{CH}\|\dots\|\mathtt{SCV}))$
$\{\mathtt{ServerCert}, \mathtt{ServerCertVfy}, \mathtt{ServerFinished}\}_{\mathsf{tk_{shs}}}$
abort if $\neg\mathsf{S.Vrfy}(pk_S, \mathsf{L}_5\|\mathsf{H}(\mathtt{CH}\|\dots\|\mathtt{SCRT}), \mathtt{SCV})$
abort if $\mathtt{SF} \neq \mathsf{HMAC}(\mathsf{fk}_S, \mathsf{H}(\mathtt{CH}\|\dots\|\mathtt{SCV}))$
ClientCert: pk_C
ClientCertVfy: $\mathtt{CCV} \leftarrow \mathsf{S.Sign}(sk_C, \mathsf{L}_7\|\mathsf{H}(\mathtt{CH}\|\dots\|\mathtt{CCRT}))$
$\mathsf{fk}_C \leftarrow \mathsf{HKDF.Expand}(\mathsf{CHTS}, \mathsf{L}_6, \mathsf{H}(\texttt{""}))$
ClientFinished: $\mathtt{CF} \leftarrow \mathsf{HMAC}(\mathsf{fk}_C, \mathsf{H}(\mathtt{CH}\|\dots\|\mathtt{CCV}))$
$\{\mathtt{ClientCert}, \mathtt{ClientCertVfy}, \mathtt{ClientFinished}\}_{\mathsf{tk_{chs}}}$
abort if $\neg\mathsf{S.Vrfy}(pk_C, \mathsf{L}_7\|\mathsf{H}(\mathtt{CH}\|\dots\|\mathtt{CCRT}), \mathtt{CCV})$
abort if $\mathtt{CF} \neq \mathsf{HMAC}(\mathsf{fk}_C, \mathsf{H}(\mathtt{CH}\|\dots\|\mathtt{CCV}))$
$\mathsf{MS} \leftarrow \mathsf{HKDF.Extract}(\mathsf{dHS}, 0)$
$\mathsf{ATS} \leftarrow \mathsf{HKDF.Expand}(\mathsf{MS}, \mathsf{L}_8, \mathsf{H}(\mathtt{CH}\|\dots\|\mathtt{SF}))$
accept with key $skey = \mathsf{ATS}$, session identifier $sid = (n_C, n_S, X, Y)$
$\mathrm{DHE} = g^{xy}$
Ext
$\mathsf{H}(\mathtt{CH}\|\mathtt{SH})$
HS
Exp
CHTS
Exp
$\mathsf{tk_{chs}}$
Exp
fk_C
$\mathsf{H}(\mathtt{CH}\|\mathtt{SH})$
Exp
Exp
SHTS
Exp
$\mathsf{tk_{shs}}$
Exp
fk_S
dHS
Ext
$\mathsf{H}(\mathtt{CH}\|\dots\|\mathtt{SF})$
MS
Exp
ATS

Protocol flow legend

MSG: Z	message MSG sent, containing Z
$\{\mathtt{MSG}\}_K$	message AEAD-encrypted with $K = \mathsf{tk_{shs}}/\mathsf{tk_{chs}}$

Message Abbreviations

CH	ClientHello
SH	ServerHello
CCRT/SCRT	Client/ServerCert
CCV/SCV	Client/ServerCertVfy
CF/SF	Client/ServerFinished

Figure 2.15. The simplified TLS 1.3 main Diffie–Hellman handshake protocol (left) and key schedule (right). Values L_i and C_i indicate bitstring labels, resp. constant values, (distinct per i). Boxes Ext and Exp denote HKDF extraction resp. expansion, dashed inputs to Exp indicating context information (see protocol figure for detailed computations).

both nonces and DH values (within $\mathtt{CH}\|\mathtt{SH}$, modulo transcript hashing) together with a unique label, and then MACs both nonces and their own identity (the latter being part of their certificate).[9] The application traffic secret ATS—which we treat as the session key *skey*, unifying secrets of both client and server—is then derived from the master secret MS through HKDF.Expand with handshake context up to the `ServerFinished` message. The master secret in turn is derived through (context-less) Expand and Extract from the handshake secret HS.

[9] Instead of using distinct labels for the client and server MAC computations, TLS 1.3 employs distinct MAC keys for client and server, achieving separation between the two MAC values this way.

2.7.2 Handling the TLS 1.3 Key Schedule

As mentioned before, the message flow of the TLS 1.3 handshake relatively closely follows the SIGMA-I design [138, 139] (cf. Figure 2.7): after exchanging nonces and DH shares (in `Hello`) from both sides, the remaining (encrypted) messages carry identities (`Certificate`), signatures over the nonces and DH shares (`CertificateVerify`), and MACs over the nonces and identities (`Finished`).

What crucially differentiates the TLS 1.3 handshake from the basic SIGMA-I design (beyond putting more under the respective signatures and MACs, which does not negatively affect the key exchange security we are after) is the way keys are derived. While SIGMA-I immediately derives a master key through a random oracle with input *both* the shared DH secret *and* the session identifying nonces and DH shares, TLS 1.3 separates them in its HKDF-based extract-then-expand key schedule: The core secrets—handshake secret (HS) and master secret (MS)—are derived without further context purely from the shared DH secret $\mathsf{DHE} = g^{xy}$ (beyond other constant inputs). Only when deriving the specific-purpose secrets—handshake traffic keys ($\mathrm{tk}_{\mathrm{chs}}$, $\mathrm{tk}_{\mathrm{shs}}$), MAC keys ($\mathrm{fk}_C$, fk_S), and session key (ATS)—is context added to the key derivation, including in particular the nonces and DH shares identifying the session. To complicate matters even further, this context is hashed before entering key derivation (or signature and MAC computation), and the final session key ATS depends on more messages than just the session-identifying ones. Since our tighter security proof for the SIGMA(-I) protocol (cf. Section 2.6) heavily makes use of (exactly) the session identifiers being input together with DH secrets to a random oracle when programming the latter, the question arises how to treat the TLS 1.3 key schedule when aiming at a similar proof strategy.

In their concurrent work, Diemert and Jager [88] satisfy this requirement by modeling the full derivation of each stage key in their multi-stage treatment as a separate random oracle. This directly connects inputs to keys, but results in a monolithic random oracle treatment of the key schedule which loses the independence of the intermediate $\mathsf{HKDF.Extract}$ and $\mathsf{HKDF.Expand}$ steps in translation.

We overcome the technical obstacle of this linking while staying closer to the structure of TLS 1.3's key schedule. First of all, we directly model both $\mathsf{HKDF.Extract}$ and $\mathsf{HKDF.Expand}$ as individual (programmable) random oracles, which leads to a slightly less excessive use of the random oracle technique. We then have to carefully orchestrate the programming of intermediate secrets and session keys in a two-level approach, connecting them through constant-time look-ups, and taking into account that inputs to deriving the session keys depend on values established through the intermediate secrets (namely, the server's `Finished` MAC). Along the way, we separately ensure that we recognize any hashed inputs of interest that the adversary might query to the random oracle, without modeling the hash function H as a random oracle itself. By tracking intermediate programming points (especially HS and MS) in the random oracles, we recover the needed capability of linking sessions and their session identifiers and DH shares exchanged to the corresponding session keys. This finally allows us to again (efficiently) determine when and on what input to query the strong Diffie–Hellman oracle when programming challenge DH shares into the TLS 1.3 key exchange execution during the proof.

2.8 Tighter Security Proof for the TLS 1.3 Handshake

We now give our second main result, the tighter-security bound for the TLS 1.3 handshake protocol.

Theorem 6. *Let $\mathscr{A}$ be a key exchange security adversary against the TLS 1.3 handshake protocol as specified in Figure 2.15 based on a hash function* H*, a signature scheme* S*, and a group* $\mathbb{G}$ *of prime order* p*, and let the* HKDF *functions* Extract *and* Expand *in the protocol be modeled as (independent) random oracles* RO_1*, resp.* RO_2*. For any* $(t, q_{\mathrm{N}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}})$*-*KE-SEC*-adversary against SIGMA-I making at most* q_{RO} *queries to the random oracle, we give algorithms* $\mathscr{B}_1$*,* $\mathscr{B}_2$*,* $\mathscr{B}_3$*, and* $\mathscr{B}_4$ *in the proof, with running times* $t_{\mathscr{B}_i} \approx t$ *(for* $i = 1,3,4$*) and* $t_{\mathscr{B}_2} \approx t + 2q_{\mathrm{RO}} \log_2 p$ *close to that of* $\mathscr{A}$*, such that*

$$\begin{aligned}\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{TLS\,1.3}}(t, q_{\mathrm{N}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}) &\leq \frac{3q_{\mathrm{S}}^2}{2^{nl+1} \cdot p} + \mathbf{Adv}^{\mathsf{CR}}_{\mathsf{H}}(t_{\mathscr{B}_1}) \\ &+ 2 \cdot \mathbf{Adv}^{\mathsf{stDH}}_{\mathbb{G}}(t_{\mathscr{B}_2}, q_{\mathrm{RO}}) + \frac{q_{\mathrm{RO}} \cdot q_{\mathrm{S}}}{2^{kl-1}} + \mathbf{Adv}^{\mathsf{mu\text{-}EUF\text{-}CMA}}_{\mathsf{S}}(t_{\mathscr{B}_3}, q_{\mathrm{N}}, q_{\mathrm{S}}, q_{\mathrm{S}}, q_{\mathrm{RL}}) \\ &+ \mathbf{Adv}^{\mathsf{mu\text{-}EUF\text{-}CMA}}_{\mathsf{HMAC}}(t_{\mathscr{B}_4}, q_{\mathrm{S}}, q_{\mathrm{S}}, 1, q_{\mathrm{S}}, 1, 0).\end{aligned}$$

Here, $nl = 256$ *is the nonce length in TLS 1.3,* kl *is the output length of* $\mathrm{RO}_2 = \mathsf{HKDF.Expand}$*,*

$\mathbb{G}$ *is the used Diffie–Hellman group of prime order* p*, and* $q_S \cdot q_{RO} \leq 2^{kl-3}$.[10]

Proof: We prove our bound by making an incremental series of changes to the key exchange security game and limiting the amount that each change affects the success probability of $\mathscr{A}$.

Game 0. The initial game, Game G_0, is the key exchange security game for TLS played by $\mathscr{A}$, using the implicit KGen, $\mathsf{Activate}$, and Run routines defined by the TLS protocol specification on the left side of Figure 2.15. (In this game, $\mathsf{HKDF.Extract}$ and $\mathsf{HKDF.Expand}$ are modeled by random oracles RO_1 and RO_2 respectively.) By definition,

$$\Pr[G_0 \Rightarrow 1] = \Pr[G^{\mathsf{KE\text{-}SEC}}_{\mathsf{TLS},\mathscr{A}} \Rightarrow 1].$$

Game 1. In game G_0, we start logging the nonces and group elements chosen by honest sessions. Whenever two honest sessions choose the same nonces or group elements, we set a flag $\mathsf{bad}[C]$. Whenever an honest responder session chooses a nonce and group element that have already been received by another session, we set a flag $\mathsf{bad}[O]$. We also make both random oracles RO_1 and RO_2 lazily sampled using internal tables H_1 and H_2. These changes only affect the values of the game's internal state, and the view of the adversary remains the same as in G_0, so

$$\Pr[G_1 \Rightarrow 1] = \Pr[G_0 \Rightarrow 1].$$

Game 2. Starting with G_2, we abort whenever two honest sessions sample the same nonce or group element and whenever an honest responder samples a nonce and group element that are already in use. Since this happens only after one of the flags $\mathsf{bad}[C]$ and $\mathsf{bad}[O]$ is set, by the identical-until-bad lemma,

$$\Pr[G_1 \Rightarrow 1] - \Pr[G_2 \Rightarrow 1] \leq \Pr[\mathsf{bad}[C] \leftarrow \mathsf{true} \text{ or } \mathsf{bad}[O] \leftarrow \mathsf{true} \text{ in } G_1].$$

[10] We simplify the factor on $\mathbf{Adv}^{\mathsf{stDH}}_{\mathbb{G}}$ to 2 by assuming $q_S \cdot q_{RO} \leq 2^{kl-3}$, which is true for any reasonable real-world parameters. See the proof for the exact bound.

One nonce and one group element is chosen in each $\mathsf{RunInit1}$ call and each $\mathsf{RunResp1}$ call, so at most one nonce and one group element is chosen for each of the q_{S} queries the adversary makes to its SEND oracle. We use the birthday bound to limit the probability of a collision (flag $\mathsf{bad}[C]$) in either the set of honest sessions' nonces or the set of honest sessions' DH shares to $\frac{q_{\mathrm{S}}^2}{2^{nl+1} \cdot p}$. Every time a responder session chooses a nonce and group element, there are at most q_{S} values have already been chosen, so by the union bound $\mathsf{bad}[O]$ is set with probability at most $\frac{q_{\mathrm{S}}^2}{2^{nl} \cdot p}$. Therefore

$$\Pr[\mathrm{G}_1 \Rightarrow 1] - \Pr[\mathrm{G}_2 \Rightarrow 1] \leq \frac{3q_{\mathrm{S}}^2}{2^{nl+1} \cdot p}.$$

Game 3. Next, we must ensure that partial transcripts between honest sessions do not collide under the hash function H. This is a step unique to the TLS proof, which hashes all of its context with a collision-resistant hash function before it is input into key-derivation. In G_3, honest sessions will log all of their hash outputs in a look-up table T: whenever an honest session computes $d = \mathsf{H}(s)$ for some string s, it sets $T[d] \leftarrow s$ if $T[d]$ has not already been defined. If $T[d]$ is not empty, then some prior honest session has computed $d = \mathsf{H}(s')$ for some string s'. The session will set a flag $\mathsf{bad}[\mathsf{H}]$ if $s' \neq s$, noting that a collision has occurred. We also remove the now superfluous $\mathsf{bad}[C]$ flag. These administrative changes do not affect the view of the adversary, so

$$\Pr[\mathrm{G}_3 \Rightarrow 1] = \Pr[\mathrm{G}_2 \Rightarrow 1].$$

Game 4. In Game G_4, we abort whenever hashes computed by honest sessions collide (i.e. the $\mathsf{bad}[\mathsf{H}]$ flag is set). By the identical-until-bad lemma,

$$\Pr[\mathrm{G}_3 \Rightarrow 1] - \Pr[\mathrm{G}_4 \Rightarrow 1] \leq \Pr[\mathsf{bad}[\mathsf{H}] \leftarrow \mathsf{true} \text{ in } \mathrm{G}_3].$$

We bound the probability that $\mathsf{bad}[\mathsf{H}]$ is set via a reduction $\mathscr{B}_1$ to the collision-resistance security of H. The reduction simulates G_3 honestly for the adversary $\mathscr{A}$. If the flag $\mathsf{bad}[\mathsf{H}]$ is set, then the reduction has obtained strings s, s', and d such that $s' \neq s$, and $\mathsf{H}(s) = \mathsf{H}(s') = d$. Then $\mathscr{B}_1$

outputs (s, s') and wins the collision-resistance game, so $\mathbf{Adv}^{\mathsf{cr}}_{\mathsf{H}}(\mathscr{B}_1) \geq \Pr[\mathsf{bad}[\mathsf{H}] \leftarrow \mathsf{true} \text{ in } \mathrm{G}_3]$. The runtime $t_{\mathscr{B}_1}$ of $\mathscr{B}_1$ approximately equals the runtime of $\mathscr{A}$ in G_3. It follows that

$$\Pr[\mathrm{G}_3 \Rightarrow 1] - \Pr[\mathrm{G}_4 \Rightarrow 1] \leq \mathbf{Adv}^{\mathsf{CR}}_{\mathsf{H}}(t_{\mathscr{B}_1}).$$

Game 5. In Game G_5, we remove the superfluous $\mathsf{bad}[\mathsf{H}]$ flag and make additional internal changes to the behavior of honest sessions. As in the SIGMA-I proof, all honest initatior sessions now log the first message they send in a set Sent, and honest responder sessions use this set to check whether their first received message came from an honest session without tampering. If so, we say the responder session has an “honest origin partner." In the SIGMA-I protocol, partnering between honest sessions was sufficient to ensure agreement on the derived master key and all subsequently computed keys, since partners are guaranteed to hold the same nonces and group elements. In TLS 1.3, partnering also ensures agreement on the handshake traffic secrets SHTS and CHTS, but it does not ensure agreement on the session key ATS. Therefore the responder only logs the handshake traffic keys $\mathsf{fk}_S, \mathsf{fk}_C, \mathsf{tk}_{\mathsf{shs}}$, and $\mathsf{tk}_{\mathsf{chs}}$ in a look-up table S under its session identifier. In addition to the session identifier, the application traffic secret ATS depends on the server's identity $\mathtt{SCRT}$, signature $\mathtt{SCV}$, and MAC tag $\mathtt{SF}$. These values are not necessarily shared by partner sessions in Game G_5, so two partnered sessions may derive different values of ATS. The responder session therefore logs its session key ATS in a second look-up table S' indexed by all of the dependencies of the session key: $sid, \mathtt{SCRT}, \mathtt{SCV}$, and $\mathtt{SF}$. All of this is just bookkeeping, so

$$\Pr[\mathrm{G}_5 \Rightarrow 1] = \Pr[\mathrm{G}_4 \Rightarrow 1].$$

Game 6. Going forward from Game G_6, honest initiators copy their key material from tables S and S' if it is consistent for them to do so. In the case where the adversary has tampered with the values of $\mathtt{SCRT}$, $\mathtt{SCV}$, or $\mathtt{SF}$, the partner's session key depends on the untampered values and should not be copied. Therefore honest initiators always copy encryption and MAC keys from the table S if they have an honest partner session, but they only copy ATS when the $\mathtt{SCRT}, \mathtt{SCV}$, and $\mathtt{SF}$ messages they received match the ones sent by their partner. The initiator session can check whether tampering occurred using the table S', which will contain a session key ATS at index $sid\|\texttt{SCRT}\|\texttt{SCV}\|\texttt{SF}$ if and only if the honest partner session computed and sent SCRT, SCV, and SF.

We argue that all copied keys are consistent with the keys that would be derived in G_5. Recall that partnered sessions agree on the nonces and the DH shares X and Y as components of sid, so they also agree on the shared DH secret Z associated with the pair (X,Y). Partnered sessions therefore agree on the handshake secret HS, which is derived from Z without context, and on the handshake traffic secrets, which are derived with the session identifier as context. Thus partnered sessions agree on the values of the handshake traffic keys $\mathsf{fk}_S, \mathsf{fk}_C, \mathsf{tk}_{\mathsf{shs}}$, and $\mathsf{tk}_{\mathsf{chs}}$ which are derived from the handshake traffic secrets. For the adversary it is hence unobservable if honest sessions compute the handshake traffic keys themselves, or copy the keys from their partners. By agreeing on the handshake secret HS, partnered sessions will also agree on the master secret MS, which is derived from HS without context. The if SCRT, SCV, and SF are left untampered, both sessions will derive the session key as $\mathrm{RO}_2(\mathsf{MS}, \mathtt{L}_8, \mathsf{H}(sid\|\texttt{SCRT}\|\texttt{SCV}\|\texttt{SF})])$. Hence it is again unobservable whether an honest initiator derives ATS itself or copies ATS from an honest partner which agrees on the values of SCRT, SCV, SF; consequently

$$\Pr[\mathrm{G}_6 \Rightarrow 1] = \Pr[\mathrm{G}_5 \Rightarrow 1].$$

Game 7. In Game G_7, all responders sample ATS, SHTS and CHTS randomly (unless their values have already been fixed by queries to random oracle RO_2 on the corresponding input), then retroactively programs random oracle RO_2 by setting its internal table H_2 on the appropriate input. Partnered initiator sessions which have not copied ATS (i.e., those who received tampered SCRT, SCV, and SF) also sample ATS randomly and program RO_2 when necessary. We choose to program ATS, SHTS, and CHTS, as opposed to only mk in the SIGMA-I proof, because these three keys are derived with context. Most importantly, the DH shares X and Y indirectly enter the key derivation for these keys, which will be critical for the reduction in the next step. This simply moves the lazy sampling process from RO_2 to RunResp1 and RunInit2 for certain queries, which is unobservable to the adversary; therefore

$$\Pr[G_7 \Rightarrow 1] = \Pr[G_6 \Rightarrow 1].$$

Game 8. The step between G_7 and G_8 is most technically involved step of this proof, and it is also the most significantly altered from the corresponding step in the proof of SIGMA-I. In G_8, partnered initiators and responder sessions with honest origin partners will stop maintaining the consistency of their keys ATS, SHTS, and CHTS with the random oracle RO_2. Specifically, responders with honest origin partners sample ATS, SHTS, and CHTS uniformly at random even if RO_2 has already been queried on the string $\mathsf{HS}, \mathtt{L}, d$ for the appropriate label and hash, and they do not retroactively program RO_2. Partnered initiator sessions which have not copied ATS from their partner also sample ATS uniformly without checking or programming RO_2. These keys are therefore completely random, and they will be inconsistent with any random oracle queries made before or after the keys are sampled.

In order to detect this inconsistency, the adversary must make a query to RO_2 that would, in G_7, return one of the unprogrammed keys. Which queries are these? They are the queries that an honest responder session with honest origin partner would use to derive SHTS, CHTS, and ATS, and the queries that an honest partnered initiator which received a tampered message would use to derive ATS. Formally, let $sid = (n, n', X, Y)$ be the session ID held by some honest responder session with honest origin partner, and let $\mathtt{SCRT}$, $\mathtt{SCV}$, $\mathtt{SF}$ be the identity, signature, and MAC tag sent by this session. Let DHE be the DH secret corresponding to the pair (X, Y). Then the adversary $\mathscr{A}$ can detect an inconsistency (in derviations of honest responders) in game G_8 if at any point during the game $\mathscr{A}$ queries RO_2 on one of the tuples

$$(\mathrm{RO}_1(\mathtt{C}_1, \mathsf{DHE}), \mathtt{L}, \mathsf{H}(sid)) \qquad \text{or} \qquad (\mathsf{MS}, \mathtt{L}_8, \mathsf{H}(sid \| \mathtt{SCRT} \| \mathtt{SCV} \| \mathtt{SF})),$$

where $\mathtt{L} \in \{\mathtt{L}_1, \mathtt{L}_2\}$ and where for some HS, dHS, we have that $\mathsf{HS} = \mathrm{RO}_1(\mathtt{C}_1, \mathsf{DHE})$, that $\mathsf{dHS} = \mathrm{RO}_2(\mathsf{HS}, \mathtt{L}_3, \mathsf{H}(\texttt{""}))$, and that $\mathsf{MS} = \mathrm{RO}_1(\mathsf{dHS}, 0)$. Otherwise (for derviations of honest initiators), let sid be the session ID held by an honest partnered initiator session, and let $\mathtt{SCRT}$, $\mathtt{SCV}$, and

$\mathtt{SF}$ be the identity, signature, and MAC tag received by that session. For initiator sessions that do not copy ATS, at least one of these values was not sent by the honest partner. Then the adversary $\mathscr{A}$ can detect an inconsistency in game G_8 if at any point it queries RO_2 on the tuple

$$(\mathsf{MS}, \mathtt{L}_8, \mathsf{H}(sid \| \mathtt{SCRT} \| \mathtt{SCV} \| \mathtt{SF})),$$

where for some HS, dHS, we have that $\mathsf{HS} = \mathrm{RO}_1(\mathtt{C}_1, \mathsf{DHE})$, that $\mathsf{dHS} = \mathrm{RO}_2(\mathsf{HS}, \mathtt{L}_3, \mathsf{H}(\texttt{""}))$, and that $\mathsf{MS} = \mathrm{RO}_1(\mathsf{dHS}, 0)$. Let event F denote the event that the adversary $\mathscr{A}$ makes at least one of the above queries. If event F does not occur, then ATS, SHTS, and CHTS are chosen uniformly at random in both G_7 and G_8, hence

$$\Pr[\mathrm{G}_7 \Rightarrow 1] - \Pr[\mathrm{G}_8] \Rightarrow 1] \leq \Pr[F \text{ occurs in } \mathrm{G}_7].$$

We bound the probability of event F via a reduction $\mathscr{B}_2$ to the strong Diffie–Hellman assumption in group $\mathbb{G}$. The reduction will make no more queries to its stDH oracle than $\mathscr{A}$ makes to its RO_2 oracles.

Given its strong DH challenge $(A = g^a, B = g^b)$ and having access to the strong Diffie–Hellman oracle stDH_a, $\mathscr{B}_2$ simulates G_7 for an adversary $\mathscr{A}$ in the following manner: In all honest initiator sessions, $\mathscr{B}_2$ samples r uniformly at random from $\mathbb{Z}_p$ and sets the session's DH share $X \leftarrow A \cdot g^r$. In all honest responder sessions with honest origin partner, $\mathscr{B}_2$ samples r' uniformly from $\mathbb{Z}_p$ and sets the session's DH share $Y \leftarrow B \cdot g^{r'}$. Both of these DH shares are still distributed uniformly over $\mathbb{Z}_p$ as long as p is prime and A and B are not the identity. To extract g^{ab} when event F occurs, the reduction $\mathscr{B}_2$ will follow the same general strategy as the reduction $\mathscr{B}_1$ in the proof of SIGMA-I, with four major points of divergence. We address these points first, before giving a full description of $\mathscr{B}_2$.

1. Since $\mathscr{B}_2$ no longer knows x or y such that $X = g^x$ or $Y = g^y$, it cannot compute the Diffie–Hellman secret DHE or the derived handshake secret HS, so it samples HS randomly for honest responder sessions with honest origin partners and for honest partnered initiator sessions. The adversary can only tell that HS was not correctly computed if it notices that

SHTS, CHTS, or dHS are derived from an incorrect value of HS. The former two cases require the adversary to make a query that triggers event F. In the latter case, dHS is not revealed to the adversary through any oracle, so the adversary must notice that ATS, which is derived indirectly from dHS via the master secret, is derived from an incorrect value of HS. This also requires $\mathscr{A}$ to make a query that triggers event F. Therefore, until event F occurs, this change is unobservable to the adversary.

2. In the TLS protocol, the context string, including the Diffie–Hellman shares X and Y, is hashed with H before it enters key derivation, so $\mathscr{B}_2$ cannot directly associate a query to RO_2 with the honest session(s) whose session ID is being used. The reduction addresses this by having each honest responder with honest origin partner and each honest partnered initiator, log the hash of its context in a reverse look-up table R. (The context does not include the handshake or master secrets.) Then in the RO_2 oracle, $\mathscr{B}_2$ can use R to efficiently check whether the hash d of a query is used to derive a handshake or application traffic key.

3. Due to TLS's complex key schedule, no one random oracle query contains both a pair of Diffie–Hellman shares and the DH secret associated with that pair. Instead, $\mathscr{B}_2$ will augment the RO_1 and RO_2 oracles to log in a reverse look-up table T the DH secret associated with each of the intermediate values HS, dHS, and MS. The DH secret for $\mathsf{dHS} = \mathrm{RO}_2(\mathsf{HS}, \mathtt{L}_3, \mathsf{H}(\texttt{""}))$ simply be copied from $T[\mathrm{HS}]$, and the DH secret for $\mathsf{MS} = \mathrm{RO}_1(\mathsf{dHS}, 0)$ will be copied from $T[\mathrm{dHS}]$. For each query to RO_2 with secret s, the reduction can efficiently check using T whether s was derived from some DH secret via RO_1.

4. The TLS key schedule uses multiple random oracle queries (if we model HKDF.Extract and HKDF.Expand as random oracles) whereas the SIGMA-I protocol uses only one. If $\mathscr{A}$ can guess the intermediate value $\mathsf{HS} = \mathrm{RO}_1(\mathtt{C}_1, \mathsf{DHE})$, where DHE is the DH secret associated to some pair of embedded shares (X, Y) chosen by honest sessions, then it can trigger event F without ever submitting DHE to an oracle. In this case, $\mathscr{A}$ can trigger event F, but $\mathscr{B}_2$ cannot win the Strong DH game. However, if $\mathrm{RO}_1(\mathtt{C}_1, \mathsf{DHE})$ is never queried, then it is uniformly random, and the probability that $\mathscr{A}$ guesses correctly is bounded by $\frac{q_{\mathrm{RO}} \cdot q_{\mathrm{S}}}{2^{kl}}$ by the birthday bound.

To compute the correct handshake and application traffic keys, $\mathscr{B}_2$ needs to be able to correctly program CHTS, SHTS, and ATS. When these keys are chosen by an honest responder with honest origin partner or a partnered initiator, $\mathscr{B}_2$ uses its strong DH oracle to check whether RO_2 has already received the query that the adversary needs to make to generate these keys. If the query has already been made, $\mathscr{B}_2$ can look up the DH secret using T and win the game. Otherwise, $\mathscr{B}_2$ hashes the session's context and logs it in R, so that future RO_2 queries can identify this session for retroactive programming. It also logs the session's randomness in a look-up table Q, to be used if event F is triggered relative to this session by a future RO_2 query.

Like in the SIGMA-I proof, $\mathscr{B}_2$ must be able to correctly compute handshake and application traffic keys for unpartnered initiator sessions. Because all initiator sessions have embedded DH shares, $\mathscr{B}_2$ cannot compute the DH secret DHE for these sessions. However, it can use its StrongDH oracle to check whether the adversary has queried such a secret and copy the expected keys to preserve consistency in this case. If no query has been made, the keys are selected randomly and the initiator session stores its context, randomness, and keys in R. In future queries to the RO_2 oracle, $\mathscr{B}_2$ will use R to efficiently check whether a query should output one of the initiator session's keys. If so, it retroactively programs the oracle using the keys from R.

Therefore, if event F occurs, reduction $\mathscr{B}_2$ wins the strong Diffie–Hellman game except with probability $\frac{q_{\mathrm{RO}} \cdot q_{\mathrm{S}}}{2^{kl}}$, resulting in $\mathbf{Adv}_{\mathbb{G}}^{\mathsf{stDH}}(t_{\mathscr{B}_2}, q_{\mathrm{RO}}) \geq (1 - \frac{q_{\mathrm{RO}} \cdot q_{\mathrm{S}}}{2^{kl}}) \cdot \Pr[F]$. Then $\Pr[F] \leq \frac{2^{kl}}{2^{kl} - q_{\mathrm{RO}} \cdot q_{\mathrm{S}}} \cdot \mathbf{Adv}_{\mathbb{G}}^{\mathsf{stDH}}(t_{\mathscr{B}_2}, q_{\mathrm{RO}})$. Otherwise, the reduction simulates G_7 perfectly except with probability $\frac{q_{\mathrm{RO}} \cdot q_{\mathrm{S}}}{2^{kl}}$.

$$\begin{aligned}
\Pr[\mathrm{G}_7 \Rightarrow 1] &= \Pr[\mathrm{G}_8 \Rightarrow 1] + \Pr[F] + (1 - \Pr[F]) \cdot \frac{q_{\mathrm{RO}} \cdot q_{\mathrm{S}}}{2^{kl}} \\
&\leq \Pr[\mathrm{G}_8 \Rightarrow 1] + \frac{2^{kl} + q_{\mathrm{RO}} \cdot q_{\mathrm{S}}}{2^{kl} - q_{\mathrm{RO}} \cdot q_{\mathrm{S}}} \cdot \mathbf{Adv}_{\mathbb{G}}^{\mathsf{stDH}}(t_{\mathscr{B}_2}, q_{\mathrm{RO}}) + \frac{q_{\mathrm{RO}} \cdot q_{\mathrm{S}}}{2^{kl}} \\
&\leq \Pr[\mathrm{G}_8 \Rightarrow 1] + 2 \cdot \mathbf{Adv}_{\mathbb{G}}^{\mathsf{stDH}}(t_{\mathscr{B}_2}, q_{\mathrm{RO}}) + \frac{q_{\mathrm{RO}} \cdot q_{\mathrm{S}}}{2^{kl}},
\end{aligned}$$

where the last simplification step assumes that $q_{\mathrm{S}} \cdot q_{\mathrm{RO}} \leq 2^{kl-2}$, which is true for any reasonable real-world parameters.

Game 9. In Game G_9, honest responders with honest origin partners sample fk_S, fk_C, $\mathsf{tk}_{\mathrm{chs}}$ and $\mathsf{tk}_{\mathrm{shs}}$ uniformly at random, so these keys are no longer consistent with RO_2. The adversary can distinguish this change if and only if it queries RO_2 on a string $\mathsf{SHTS}, \mathtt{L}, \mathsf{H}(\texttt{""})$, or $\mathsf{CHTS}, \mathtt{L}, \mathsf{H}(\texttt{""})$, where $\mathtt{L} \in \{\mathtt{L}_4, \mathtt{L}_6\}$, and SHTS and CHTS are chosen by an honest responder sessions with honest origin partner. Call this event E. In these sessions, SHTS and CHTS are chosen uniformly at random by G_8, and they are never revealed by any oracle. Therefore the probability of event E is at most $\frac{q_{\mathrm{RO}} \cdot q_{\mathrm{S}}}{2^{kl}}$ by the birthday bound, hence

$$\Pr[\mathrm{G}_8] \leq \Pr[\mathrm{G}_9] + \frac{q_{\mathrm{RO}} \cdot q_{\mathrm{S}}}{2^{kl}}.$$

Note that this step in the SIGMA-I proof introduced a multi-user PRF security bound due to final keys being derived through a PRF, not the random oracle. Modeling $\mathsf{HKDF.Expand}$ as random oracle RO_2, we here instead incur a birthday bound under the random oracle instead of a multi-user PRF security bound for $\mathsf{HKDF.Expand}$.

The remaining game hops are identical to those in the proof of SIGMA-I, so we discuss them only briefly.

Game 10. In Game G_{10}, we log all messages signed by an honest session in a look-up table Q_S, and we set a flag $\mathsf{bad}[S]$ whenever a partnered session verifies a signature with an uncorrupted public key on a message that was not in Q_S. This is just administrative, so

$$\Pr[\mathrm{G}_{10} \Rightarrow 1] = \Pr[\mathrm{G}_9 \Rightarrow 1].$$

Game 11. In Game G_{11}, we abort if the flag $\mathsf{bad}[S]$ is set. In this case, an honest partnered session received a signature which was not computed by an honest session, and which was verified by an uncorrupted public key. We can give a straightforward reduction $\mathscr{B}_3$ to the multi-user EUF-CMA security of the signature scheme that wins whenever $\mathsf{bad}[S]$ is set and has runtime approximately equal to that of $\mathscr{A}$ in G_{10}. By the identical-until-bad lemma,

$$Pr[\mathrm{G}_{10} \Rightarrow 1] - \Pr[\mathrm{G}_{11} \Rightarrow 1] \leq \mathbf{Adv}_{\mathsf{S}}^{\mathsf{mu\text{-}EUF\text{-}CMA}}(t_{\mathscr{B}_3}, q_{\mathrm{N}}, q_{\mathrm{S}}, q_{\mathrm{S}}, q_{\mathrm{RL}}).$$

Interestingly and in contrast to the SIGMA-I proof, soundness is still not guaranteed after this game hop, because we do not require the signature scheme to be strongly unforgeable. Therefore the adversary may be able to produce a new signature on a message that had been signed by an honest session, allowing it to tamper with SCV without setting the $\mathsf{bad}[S]$ flag.

Game 12. In Game G_{12} we log all messages for which an honest session computed a MAC tag in a look-up table Q_M. We remove the $\mathsf{bad}[S]$ flag and instead set a flag $\mathsf{bad}[M]$ if an honest partnered session verifies a MAC on a message that is not in Q_M. Again, this is only bookkeeping and does not impact the view of $\mathscr{A}$, hence

$$\Pr[\mathrm{G}_{12} \Rightarrow 1] = \Pr[\mathrm{G}_{11} \Rightarrow 1].$$

Game 13. Finally, in Game G_{13}, we abort if an honest session with an honest partner verifies a MAC tag on a message which was not tagged by any honest session; i.e if the $\mathsf{bad}[M]$ flag is set. We can give a simple reduction $\mathscr{B}_4$ to multi-user MAC security. The reduction $\mathscr{B}_4$ assigns a pair of indices $i, i+1$ to each session identifier held by an honest session with honest origin partner. When an honest session with honest origin partner needs to compute a server MAC tag, $\mathscr{B}_4$ finds the pair $(i, i+1)$ using the session identifier and calls its Tag oracle with user identity i. When the session needs a client MAC tag $\mathscr{B}_4$ calls Tag with user identity $i+1$. The reduction calls its Tag oracle no more than twice for every query $\mathscr{A}$ makes to SEND (once to generate a tag, and once to verify a tag). Since by Game G_9 all honest sessions with honest origin partners sample their MAC keys fk_S and fk_C uniformly at random, the keys implicitly generated by the MAC security game are consistent with the expected operation of Game G_{13}. When the flag $\mathsf{bad}[M]$ is set, a partnered session has received a valid tag on a message which was never logged in Q_M. The reduction can look up the pair $(i, i+1)$ using the session identifier of whichever session set

$\mathsf{bad}[M]$. Since $\mathscr{B}_4$ logs every message for which it calls its Tag oracle, this is a valid forgery for either identity i or identity $i+1$, and $\mathscr{B}_4$ will win. Then

$$\Pr[\mathrm{G}_{12} \Rightarrow 1] - \Pr[\mathrm{G}_{13} \Rightarrow 1] \leq \mathbf{Adv}_{\mathsf{M}}^{\mathsf{mu\text{-}EUF\text{-}CMA}}(t_{\mathscr{B}_4}, q_{\mathrm{S}}, q_{\mathrm{S}}, 1, q_{\mathrm{S}}, 1, 0).$$

The runtime of $\mathscr{B}_4$ is about that of $\mathscr{A}$ in G_{12}.

We can now finally argue that the advantage of $\mathscr{A}$ in G_{13} is zero. The adversary $\mathscr{A}$ would win game G_{13} with probability better than $\frac{1}{2}$ in one of three ways:

1. Sound is false,
2. $\mathsf{ExplicitAuth}$ is false, or
3. Fresh is true and $b' = b$.

Soundness.

The flag Sound is set if (1) three honest sessions hold the same session identifier, or if (2) two partnered sessions accept with different session keys. By Game G_2, each session identifier is held by at most one session of each role. There are only two roles so case (1) never occurs. If two partnered sessions π_1 and π_2 accept, the initiator session π_1 verified a MAC tag τ on the message $m = n \| n' \| X \| Y \| \texttt{SCRT} \| \texttt{SCV}$. Because τ was verified by an honsest partnered session, by Game G_{13}, this message was tagged by an honest session. Honest sessions only tag strings including their own nonce, and by Game G_2, the only honest session with nonce n' is π_2. Then π_2 must have tagged the message m, so π_1 and π_2 agree on both τ and m. Since the DH shares X and Y are components of m, π_1 and π_2 also agree on the DH secret DHE associated with the pair (X,Y). Consequently, π_1 and π_2 will agree on any value derived deterministically from m, τ, and DHE, including the session key ATS. Then the flag Sound is always true in G_{13}.

Explicit authentication.

The flag $\mathsf{ExplicitAuth}$ is set if there exists a session π_u^i that accepts with uncorrupted peer identity v, and either (1) no honest session π_v^j is partnered with π_u^i, or (2) a session π_v^j is partnered with

π_u^i but accepts with peer identity $w \neq u$. To have accepted with peer identity v, the session π_u^i must have received and verified a signature σ using the public key of identity v on a message m containing the session identifier of π_u^i. As v was uncorrupted at the time that π_u^i accepted, by Game G_{11}, the message m must have been signed by some honest session π_v^j. As honest sessions only sign messages containing their own session identifiers, $\pi_v^j.sid = \pi_u^i.sid$, so π_v^j and π_u^i are partnered. If case (2) occurs, π_v^j must have accepted a MAC tag τ on message m' containing its session ID and the identity w of its peer. We know that π_v^j is a partnered session, so by G_{13}, m' was tagged by some honest session. Honest sessions tag only messages containing their own session identifiers, so by G_2, the message m' must have been tagged by either π_u^i or π_v^j. In SIGMA-I, the messages tagged by these two sessions are differentiated by there labels. Here, they are differentiated by their length: one role signs a message including values `SF`, `CCRT`, and `CCV`, while the other signs a message which does not contain these values. For this reason π_v^j will not verify the tag on a message it signed itself. Therefore m' must have been tagged by π_u^i, so m' contains the identity u. This contradicts the assumption that $w \neq u$, so case (2) never occurs, and the flag ExplicitAuth is always false in G_{13}.

Guessing the challenge bit.

Now the adversary can only win with advantage better than zero is by guessing the correct value of b when the Fresh flag is set to true. This requirement ensures that all tested sessions accepted with uncorrupted peer identities. Since ExplicitAuth is true, each tested session must therefore have an honest session with which it is partnered, and by Sound, this session holds the same session key. Then by G_6, each tested initiator session copies the session key of its partner. By G_8 each tested responder session, and each responder session partnered with a tested initiator session chooses its session key uniformly at random. By Fresh, the partners of tested sessions were not tested or revealed. Then the session keys of all tested sessions are sampled uniformly and never revealed to the adversary by any oracle. Therefore the key returned by each TEST query is uniformly random and independent of the bit b. The adversary's view is independent of the bit b, so it will win G_{13} with probability $\frac{1}{2}$, and consequently its advantage is 0.

Collecting the bounds across all game hops gives the theorem statement. ▮

2.9 Evaluation

Tighter security results in terms of loss factors are practically meaningful only if they materialize in better concrete advantage bounds when taking the underlying assumptions into account. In our case, this amounts to the question: How does the overall concrete security of the SIGMA/SIGMA-I and the TLS 1.3 key exchange protocols improve based on our tighter security proofs?

Parameter selection.

In order to evaluate our and prior bounds pratically, we need to make concrete choices for each of the parameters entering the bounds. Let us explain the choices we made in our evaluation:

Runtime $t \in \{2^{40}, 2^{60}, 2^{80}\}$. We parameterize the adversary's runtime between well within computational reach (2^{40}) and large-scale attackers (2^{80}).

Number of users $\#U = q_{\mathrm{N}} \in \{2^{20}, 2^{30}\}$. We consider the number of users a global-scale adversary may interact with to be in the order of active public-key certificates on the Internet, reported at 130–150 million[11] ($\approx 2^{27}$).

Number of sessions $\#S \approx q_{\mathrm{S}} \in \{2^{35}, 2^{45}, 2^{55}\}$. Chrome[12] and Firefox[13] report that 76–98% of all web page accesses through these browsers are encrypted, with an active daily base of about 2 billion ($\approx 2^{30}$) users.We consider adversaries may easily see 2^{35} sessions and a global-scale attacker may have access to 2^{55} sessions over an extended timespan. Note that the number of send queries essentially corresponds to the number of sessions.

Number of RO queries $\#RO = q_{\mathrm{RO}} = \frac{t}{2^{10}}$. We fix this bound at a 2^{10}-fraction of the overall runtime accounting for all adversarial steps.

Diffie–Hellman groups and group order p. We consider all five elliptic curves standardized for use with TLS 1.3 (bit-security level b and group order p in parentheses): `secp256r1`

[11] https://letsencrypt.org/stats/, https://trends.builtwith.com/ssl/traffic/Entire-Internet
[12] https://transparencyreport.google.com/https/
[13] https://telemetry.mozilla.org/

($b = 128,\ p \approx 2^{256}$), `secp384r1` ($b = 192,\ p \approx 2^{384}$), `secp521r1` ($b = 256,\ p \approx 2^{521}$), `x25519` ($b = 128,\ p \approx 2^{252}$), and `x448` ($b = 224,\ p \approx 2^{446}$). We focus on elliptic curve groups only, as they provide high efficiency and the best known algorithms for solving discrete-log and DH problems are generic, allowing us to apply GGM bounds for the involved DDH and strong DH assumptions.

Signature schemes. In order to unify the underlying hardness assumptions, we consider the ECDSA/EdDSA signature schemes standardized for use with TLS 1.3, based on the five elliptic curves above, treating their single-user unforgeability as equally hard as the corresponding discrete logarithm problem.

Symmetric schemes and key/output/nonce lengths kl, ol, nl. Since our focus is mostly on evaluating ECDH parameters, we idealize the symmetric primitives (PRF, MAC, and hash function) in the random oracle model. Applying lengths standardized for TLS 1.3, we set the key and output length to $kl = ol = 256$ bits for 128-bit security curves and 384 bits for higher-security curves, corresponding to ciphersuites using SHA-256 or SHA-384. The nonce length is fixed to $nl = 256$ bits, again as in TLS 1.3.

Reveal and Test queries q_{RS}, q_{RL}, q_{T}. Using a generic reduction to single-user signature unforgeability, the number of RevLongTermKey, RevSessionKey, and Test queries do not affect the bounds; we hence do not place any constraints on them.

Fully-quantitative CK/DFGS bounds for SIGMA/TLS 1.3.

For our evaluation, we need to reconstruct fully-quantitative security bounds from the more abstract prior security proofs for SIGMA by Canetti-Krawczyk [68] and for TLS 1.3 by Dowling et al. [92]. We report them in Appendix 2.10 for reference. In terms of their reduction to underlying DH problems, the CK SIGMA bound reduces to the DDH problem with a loss of $\#U \cdot \#S$, whereas the DFGS TLS 1.3 bound reduces to the strong DH problem with a loss of $(\#S)^2$.

Numerical advantage bounds for CK, DFGS, and ours.

We report the numerical advantage bounds for SIGMA and TLS 1.3 based on prior (CK [68], DFGS [92]) and our bounds when ranging over the full parameter space detailed above

Adv. resources						SIGMA		TLS 1.3	
t	$\#U$	$\#S$	$\#RO$	Curve (bit security b, group order p)	Target $t/2^b$	CK [68]	Us (Thm. 5)	DFGS [92]	Us (Thm. 6)
2^{40}	2^{20}	2^{35}	2^{30}	`secp256r1` ($b=128$, $p\approx2^{256}$)	2^{-88}	$\approx2^{-101}$	$\approx2^{-156}$	$\approx2^{-104}$	$\approx2^{-156}$
2^{40}	2^{20}	2^{45}	2^{30}	`secp256r1` ($b=128$, $p\approx2^{256}$)	2^{-88}	$\approx2^{-91}$	$\approx2^{-156}$	$\approx2^{-84}$	$\approx2^{-156}$
2^{40}	2^{20}	2^{55}	2^{30}	`secp256r1` ($b=128$, $p\approx2^{256}$)	2^{-88}	$\approx2^{-81}$	$\approx2^{-156}$	$\approx2^{-64}$	$\approx2^{-156}$
2^{40}	2^{30}	2^{35}	2^{30}	`secp256r1` ($b=128$, $p\approx2^{256}$)	2^{-88}	$\approx2^{-81}$	$\approx2^{-146}$	$\approx2^{-104}$	$\approx2^{-146}$
2^{40}	2^{30}	2^{45}	2^{30}	`secp256r1` ($b=128$, $p\approx2^{256}$)	2^{-88}	$\approx2^{-71}$	$\approx2^{-146}$	$\approx2^{-84}$	$\approx2^{-146}$
2^{40}	2^{30}	2^{55}	2^{30}	`secp256r1` ($b=128$, $p\approx2^{256}$)	2^{-88}	$\approx2^{-61}$	$\approx2^{-146}$	$\approx2^{-64}$	$\approx2^{-146}$
2^{40}	2^{20}	2^{35}	2^{30}	`secp384r1` ($b=192$, $p\approx2^{384}$)	2^{-152}	$\approx2^{-229}$	$\approx2^{-284}$	$\approx2^{-232}$	$\approx2^{-284}$
2^{40}	2^{20}	2^{45}	2^{30}	`secp384r1` ($b=192$, $p\approx2^{384}$)	2^{-152}	$\approx2^{-219}$	$\approx2^{-284}$	$\approx2^{-212}$	$\approx2^{-284}$
2^{40}	2^{20}	2^{55}	2^{30}	`secp384r1` ($b=192$, $p\approx2^{384}$)	2^{-152}	$\approx2^{-209}$	$\approx2^{-284}$	$\approx2^{-192}$	$\approx2^{-284}$
2^{40}	2^{30}	2^{35}	2^{30}	`secp384r1` ($b=192$, $p\approx2^{384}$)	2^{-152}	$\approx2^{-209}$	$\approx2^{-274}$	$\approx2^{-232}$	$\approx2^{-274}$
2^{40}	2^{30}	2^{45}	2^{30}	`secp384r1` ($b=192$, $p\approx2^{384}$)	2^{-152}	$\approx2^{-199}$	$\approx2^{-274}$	$\approx2^{-212}$	$\approx2^{-274}$
2^{40}	2^{30}	2^{55}	2^{30}	`secp384r1` ($b=192$, $p\approx2^{384}$)	2^{-152}	$\approx2^{-189}$	$\approx2^{-274}$	$\approx2^{-192}$	$\approx2^{-274}$
2^{40}	2^{20}	2^{35}	2^{30}	`secp521r1` ($b=256$, $p\approx2^{521}$)	2^{-216}	$\approx2^{-298}$	$\approx2^{-318}$	$\approx2^{-282}$	$\approx2^{-317}$
2^{40}	2^{20}	2^{45}	2^{30}	`secp521r1` ($b=256$, $p\approx2^{521}$)	2^{-216}	$\approx2^{-288}$	$\approx2^{-308}$	$\approx2^{-262}$	$\approx2^{-307}$
2^{40}	2^{20}	2^{55}	2^{30}	`secp521r1` ($b=256$, $p\approx2^{521}$)	2^{-216}	$\approx2^{-278}$	$\approx2^{-298}$	$\approx2^{-242}$	$\approx2^{-297}$
2^{40}	2^{30}	2^{35}	2^{30}	`secp521r1` ($b=256$, $p\approx2^{521}$)	2^{-216}	$\approx2^{-288}$	$\approx2^{-318}$	$\approx2^{-282}$	$\approx2^{-317}$
2^{40}	2^{30}	2^{45}	2^{30}	`secp521r1` ($b=256$, $p\approx2^{521}$)	2^{-216}	$\approx2^{-278}$	$\approx2^{-308}$	$\approx2^{-262}$	$\approx2^{-307}$
2^{40}	2^{30}	2^{55}	2^{30}	`secp521r1` ($b=256$, $p\approx2^{521}$)	2^{-216}	$\approx2^{-268}$	$\approx2^{-298}$	$\approx2^{-242}$	$\approx2^{-297}$
2^{40}	2^{20}	2^{35}	2^{30}	`x25519` ($b=128$, $p\approx2^{252}$)	2^{-88}	$\approx2^{-97}$	$\approx2^{-152}$	$\approx2^{-100}$	$\approx2^{-152}$
2^{40}	2^{20}	2^{45}	2^{30}	`x25519` ($b=128$, $p\approx2^{252}$)	2^{-88}	$\approx2^{-87}$	$\approx2^{-152}$	$\approx2^{-80}$	$\approx2^{-152}$
2^{40}	2^{20}	2^{55}	2^{30}	`x25519` ($b=128$, $p\approx2^{252}$)	2^{-88}	$\approx2^{-77}$	$\approx2^{-152}$	$\approx2^{-60}$	$\approx2^{-152}$
2^{40}	2^{30}	2^{35}	2^{30}	`x25519` ($b=128$, $p\approx2^{252}$)	2^{-88}	$\approx2^{-77}$	$\approx2^{-142}$	$\approx2^{-100}$	$\approx2^{-142}$
2^{40}	2^{30}	2^{45}	2^{30}	`x25519` ($b=128$, $p\approx2^{252}$)	2^{-88}	$\approx2^{-67}$	$\approx2^{-142}$	$\approx2^{-80}$	$\approx2^{-142}$
2^{40}	2^{30}	2^{55}	2^{30}	`x25519` ($b=128$, $p\approx2^{252}$)	2^{-88}	$\approx2^{-57}$	$\approx2^{-142}$	$\approx2^{-60}$	$\approx2^{-142}$
2^{40}	2^{20}	2^{35}	2^{30}	`x448` ($b=224$, $p\approx2^{446}$)	2^{-184}	$\approx2^{-291}$	$\approx2^{-318}$	$\approx2^{-282}$	$\approx2^{-317}$
2^{40}	2^{20}	2^{45}	2^{30}	`x448` ($b=224$, $p\approx2^{446}$)	2^{-184}	$\approx2^{-281}$	$\approx2^{-308}$	$\approx2^{-262}$	$\approx2^{-307}$
2^{40}	2^{20}	2^{55}	2^{30}	`x448` ($b=224$, $p\approx2^{446}$)	2^{-184}	$\approx2^{-271}$	$\approx2^{-298}$	$\approx2^{-242}$	$\approx2^{-297}$
2^{40}	2^{30}	2^{35}	2^{30}	`x448` ($b=224$, $p\approx2^{446}$)	2^{-184}	$\approx2^{-271}$	$\approx2^{-318}$	$\approx2^{-282}$	$\approx2^{-317}$
2^{40}	2^{30}	2^{45}	2^{30}	`x448` ($b=224$, $p\approx2^{446}$)	2^{-184}	$\approx2^{-261}$	$\approx2^{-308}$	$\approx2^{-262}$	$\approx2^{-307}$
2^{40}	2^{30}	2^{55}	2^{30}	`x448` ($b=224$, $p\approx2^{446}$)	2^{-184}	$\approx2^{-251}$	$\approx2^{-298}$	$\approx2^{-242}$	$\approx2^{-297}$
2^{60}	2^{20}	2^{35}	2^{50}	`secp256r1` ($b=128$, $p\approx2^{256}$)	2^{-68}	$\approx2^{-61}$	$\approx2^{-116}$	$\approx2^{-64}$	$\approx2^{-116}$
2^{60}	2^{20}	2^{45}	2^{50}	`secp256r1` ($b=128$, $p\approx2^{256}$)	2^{-68}	$\approx2^{-51}$	$\approx2^{-116}$	$\approx2^{-44}$	$\approx2^{-116}$
2^{60}	2^{20}	2^{55}	2^{50}	`secp256r1` ($b=128$, $p\approx2^{256}$)	2^{-68}	$\approx2^{-41}$	$\approx2^{-116}$	$\approx2^{-24}$	$\approx2^{-116}$
2^{60}	2^{30}	2^{35}	2^{50}	`secp256r1` ($b=128$, $p\approx2^{256}$)	2^{-68}	$\approx2^{-41}$	$\approx2^{-106}$	$\approx2^{-64}$	$\approx2^{-106}$
2^{60}	2^{30}	2^{45}	2^{50}	`secp256r1` ($b=128$, $p\approx2^{256}$)	2^{-68}	$\approx2^{-31}$	$\approx2^{-106}$	$\approx2^{-44}$	$\approx2^{-106}$
2^{60}	2^{30}	2^{55}	2^{50}	`secp256r1` ($b=128$, $p\approx2^{256}$)	2^{-68}	$\approx2^{-21}$	$\approx2^{-106}$	$\approx2^{-24}$	$\approx2^{-106}$
2^{60}	2^{20}	2^{35}	2^{50}	`secp384r1` ($b=192$, $p\approx2^{384}$)	2^{-132}	$\approx2^{-189}$	$\approx2^{-244}$	$\approx2^{-192}$	$\approx2^{-244}$
2^{60}	2^{20}	2^{45}	2^{50}	`secp384r1` ($b=192$, $p\approx2^{384}$)	2^{-132}	$\approx2^{-179}$	$\approx2^{-244}$	$\approx2^{-172}$	$\approx2^{-244}$
2^{60}	2^{20}	2^{55}	2^{50}	`secp384r1` ($b=192$, $p\approx2^{384}$)	2^{-132}	$\approx2^{-169}$	$\approx2^{-244}$	$\approx2^{-152}$	$\approx2^{-244}$
2^{60}	2^{30}	2^{35}	2^{50}	`secp384r1` ($b=192$, $p\approx2^{384}$)	2^{-132}	$\approx2^{-169}$	$\approx2^{-234}$	$\approx2^{-192}$	$\approx2^{-234}$
2^{60}	2^{30}	2^{45}	2^{50}	`secp384r1` ($b=192$, $p\approx2^{384}$)	2^{-132}	$\approx2^{-159}$	$\approx2^{-234}$	$\approx2^{-172}$	$\approx2^{-234}$
2^{60}	2^{30}	2^{55}	2^{50}	`secp384r1` ($b=192$, $p\approx2^{384}$)	2^{-132}	$\approx2^{-149}$	$\approx2^{-234}$	$\approx2^{-152}$	$\approx2^{-234}$
2^{60}	2^{20}	2^{35}	2^{50}	`secp521r1` ($b=256$, $p\approx2^{521}$)	2^{-196}	$\approx2^{-278}$	$\approx2^{-298}$	$\approx2^{-250}$	$\approx2^{-285}$
2^{60}	2^{20}	2^{45}	2^{50}	`secp521r1` ($b=256$, $p\approx2^{521}$)	2^{-196}	$\approx2^{-268}$	$\approx2^{-288}$	$\approx2^{-240}$	$\approx2^{-285}$
2^{60}	2^{20}	2^{55}	2^{50}	`secp521r1` ($b=256$, $p\approx2^{521}$)	2^{-196}	$\approx2^{-258}$	$\approx2^{-278}$	$\approx2^{-222}$	$\approx2^{-277}$
2^{60}	2^{30}	2^{35}	2^{50}	`secp521r1` ($b=256$, $p\approx2^{521}$)	2^{-196}	$\approx2^{-268}$	$\approx2^{-298}$	$\approx2^{-250}$	$\approx2^{-285}$
2^{60}	2^{30}	2^{45}	2^{50}	`secp521r1` ($b=256$, $p\approx2^{521}$)	2^{-196}	$\approx2^{-258}$	$\approx2^{-288}$	$\approx2^{-240}$	$\approx2^{-285}$
2^{60}	2^{30}	2^{55}	2^{50}	`secp521r1` ($b=256$, $p\approx2^{521}$)	2^{-196}	$\approx2^{-248}$	$\approx2^{-278}$	$\approx2^{-222}$	$\approx2^{-277}$
2^{60}	2^{20}	2^{35}	2^{50}	`x25519` ($b=128$, $p\approx2^{252}$)	2^{-68}	$\approx2^{-57}$	$\approx2^{-112}$	$\approx2^{-60}$	$\approx2^{-112}$
2^{60}	2^{20}	2^{45}	2^{50}	`x25519` ($b=128$, $p\approx2^{252}$)	2^{-68}	$\approx2^{-47}$	$\approx2^{-112}$	$\approx2^{-40}$	$\approx2^{-112}$
2^{60}	2^{20}	2^{55}	2^{50}	`x25519` ($b=128$, $p\approx2^{252}$)	2^{-68}	$\approx2^{-37}$	$\approx2^{-112}$	$\approx2^{-20}$	$\approx2^{-112}$
2^{60}	2^{30}	2^{35}	2^{50}	`x25519` ($b=128$, $p\approx2^{252}$)	2^{-68}	$\approx2^{-37}$	$\approx2^{-102}$	$\approx2^{-60}$	$\approx2^{-102}$
2^{60}	2^{30}	2^{45}	2^{50}	`x25519` ($b=128$, $p\approx2^{252}$)	2^{-68}	$\approx2^{-27}$	$\approx2^{-102}$	$\approx2^{-40}$	$\approx2^{-102}$
2^{60}	2^{30}	2^{55}	2^{50}	`x25519` ($b=128$, $p\approx2^{252}$)	2^{-68}	$\approx2^{-17}$	$\approx2^{-102}$	$\approx2^{-20}$	$\approx2^{-102}$
2^{60}	2^{20}	2^{35}	2^{50}	`x448` ($b=224$, $p\approx2^{446}$)	2^{-164}	$\approx2^{-251}$	$\approx2^{-298}$	$\approx2^{-250}$	$\approx2^{-285}$
2^{60}	2^{20}	2^{45}	2^{50}	`x448` ($b=224$, $p\approx2^{446}$)	2^{-164}	$\approx2^{-241}$	$\approx2^{-288}$	$\approx2^{-234}$	$\approx2^{-285}$
2^{60}	2^{20}	2^{55}	2^{50}	`x448` ($b=224$, $p\approx2^{446}$)	2^{-164}	$\approx2^{-231}$	$\approx2^{-278}$	$\approx2^{-214}$	$\approx2^{-277}$
2^{60}	2^{30}	2^{35}	2^{50}	`x448` ($b=224$, $p\approx2^{446}$)	2^{-164}	$\approx2^{-231}$	$\approx2^{-296}$	$\approx2^{-250}$	$\approx2^{-285}$
2^{60}	2^{30}	2^{45}	2^{50}	`x448` ($b=224$, $p\approx2^{446}$)	2^{-164}	$\approx2^{-221}$	$\approx2^{-288}$	$\approx2^{-234}$	$\approx2^{-285}$
2^{60}	2^{30}	2^{55}	2^{50}	`x448` ($b=224$, $p\approx2^{446}$)	2^{-164}	$\approx2^{-211}$	$\approx2^{-278}$	$\approx2^{-214}$	$\approx2^{-277}$
2^{80}	2^{20}	2^{35}	2^{70}	`secp256r1` ($b=128$, $p\approx2^{256}$)	2^{-48}	$\approx2^{-21}$	$\approx2^{-76}$	$\approx2^{-24}$	$\approx2^{-76}$
2^{80}	2^{20}	2^{45}	2^{70}	`secp256r1` ($b=128$, $p\approx2^{256}$)	2^{-48}	$\approx2^{-11}$	$\approx2^{-76}$	$\approx2^{-4}$	$\approx2^{-76}$
2^{80}	2^{20}	2^{55}	2^{70}	`secp256r1` ($b=128$, $p\approx2^{256}$)	2^{-48}	$\approx2^{-1}$	$\approx2^{-76}$	1	$\approx2^{-76}$
2^{80}	2^{30}	2^{35}	2^{70}	`secp256r1` ($b=128$, $p\approx2^{256}$)	2^{-48}	$\approx2^{-1}$	$\approx2^{-66}$	$\approx2^{-24}$	$\approx2^{-66}$
2^{80}	2^{30}	2^{45}	2^{70}	`secp256r1` ($b=128$, $p\approx2^{256}$)	2^{-48}	1	$\approx2^{-66}$	$\approx2^{-4}$	$\approx2^{-66}$
2^{80}	2^{30}	2^{55}	2^{70}	`secp256r1` ($b=128$, $p\approx2^{256}$)	2^{-48}	1	$\approx2^{-66}$	1	$\approx2^{-66}$
2^{80}	2^{20}	2^{35}	2^{70}	`secp384r1` ($b=192$, $p\approx2^{384}$)	2^{-112}	$\approx2^{-149}$	$\approx2^{-204}$	$\approx2^{-152}$	$\approx2^{-204}$
2^{80}	2^{20}	2^{45}	2^{70}	`secp384r1` ($b=192$, $p\approx2^{384}$)	2^{-112}	$\approx2^{-139}$	$\approx2^{-204}$	$\approx2^{-132}$	$\approx2^{-204}$
2^{80}	2^{20}	2^{55}	2^{70}	`secp384r1` ($b=192$, $p\approx2^{384}$)	2^{-112}	$\approx2^{-129}$	$\approx2^{-204}$	$\approx2^{-112}$	$\approx2^{-204}$
2^{80}	2^{30}	2^{35}	2^{70}	`secp384r1` ($b=192$, $p\approx2^{384}$)	2^{-112}	$\approx2^{-129}$	$\approx2^{-194}$	$\approx2^{-152}$	$\approx2^{-194}$
2^{80}	2^{30}	2^{45}	2^{70}	`secp384r1` ($b=192$, $p\approx2^{384}$)	2^{-112}	$\approx2^{-119}$	$\approx2^{-194}$	$\approx2^{-132}$	$\approx2^{-194}$
2^{80}	2^{30}	2^{55}	2^{70}	`secp384r1` ($b=192$, $p\approx2^{384}$)	2^{-112}	$\approx2^{-109}$	$\approx2^{-194}$	$\approx2^{-112}$	$\approx2^{-194}$
2^{80}	2^{20}	2^{35}	2^{70}	`secp521r1` ($b=256$, $p\approx2^{521}$)	2^{-176}	$\approx2^{-258}$	$\approx2^{-278}$	$\approx2^{-210}$	$\approx2^{-245}$
2^{80}	2^{20}	2^{45}	2^{70}	`secp521r1` ($b=256$, $p\approx2^{521}$)	2^{-176}	$\approx2^{-248}$	$\approx2^{-268}$	$\approx2^{-200}$	$\approx2^{-245}$
2^{80}	2^{20}	2^{55}	2^{70}	`secp521r1` ($b=256$, $p\approx2^{521}$)	2^{-176}	$\approx2^{-238}$	$\approx2^{-258}$	$\approx2^{-190}$	$\approx2^{-245}$
2^{80}	2^{30}	2^{35}	2^{70}	`secp521r1` ($b=256$, $p\approx2^{521}$)	2^{-176}	$\approx2^{-248}$	$\approx2^{-278}$	$\approx2^{-210}$	$\approx2^{-245}$
2^{80}	2^{30}	2^{45}	2^{70}	`secp521r1` ($b=256$, $p\approx2^{521}$)	2^{-176}	$\approx2^{-238}$	$\approx2^{-268}$	$\approx2^{-200}$	$\approx2^{-245}$
2^{80}	2^{30}	2^{55}	2^{70}	`secp521r1` ($b=256$, $p\approx2^{521}$)	2^{-176}	$\approx2^{-228}$	$\approx2^{-258}$	$\approx2^{-190}$	$\approx2^{-245}$
2^{80}	2^{20}	2^{35}	2^{70}	`x25519` ($b=128$, $p\approx2^{252}$)	2^{-48}	$\approx2^{-17}$	$\approx2^{-72}$	$\approx2^{-20}$	$\approx2^{-72}$
2^{80}	2^{20}	2^{45}	2^{70}	`x25519` ($b=128$, $p\approx2^{252}$)	2^{-48}	$\approx2^{-7}$	$\approx2^{-72}$	1	$\approx2^{-72}$
2^{80}	2^{20}	2^{55}	2^{70}	`x25519` ($b=128$, $p\approx2^{252}$)	2^{-48}	1	$\approx2^{-72}$	1	$\approx2^{-72}$
2^{80}	2^{30}	2^{35}	2^{70}	`x25519` ($b=128$, $p\approx2^{252}$)	2^{-48}	1	$\approx2^{-62}$	$\approx2^{-20}$	$\approx2^{-62}$
2^{80}	2^{30}	2^{45}	2^{70}	`x25519` ($b=128$, $p\approx2^{252}$)	2^{-48}	1	$\approx2^{-62}$	1	$\approx2^{-62}$
2^{80}	2^{30}	2^{55}	2^{70}	`x25519` ($b=128$, $p\approx2^{252}$)	2^{-48}	1	$\approx2^{-62}$	1	$\approx2^{-62}$
2^{80}	2^{20}	2^{35}	2^{70}	`x448` ($b=224$, $p\approx2^{446}$)	2^{-144}	$\approx2^{-211}$	$\approx2^{-266}$	$\approx2^{-210}$	$\approx2^{-245}$
2^{80}	2^{20}	2^{45}	2^{70}	`x448` ($b=224$, $p\approx2^{446}$)	2^{-144}	$\approx2^{-201}$	$\approx2^{-266}$	$\approx2^{-194}$	$\approx2^{-245}$
2^{80}	2^{20}	2^{55}	2^{70}	`x448` ($b=224$, $p\approx2^{446}$)	2^{-144}	$\approx2^{-191}$	$\approx2^{-258}$	$\approx2^{-174}$	$\approx2^{-245}$
2^{80}	2^{30}	2^{35}	2^{70}	`x448` ($b=224$, $p\approx2^{446}$)	2^{-144}	$\approx2^{-191}$	$\approx2^{-256}$	$\approx2^{-210}$	$\approx2^{-245}$
2^{80}	2^{30}	2^{45}	2^{70}	`x448` ($b=224$, $p\approx2^{446}$)	2^{-144}	$\approx2^{-181}$	$\approx2^{-256}$	$\approx2^{-194}$	$\approx2^{-245}$
2^{80}	2^{30}	2^{55}	2^{70}	`x448` ($b=224$, $p\approx2^{446}$)	2^{-144}	$\approx2^{-171}$	$\approx2^{-256}$	$\approx2^{-174}$	$\approx2^{-245}$

Table 2.2. Advantages of a key exchange adversary with given resources t (running time), $\#U$ (number of users), $\#S$ (number of sessions), and $\#RO$ (number of random oracle queries), in breaking the security of the SIGMA and TLS 1.3 protocols when instantiated with the given curves (bit security b and group order p in parentheses), based on the prior bounds by Canetti-Krawczyk [68] resp. Dowling et al. [92], and the bounds we establish (Theorem 5 and 6). Target indicates the maximal advantage $t/2^b$ tolerable when aiming for the respective curve's security level b; entries in red-shaded cells miss that target. See Section 3.7 for further details.

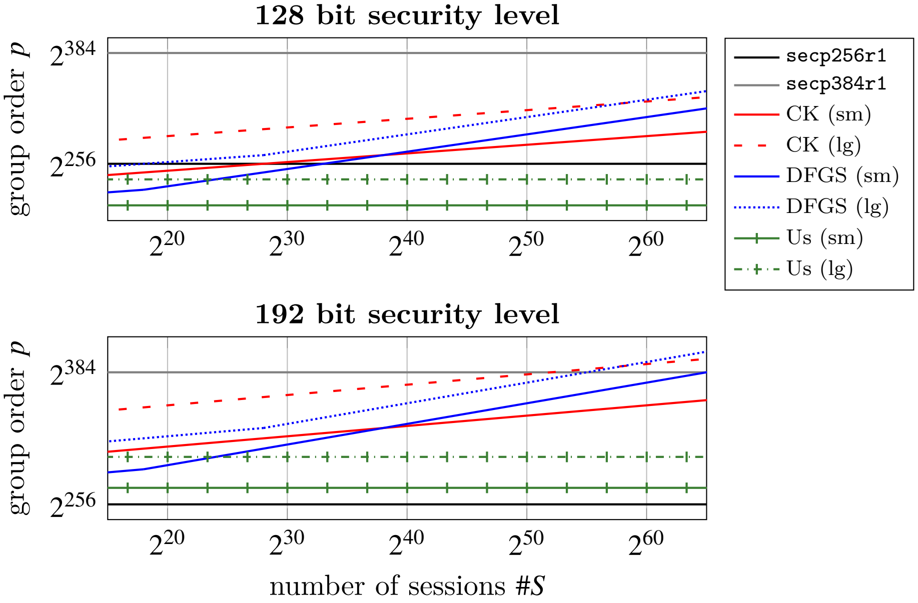


Figure 2.16. Elliptic curve group order (y axis) required to achieve 128-bit (top) and 192-bit (bottom) AKE security for SIGMA and TLS 1.3 based on the CK [68] SIGMA, DFGS [92] TLS 1.3, and our bounds (ours giving the same result for SIGMA and TLS 1.3), for a varying number of sessions $\#S$ (x axis). Both axes are in log-scale.
For each security and bound, we plot a smaller-resource "(sm)" setting with runtime $t = 2^{60}$, number of users $\#U = 2^{20}$, and number of random oracle queries $\#RO = 2^{50}$ and a larger-resource "(lg)" setting with $t = 2^{80}$, $\#U = 2^{30}$, and $\#RO = 2^{70}$. We let symmetric key/output lengths be 256 bits for 128-bit security and 384-bits for 192-bit security; nonce length is 256 bits. The group orders of NIST elliptic curves `secp256r1` ($p \approx 2^{256}$) and `secp384r1` ($p \approx 2^{384}$) are shown as horizontal lines for context.

in Table 2.2. Table 2.1 summarizes the key data points for 128-bit and 192-bit security levels.

Throughout Table 2.2, we assume that an adversary with running time t makes no more than $t \cdot 2^{-10}$ queries to its random oracles. We target the bit-security of whatever curve we use; this means that for b bits of security we want an advantage of $t/2^b$. If a bound does not achieve this goal, we color it red. We consider runtimes between 2^{40} and 2^{80}, a total number of users between to vary between 2^{20} and 2^{30}, and a total number of sessions between 2^{35} and 2^{55} (see above for the discussion of these parameter choices). We evaluate these parameters in relation to all of the elliptic curve groups standardized for use with TLS 1.3. We idealize symmetric primitives, assuming the use of 256-bit keys in conjunction with 128-bit security curves and 384-bit keys in conjunction with higher-security curves, this corresponds to the available SHA-256 and SHA-384 functions in TLS 1.3. The nonce length is fixed to 256 bits (as in TLS 1.3).

Our bounds do better than the CK [68] and DFGS [92] bounds across all considered parameters and always meet the security targets, which these prior bounds fail to meet for secp256r1 and x25519 for almost all parameters, but notably also for the 192-bit security level of curve secp384r1 for large-scale parameters. We improve over prior bounds by at least 20 and up to 85 bits of security for SIGMA, and by at least 35 and up to 92 bits of security for TLS 1.3.

In comparison, the TLS 1.3 bounds from the concurrent work by Diemert and Jager [88] yield bit security levels similar to ours for TLS 1.3: While some sub-terms in their bound are slightly worse (esp. for strong DH), the dominating sub-terms are the same.

Group size requirements based on CK, DFGS, and our bound.

Finally, let us take a slightly different perspective on what the prior and our bounds tell us: Figure ?? illustrates the group size required to achieve 128-bit resp. 192-bit AKE security for SIGMA and TLS 1.3 based on the different bounds, dependent on a varying number of sessions $\#S$. The CK SIGMA and our SIGMA and TLS 1.3 bounds are dominated by the signature scheme advantage (with a $\#S \cdot (\#U)^2$ loss for CK and a $\#U$ loss for our bound); the DFGS TLS 1.3 bound instead is mostly dominated by the $(\#S)^2$–loss reduction to strong DH. The CK and DFGS bounds require the use of larger, less efficient curves to achieve 128-bit security even for 2^{35} sessions. For large-scale attackers, they similarly require a larger curve than secp384r1 above about 2^{55} sessions. We highlight that, in contrast, with our bounds a curve with 128-bit, resp. 192-bit, security is sufficient to guarantee the same security level for SIGMA and TLS 1.3, for both small- and large-scale adversaries and for very conservative bounds on the number of sessions.

2.10 Evaluation Details

2.10.1 Fully-quantitative CK SIGMA Bound

Recall our security bound for SIGMA/SIGMA-I from Theorem 5: **ADDTHISBACKIN**

Comparing this boundto the original security proof for SIGMA by Canetti and Krawczyk [68] (CK) faces two complications. First, we must reconstruct a concrete security bound from the CK proof, which merely refers to the decisional Diffie–Hellman and "standard security notions" for digital signatures, MACs, and PRFs (i.e., single-user EUF-CMA and PRF security). Second, the CK result is given in a stronger security model for key exchange [67] which allows state-reveal attacks. Further, the CK proof assumes out-of-band unique session identifiers, whereas protocols in practice have to establish those from, e.g., nonces (introducing a corresponding collision bound as in our analysis). We are therefore inherently constrained to compare qualitatively different security properties here.

Let us informally consider a game-based definition of the CK model [67] in the same style as our AKE model (cf. Definition 1), capturing the same oracles plus an additional state-reveal oracle, with $q_{\mathrm{RS_T}}$ denoting the number of queries to this oracle, and session identifiers that, like ours, consist of the session and peers' nonces and DH shares. Translating the SIGMA-I security proof from [68, Theorem 6 in the full version], we obtained the following concrete security bound:

$$\begin{aligned}
&\mathbf{Adv}^{\mathrm{CK}}_{\mathrm{SIGMA\text{-}I}}(t, q_{\mathrm{N}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{RS_T}}, q_{\mathrm{T}}) \\
&\quad \le \frac{2q_{\mathrm{S}}^2}{2^{nl} \cdot p} + \mathbf{Adv}^{\mathsf{mu\text{-}EUF\text{-}CMA}}_{\mathsf{S}}(t_{\mathscr{B}_1}, q_{\mathrm{N}}, q_{\mathrm{S}}, q_{\mathrm{S}}, q_{\mathrm{RL}}) \qquad \text{// sid collision \& property P1} \\
&\qquad + q_{\mathrm{N}} \cdot q_{\mathrm{S}} \cdot \Big(\mathbf{Adv}^{\mathsf{DDH}}_{\mathbb{G}}(t_{\mathscr{B}_2}) + \mathbf{Adv}^{\mathsf{mu\text{-}PRF}}_{\mathsf{PRF}}(t_{\mathscr{B}_5}, 1, 3) \qquad \text{// property P2 \dots} \\
&\qquad + (q_{\mathrm{N}} + 1) \cdot \mathbf{Adv}^{\mathsf{mu\text{-}EUF\text{-}CMA}}_{\mathsf{S}}(t_{\mathscr{B}_3}, 1, q_{\mathrm{S}}, q_{\mathrm{S}}, 0) + \mathbf{Adv}^{\mathsf{mu\text{-}EUF\text{-}CMA}}_{\mathsf{M}}(t_{\mathscr{B}_4}, 1, 2, 2, 2, 2, 0) \Big),
\end{aligned}$$

where nl is the nonce length, $\mathbb{G}$ the used Diffie–Hellman group of prime order p, the number of test queries is restricted to $q_{\mathrm{T}} = 1$, and $\mathscr{B}_i$ (for $i = 1, \dots, 5$) are the described reductions for property P1 and P2 in [68, Theorem 6 in the full version] all running in time $t_{\mathscr{B}_i} \approx t$. For simplicity, we present the above bound in terms of "multi-user" PRF, signature, and MAC advantages for a single user $q_{\mathrm{Nw}} = 1$, which are equivalent to the corresponding single-user advantages (cf. Section 2.3).

2.10.2 Fully-quantitative DFGS TLS 1.3 Bound

Recall our security bound for TLS 1.3 from Theorem 6: **ADDTHISBACKIN** We compare this bound with the bound of Dowling et al. [92] (DFGS). Note that this bound is established in a multi-stage key exchange model [100], here we focus only on the main application key derivation, as in our proof. The DFGS bound needs instantiation through the random oracle only in one step (the PRF-ODH assumption on HKDF.Extract) while other PRF steps remain in the standard model. Our proof instead models both HKDF.Extract and HKDF.Expand as random oracles. Translating the bound from [92, Theorems 5.1, 5.2] yields:

$$
\begin{aligned}
&\mathbf{Adv}^{\mathrm{DFGS}}_{\mathsf{TLS\,1.3}}(t, q_{\mathrm{N}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}) \\
&\quad \leq \frac{q_{\mathrm{S}}^2}{2^{nl} \cdot p} + q_{\mathrm{S}} \cdot \Bigg(\mathbf{Adv}^{\mathsf{CR}}_{\mathsf{H}}(t_{\mathscr{B}_1}) + q_{\mathrm{N}} \cdot \mathbf{Adv}^{\mathsf{mu\text{-}EUF\text{-}CMA}}_{\mathsf{S}}(t_{\mathscr{B}_2}, 1, q_{\mathrm{S}}, q_{\mathrm{S}}, 0) \\
&\quad + q_{\mathrm{S}} \cdot \Big(\mathbf{Adv}^{\mathsf{dual\text{-}snPRF\text{-}ODH}}_{\mathsf{HKDF.Extract},\mathbb{G}}(t_{\mathscr{B}_3}) + \mathbf{Adv}^{\mathsf{mu\text{-}PRF}}_{\mathsf{HKDF.Expand}}(t_{\mathscr{B}_4}, 1, 3, 3, 0) \\
&\quad + 2 \cdot \mathbf{Adv}^{\mathsf{mu\text{-}PRF}}_{\mathsf{HKDF.Expand}}(t_{\mathscr{B}_5}, 1, 2, 2, 0) + \mathbf{Adv}^{\mathsf{mu\text{-}PRF}}_{\mathsf{HKDF.Extract}}(t_{\mathscr{B}_6}, 1, 1, 1, 0) \\
&\quad + \mathbf{Adv}^{\mathsf{mu\text{-}PRF}}_{\mathsf{HKDF.Expand}}(t_{\mathscr{B}_7}, 1, 1, 1, 0) \Big) \Bigg).
\end{aligned}
$$

Let us further unpack the PRF-ODH term. Following Brendel et al. [63], it can be reduced to the strong Diffie–Hellman assumption modeling $\mathsf{HKDF.Extract}$ as a random oracle.[14] In this reduction, the single DH oracle query is checked against each random oracle query via the strong-DH oracle, hence establishing the following bound:

$$
\mathbf{Adv}^{\mathsf{dual\text{-}snPRF\text{-}ODH}}_{\mathrm{RO},\mathbb{G}}(t_{\mathscr{B}_3}, q_{\mathrm{RO}}) \leq \mathbf{Adv}^{\mathsf{stDH}}_{\mathbb{G}}(t_{\mathscr{B}_3}, q_{\mathrm{RO}}).
$$

Acknowledgements

We thank Mihir Bellare for insightful discussions and helpful comments, and Denis Diemert and Tibor Jager for their kind handling of our concurrent work. We thank the anonymous reviewers for valuable comments. Both authors were supported in part by National Science Foundation (NSF) grant CNS-1717640. Felix Günther has been supported in part by Research Fellowship grant GU 1859/1-1 of the German Research Foundation (DFG).

[14] The same paper suggests that a standard-model instantiation of the PRF-ODH assumption via an algebraic black-box reduction to common cryptographic problems is implausible.

Chapter 3

On the concrete security of TLS 1.3 PSK Mode

3.1 Introduction

The *Transport Layer Security* (TLS) protocol is probably the most widely-used cryptographic protocol. It provides a secure channel between two endpoints (*client* and *server*) for arbitrary higher-layer application protocols. Its most recent version, TLS 1.3 [186], specifies two different "modes" for the initial handshake establishing a secure session key: the main handshake mode based on a Diffie–Hellman key exchange and public-key authentication via digital signatures, and a *pre-shared key* (PSK) mode, which performs authentication based on symmetric keys. The latter is mainly used for two purposes:

Session resumption. Here, a prior TLS connection established a secure channel along with a pre-shared key PSK, usually via a full handshake. Subsequent TLS resumption sessions use this key for authentication and key derivation. For example, modern web browsers typically establish multiple TLS connections when loading a web site. Using public-key authentication only in an initial session and PSK-mode in subsequent ones minimizes the number of relatively expensive public-key computations and significantly improves performance for both clients and servers.

Out-of-band establishment. PSKs can also be established out-of-band, e.g., by manual configuration of devices or with a separate key establishment protocol. This enables secure communication in settings where a complex public-key infrastructure (PKI) is unsuitable, such as IoT applications.

TLS 1.3 provides two variants of the PSK handshake mode: *PSK-only* and *PSK-(EC)DHE*. The PSK-only mode is purely based on symmetric-key cryptography. This makes TLS accessible to resource-constrained low-cost devices, and other applications with strict performance requirements, but comes at the cost of not providing *forward secrecy* [112], since the latter is not achievable with static symmetric keys.[1] The PSK-(EC)DHE mode in turn achieves forward secrecy by additionally performing an (elliptic-curve) Diffie–Hellman key exchange, authenticated via the PSK (i.e., still avoiding inefficient public-key signatures). This compromise between performance and security is the suggested choice for TLS 1.3 session resumption on the Internet.

Concrete security and tightness.

Classical, complexity-theoretic security proofs considered the security of cryptosystems *asymptotically.* They are satisfied with security reductions running in polynomial time and having non-negligible success probability. However, it is well-known that this only guarantees that a sufficiently large security parameter exists *asymptotically*, but it does not guarantee that a deployed real-world cryptosystem with standardized parameters—such as concrete key lengths, sizes of algebraic groups, moduli, etc.—can achieve a certain expected security level. In contrast, a *concrete security* approach makes all bounds on the running time and success probability of adversaries explicit, for example, with a bound of the form

$$\mathbf{Adv}(\mathscr{A}) \leq f(\mathscr{A}) \cdot \mathbf{Adv}(\mathscr{B}),$$

where f is a function of the adversary's resources and $\mathscr{B}$ is an adversary against some underlying cryptographic hardness assumption.

The concrete security approach makes it possible to determine concrete deployment parameters that are supported by a formal security proof. As an intuitive toy example, suppose we want to achieve "128-bit security", that is, we want a security proof that guarantees (for any $\mathscr{A}$ in a certain class of adversaries) that $\mathbf{Adv}(\mathscr{A}) \leq 2^{-128}$. Suppose we have a cryptosystem with a reduction that loses "40 bits of security" because we can only prove a bound of $f(\mathscr{A}) \leq 2^{40}$. This means that we have to instantiate the scheme with an underlying hardness assumption

[1] See [**?**, 61] for recent work discussing symmetric key exchange and forward secrecy.

that achieves $\mathbf{Adv}(\mathscr{B}) \leq 2^{-168}$ for any $\mathscr{B}$ in order to upper bound $\mathbf{Adv}(\mathscr{A})$ by 2^{-128} as desired. Hence, the 40-bit security loss of the bound is compensated by larger parameters that provide "168-bit security".

This yields a theoretically-sound choice of deployment parameters, but it might incur a very significant performance loss, as it requires the choice of larger groups, moduli, or key lengths. For example, the size of an elliptic curve group scales quadratically with the expected bit security, so we would have to choose $|\mathbb{G}| \approx 2^{2 \cdot 168} = 2^{336}$ instead of the optimal $|\mathbb{G}| \approx 2^{2 \cdot 128} = 2^{256}$. The performance penalty is even more significant for finite field groups, RSA or discrete logarithms "modulo p". This could lead to parameters which are either too large for practical use, or too small to be supported by the formal security analysis of the cryptosystem. We demonstrate this below for security proofs of TLS.

Even worse, for a given security proof the concrete loss ℓ may not be a constant, as in the above example, but very often ℓ depends on other parameters, such as the number of users or protocol sessions, for example. This makes it difficult to choose theoretically-sound parameters when bounds on these other parameters are not exactly known at the time of deployment. If then a concrete value for ℓ is estimated too small (e.g., because the number of users is underestimated), then the derived parameters are not backed by the security analysis. If ℓ is chosen too large, then it incurs an unnecessary performance overhead.

Therefore we want to have *tight* security proofs, where ℓ is a small constant, independent of any parameters that are unknown when the cryptosystem is deployed. This holds in particular for cryptosystems and protocols that are designed to maximize performance, such as the PSK modes of TLS 1.3 for session resumption or resource-constrained devices.

Previous analyses of the TLS handshake protocol and their tightness.

TLS 1.3 is the first TLS version that was developed in a close collaboration between academia and industry. Early TLS 1.3 drafts were inspired by the OPTLS design by Krawczyk and Wee [147], and several draft revisions as well as the final TLS 1.3 standard in RFC 8446 [186] were analyzed by many different research groups, including computational/reductionist analyses of the full and PSK modes in [93, 95, 101, 92]. All reductions in these papers are however highly non-tight, having up to a quadratic security loss in the number of TLS sessions and adversary can interact with. For example, [?] explains that for “128-bit security” and plausible numbers of users and sessions, an RSA modulus of more than 10,000 bits would be necessary to compensate the loss of previous security proofs for TLS, even though 3072 bits are usually considered sufficient for “128-bit security” when the loss of reductions is not taken into account. Likewise, [82] argues that the tightness loss to the underlying Diffie–Hellman hardness assumption lets these bounds fail to meet the standardized elliptic curves' security target, and for large-scale adversary even yields completely vacuous bounds.

Recently, Davis and Günther [82] and Diemert and Jager [?] gave new, tight security proofs for the TLS 1.3 full handshake based on Diffie–Hellman key exchange and digital signatures (not PSKs). However, their results required very strong assumptions. One is that the underlying digital signature scheme is tightly secure in a multi-user setting with adaptive corruptions. While such signature schemes do exist [21, 108, ?, ?], this is not known for any of the signature schemes standardized for TLS 1.3, which are subject to the tightness lower bounds of [22] as their public keys uniquely determine the matching secret key.

Even more importantly, both [82] and [?] modeled the TLS key schedule or components thereof as *independent* random oracles. This was done to overcome the technical challenge that the Diffie–Hellman secret and key shares need to be *combined* in the key derivation to apply their tight security proof strategy, following Cohn-Gordon et al. [73], yet in TLS 1.3 those values enter key derivation through *separate* function calls. But neither work provided formal justification for their modeling, and both neglected to address potential dependencies between the use of a hash function in the key schedule and elsewhere in the protocol.

Our contributions

In this paper, we describe a new perspective on TLS 1.3, which enables a modular security analysis with tight security proofs.

New abstraction of the TLS 1.3 key schedule.

We first describe a new abstraction of the TLS 1.3 key schedule used in the PSK modes (in Section 3.2), where different steps of the key schedule are modeled as *independent* random oracles (12 random oracles in total). This makes it significantly easier to rigorously analyze the security of TLS 1.3, since it replaces a significant part of the complexity of the protocol with what the key schedule intuitively provides, namely "as-good-as-independent cryptographic keys", deterministically derived from pre-shared keys, Diffie–Hellman values (in PSK-(EC)DHE mode), protocol messages, and the randomness of communicating parties.

Most importantly, in contrast to prior works on TLS 1.3's tightness that abstracted (parts of or the entire) key schedule as random oracles [?, 82] to enable the tight proof technique of Cohn-Gordon et al. [73], we support this new abstraction formally. Using the *indifferentiability* framework of Maurer et al. [156] in its recent adaptation by Bellare et al. [?] that treats *multiple* random oracles, in Section 3.4 we prove our abstraction *indifferentiable* from TLS 1.3 with *only* the underlying cryptographic hash function modeled as a random oracle, and this proof is *tight*. This accounts for possible interdependencies between the use of a hash function in multiple contexts, which were not considered in [?, 82].

Identifying a lack of domain separation.

A noteworthy subtlety is that, to our surprise, we identify that for a certain choice of TLS 1.3 PSK mode and hash function (namely, PSK-only mode with SHA384), a lack of *domain separation* [?] in the protocol does *not* allow us to prove indifferentiability for this case. We discuss the details of why domain separation is achieved for all but this case in Appendix 3.8.

This gap could be closed by more careful domain separation in the key schedule, which we consider an interesting insight for designers of future versions of TLS or other protocols. Concretely, the ideal domain separation method would be to add a unique prefix or suffix to each hash function call made by the protocol. However, existing standard primitives like HMAC and HKDF do not permit the use of such labels, so this advice is not practical for TLS 1.3 or similar protocols. For these, a combination of labels (where possible) and padding for domain separation seems advisable, where the padding ensures that the protocol's direct hash calls have strictly longer inputs than the internal hash calls in HMAC and HKDF. We outline this method in more detail in Appendix 3.8.5.

Modularization of record layer encryption.

Like most of the prior computational TLS 1.3 analyses [93, 101, 92, ?], we use a *multi-stage key exchange* (MSKE) security model [100] to capture the complex and fine-grained security aspects of TLS 1.3. These aspects include cleverly distinguishing between "external" keys established in the handshake for subsequent use (by, e.g., application data encryption, resumption, etc.) and "internal" keys, used within the handshake itself (in TLS 1.3 for encrypting most of the handshake through the protocol's record layer) to avoid complex security models such as the ACCE model [127] which monolithically treat handshake and record-layer encryption.

As a generic simplification step for MSKE models, we show (in Section 3.5) that for a certain class of *transformations* using the internal keys, we can even avoid the somewhat involved handling of internal keys altogether. We use this to simplify our analysis of the TLS 1.3 handshake (treating the TLS 1.3 record-layer encryption as such transformation). The result itself however is not specific to TLS 1.3, but general and of independent interest; it furthermore is *tight*.

Tight security of TLS 1.3 PSK modes.

We leverage the new perspective on the TLS 1.3 key schedule and the fact that we can ignore record-layer encryption to give our main results: the first *tight* security proofs for the PSK-only and PSK-(EC)DHE handshake modes of TLS 1.3.

Evaluation.

Finally, we evaluate our new bounds and prior ones from [92] over a wide range of fully concrete resource parameters, following the approach of Davis and Günther [82]. Our bounds improve on previous analyses of the PSK-only handshake by between 15 and 53 bits of security, and those of the PSK-(EC)DHE handshake by 60 and 131 bits of security across all our parameters evaluated.

Further related work and scope of our analysis

Several previous works gave security proofs for the previous protocol version TLS 1.2 [127, 145, 105, 146, 153, 53], including its PSK-modes [153]; all reductions in these works are highly non-tight.

Brzuska et al. [?] recently proposed a stand-alone security model for the TLS 1.3 key schedule, likewise aiming at a new abstraction perspective on the latter to support formal protocol analysis. While their treatment focuses solely on the key schedule and only briefly argues its application to a key exchange security result, it is more general and covers the negotiation of parameters [96, 52] and agile usage of various algorithms.

Our focus is on the TLS 1.3 PSK modes. Hence, our abstraction of the key schedule and the careful indifferentiability treatment is tailored to that mode and cannot be directly translated to the full handshake (without PSKs). We are confident that our approach can be adapted to achieve similar results for the full handshake, but leave revisiting the results in [?, 82] in that way to future work.

Like many previous cryptographic analyses [127, 145, 93, 95, 101, 92, ?, 82] of the TLS handshake, our work focuses on the “cryptographic core” of the TLS 1.3 PSK handshake modes (in particular, we consider fixed parameters like the Diffie–Hellman group, TLS ciphersuite, etc.). Our abstraction of the key schedule is designed for easy composition with our tight key exchange proof, and our indifferentiability treatment is important confirmation of that abstraction’s soundness. We do not consider, e.g., ciphersuite and version negotiation [96] or backwards compatibility issues in settings where multiple TLS versions are used in parallel, such as [128]. We also do not treat the security of the TLS record layer; instead we explain how to avoid the necessity to do so in order to achieve more modular security analyses, and we refer to compositional results [100, 93, 113, 92, ?] treating the combined security when subsequent protocols use the session keys established in an MSKE protocol.

Numerous authenticated key exchange protocols [108, 73, ?, 126, ?] were recently proposed that can be proven (almost) tightly secure. However, these protocols were specifically designed to be tightly secure and none is standardized.

3.2 The TLS 1.3 Pre-shared Key Handshake Protocol

Overview.

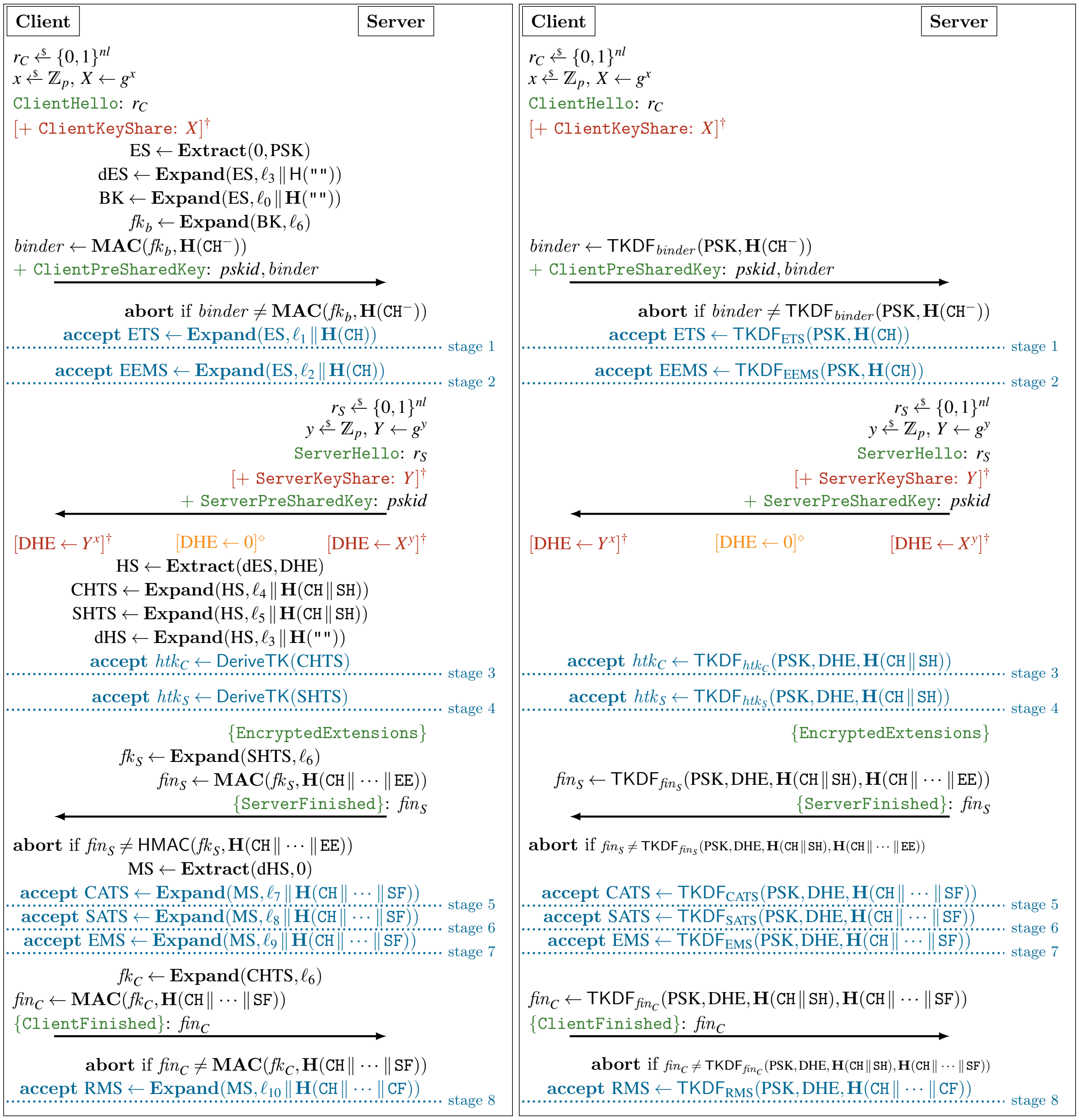
We consider the pre-shared key mode of TLS 1.3, used in a setting where both client and server already share a common secret, a so-called *pre-shared key* (PSK). A PSK is a cryptographic key which may either be manually configured, negotiated out-of-band, or (and most commonly) be obtained from a prior and possibly not PSK-based TLS session to enable fast *session resumption*. The TLS 1.3 PSK handshake comes in two flavors: PSK-only, where security is established from the pre-shared key alone, and PSK-(EC)DHE, which includes an (finite-field or elliptic-curve) Diffie–Hellman key exchange for added forward secrecy. Both PSK handshakes essentially consist of two phases (cf. Figure 3.1).

1. The client sends a random nonce and a list of offered pre-shared keys to the server, where each key is identified by a (unique) identifier *pskid*.[2] The server then selects one *pskid* from the list, and responds with another random nonce and the selected *pskid*. In PSK-(EC)DHE mode, client and server additionally perform a Diffie–Hellman key exchange, sending group elements along with the nonces and PSK identifiers. In both modes, the client also sends a so-called binder value, which applies a *message authentication code* (MAC) to the client's nonce and *pskid* (and the Diffie–Hellman share in PSK-(EC)DHE mode) and binds the PSK handshake to the (potential) prior handshake in which the used pre-shared key was established (see [**?**, 142] for analysis rationale behind the binder value).

2. Then client and server derive *unauthenticated* cryptographic keys from the PSK and the established Diffie–Hellman key (the latter only in (EC)DHE mode, of course). This includes, for instance, the *client* and *server handshake traffic keys* (htk_C and htk_S) used to encrypt the subsequent handshake messages, as well as *finished keys* (fk_C and fk_S) used to compute and exchange *finished messages*. The finished messages are MAC tags over all previous messages, ensuring that client and server have received all previous messages exactly as they were sent.

   After verifying the finished messages, client and server "accept" *authenticated* cryptographic keys, including the *client* and *server application traffic secret* (CATS and SATS), the *exporter master secret* (EMS), and the *resumption master secret* (RMS) for future session resumptions.

[2] In this work, we do not consider negotiation of pre-shared keys in situations where client and server share multiple keys, but focus on the case where client and server share only one PSK and the client therefore offers only a single *pskid*. However, we expect that our results extend to the general case as well.


Client
Server
$r_C \xleftarrow{\$} \{0,1\}^{nl}$
$x \xleftarrow{\$} \mathbb{Z}_p$, $X \leftarrow g^x$
ClientHello: r_C
[+ ClientKeyShare: X]†
$\mathrm{ES} \leftarrow \mathbf{Extract}(0, \mathrm{PSK})$
$\mathrm{dES} \leftarrow \mathbf{Expand}(\mathrm{ES}, \ell_3 \| \mathsf{H}(""))$
$\mathrm{BK} \leftarrow \mathbf{Expand}(\mathrm{ES}, \ell_0 \| \mathbf{H}(""))$
$fk_b \leftarrow \mathbf{Expand}(\mathrm{BK}, \ell_6)$
$binder \leftarrow \mathbf{MAC}(fk_b, \mathbf{H}(\mathtt{CH}^-))$
+ ClientPreSharedKey: $pskid, binder$
abort if $binder \neq \mathbf{MAC}(fk_b, \mathbf{H}(\mathtt{CH}^-))$
accept $\mathrm{ETS} \leftarrow \mathbf{Expand}(\mathrm{ES}, \ell_1 \| \mathbf{H}(\mathtt{CH}))$
stage 1
accept $\mathrm{EEMS} \leftarrow \mathbf{Expand}(\mathrm{ES}, \ell_2 \| \mathbf{H}(\mathtt{CH}))$
stage 2
$r_S \xleftarrow{\$} \{0,1\}^{nl}$
$y \xleftarrow{\$} \mathbb{Z}_p$, $Y \leftarrow g^y$
ServerHello: r_S
[+ ServerKeyShare: Y]†
+ ServerPreSharedKey: $pskid$
[DHE $\leftarrow Y^x$]†
[DHE $\leftarrow 0$]$^\diamond$
[DHE $\leftarrow X^y$]†
$\mathrm{HS} \leftarrow \mathbf{Extract}(\mathrm{dES}, \mathrm{DHE})$
$\mathrm{CHTS} \leftarrow \mathbf{Expand}(\mathrm{HS}, \ell_4 \| \mathbf{H}(\mathtt{CH} \| \mathtt{SH}))$
$\mathrm{SHTS} \leftarrow \mathbf{Expand}(\mathrm{HS}, \ell_5 \| \mathbf{H}(\mathtt{CH} \| \mathtt{SH}))$
$\mathrm{dHS} \leftarrow \mathbf{Expand}(\mathrm{HS}, \ell_3 \| \mathbf{H}(""))$
accept $htk_C \leftarrow \mathsf{DeriveTK}(\mathrm{CHTS})$
stage 3
accept $htk_S \leftarrow \mathsf{DeriveTK}(\mathrm{SHTS})$
stage 4
{EncryptedExtensions}
$fk_S \leftarrow \mathbf{Expand}(\mathrm{SHTS}, \ell_6)$
$fin_S \leftarrow \mathbf{MAC}(fk_S, \mathbf{H}(\mathtt{CH} \| \cdots \| \mathtt{EE}))$
{ServerFinished}: fin_S
abort if $fin_S \neq \mathsf{HMAC}(fk_S, \mathbf{H}(\mathtt{CH} \| \cdots \| \mathtt{EE}))$
$\mathrm{MS} \leftarrow \mathbf{Extract}(\mathrm{dHS}, 0)$
accept $\mathrm{CATS} \leftarrow \mathbf{Expand}(\mathrm{MS}, \ell_7 \| \mathbf{H}(\mathtt{CH} \| \cdots \| \mathtt{SF}))$
stage 5
accept $\mathrm{SATS} \leftarrow \mathbf{Expand}(\mathrm{MS}, \ell_8 \| \mathbf{H}(\mathtt{CH} \| \cdots \| \mathtt{SF}))$
stage 6
accept $\mathrm{EMS} \leftarrow \mathbf{Expand}(\mathrm{MS}, \ell_9 \| \mathbf{H}(\mathtt{CH} \| \cdots \| \mathtt{SF}))$
stage 7
$fk_C \leftarrow \mathbf{Expand}(\mathrm{CHTS}, \ell_6)$
$fin_C \leftarrow \mathbf{MAC}(fk_C, \mathbf{H}(\mathtt{CH} \| \cdots \| \mathtt{SF}))$
{ClientFinished}: fin_C
abort if $fin_C \neq \mathbf{MAC}(fk_C, \mathbf{H}(\mathtt{CH} \| \cdots \| \mathtt{SF}))$
accept $\mathrm{RMS} \leftarrow \mathbf{Expand}(\mathrm{MS}, \ell_{10} \| \mathbf{H}(\mathtt{CH} \| \cdots \| \mathtt{CF}))$
stage 8
Client
Server
$r_C \xleftarrow{\$} \{0,1\}^{nl}$
$x \xleftarrow{\$} \mathbb{Z}_p$, $X \leftarrow g^x$
ClientHello: r_C
[+ ClientKeyShare: X]†
$binder \leftarrow \mathsf{TKDF}_{binder}(\mathrm{PSK}, \mathbf{H}(\mathtt{CH}^-))$
+ ClientPreSharedKey: $pskid, binder$
abort if $binder \neq \mathsf{TKDF}_{binder}(\mathrm{PSK}, \mathbf{H}(\mathtt{CH}^-))$
accept $\mathrm{ETS} \leftarrow \mathsf{TKDF}_{\mathrm{ETS}}(\mathrm{PSK}, \mathbf{H}(\mathtt{CH}))$
stage 1
accept $\mathrm{EEMS} \leftarrow \mathsf{TKDF}_{\mathrm{EEMS}}(\mathrm{PSK}, \mathbf{H}(\mathtt{CH}))$
stage 2
$r_S \xleftarrow{\$} \{0,1\}^{nl}$
$y \xleftarrow{\$} \mathbb{Z}_p$, $Y \leftarrow g^y$
ServerHello: r_S
[+ ServerKeyShare: Y]†
+ ServerPreSharedKey: $pskid$
[DHE $\leftarrow Y^x$]†
[DHE $\leftarrow 0$]$^\diamond$
[DHE $\leftarrow X^y$]†
accept $htk_C \leftarrow \mathsf{TKDF}_{htk_C}(\mathrm{PSK}, \mathrm{DHE}, \mathbf{H}(\mathtt{CH} \| \mathtt{SH}))$
stage 3
accept $htk_S \leftarrow \mathsf{TKDF}_{htk_S}(\mathrm{PSK}, \mathrm{DHE}, \mathbf{H}(\mathtt{CH} \| \mathtt{SH}))$
stage 4
{EncryptedExtensions}
$fin_S \leftarrow \mathsf{TKDF}_{fin_S}(\mathrm{PSK}, \mathrm{DHE}, \mathbf{H}(\mathtt{CH} \| \mathtt{SH}), \mathbf{H}(\mathtt{CH} \| \cdots \| \mathtt{EE}))$
{ServerFinished}: fin_S
abort if $fin_S \neq \mathsf{TKDF}_{fin_S}(\mathrm{PSK}, \mathrm{DHE}, \mathbf{H}(\mathtt{CH} \| \mathtt{SH}), \mathbf{H}(\mathtt{CH} \| \cdots \| \mathtt{EE}))$
accept $\mathrm{CATS} \leftarrow \mathsf{TKDF}_{\mathrm{CATS}}(\mathrm{PSK}, \mathrm{DHE}, \mathbf{H}(\mathtt{CH} \| \cdots \| \mathtt{SF}))$
stage 5
accept $\mathrm{SATS} \leftarrow \mathsf{TKDF}_{\mathrm{SATS}}(\mathrm{PSK}, \mathrm{DHE}, \mathbf{H}(\mathtt{CH} \| \cdots \| \mathtt{SF}))$
stage 6
accept $\mathrm{EMS} \leftarrow \mathsf{TKDF}_{\mathrm{EMS}}(\mathrm{PSK}, \mathrm{DHE}, \mathbf{H}(\mathtt{CH} \| \cdots \| \mathtt{SF}))$
stage 7
$fin_C \leftarrow \mathsf{TKDF}_{fin_C}(\mathrm{PSK}, \mathrm{DHE}, \mathbf{H}(\mathtt{CH} \| \mathtt{SH}), \mathbf{H}(\mathtt{CH} \| \cdots \| \mathtt{SF}))$
{ClientFinished}: fin_C
abort if $fin_C \neq \mathsf{TKDF}_{fin_C}(\mathrm{PSK}, \mathrm{DHE}, \mathbf{H}(\mathtt{CH} \| \mathtt{SH}), \mathbf{H}(\mathtt{CH} \| \cdots \| \mathtt{SF}))$
accept $\mathrm{RMS} \leftarrow \mathsf{TKDF}_{\mathrm{RMS}}(\mathrm{PSK}, \mathrm{DHE}, \mathbf{H}(\mathtt{CH} \| \cdots \| \mathtt{CF}))$
stage 8

Legend

MSG: Y	message MSG sent, containing Y	$\mathtt{CH}^-$	partial ClientHello up to (incl.) $pskid$
+ MSG	extension sent within previous message	ℓ_x	label value, distinct for distinct x
{MSG}	MSG sent AEAD-encrypted with htk_C/htk_S		
$[\ldots]^\dagger$	present only in PSK-(EC)DHE		
$[\ldots]^\diamond$	present only in PSK		

$\mathsf{DeriveTK}(HTS) := \mathbf{Expand}(HTS, \ell_{11} \| \mathsf{Th}(""), hl) \| \mathbf{Expand}(HTS, \ell_{12} \| \mathsf{Th}(""), ivl)$
(traffic key computation, deriving a hl-bit key and a ivl-bit IV)

Figure 3.1. TLS 1.3 PSK and PSK-(EC)DHE handshake modes with (optional) 0-RTT keys (stages 1 and 2), with detailed key schedule (left) and our representation of the key schedule through functions TKDF_x (right), explained in the text. Centered computations are executed by both client and server with their respective messages received, and possibly at different points in time. Dotted lines indicate the derivation of session (stage) keys together with their stage number. The labels ℓ_x are distinct for distinct index x, see Table 3.1 for their definition.

Value	Label	Value	Label
dES	$\ell_3 =$ `"derived"`	htk_C	$\ell_{11} =$ `"key"` & $\ell_{12} =$ `"iv"`
BK	$\ell_0 =$ `"ext binder"` / `"res binder"`	htk_S	$\ell_{11} =$ `"key"` & $\ell_{12} =$ `"iv"`
fk_b	$\ell_6 =$ `"finished"`	fk_S	$\ell_6 =$ `"finished"`
ETS	$\ell_1 =$ `"c e traffic"`	CATS	$\ell_7 =$ `"c ap traffic"`
EEMS	$\ell_2 =$ `"e exp master"`	SATS	$\ell_8 =$ `"s ap traffic"`
CHTS	$\ell_4 =$ `"c hs traffic"`	EMS	$\ell_9 =$ `"exp master"`
SHTS	$\ell_5 =$ `"s hs traffic"`	fk_C	$\ell_6 =$ `"finished"`
dHS	$\ell_3 =$ `"derived"`	RMS	$\ell_{10} =$ `"res master"`

Table 3.1. Definitions of the short labels used in Figure 3.1. We simplify the labeling of **Expand** in our presentation. In the specification each **Expand** is not only labeled by $\ell \| H$ for some label ℓ and some hash H, but it is prefixed by the output length of the respecitive **Expand** call and the constant label "`tls13` ". As the output length for all of the above calls is equal (namely, the output length hl of $\mathbf{H}$), we leave this constant prefix out to reduce complexity.

Detailed specification.

For our proofs we will need fully-specified descriptions for each of the TLS 1.3 PSK and PSK-(EC)DHE handshake protocols. Pseudocode for these protocols can be found in Figure 3.1, where we let $(\mathbb{G}, p, g)$ be a cyclic group of prime order p such that $\mathbb{G} = \langle g \rangle$.

The two descriptions on the left and right in Figure 3.1 show the same protocol, but they use different abstractions to highlight how we capture the complex way TLS 1.3 calls its hash function. This one hash function is used in some places to condense transcripts, in others to help derive session keys, and in still others as part of a message authentication code. We call this function H, and let its output length be hl bits so that we have $\mathbf{H}\colon \{0,1\}^* \to \{0,1\}^{hl}$. Depending on the choice of ciphersuite, TLS 1.3 instantiates H with either `SHA256` or `SHA384` [171]. In our security analysis, we will model H as a random oracle.

On the left-hand side of Figure 3.1, we distinguish four named subroutines of TLS 1.3 which use H for different purposes:

- A message authentication code $\mathbf{MAC}\colon \{0,1\}^{hl} \times \{0,1\}^* \to \{0,1\}^{hl}$, which calls H via the HMAC function $\mathbf{MAC}(K,M) := \mathsf{HMAC}[\mathsf{H}](K,M)$ where

$$\mathsf{HMAC}[\mathsf{H}](K,M) := \mathsf{H}((K \| 0^{bl-hl}) \oplus \mathsf{opad}) \| \mathsf{H}((K \| 0^{bl-hl} \oplus \mathsf{ipad}) \| M))$$

Here opad and ipad are bl-bit strings, where each byte of opad and ipad is set to the hexadecimal value `0x5c`, resp. `0x36`. We have $bl = 512$ when `SHA256` is used and $bl = 512$ for `SHA384`. When modeling `SHA256` resp. `SHA384` as a random oracle, we keep the corresponding value of bl.

- $\mathbf{Extract}, \mathbf{Expand} \colon \{0,1\}^{hl} \times \{0,1\}^* \to \{0,1\}^{hl}$, two subroutines for *extracting* and *expanding* key material in the key schedule, following the HKDF key derivation paradigm of Krawczyk [141, 144]. These functions are defined
  - $\mathbf{Extract}(K,M) := \mathsf{HKDF.Extract}(K,M) = \mathbf{MAC}(K,M)$.
  - $\mathbf{Expand}(K,M) := \mathsf{HKDF.Expand}(K,M) = \mathbf{MAC}(K,M \| \texttt{0x01})$.[3]

Despite the new naming conventions, this abstraction closely mimics the TLS 1.3 standard: **MAC**, **Extract**, and **Expand** can be read as more generic ways of referring to the HMAC, HKDF.Extract, and HKDF.Expand algorithms [143, 144].

The right-hand side of Figure 3.1 separates the key derivation functions for each first-class key as well as the binder and finished MAC values derived. This way of modeling TLS 1.3 makes it easier to establish key independence for the many keys computed in the key schedule, as we will see in Section 3.4. We introduce 11 functions TKDF_{binder}, $\mathsf{TKDF}_{\mathrm{ETS}}$, $\mathsf{TKDF}_{\mathrm{EEMS}}$, TKDF_{htk_C}, TKDF_{fin_C}, TKDF_{htk_S}, TKDF_{fin_S}, $\mathsf{TKDF}_{\mathrm{CATS}}$, $\mathsf{TKDF}_{\mathrm{SATS}}$, $\mathsf{TKDF}_{\mathrm{EMS}}$, and $\mathsf{TKDF}_{\mathrm{RMS}}$ (indexed by the value they derive) and use them to abstract away many intermediate computations. Note that we are not changing the protocol, though: we define each TKDF function to capture the same steps it replaces.

Take as an example TKDF_{fin_S}, the function used to derive the MAC in the `ServerFinished` message. In the prior abstraction, a session would first use the key schedule to derive a finished key fk_S from the hashed transcript and the secrets PSK and DHE. It would then call **MAC**, keyed with fk_S, to generate the `ServerFinished` message authentication code on the hashed transcript and encrypted extensions. Accordingly, we define $\mathsf{TKDF}_{fin_S} \colon \{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$ as in Figure 3.2. In the protocol, TKDF_{fin_S} takes inputs the pre-shared key PSK and

[3] HKDF.Expand [144] is defined for any output length (given as third parameter). In TLS 1.3, **Expand** always derives at most hl bits, which can be trimmed from a hl-bit output; we hence in most places omit the output length parameter.

```
$\mathsf{TKDF}_{\mathit{fin}_S}(\mathrm{PSK}, \mathrm{DHE}, \mathsf{h}_1, \mathsf{h}_2)$:
    $\mathrm{ES} \leftarrow \mathbf{Extract}(0, \mathrm{PSK})$
    $\mathrm{dES} \leftarrow \mathbf{Expand}(\mathrm{ES}, \ell_3 \,\|\, \mathsf{Th}(\texttt{""}))$
    $\mathrm{HS} \leftarrow \mathbf{Extract}(\mathrm{dES}, \mathrm{DHE})$
    $\mathrm{SHTS} \leftarrow \mathbf{Expand}(\mathrm{HS}, \ell_5 \,\|\, \mathsf{h}_1)$
    $\mathit{fk}_S \leftarrow \mathbf{Expand}(\mathrm{SHTS}, \ell_6)$
    $\mathit{fin}_S \leftarrow \mathbf{MAC}(\mathit{fk}_S, \mathsf{h}_2)$
    return $\mathit{fin}_S$
```

Figure 3.2. Definition of $\mathsf{TKDF}_{\mathit{fin}_S}$, deriving the `ServerFinished` MAC.

Diffie–Hellman secret DHE and hash digests $\mathsf{h}_1 = \mathsf{Th}(\texttt{CH} \,\|\, \texttt{SH})$ and $\mathsf{h}_2 = \mathsf{Th}(\texttt{CH} \,\|\, \cdots \,\|\, \texttt{EE})$, and it outputs a MAC tag for the `ServerFinished` message. The remaining key derivation functions are defined the same way; we give their signatures

below for completeness.

1. $\mathsf{TKDF}_{\mathit{binder}}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
2. $\mathsf{TKDF}_{\mathrm{ETS}}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
3. $\mathsf{TKDF}_{\mathrm{EEMS}}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
4. $\mathsf{TKDF}_{\mathit{htk}_C}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl+ivl}$
5. $\mathsf{TKDF}_{\mathit{fin}_C}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
6. $\mathsf{TKDF}_{\mathit{htk}_S}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl+ivl}$
7. $\mathsf{TKDF}_{\mathit{fin}_S}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
8. $\mathsf{TKDF}_{\mathrm{CATS}}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
9. $\mathsf{TKDF}_{\mathrm{SATS}}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
10. $\mathsf{TKDF}_{\mathrm{EMS}}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
11. $\mathsf{TKDF}_{\mathrm{RMS}}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$

Note that the definition of the 11 functions induces a lot of redundancy as we derive every value independently and therefore compute intermediate values (e.g., ES, dES, and HS) multiple times over the execution of the handshake. However, this is only conceptual. Since the computations of these intermediate values are deterministic, the intermediate values will be the same for the same inputs and could be cached.

3.3 Code-based MSKE Model for PSK Modes

We formalize security of the TLS 1.3 PSK modes in a game-based multi-stage key exchange (MSKE) model, adapted primarily from that of Dowling et al. [92]. We fully specify our model in pseudocode in Figures 3.3 and 3.4. We adopt the explicit authentication property from the model of Davis and Günther [82] and capture forward secrecy by following the model of Schwabe et al. [?].

3.3.1 Key Exchange Syntax

In our security model, the adversary interacts with *sessions* executing a key exchange protocol KE. For the definition of the security experiment it will be useful to have a unified, generic interface to the algorithms implementing KE, which can then be called from the various procedures defining the security experiment to run KE. Therefore, we first formalize a general syntax for protocols.

We assume that pairs of users share long-term symmetric keys (pre-shared keys), which are chosen uniformly at random from a set $\mathsf{KE.PSKS}$.[4] We allow users to share multiple pre-shared keys, maintained in a list pskeys, and require that each user uses any key only in a fixed role (i.e., as client *or* server) to avoid the Selfie attack [?]. We do not cover PSK negotiation; each session will know at the start of the protocol which key it intends to use.

New sessions are created via the algorithm $\mathsf{Activate}$. This algorithm takes as input the new session's own user, identified by some ID u, the user ID *peerid* of the intended communication partner, a pre-shared key PSK, and a role identifier—$\mathsf{initiator}$ (client) or $\mathsf{responder}$ (server)—that determines whether the session will send or receive the first protocol message. It returns the new session π_u^i, which is identified by its user ID u and a unique index i so that a single user can execute many sessions.

Existing sessions send and receive messages by executing the algorithm Run. The inputs to Run are an existing session π_u^i and a message m it has received. The algorithm processes the message, updates the state of π_u^i, and returns the next protocol message m' on behalf of the

[4] While our results can be generalized to any distribution on $\mathsf{KE.PSKS}$ (based on its min-entropy), for simplicity, we focus on the uniform distribution in this work.

session. Run also maintains the status of π_u^i, which can have one of three values: $\mathsf{running}$ when it is awaiting the next protocol message, $\mathsf{accepted}$ when it has established a session key, and $\mathsf{rejected}$ if the protocol has terminated in failure.

In a multi-stage protocol, sessions accept multiple session keys while running; we identify each with a numbered *stage*. A protocol may accept several stages/keys while processing a single message, and TLS 1.3 does this. In order to handle each stage individually, our model adds artificial pauses after each acceptance to allow the adversary to interact with the sessions upon each stage accepting (beyond, as usual, each message exchanged). When a session π_u^i accepts in stage s while executing Run, we require Run to set the status of π_u^i to $\mathsf{accepted}_s$ and terminate. We then define a special "continue" message. When session π_u^i in state $\mathsf{accepted}_s$, receives this message it calls Run again, updates its status to $\mathsf{running}_{s+1}$ and continues processing from the point where it left off.

3.3.2 Key Exchange Security

We define key exchange security via a real-or-random security game, formalized through Figures 3.3 and 3.4.

Game oracles.

In this security game, the adversary $\mathscr{A}$ has access to seven oracles: INIT, NEWSECRET, SEND, REVSESSIONKEY, REVLONGTERMKEY, TEST, and FIN, as well as any random oracles the protocol defines. The game begins with a call to INIT, which samples a challenge bit b. It ends when the adversary calls FIN with a guess b' at the challenge bit. We say the adversary "wins" the game if FIN returns true.

The adversary can establish a random pre-shared key between two users by calling NEWSECRET.[5] It can corrupt existing users' pre-shared keys via the oracle REVLONGTERMKEY. The SEND oracle creates new protocol sessions and processes protocol messages on the behalf of existing sessions. The REVSESSIONKEY oracle reveals a session's accepted session key. Finally,

[5] Our model stipulates that pre-shared keys are sampled uniformly random and honestly. One could additionally allow the registration of biased or malicious PSKs, akin to models treating, e.g., the certification of public keys [60]. While this would yield a theoretically stronger model, we consider a simpler model reasonable, because we expect most PSKs used in practice to be random keys established in prior protocol sessions. Furthermore, we consider tightness as particularly interesting when "good" PSKs are used, since low-entropy PSKs might decrease the security below what is achieved by (non)-tight security proofs, anyway.

```
$G^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE},\mathscr{A}}$

INIT:
  $\mathsf{time} \leftarrow 0;$
  $b \xleftarrow{\$} \{0,1\}$

NEWSECRET$(u, v, pskid)$:
  $\mathsf{time} \leftarrow \mathsf{time} + 1$
  if $\mathsf{pskeys}[(u, v, pskid)] \neq \bot$
    return $\bot$
  $\mathsf{pskeys}[(u, v, pskid)] \leftarrow\!\!\$\ \mathsf{KE.PSKS}$
  $\mathsf{revpsk}_{(u,v,pskid)} \leftarrow \infty$
  return $pskid$

SEND$(u, i, m)$:
  $\mathsf{time} \leftarrow \mathsf{time} + 1$
  if $\pi_u^i = \bot$ then
    $(peerid, pskid, role) \leftarrow m$
    if $role = \mathsf{initiator}$
      then $psk \leftarrow \mathsf{pskeys}[(u, peerid, pskid)]$
      else $psk \leftarrow \mathsf{pskeys}[(peerid, u, pskid)]$
    $(\pi_u^i, m') \leftarrow\!\!\$\ \mathsf{Activate}(u, peerid, psk, role)$
  else
    $(\pi_u^i, m') \leftarrow\!\!\$\ \mathsf{Run}(u, \pi_u^i.psk, \pi_u^i, m)$
  if $\pi_u^i.status = \mathsf{accepted}_{\pi_u^i.stage}$ then
    $stage \leftarrow \pi_u^i.stage$
    $\pi_u^i.\mathsf{accepted}[stage] \leftarrow \mathsf{time}$
    if $\mathsf{repr}[\pi_u^i.sid[stage]] \neq \bot$ then
      $\pi_u^i.skey[stage] \leftarrow \mathsf{repr}[\pi_u^i.sid[stage]]$
    $\pi_u^i.\mathsf{untampered}[stage] \leftarrow \exists \pi_v^j$ with $\pi_v^j.cid_{\pi_u^i.role}[stage] =$
  $\pi_u^i.cid_{\pi_u^i.role}[stage]$
  return $m'$

REVSESSIONKEY$(u, i, s)$:
  $\mathsf{time} \leftarrow \mathsf{time} + 1$
  if $\pi_u^i = \bot$ or $\pi_u^i.\mathsf{t_{acc}}[s] = \infty$ then
    return $\bot$
  $\pi_u^i.\mathsf{revealed}[s] \leftarrow \mathsf{true}$
  return $\pi_u^i.skey[s]$
```

```
REVLONGTERMKEY$(u, v, pskid)$:
  $\mathsf{time} \leftarrow \mathsf{time} + 1$
  $\mathsf{revpsk}_{(u,v,pskid)} \leftarrow \mathsf{time}$
  return $\mathsf{pskeys}[(u, v, pskid)]$

TEST$(u, i, s)$:
  $\mathsf{time} \leftarrow \mathsf{time} + 1$
  if $s \in \mathsf{INT}$
        and $\exists \pi_v^j : \pi_v^j.sid[s] = \pi_u^i.sid[s]$
        and $\pi_v^j.\mathsf{t_{acc}}[s] < \infty$
        and $\pi_v^j.status \neq \mathsf{accepted}_s$ then
    return $\bot$
    // can only test internal keys if all sessions
  having accepted that key have not moved
  on with the protocol
  if $\pi_u^i = \bot$ or $\pi_u^i.\mathsf{t_{acc}}[s] = \infty$ or $\neg \pi_u^i.\mathsf{tested}[s]$ then
    return $\bot$
  $\pi_u^i.\mathsf{tested}[s] \leftarrow \mathsf{time}$
  $\mathscr{T} \leftarrow \mathscr{T} \cup \{(\pi_u^i, s)\}$
  $k_0 \leftarrow \pi_u^i.skey[s]$
  $k_1 \xleftarrow{\$} \mathsf{KE.KS}[s]$
  if $s \in \mathsf{INT}$ then
    $\forall \pi_v^j : \pi_v^j.sid[s] = \pi_u^i.sid[s]$
        and $\pi_v^j.status = \mathsf{accepted}_s$
      $\pi_v^j.skey[s] \leftarrow k_b$
    $\mathsf{repr}[\pi_u^i.sid[s]] \leftarrow k_b$
  return $k_b$

FIN$(b')$:
  if $\neg\mathsf{Sound}$ then
    return 1
  if $\neg\mathsf{ExplicitAuth}$ then
    return 1
  if $\neg\mathsf{Fresh}$ then
    $b' \leftarrow 0$
  return $[[b = b']]$

RO$(i, X)$:
  $\mathsf{time} \leftarrow \mathsf{time} + 1$
  return $\mathrm{RO}_i(X)$
```

Figure 3.3. Multi-stage key exchange (MSKE) security game for a key exchange protocol KE with pre-shared keys. Predicates Fresh, ExplicitAuth, and Sound are defined in Figure 3.4. The functions RO_i correspond to the (independent) random oracles available to the adversary.

the TEST oracle servers as the challenge oracle: it returns the real session key of a target session or an independent one sampled randomly from the session key space $\mathsf{KE.KS}[s]$ of the respective stage s, depending on the value of the challenge bit b.

Fresh:

```
for each $(\pi_u^i, s) \in \mathscr{T}$
    $\mathsf{t_{Test}} \leftarrow \pi_u^i.\mathsf{tested}[s]$
    if $\pi_u^i.\mathsf{revealed}[s]$ then
        return false    // tested session may not be revealed
    if $\exists \pi_v^j \neq \pi_u^i : \pi_v^j.sid[s] = \pi_u^i.sid[s]$
        and $(\pi_v^j.\mathsf{tested}[s]$ or $\pi_v^j.\mathsf{revealed}[s])$ then
        return false    // tested session's partnered session may not be tested or revealed
    if $\pi_u^i.\mathsf{t_{acc}}[\mathsf{FS}[s, \mathsf{fs}]] < \mathsf{t_{Test}}$
        if $\mathsf{revpsk}_{(u, \pi_u^i.peerid, \pi_u^i.pskid)} < \pi_u^i.\mathsf{t_{acc}}[\mathsf{FS}[s, \mathsf{fs}]]$ and $\neg\pi_u^i.\mathsf{untampered}[\mathsf{FS}[s, \mathsf{fs}]]$ then
        return false    // Sessions with forward secrecy are fresh if they attained fs before their PSK was corrupted, or if they have a contributive partner (no tampering).
    else if $\pi_u^i.\mathsf{t_{acc}}[\mathsf{FS}[s, \mathsf{wfs2}]] < \mathsf{t_{Test}}$
        if $\mathsf{revpsk}_{(u, \pi_u^i.peerid, \pi_u^i.pskid)}$ and $\neg\pi_u^i.\mathsf{untampered}[\mathsf{FS}[s, \mathsf{wfs2}]]$ then
            return false    // Sessions with weak forward secrecy 2 are fresh if the PSK was never corrupted, or if they have a contributive partner.
    else if $\mathsf{revpsk}_{\{u, \pi_u^i.peerid\}, \pi_u^i.pskid}$ then
        return false    // Sessions with no forward secrecy are fresh if the PSK was never corrupted.
return true
```

ExplicitAuth:

```
if $\forall \pi_u^i, s$:
$s' \leftarrow \mathsf{EAUTH}[\pi_u^i.role, s]$
$\pi_u^i.\mathsf{t_{acc}}[s'] < \infty$
    and $\pi_u^i.\mathsf{t_{acc}}[s] < \infty$
    and $\pi_u^i.\mathsf{t_{acc}}[s'] < \mathsf{revpsk}_{(u, \pi_u^i.peerid, \pi_u^i.pskid)}$
    and $\pi_u^i.\mathsf{t_{acc}}[s'] < \infty$
// all sessions accepting in explicitly authenticated stages whose PSK was not corrupted before acceptance of the stage at which explicit authentication was (perhaps retroactively) established...
$\implies \exists \pi_v^j : \pi_u^i.sid[s'] = \pi_v^j.sid[s']$
        and $\pi_u^i.peerid = v$
        and $\pi_u^i.pskid = \pi_v^j.pskid$
// ... have a partnered session in that stage ...
// ... agreeing on the peerid and pre-shared key...
        and $(\pi_v^j.\mathsf{t_{acc}}[s] < \mathsf{time} \implies \pi_v^j.sid[s] = \pi_u^i.sid[s])$
// ... and partnered in stage $s$ (upon acceptance)
    return true
```

Sound:

```
if $\exists s$, distinct $\pi_u^i$, $\pi_v^j$, $\pi_w^k$ with $\pi_u^i.sid[s] = \pi_v^j.sid[s] = \pi_w^k.sid[s] \neq \bot$
and $\mathsf{REPLAY}[s] = \mathsf{false}$ then
    return false
// no triple sid match, except for replayable stages
if $\exists \pi_u^i, \pi_v^j$, $s$ with
    $\pi_u^i.sid[s] = \pi_v^j.sid[s] \neq \bot$ and
    $\pi_u^i.role = \pi_v^j.role$ and
    $(\mathsf{REPLAY}[s] = \mathsf{false}$ or $\pi_u^i.role = \mathsf{initiator})$ then
    return false
// partnering implies different roles (except for responders in replayable stages)
if $\exists \pi_u^i, \pi_v^j$, $s$ with
    $\pi_u^i.sid[s] = \pi_v^j.sid[s] \neq \bot$ and
$(\pi_u^i.cid_{\mathsf{initiator}}[s] \neq \pi_v^j.cid_{\mathsf{initiator}}[s]$ or $\pi_u^i.cid_{\mathsf{responder}}[s] \neq \pi_v^j.cid_{\mathsf{responder}}[s])$
    return false
// partnering implies matching cids
if $\exists \pi_u^i, \pi_v^j$ and $s \neq t$ such that
    $\pi_u^i.sid[s] = \pi_v^j.sid[t]$
    return false
// different stages implies different sids
if $\exists \pi_u^i, \pi_v^j$, $s$ with
    $\pi_u^i.sid[s] = \pi_v^j.sid[s] \neq \bot$
    and $\pi_u^i.peerid \neq v$
    or $\pi_v^j.peerid \neq u$ or $\pi_u^i.pskid \neq \pi_v^j.pskid$ then
// partnering implies agreement on peer IDs and PSKs
    return false
if $\exists \pi_u^i, \pi_v^j$, $s$ with
    $\pi_u^i.\mathsf{t_{acc}}[s] < \mathsf{time}$
    and $\pi_v^j.\mathsf{t_{acc}}[s] < \mathsf{time}$
    and $\pi_u^i.sid[s] = \pi_v^j.sid[s] \neq \bot$,
    but $\pi_u^i.skey[s] \neq \pi_v^j.skey[s]$ then
// partnering implies same key
    return false
return true
```

Figure 3.4. Predicates Fresh, ExplicitAuth, and Sound for the MSKE pre-shared key model.

Protocol properties.

Keys established in different stages possess different security attributes, which are defined as part of the key exchange protocol: replayability, forward secrecy level, and authentication level. Certain stages, whose indices are tracked in a list INT, produce "internal" keys intended for use only within the key exchange protocol; these keys may only be TESTed at the time of acceptance of this particular key, but not later. This is because otherwise such keys may be trivially distinguishable from random, e.g., via trial decryption, due to the fact that they are used within the protocol. To avoid a trivial distinguishing attack, we force the rest of the protocol execution to be consistent with the result of such a TEST. That is, a tested internal key is replaced in the protocol with whatever the TEST returns to the adversary (which is either the real internal key or an independent random key). The remaining stages produce "external" keys which may be tested at any time after acceptance.

For some protocols, it may be possible that a trivial replay attack can achieve that several sessions agree on the same session key for stage s, but this is not considered an "attack". For example, in TLS 1.3 PSK an adversary can always replay the `ClientHello` message to multiple sessions of the same server, which then all derive the same ETS and EEMS keys (cf. Figure 3.1). To specify that such a replay is not considered a protocol weakness, and thus should not be considered a valid "attack", the protocol specification may define $\mathsf{REPLAY}[s]$ to true for a stage s. $\mathsf{REPLAY}[s]$ is set to false by default.

As we focus on protocols which rely on (pre-authenticated) pre-shared keys, our model encodes that all protocol stages are at least *implicitly* mutually authenticated in the sense of Krawczyk [140], i.e., a session is guaranteed that any established key can only be known by the intended partner. Some stages will further be *explicitly* authenticated, either immediately upon acceptance or retroactively upon acceptance of a later state. Additionally, the stage at which explicit authentication is achieved may differ between the initiator and responder roles. For each stage s and role r, the key exchange protocol specification states in $\mathsf{EAUTH}[r,s]$ the stage t from whose acceptance stage s derives explicit authentication for the session in role r. Note that the stage-s key is not authenticated until both stages s and $\mathsf{EAUTH}[r,s]$ have been accepted. If the stage-s key will never be explicitly authenticated for role r, we set $\mathsf{EAUTH}[r,s] = \infty$.

We use a predicate ExplicitAuth (cf. Figure 3.4) to require the existence of an honest partner for explicitly authenticated stages upon both parties' completion of the protocol, except when the session's pre-shared key was corrupted prior to accepting the explicitly-authenticating stage (as in that case, we anticipate the adversary can trivially forge any authentication mechanism).

Motivated by TLS 1.3, it might be the case that initiator and responder sessions achieve slightly different guarantees of authentication. While responders in TLS 1.3 are guaranteed the existence of an honest partner in any explicitly authenticated stage, initiators cannot guarantee that their partner has received their final message. This issue was first raised by FGSW [102] and led to their definitions of "full" and "almost-full" key confirmation; it was then extended to "full" and "almost-full" explicit authentication by DFW [**?**]. Our definitions for responders and initiators respectively resemble the latter two notions most closely, but we rely on session identifiers instead of "key confirmation identifiers".

We consider three levels of forward secrecy inspired by the KEMTLS work of Schwabe, Stebila, and Wiggers [**?**]: no forward secrecy, weak forward secrecy 2 (wfs2), and full forward secrecy (fs). As for authentication, each stage may retroactively upgrade its level of forward secrecy upon the acceptance of later stages, and forward secrecy may be established at different stages for each role. For each stage s and role r, the stage at which wfs2, resp. fs, is achieved is stated in $\mathsf{FS}[r, s, \mathsf{wfs2}]$, resp. $\mathsf{FS}[r, s, \mathsf{fs}]$, by the key exchange protocol.

The definition of weak forward secrecy 2 states that a session key with wfs2 should be indistinguishable as long as (1) that session has received the relevant messages from an honest partner (formalized via matching contributive identifiers below, we say: "has an honest contributive partner") or (2) the pre-shared key was never corrupted. Full forward secrecy relaxes condition (2) to forbid corruption of the pre-shared key only before acceptance of the stage that retroactively provides full forward secrecy. We capture these notions of forward secrecy in a predicate Fresh(cf. Figure 3.4), which uses the log of events to check whether any tested session key is trivially distinguishable (e.g., through the session or its partnered being revealed, or forward secrecy requirements violated). With forward secrecy encoded in Fresh, our long-term key corruption oracle (RevLongTermKey), unlike in the model of [92], handles all corruptions the same way, regardless of forward secrecy.

Session and game variables.

Sessions π_u^i and the security game itself maintain several variables; we indicate the former in *italics*, the latter in sans-serif font.

The game uses a counter time, initialized to $\mathsf{0}$ and incremented with any oracle query the adversary makes, to order events in the game log for later analysis. When we say that an event happens at a certain "time", we mean the current value of the time counter. The list pskeys contains, as discussed above, all pre-shared keys, indexed by a tuple $(u, v, pskid)$ containing the two users' IDs (u using the key only in the initiator role, v only in the reponder role), and a unique string identifier. The list revpsk, indexed like pskeys, tracks the time of each pre-shared key corruption, initialized to $\mathsf{revpsk}_{(u,v,pskid)} \leftarrow \infty$. (In boolean expressions, we write $\mathsf{revpsk}_{(u,v,pskid)}$ as a shorthand for $\mathsf{revpsk}_{(u,v,pskid)} \neq \infty$.)

Each session π_u^i, identified by (adversarially chosen) user ID and a unique session ID, furthermore tracks the following variables:

- $status \in \{\mathsf{running}_s, \mathsf{accepted}_s, \mathsf{rejected}_s \mid s \in [1, \ldots, \mathsf{STAGES}]\}$, where STAGES is the total number of stages of the considered protocol. The status should be $\mathsf{accepted}_s$ immediately after the session accepts the stage-s key, $\mathsf{rejected}_s$ after it rejects stage s (but may continue running; e.g., rejecting 0-RTT data), and $\mathsf{running}_s$ for some stage s otherwise.

- $peerid$. The identity of the session's intended communication partner.

- $pskid$. The identifier of the session's pre-shared key.

- $\mathsf{t_{acc}}[s]$. For each stage s, the time (i.e., the value of the time counter) at which the stage s key was accepted. Initialized to ∞.

- $\mathsf{revealed}[s]$. A boolean denoting whether the stage s key has been leaked through a RevSessionKey query. Initialized to false.

- $\mathsf{tested}[s]$. The time at which the stage s key was tested. Initialized to ∞ before any Test query occurs. (In boolean expressions, we write $\mathsf{tested}[s]$ as a shorthand for $\mathsf{tested}[s] \neq \infty$.)

- $sid[s]$. The session identifier for each stage s, used to match honest communication partners within each stage.

- $\mathit{skey}[s]$. The key accepted at each stage.

- $\mathit{cid}_{\mathsf{initiator}}[s]$ and $\mathit{cid}_{\mathsf{responder}}[s]$. The contributive identifiers for each stage s, where $\mathit{cid}_{\mathit{role}}[s]$ identifies the communication part that a session in role *role* must have honestly received in order to be allowed to be tested in certain scenarios (cf. the freshness definition in the Fresh predicate). Unlike prior models, each session maintains a contributive identifiers for each role; one for itself and one for its intended partner. This enables more fine-grained testing of session stages in our model.

The predicate Sound (cf. Figure 3.4) captures that variables are properly assigned, in particular that session identifiers uniquely identify a partner session (except for replayable stages) and that partnering implies agreement on (distinct) roles, contributive identifiers, peer identities and the pre-shared key used, as well as the established session key.

Definition 10 (Multi-stage key exchange security)**.** Let KE be a key exchange protocol and $G^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE},\mathscr{A}}$ be the key exchange security game defined in Figures 3.3 and 3.4. We define

$$\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) := 2 \cdot \max_{\mathscr{A}} \Pr\left[\mathrm{G}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE},\mathscr{A}} \Rightarrow 1\right] - 1,$$

where the maximum is taken over all adversaries, denoted $(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}})$-$\mathsf{KE\text{-}SEC}$-*adversaries*, running in time at most t and making at most q_{NS}, q_{S}, q_{RS}, q_{RL}, q_{T}, resp. q_{RO} queries to their respective oracles NewSecret, Send, RevSessionKey, RevLongTermKey, Test, and RO.

In the random oracle model, we treat hash functions like `SHA256` as uniformly sampled random functions. Honest parties and adversaries alike access these functions via additional oracles in the security game. These are the *random oracles*. These random functions will be sampled from a set called a *function space* at the start of a security game. Alternatively, the random oracle can *lazily sample* responses to each query as they are needed. While we typically use the latter (lazily-sampled) model in key exchange security proofs, we will focus on the former conceptual view here.

Let us give an example. When we model the TLS 1.3 protocol in the ROM, we will equip our protocol definition with a function space parameter FS. We set this parameter according to the portion of the protocol we wish to model as a random oracle. If we wish to replace the hash function H with a random oracle RO_{H}, then we would set FS to be the set of all functions the set of all functions with domain $\{0,1\}^*$ and range $\{0,1\}^{hl}$. The KE security game would sample RO_{H} from FS in its INIT routine, then provide oracle access to RO_{H} to all parties. This notation also captures protocols which use multiple random oracles. If we wish to use two independent random oracles, say RO_1 and RO_2, then we would define an *arity*-2 function space FS, which is a set of tuples each containing two functions. Let FS_1, resp. FS_2 be the set from which RO_1, RO_2 should be drawn. Then we set $\mathsf{FS} = \{(F_1, F_2) : F_1 \in \mathsf{FS}_1 \text{ and } F_2 \in \mathsf{FS}_2\}$. We call FS_1 and FS_2 the subspaces of FS. A security game provides access to F_1 and F_2 through a single oracle RO that takes two arguments; the first is the index of the function to be queried and the second is the contents of the query. So $\mathrm{RO}(i,X)$ will return $F_i(X)$. We can also cast an arity-1 function space in this notation by identifying each function F with the tuple (F), but we will typically omit the parentheses and index argument when only one random oracle is used.

Indifferentiability was originally developed by Maurer, Renner, and Holenstein [156], and it has been used to prove security for hash functions built from public compression functions. More generally, it gives a framework to show the security of a transition between any two function spaces. We'll call these spaces SS (for "starting space") and ES (for "ending space"). A *construction* of ES from SS is an algorithm $\mathbf{C}$ that outputs elements of ES given an oracle $\mathrm{RO}_{\mathsf{SS}} \in \mathsf{SS}$. We may use the notation $\mathbf{C} : \mathsf{SS} \to \mathsf{ES}$. We then say that $\mathbf{C}$ is "indifferentiable" if for any function $\mathrm{RO}_{\mathsf{SS}}$ sampled from SS, $\mathbf{C}[\mathrm{RO}]$ behaves indistinguishably from a function $\mathrm{RO}_{\mathsf{ES}}$ sampled from ES. Indifferentiability requires this behavior to hold even when the adversary can access *both* $\mathbf{C}[\mathrm{RO}_{\mathsf{SS}}]$ *and* $\mathrm{RO}_{\mathsf{SS}}$ without any restriction. Once we have an indifferentiable construction between two function spaces, we can use the indifferentiability "composition theorem" to prove that (almost) any protocol is as secure when it uses $\mathbf{C}[\mathrm{RO}_{\mathsf{SS}}]$ as its random oracle as when it uses $\mathrm{RO}_{\mathsf{ES}}$.[6]

How do we check whether a construction $\mathbf{C}$ is indifferentiable? From the earlier intuition, we set up a security game with two worlds. In one world, often called the "real world", the

[6] As Ristenpart, Shacham, and Shrimpton [187] showed, indifferentiability composition does not cover what they call "multi-stage games," meaning games in which the adversary is split into distinct algorithms with restricted communication. Our multi-stage AKE security game is actually a "single-stage" game in the RSS terminology; indifferentiability composition does apply to our results without issue.

Game $\mathsf{G}^{\mathsf{indiff}}_{\mathbf{C},\mathsf{Sim},\mathsf{SS},\mathsf{ES}}$

```
INIT():
  $b \leftarrow\!\!\$ \{0,1\}$
  $\mathrm{RO}_{\mathsf{SS}} \leftarrow\!\!\$ \mathsf{SS}$
  $\mathrm{RO}_{\mathsf{ES}} \leftarrow\!\!\$ \mathsf{ES}$
  $state \leftarrow\!\!\$ \varepsilon$

FIN($b'$):
  return $b'$

PUB($i, Y$):
  if $b = 0$ then
    $(z, state) \leftarrow \mathsf{Sim}[\textsc{Priv}](i, Y, state)$
    return $z$
  else return $\mathrm{RO}_{\mathsf{SS}}(i, Y)$

PRIV($i, X$):
  if $b = 0$ then return $\mathrm{RO}_{\mathsf{SS}}(i, X)$
  else return $\mathbf{C}[\mathrm{RO}_{\mathsf{ES}}](i, X)$
```

Figure 3.5. The game $\mathsf{G}^{\mathsf{indiff}}_{\mathbf{C},\mathsf{Sim},\mathsf{SS},\mathsf{ES}}$ measuring indifferentiability of a construct $\mathbf{C}$ that transforms function space SS into ES. The game is parameterized by a simulator Sim.

adversary has oracle access to $\mathrm{RO}_{\mathsf{SS}}$ (drawn from SS) and $\mathbf{C}[\mathrm{RO}_{\mathsf{SS}}]$. In the other, the "ideal world", it has oracle access to $\mathrm{RO}_{\mathsf{ES}}$, a random oracle sampled from ES. The adversary's task is then to return a bit indicating which world it is in.

This intuition is obviously incomplete: the adversary can distinguish between worlds just by counting its oracles. We need a second oracle in the ideal world. This second oracle, Pub, must behave indistinguishably from $\mathrm{RO}_{\mathsf{SS}}$, but its responses must also be consistent with the view of $\mathrm{RO}_{\mathsf{ES}}$ (accessed via the first oracle, Priv) as a construction of Pub. The algorithm that does this is called a "simulator". Every construction requires a different simulator Sim, so we make it a parameter of the definition. We can now give pseudocode for the full indifferentiability security game, shown in Figure 3.5.

Definition 11 (Indifferentiability)**.** Let SS and ES be function spaces, and let $\mathbf{C}$ be a construction of ES from SS. Then for any simulator Sim and any adversary $\mathscr{D}$ which makes $\mathsf{q}_{\textsc{Priv}}$ queries to the Priv oracle and $\mathsf{q}_{\textsc{Pub}}$ queries to the Pub oracle, the indifferentiability advantage of $\mathscr{D}$ is

$$\mathbf{Adv}^{\mathsf{indiff}}_{\mathbf{C},\mathsf{Sim},\mathsf{q}_{\textsc{Priv}},\mathsf{q}_{\textsc{Pub}}}(\mathscr{D}) := \Pr[\mathsf{G}^{\mathsf{indiff}}_{\mathbf{C},\mathsf{Sim}}(\mathscr{D}) \Rightarrow 1 | b = 1] - \Pr[\mathsf{G}^{\mathsf{indiff}}_{\mathbf{C},\mathsf{Sim}}(\mathscr{D}) \Rightarrow 1 | b = 0].$$

Indifferentiability is useful because of the following theorem of Maurer et al. [156]. In our presentation, we consider only the authenticated key exchange game, although the theorem applies equally well to any single-stage game [187].

Theorem 7. *Let* KE *be a key exchange protocol using function space* ES*. Let* $\mathbf{C}$ *be an indifferentiable construct of* ES *from* SS *with respect to simulator* Sim*, and let* t' *be the runtime of* Sim

on a single query. We define KE' *to be the following key exchange protocol with function space* SS*:* KE' *runs* KE*, but wherever* KE *would call its random oracle,* KE' *instead computes* $\mathbf{C}$ *using its own random oracle. For any adversary* $\mathscr{A}$ *against the* $\mathsf{KE\text{-}SEC}$ *security of* KE' *with runtime* $t_{\mathscr{A}}$ *and making* q *random oracle queries, there exists an adversary* $\mathscr{B}$ *and a distinguisher* $\mathscr{D}$ *with runtime approximately* $t_{\mathscr{A}} + q \cdot t$ *such that*

$$\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}'}(\mathscr{A}) \leq \mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}}(\mathscr{B}) + \mathbf{Adv}^{\mathsf{indiff}}_{\mathbf{C},\mathsf{Sim}}(\mathscr{D}).$$

Proof: Adversary $\mathscr{B}$ is a wrapper for $\mathscr{A}$ whenever $\mathscr{A}$ makes a query to its random oracle RO, $\mathscr{B}$ responds by running the simulator with its own random oracle. The distinguisher $\mathscr{D}$ simulates the $\mathsf{KE} - \mathit{Sec}$ game of KE for $\mathscr{A}$, with two differences: instead of an RO, it gives $\mathscr{A}$ oracle access to Pub, and where KE would query its own RO, it instead queries Priv. We claim that when $b = 1$ in the indifferentiability game (the real world), $\mathscr{D}$ perfectly simulates the $\mathsf{KE\text{-}SEC}$ game of KE' for $\mathscr{A}$. This works because the Priv oracle computes $\mathbf{C}$ for KE', and the Pub oracle is indeed an RO as $\mathscr{A}$ expects. When $b = 0$, $\mathscr{D}$ perfectly simulates $\mathsf{KE\text{-}SEC}$ of KE for $\mathscr{B}$. The Pub oracle answers all of $\mathscr{A}$'s queries using the simulator, so it properly executes the wrapper code that makes up $\mathscr{B}$. The rest of the simulation is honest, down to the random oracle accessed via Priv. ▌

3.4 Key-Schedule Indifferentiability

In this section we will argue that the key schedule of TLS 1.3 PSK modes, where the underlying cryptographic hash function is modeled as a random oracle (i.e., the left-hand side of Figure 3.1 with the underlying hash function modeled as a random oracle), is *indifferentiable* [156] from a key schedule that uses *independent* random oracles for each step of the key derivation (i.e., the right-hand side of Figure 3.1 with all TKDF_x functions modeled as independent random oracles). We stress that this step not only makes our main security proof in Section 3.6 significantly simpler and cleaner, but also it puts the entire protocol security analysis on a firmer theoretical ground than previous works. For some background on the indifferentiability framework, see Section **??**.

In their proof of tight security, Diemert and Jager [?] previously modeled the TLS 1.3 key schedule as four independent random oracles. Davis and Günther [82] concurrently modeled the functions HKDF.Extract and HKDF.Expand used by the key schedule as two independent random oracles. Neither work provided formal justification for their modeling. Most importantly, both neglected potential dependencies between the use of the hash function in multiple contexts in the key schedule and elsewhere in the protocol. In particular, no construction of HKDF.Extract and HKDF.Expand as independent ROs from one hash function could be indifferentiable, because HKDF.Extract and HKDF.Expand both call HMAC directly on their inputs, with HKDF.Expand only adding a counter byte. Hence, the two functions are inextricably correlated by definition. We do not claim that the analyses of [?, 82] are incorrect or invalid, but merely point out that their modeling of independent random oracles is currently not justified and might not be formally reachable if one only wants to treat the hash function itself as a random oracle. This is undesirable because the gap between an instantiated protocol and its abstraction in the random oracle model can camouflage serious attacks, as Bellare et al. [?] found for the NIST PQC KEMs. Their attacks exploited dependencies between functions that were also modeled as independent random oracles but instantiated with a single hash function.

In contrast, in this section we will show that our modeling of the TLS 1.3 key schedule is indifferentiable from the key schedule when the underlying cryptographic hash function is modeled as a random oracle. To this end, we will require that inputs to the hash function do not appear in multiple contexts. For instance, a protocol transcript might collide with a Diffie–Hellman group element or an internal key (i.e., both might be represented by exactly the same bit string, but in different contexts). For most parameter settings, we can rule out such collisions by exploiting serendipitous formatting, but for one choice of parameters (the PSK-only handshake using SHA384 as hash function), an adversary could conceivably force this type of collision to occur; see Appendix 3.8 for a detailed discussion. While this does not lead to any known attack on the handshake, it precludes our indifferentiability approach for that case.

Insights for the design of cryptographic protocols.

One interesting insight for protocol designers that results from our attempt of closing this gap with a careful indifferentiability-based analysis is that proper domain separation might enable a cleaner and simpler analysis, whereas a lack of domain separation leads to uncertainty in the security analysis. No domain separation means stronger assumptions in the best case, and an insecure protocol in the worst case, due to the potential for overlooked attack vectors in the hash functions. A simple prefix can avoid this with hardly any performance loss.

Indifferentiability of the TLS 1.3 key schedule.

Via the indifferentiability framework, we replace the complex key schedule of TLS 1.3 with 12 independent random oracles: one for each first-class key and MAC tag, and one more for computing transcript hashes. In short, we relate the security of TLS 1.3 as described in the left-hand side of Figure 3.1 to that of the simplified protocol on the right side of Figure 3.1 with the key derivation and MAC functions TKDF_x and modeled as independent random oracles. We prove the following theorem, which formally justifies our abstraction of the key exchange protocol by reducing its security to that of the original key exchange game.

Theorem 8. *Let* $\mathrm{RO}_{\mathsf{H}}\colon \{0,1\}^* \to \{0,1\}^{hl}$ *be a random oracle. Let* KE *be the TLS 1.3 PSK-only or PSK-(EC)DHE handshake protocol described on the left hand side of Figure 3.1 with* $\mathbf{H} := \mathrm{RO}_{\mathsf{H}}$ *and* **MAC**, **Extract**, *and* **Expand** *defined from* **H** *as in Section 3.2. Let* KE' *be the corresponding (PSK-only or PSK-(EC)DHE) handshake protocol on the right hand side of Figure 3.1, with* $\mathbf{H} := \mathrm{RO}_{\mathsf{Th}}$ *and* $\mathsf{TKDF}_x := \mathrm{RO}_x$, *where* $\mathrm{RO}_{\mathsf{Th}}$, RO_{binder}, $\ldots$, $\mathrm{RO}_{\mathsf{RMS}}$ *are random oracles with the appropriate signatures (cf. Section 3.4.1 for the signature details). Then,*

$$\mathbf{Adv}_{\mathsf{KE}}^{\mathsf{KE\text{-}SEC}}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) \leq \mathbf{Adv}_{\mathsf{KE}'}^{\mathsf{KE\text{-}SEC}}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) \\ + \frac{2(12q_{\mathrm{S}} + q_{\mathrm{RO}})^2}{2^{hl}} + \frac{2q_{\mathrm{RO}}^2}{2^{hl}} + \frac{8(q_{\mathrm{RO}} + 36q_{\mathrm{S}})^2}{2^{hl}}.$$

We establish this result via three modular steps in the indifferentiability framework introduced by Maurer, Renner, and Holenstein [156]. More specifically we will leverage a recent generalization proposed by Bellare, Davis, and Günther (BDG) [**?**], which in particular formalizes indifferentiability for constructions of *multiple* random oracles.

3.4.1 Indifferentiability for the TLS 1.3 Key Schedule in Three Steps

We move from the left of Figure 3.1 to the right via three steps. Each step introduces a new variant of the TLS 1.3 protocol with a different set of random oracles by changing how we implement **H**, **MAC**, **Expand**, **Extract**, and eventually the whole key schedule. Then we view the prior implementations of these functions as constructions of new, independent random oracles. We prove security for each intermediate protocol in two parts: first, we bound the indifferentiability advantage against that step's construction; then we apply the indifferentiability composition theorem based on [156] (cf. Section **??**, Theorem 7) to bound the multi-stage key exchange ($\mathsf{KE\text{-}SEC}$) security of the new protocol.

We give a brief description of each step; all details and formal theorem statements and proofs can be found in Sections 3.4.1, 3.4.1, and 3.4.1, respectively.

From one random oracle to two. TLS 1.3 calls its hash function **H**, which we initially model as random oracle RO_{H}, for two purposes: to hash protocol transcripts, and as a component of **MAC**, **Extract**, and **Expand** which are implemented using $\mathsf{HMAC}[\mathbf{H}]$. Our eventual key exchange proof needs to make full use of the random oracle model for the latter category of hashes, but we require only collision resistance for transcript hashes.

Our first intermediate handshake variant, KE_{I}, replaces **H** with two new functions: Th for hashing transcripts, and Ch for use within **MAC**, **Extract**, or **Expand**. While KE uses the same random oracle RO_{H} to implement Th and Ch, the KE_{I} protocol instead uses two independent random oracles $\mathrm{RO}_{\mathsf{Th}}$ and $\mathrm{RO}_{\mathsf{HMAC}}$. To accomplish this without loss in $\mathsf{KE\text{-}SEC}$ security, we exploit some possibly unintentional domain separation in how inputs to these functions are formatted in TLS 1.3 to define a so-called *cloning functor*, following BDG [**?**]. Effectively, we partition the domain $\{0,1\}^*$ of RO_{H} into two sets $\mathsf{Dom}_{\mathsf{Th}}$ and $\mathsf{Dom}_{\mathsf{Ch}}$ such that $\mathsf{Dom}_{\mathsf{Th}}$ contains all valid transcripts and $\mathsf{Dom}_{\mathsf{Ch}}$ contains all possible inputs to **H** from HMAC. We then leverage Theorem 1 of [**?**] that guarantees composition for any scheme that only queries $\mathrm{RO}_{\mathsf{Ch}}$ within the set $\mathsf{Dom}_{\mathsf{Ch}}$ and $\mathrm{RO}_{\mathsf{Th}}$ within the set $\mathsf{Dom}_{\mathsf{Th}}$.

We defer details on the exact domain separation to Appendix 3.8, but highlight that the PSK-only handshake with hash function `SHA384` *fails* to achieve this domain separation and consequently this proof step cannot be applied and leaves a gap for that configuration of TLS 1.3.

From SHA to HMAC. Our second variant protocol, KE_2, rewrites the **MAC** function. Instead of computing $\mathsf{HMAC}[\mathrm{RO}_{\mathsf{Ch}}]$, **MAC** now directly queries a new random oracle $\mathrm{RO}_{\mathsf{HMAC}} \colon \{0,1\}^{hl} \times \{0,1\}^* \to \{0,1\}^{hl}$. Since $\mathrm{RO}_{\mathsf{Ch}}$ was only called by **MAC**, we drop it from the protocol, but we do continue to use $\mathrm{RO}_{\mathsf{Th}}$, i.e., KE_2 uses two random oracles: $\mathrm{RO}_{\mathsf{Th}}$ and $\mathrm{RO}_{\mathsf{HMAC}}$. The security of this replacement follows directly from Theorem 4.3 of Dodis et al. [91], which proves the indifferentiability of HMAC with fixed-length keys.[7]

From two random oracles to 12. Finally, we apply a "big" indifferentiability step which yields 12 independent random oracles and moves us to the right-hand side of Figure 3.1. The 12 ROs include the transcript-hash oracle $\mathrm{RO}_{\mathsf{Th}}$ and 11 oracles that handle each key(-like) output in TLS 1.3's key derivation, named RO_{binder}, $\mathrm{RO}_{\mathsf{ETS}}$, $\mathrm{RO}_{\mathsf{EEMS}}$, RO_{htk_C}, $\mathrm{RO}_{\mathsf{CF}}$, RO_{htk_S}, $\mathrm{RO}_{\mathsf{SF}}$, $\mathrm{RO}_{\mathsf{CATS}}$, $\mathrm{RO}_{\mathsf{SATS}}$, $\mathrm{RO}_{\mathsf{EMS}}$, and $\mathrm{RO}_{\mathsf{RMS}}$. (The signatures for these oracles are given in Appendix 3.4.1.) For this step, we view TKDF as a construction of 11 random oracles from a single underlying oracle ($\mathrm{RO}_{\mathsf{HMAC}}$). We then give our a simulator in pseudocode and prove the indifferentiability of TKDF with respect to this simulator. Our simulator uses look-up tables to efficiently identify intermediate values in the key schedule and consistently program the final keys and MAC tags.

Combining these three steps yields the result in Theorem 8. In the remainder of the paper, we can therefore now work with the right-hand side of Figure 3.1, modeling **H** and the TKDF functions as 12 independent random oracles.

Step 1: Domain-separating the Transcript Hash

In the original TLS 1.3 PSK/PSK-(EC)DHE handshake, the hash function **H** is used in two different ways. It is used directly to compute digests of a *transcript* and it is used as a *component* of **MAC**, **Extract**, and **Expand**. We will argue now that these two uses are entirely distinct, and we can accordingly write two functions Th and Ch in place of the two uses of **H**,

[7] This requires PSKs to be elements of $\{0,1\}^{hl}$, which is true of resumption keys but possibly not for out-of-band PSKs.

and, following BDG [?], go from modeling $\mathbf{H}$ as one random oracle to modeling Th and Ch as two independent random oracles.

We will refer to our two new random oracles as $\mathrm{RO}_{\mathsf{Th}}$ (modeling the *transcript hash* function Th) and $\mathrm{RO}_{\mathsf{Ch}}$ (modeling the *component hash* Ch). Because TLS 1.3 fully specifies the inputs to each hash function call, we can show that in PSK-(EC)DHE mode and in PSK-only mode when $hl = 256$, TLS 1.3 will never call the same string as an input to both Th and Ch. This is due to some fortunate coincidences of formatting in the standard, which we describe in full in Appendix 3.8. We can therefore define two disjoint sets $\mathsf{Dom}_{\mathsf{Th}}$ and $\mathsf{Dom}_{\mathsf{Ch}}$ such that $\mathsf{Dom}_{\mathsf{Th}} \cup \mathsf{Dom}_{\mathsf{Th}} = \{0,1\}^*$ split up H's domain.

If we define the domain of $\mathrm{RO}_{\mathsf{Th}}$ to be $\mathsf{Dom}_{\mathsf{Th}}$ and the domain of $\mathrm{RO}_{\mathsf{Ch}}$ to be $\mathsf{Dom}_{\mathsf{Ch}}$, we could prove indifferentiability using a construction called the *identity (cloning) functor* $\mathbf{I}$ from [?]. The identity functor constructs two or more random oracles $\mathrm{RO}_1, \mathrm{RO}_2, \ldots$ from RO_{H} by forwarding all RO_i queries to RO_{H} unchanged. However, the definitions of sets $\mathsf{Dom}_{\mathsf{Th}}$ and $\mathsf{Dom}_{\mathsf{Ch}}$ are somewhat complex, especially in PSK-only mode. We would instead prefer to define both $\mathrm{RO}_{\mathsf{Th}}$ and $\mathrm{RO}_{\mathsf{Ch}}$ with domains $\{0,1\}^*$. This would greatly simplify our later use of $\mathrm{RO}_{\mathsf{Ch}}$ as a component of HMAC. Unfortunately, when the domains of $\mathrm{RO}_{\mathsf{Th}}$ and $\mathrm{RO}_{\mathsf{Ch}}$ overlap, the identity functor is *not* indifferentiable. We can however still provide the desired result by turning to the read-only indifferentiability framework of Bellare, Davis, and Günther [?].

Read-only indifferentiability (a.k.a. $\mathsf{rd\text{-}indiff}$) is similar to standard indifferentiability [156]. One notable change (and the one we will leverage here) is that it is parameterized by a set $\mathscr{W}$ called the "working domain." The security game places a restriction on the $\textsc{Priv}$ oracle so that it only responds to queries within $\mathscr{W}$. Read-only indifferentiability supports a broader composition thoerem than Theorem 7, which covers security games which call their random oracles only within the working domain. BDG prove [?, Theorem 1], which states that when $\mathscr{W}$ consists of disjoint sets like $\mathsf{Dom}_{\mathsf{Th}}$ and $\mathsf{Dom}_{\mathsf{Ch}}$, the identity functor is read-only indifferentiable even when the full domains of $\mathrm{RO}_{\mathsf{Th}}$ and $\mathrm{RO}_{\mathsf{Ch}}$ are not disjoint. Furthermore, the read-only indifferentiability advantage is upper-bounded by 0, and BDG give a simulator that runs in linear time on the length of its inputs and makes at most one query per execution. When we apply the read-only indifferentiability composition theorem, the adversary's runtime and query bounds will not increase.

We formalize this with a lemma:

Lemma 1. *Let* KE *be the TLS 1.3 key exchange protocol of Theorem 8. Let* $\mathrm{RO}_{\mathsf{Th}}, \mathrm{RO}_{\mathsf{Ch}}\colon \{0,1\}^* \to \{0,1\}^{hl}$ *be two random oracles, and let* KE_1 *be the protocol on the left-hand side of Figure 3.1, where*

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

and **Expand** *and* **Extract** *are as in* KE *(using the new definition of* **MAC***). Let* $\mathsf{Dom}_{\mathsf{Th}}$ *and* $\mathsf{Dom}_{\mathsf{Ch}}$ *be two disjoint sets such that* $\mathsf{KE}_1.\mathsf{Run}$ *only queries* $\mathrm{RO}_{\mathsf{Th}}$*, resp.* $\mathrm{RO}_{\mathsf{Ch}}$ *in* $\mathsf{Dom}_{\mathsf{Th}}$*, resp.* $\mathsf{Dom}_{\mathsf{Ch}}$*, and* $\mathsf{Dom}_{\mathsf{Th}} \cup \mathsf{Dom}_{\mathsf{Ch}} = \{0,1\}^*$*. Furthermore, let* $\mathsf{Dom}_{\mathsf{Th}}$ *have an efficient membership function.*

Let $\mathscr{A}$ *be an adversary against the* $\mathsf{KE\text{-}SEC}$ *security of* KE*, running in time* $t_{\mathscr{A}}$ *and making* q_{RO} *and* q_{S} *queries to its random oracle resp.* SEND *oracle. Then there exists an adversary* $\mathscr{B}$ *against the security of* KE'*, such that*

$$\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}}(\mathscr{A}) \leq \mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_1}(\mathscr{B}).$$

Adversary $\mathscr{B}$*'s runtime is* $\mathscr{O}(t_{\mathscr{A}} + q_{\mathrm{RO}})$*, and it makes the same number of queries to each of its oracles as* $\mathscr{A}$ *in the* $\mathsf{KE\text{-}SEC}$ *game.*

Proof: The function space of KE is $\mathsf{SS} = \mathsf{FUNC}((,\{0,1\})^*, \{0,1\}^{hl})$, and the function space of KE_1 is $\mathsf{ES} = \mathsf{FUNC}((,\{)\mathsf{Th},\mathsf{Ch}\} \times \{0,1\}^*, \{0,1\}^{hl})$. We can construct ES from SS via a construction called the "identity functor" defined by BDG [?]. This construction is parameterized by a set $\mathscr{W} := (\{\mathsf{Th}\} \times \mathsf{Dom}_{\mathsf{Th}}) \cup (\{\mathsf{Ch}\} \times \mathsf{Dom}_{\mathsf{Ch}})$. To answer any query (i,s), the identity functor simply forwards s to its own oracle, regardless of whether i is Th or Ch. Because $\mathscr{W}$ is the union of two disjoint sets with efficient membership functions, the simulator Sim defined by BDG's Theorem 1 has the property that for any distinguisher $\mathscr{D}$,

$$\mathbf{Adv}^{\mathsf{rd\text{-}indiff}}_{\mathbf{I}_{\mathscr{W}},\mathscr{W},\mathsf{Sim}}(\mathscr{D}) = 0.$$

Sim works by using the membership function of $\mathsf{Dom}_{\mathsf{Th}}$ to check which of the two oracles is being simulated; then it forwards the query to the appropriate oracle.

For this (or any) simulator, the composition theorem for read-only indifferentiability grants the existence of adversary $\mathscr{B}$ and a distingisher $\mathscr{D}$ such that

$$\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}}(\mathscr{A}) \leq \mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_1}(\mathscr{B}) + \mathbf{Adv}^{\mathsf{rd\text{-}indiff}}_{\mathbf{I}_{\mathscr{W}},\mathscr{W},\mathsf{Sim}}(\mathscr{D}) \leq \mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_1}(\mathscr{B}).$$

This composition theorem crucially rests on the fact that $\mathsf{KE}_1.\mathsf{Run}$ queries $\mathrm{RO}_{\mathsf{Th}}$ and $\mathrm{RO}_{\mathsf{Ch}}$ only within $\mathscr{W}$. The lemma follows.

We require that $\mathsf{Dom}_{\mathsf{Th}}$ and $\mathsf{Dom}_{\mathsf{Ch}}$ are disjoint sets. We define specific choices of $\mathsf{Dom}_{\mathsf{Th}}$ and $\mathsf{Dom}_{\mathsf{Ch}}$ based on the low-level formatting of TLS 1.3 in Appendix 3.8, and there we give detailed arguments that the sets are disjoint for 3 of 4 standardized settings of the PSK/PSK-(EC)DHE handshake.

In the fourth setting, PSK-only mode with hash function `SHA384`, there are no disjoint choices for $\mathsf{Dom}_{\mathsf{Th}}$ and $\mathsf{Dom}_{\mathsf{Ch}}$ with efficient membership functions. This is due to a lack of careful domain separation of the hash function calls in TLS 1.3. We therefore cannot apply this indifferentiability step for the PSK-only/`SHA384` handshake protocol. Any security proof of this handshake must either rely on stronger, possibly falsifiable abstractions in the random oracle model, or use a model `SHA384` as a single random oracle, with no guarantees of independence. We avoid the latter approach in order to maintain a modular and readable proof.

The second inequality follows from our choice of simulator and Theorem 1 of [**?**], which makes at most one query to its random oracle per execution. Their simulator, as mentioned above, must efficiently determine for every query s whether to query $\mathrm{RO}_{\mathsf{Th}}$ or $\mathrm{RO}_{\mathsf{Ch}}$. This induces the requirement that $\mathsf{Dom}_{\mathsf{Th}} \cup \mathsf{Dom}_{\mathsf{Ch}} = \{0,1\}^*$, so every possible query can be routed appropriately, and the requirement that $\mathsf{Dom}_{\mathsf{Th}}$ has an efficient membership function so that the simulator is itself efficient. $\mathsf{Dom}_{\mathsf{Th}}$ and $\mathsf{Dom}_{\mathsf{Ch}}$ satisfy these requirements thanks to the rules given in Appendix 3.8. ▮

Step 2: Applying the Indifferentiability of HMAC

Our next key exchange protocol, KE_2, replaces the construction $\mathsf{HMAC}[\mathsf{Ch}]$ with a single random oracle $\mathrm{RO}_{\mathsf{HMAC}}$ in the implementation of **MAC** and by extension **Extract** and **Expand**. We rely on the proof of HMAC's indifferentiability by Dodis et al. [91, Theorem 3]. As a prerequisite for this theorem, we need to restrict HMAC to keys of a fixed length less than the block length of the hash function (512 bits for `SHA256` and 1024 bits for `SHA384`). This is consistent with HMAC's usage in TLS 1.3, where the keys are almost always of length $hl \in \{256, 384\}$. The only exception is when pre-shared keys of another length are negotiated out-of-band; we exclude this case.

Lemma 2. *Let* $\mathrm{RO}_{\mathsf{Th}}, \mathrm{RO}_{\mathsf{Ch}} \colon \{0,1\}^* \to \{0,1\}^{hl}$ *and* $\mathrm{RO}_{\mathsf{HMAC}} \colon \{0,1\}^{hl} \times \{0,1\}^* \to \{0,1\}^{hl}$ *be random oracles. Let* KE_1 *be the TLS 1.3 key exchange protocol described in Theorem 1 using random oracles* $\mathrm{RO}_{\mathsf{Th}}$ *and* $\mathrm{RO}_{\mathsf{Ch}}$*. Let* KE_2 *be the key exchange protocol given on the left-hand side of Figure 3.1, where*

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

and **Extract** *and* **Expand** *are defined as Section 3.2. Let* $\mathscr{A}$ *be an adversary against the* KE-SEC *security of* KE_1*, running in time* $t_{\mathscr{A}}$ *and making* q_{RO} *and* q_{S} *queries to its random oracle resp.* Send *oracle. Then there exists an adversary* $\mathscr{B}$ *against the security of* KE_2 *such that*

$$\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_1}(\mathscr{A}) \leq \mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_2}(\mathscr{B}) + \frac{2(12q_{\mathrm{S}} + q_{\mathrm{RO}})^2}{2^{hl}}.$$

Adversary $\mathscr{B}$ *has runtime* $\mathscr{O}(t_{\mathscr{A}} + q_{\mathrm{RO}})$ *and makes the same number of queries to each of its oracles as* $\mathscr{A}$ *in the* KE-SEC *game.*

Proof: KE_1 uses function space ES, defined in the proof of Lemma 1, and KE_2 uses function space $\mathsf{ES}_2 = \mathrm{FUNC}((,()\{\mathsf{Th}\} \times \{0,1\}^*) \cup (\{\mathsf{HMAC}\} \times \{0,1\}^{hl} \times \{0,1\}^*), \{0,1\}^{hl})$. The construction $\mathbf{C}$ of ES_2 from ES simply forwards all queries to $\mathrm{RO}_{\mathsf{Th}}$. It answers $\mathrm{RO}_{\mathsf{HMAC}}$ queries with $\mathsf{HMAC}[\mathrm{RO}_{\mathsf{Ch}}]$.

For any simulator Sim, Theorem 5 grants the existence of a distinguisher $\mathscr{D}$ and an adversary $\mathscr{B}$ such that

$$\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_1}(\mathscr{A}) \leq \mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_2}(\mathscr{B}) + \mathbf{Adv}^{\mathsf{indiff}}_{\mathsf{C},\mathsf{Sim}}(\mathscr{D}).$$

The distinguisher $\mathscr{D}$ makes up to 12 queries to Priv for each Send query made by $\mathscr{A}$, and makes one Pub query for each RO query of $\mathscr{A}$.

We consider the simulator Sim_2 defined by Dodis et al. for [90, Theorem 4.3] (the full version of [91, Theorem 3]). This simulator relies on the requirement that HMAC keys are a fixed length, and shorter than the block length of the underlying hash function. HMAC pads its keys with zero bits up to the block length, so each hash function call made by HMAC contains a segment containing the byte `0x36` for the first of the two calls and `0x5c` for the second. Sim_2 uses this segment to identify whether a particular query is intended to simulate the first or second hash function call. It answers the "first" calls with random strings and logs these responses. Then it programs the "second" calls by using its stored intermediate values to find which $\mathrm{RO}_{\mathsf{HMAC}}$ query should be simulated. We augment the simulator to forward all queries to $\mathrm{RO}_{\mathsf{Th}}$; this does not change its runtime or effectiveness. This simulator works perfectly unless there is a collision among the $2\mathsf{q}_{\text{Priv}} + \mathsf{q}_{\text{Pub}}$ intermediate values, which Dodis et al. bound with a birthday bound. That theorem states that for a distinguisher $\mathscr{D}$ making $12q_{\mathrm{S}}$ queries to Priv and q_{RO} queries to Pub,

$$\mathbf{Adv}^{\mathsf{indiff}}_{\mathsf{C},\mathsf{Sim}}(\mathscr{D}) \leq \frac{2(12q_{\mathrm{S}} + q_{\mathrm{RO}})^2}{2^{hl}}.$$

The lemma follows. ▮

Step 3: Applying Indifferentiability to the TLS Key Schedule

In the last step, we move to the right-hand side of Figure 3.1 and introduce 11 new independent random oracles to model the key schedule. We start by rephrasing the TLS key schedule and message authentication codes as eleven functions TKDF_{binder}, ..., $\mathsf{TKDF}_{\mathsf{RMS}}$ as in Section 3.2. This abstraction does not change any of the operations performed by the key schedule; the TKDF functions simply rename the key derivation steps already performed by KE_2. In our last key exchange protocol KE', we model each TKDF function as a independent random oracle. We name these oracles after the keys or values they derive:

1. $\mathrm{RO}_{binder}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
2. $\mathrm{RO}_{\mathrm{ETS}}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
3. $\mathrm{RO}_{\mathrm{EEMS}}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
4. $\mathrm{RO}_{htk_C}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl+ivl}$
5. $\mathrm{RO}_{fin_C}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
6. $\mathrm{RO}_{htk_S}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl+ivl}$
7. $\mathrm{RO}_{fin_S}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
8. $\mathrm{RO}_{\mathsf{CATS}}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
9. $\mathrm{RO}_{\mathsf{SATS}}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
10. $\mathrm{RO}_{\mathrm{EMS}}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$
11. $\mathrm{RO}_{\mathsf{RMS}}[\mathrm{RO}_{\mathsf{HMAC}}]$: $\{0,1\}^{hl} \times \mathbb{G} \times \{0,1\}^{hl} \to \{0,1\}^{hl}$

The 12th random oracle is $\mathrm{RO}_{\mathsf{Th}}$, used to hash transcripts as in KE_1 and KE_2.

Now we can state Lemma 3.

Lemma 3. *Let KE_2 be the key exchange protocol of Lemma 2, and let KE' be the key exchange protocol of Theorem 8.*

For any adversary $\mathscr{A}$ against the $\mathsf{KE\text{-}SEC}$ security of KE_2, with runtime t and making q_{RO} random oracle queries and q_{S} queries to Send, *there exists adversary $\mathscr{B}$ against the $\mathsf{KE\text{-}SEC}$ security of KE' such that*

$$\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_1}(\mathscr{A}) \leq \mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_2}(\mathscr{B}) + \frac{2\mathsf{q}^2_{\mathrm{PUB}}}{2^{hl}} + \frac{8(\mathsf{q}_{\mathrm{PUB}} + 6\mathsf{q}_{\mathrm{PRIV}})^2}{2^{hl}}.$$

Adversary $\mathscr{B}$ runs in time at most $t + q_{\mathrm{RO}} t_{\mathbb{G}}$, where $t_{\mathbb{G}}$ is the time to perform one group operation in the Diffie–Hellman group $\mathbb{G}$. It makes no more queries to each of the oracles in the $\mathsf{KE\text{-}SEC}$ game than does $\mathscr{A}$.

Proof: We view TKDF as defined in Section 3.2 as a construction of the function space ES'

of KE': the arity-12 function space whose first subspace is $\mathsf{FUNC}((,\{0,1\})^*,\{0,1\}^{hl})$ and whose remaining 11 subspaces are the spaces of all functions with the domains and ranges specified in the above list. This TKDF construction takes an oracle from ES_2, the function space of KS_2.

As in the prior two steps, we consider a particular simulator Sim (cf. Figure ??) and rely on Theorem 5 for the existence of a distinguisher $\mathscr{D}$ and an adversary $\mathscr{B}$ such that

$$\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_2}(\mathscr{A}) \leq \mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}'}(\mathscr{B}) + \mathbf{Adv}^{\mathsf{indiff}}_{\mathsf{TKDF},\mathsf{Sim}}(\mathscr{D}).$$

The distinguisher $\mathscr{D}$ will make no more than 12 queries to Priv for each Send query made by $\mathscr{A}$ and one query to Pub per RO query.

Via a sequence of code-based games, we will show that the indifferentiability advantage of any distinguisher $\mathscr{D}$ making $\mathsf{q}_{\textsc{Priv}}$ queries to the Priv oracle and $\mathsf{q}_{\textsc{Pub}}$ queries to the Pub oracle is

$$\mathbf{Adv}^{\mathsf{indiff}}_{\mathsf{TKDF},\mathsf{SS},\mathsf{ES},\mathsf{Sim}}(\mathscr{D}) \leq \frac{2\mathsf{q}_{\textsc{Pub}}^2}{2^{hl}} + \frac{8(\mathsf{q}_{\textsc{Pub}} + 6\mathsf{q}_{\textsc{Priv}})^2}{2^{hl}}.$$

We give fully specified pseudocode for each of our games.

First, we explain the high-level strategy of our simulator. Our simulator takes two inputs: an index $i \in \{\mathsf{Th},\mathsf{HMAC}\}$ and a string $s \in \{0,1\}^*$. When $i = \mathsf{Th}$, the simulator simulates $\mathrm{RO}_{\mathsf{Th}}(s)$ easily; it simply forwards the query to its own random oracle $\mathrm{RO}_{\mathsf{Th}}$. When $i = \mathsf{HMAC}$, the simulator will parse s into a key $K \in \{0,1\}^{hl}$ and a context string $Y \in \{0,1\}^*$ and simulate $\mathrm{RO}_{\mathsf{HMAC}}(K,Y)$. This simulation should be compatible with a view of the random oracles RO_x as computing $\mathsf{TKDF}_x[\mathrm{RO}_{\mathsf{HMAC}}]$.

Initially, Sim randomly samples the response y to any simulated $\mathrm{RO}_{\mathsf{HMAC}}$ query from $\{0,1\}^{hl}$. Repeated queries are cached in a table M. Next, Sim checks whether the query could be part of an attempt to compute $\mathsf{TKDF}_x[\mathsf{Sim}]$ for some x. If so, it may have to program its response for consistency with RO_x, or it may store its response in a lookup table T to enable future programming.

The only values that need programming are the first-class keys and MAC values. These are all outputs of $\mathbf{Expand}[\mathrm{RO}_{\mathsf{HMAC}}]$. Sim can tell if a particular $\mathrm{RO}_{\mathsf{HMAC}}$ query is made by $\mathbf{Expand}$

by checking its formatting. The inputs Y of all **Expand**'s queries in the key schedule start with 3 bytes of fixed values and a label ℓ between 8 and 18 bytes long that starts with the string `"tls13"`. They end with a 1 byte counter that TLS 1.3 fixes to `0x01`. Sim pattern-matches this label to determine which key is being derived. It has a subroutine $\mathscr{L}$ to translate the few labels which are used in the last derivation step for multiple keys.

Whenever Sim detects the label of an intermediate key derivation query like the **Expand** calls used to compute ES, HS, or MS, it stores the response to this query in table T under the name of the key in question. If $\mathscr{D}$ computes TKDF honestly, these tables will allow the simulator to backtrack through the execution to identify all of the inputs to TKDF. Inputs to $\mathrm{RO}_{\mathsf{HMAC}}$ queries made by $\mathsf{HKDF.Extract}$ do not contain labels, so some tables contain multiple intermediate values. Even without labels, each intermediate value should only appear in one key derivation except in the unlikely event of a collision in $\mathrm{RO}_{\mathsf{HMAC}}$.

The first game in our sequence is G_0 which is the "ideal world" setting of the indifferentiability game. Here, $\textsc{Priv}$ queries are answered using a random function RO drawn from ES, and $\textsc{Pub}$ queries are answered with $\mathsf{Sim}[\mathrm{RO}]$.

In G_1 (cf. Figure ??), we set a bad flag bad_C and abort whenever Sim samples a random answer y that collides with the input or output of any previous simulator query. We track these inputs and outputs in a list L. For each new query, there are at most $2\mathsf{q}_{\textsc{Pub}}$ points to collide with. Since y is sampled uniformly from $\{0,1\}^{hl}$, the probability of such a collision over all queries is at most $\frac{2\mathsf{q}^2_{\textsc{Pub}}}{2^{hl}}$ by a birthday and union bound). Then

$$|\Pr[\mathrm{G}_1] - \Pr[\mathrm{G}_0]| \leq \frac{2\mathsf{q}^2_{\textsc{Pub}}}{2^{hl}}.$$

In G_2 (Figure??), the $\textsc{fin}$ oracle computes $\mathsf{TKDF}[\mathrm{RO}_{\mathsf{HMAC}}]$ on the input to every query to the $\textsc{Priv}$ oracle, using $\textsc{Pub}$ as its hash function. It discards the results of this computation, so this change can affect the outcome of the game only if one of the additional $\textsc{Pub}$ queries sets the bad_C flag. The TKDF function queries its oracle at most 6 times per execution, so there are no more than $6\mathsf{q}_{\textsc{Priv}}$ new queries. There are now a total of $\mathsf{q}_{\textsc{Pub}} + 6\mathsf{q}_{\textsc{Priv}}$ queries to $\textsc{Pub}$, so the

$\mathsf{Sim}(i,s)$

```
$\mathsf{Sim}[\mathrm{RO}](i,s)$:
if $M[s] \neq \bot$
    then return $M[s]$
if $i = \mathsf{Th}$ then return $\mathrm{RO}_{\mathsf{Th}}(K \| Y)$
    // If not, this query should simulate $\mathrm{RO}_{\mathsf{HMAC}}$
$K,\ Y \leftarrow s$
    // Randomly sample a response
$y \leftarrow\$ \{0,1\}^{hl}$
if $Y = 0$
    $T_{\mathrm{PSK}}[y] \leftarrow K$
else if $K = 0$
    $T_{\mathrm{dHS}}[y] \leftarrow Y$
else if $T_{fk_b/fk_C/fk_S}[K] \neq \bot$
    $\mathrm{ES} \leftarrow T_{\mathrm{ES}}[T_{\mathrm{BK/CHTS/SHTS}}[K]]$
    $\mathrm{PSK} \leftarrow T_{\mathrm{PSK}}[\mathrm{ES}]$
    if $\mathrm{PSK} \neq \bot$
        $y \leftarrow \mathrm{RO}_{binder}(\mathrm{PSK}, Y)$
        $\mathrm{HTS} \leftarrow T_{\mathrm{BK/CHTS/SHTS}}[K]$
    $(\ell', \mathrm{HS}, \mathsf{H}_2) \leftarrow T_{\mathrm{HS}/d}[\mathrm{HTS}]$
    $(\mathrm{dES}, \mathrm{DHE}) \leftarrow T_{\mathrm{dES/DHE}}[\mathrm{HS}]$
    $\mathrm{PSK} \leftarrow T_{\mathrm{PSK}}[T_{\mathrm{ES/HS}}[\mathrm{dES}]]$
    if $\mathrm{PSK} \neq \bot$
        $y \leftarrow \mathrm{RO}_{\ell'[1]}(\mathrm{PSK}, \mathrm{DHE}, \mathsf{H}_2, Y)[\mathscr{L}(\ell)]$
else $T_{\mathrm{dES/DHE}}[y] \leftarrow (K, Y)$
if $(Y[0\ldots2] \neq hl)$
    $\vee Y[2] < 8) \vee (Y[2] > 18)$
    $\vee (Y[3\ldots9] \neq$ "tls13")
    $\vee (Y[|Y|-1] \neq 1)$
        // This query does not match $\mathsf{HKDF.Expand}$ formatting.
        $M[s] \leftarrow y$
        return $y$
    // Parse the **Expand** formatting to find the label.
$\mathrm{len}_\ell \leftarrow Y[2]$
$\ell \leftarrow Y[3\ldots(3+\mathrm{len}_\ell)]$
$d \leftarrow Y[(3+\mathrm{len}_\ell)\ldots|Y|]$
...  // continued in next column

$\mathsf{Sim}[\mathrm{RO}](i,s)$  // continued:
if $\ell = \ell_{binder}$ and $d = \mathsf{H}("")$
    $T_{\mathrm{ES}}[y] \leftarrow K$
else if $\ell = \ell_{\mathrm{dES/dHS}}$ and $d = \mathsf{H}("")$
    $T_{\mathrm{ES/HS}}[y] \leftarrow K$
else if $\ell \in \{\ell_{\mathrm{CHTS}}, \ell_{\mathrm{SHTS}}\}$
    $T_{\mathrm{HS}/d}[y] \leftarrow (\mathscr{L}(\ell), K, d)$
else if $\exists k \in \{\mathrm{ETS}, \mathrm{EEMS}\}$ with $\ell = \ell_k$ and $T_{\mathrm{PSK}}[K] \neq \bot$
    $y \leftarrow \mathrm{RO}_k(T_{\mathrm{PSK}}[K], d)$
else if $\exists k \in \{\mathrm{CATS}, \mathrm{SATS}, \mathrm{EMS}, \mathrm{RMS}\}$ with $\ell = \ell_k$
    $(\mathrm{dES}, \mathrm{DHE}) \leftarrow T_{\mathrm{dES/DHE}}[T_{\mathrm{ES/HS}}[T_{\mathrm{dHS}}[K]]]$
    $\mathrm{PSK} \leftarrow T_{\mathrm{PSK}}[T_{\mathrm{ES/HS}}[\mathrm{dES}]]$
    if $\mathrm{PSK} \neq \bot$
        $y \leftarrow \mathrm{RO}_k(\mathrm{PSK}, \mathrm{DHE}, d)$
else if $\ell = \ell_{fk}$ and $d = ""$
    $T_{\mathrm{BK/CHTS/SHTS}}[y] \leftarrow K$
else if $\ell \in \{$"tls13 key", "tls13 iv"$\}$
    and $d = \mathsf{H}("")$
    $(\ell', \mathrm{HS}, \mathsf{H}_2) \leftarrow T_{\mathrm{HS}/d}[K]$
    $(\mathrm{dES}, \mathrm{DHE}) \leftarrow T_{\mathrm{dES/DHE}}[\mathrm{HS}]$
    $\mathrm{PSK} \leftarrow T_{\mathrm{PSK}}[T_{\mathrm{ES/HS}}[\mathrm{dES}]]$
    if $\mathrm{PSK} \neq \bot$
        $y \leftarrow \mathrm{RO}_{\ell'[0]}(\mathrm{PSK}, \mathrm{DHE}, \mathsf{H}_2)[\mathscr{L}(\ell)]$

$M[s] \leftarrow y$
return $y$

Label translator $\mathscr{L}(\ell)$:
if $\ell = \ell_{\mathrm{CHTS}}$
    return $htk_C$, ClientFinished
if $\ell = \ell_{\mathrm{SHTS}}$
    return $htk_S$, ServerFinished
if $\ell =$ "tls13 key"
    return 0
if $\ell =$ "tls13 iv"
    return 1
return $\bot$
```

Figure 3.6. Simulator Sim used in the proof of Lemma 3.

probability that bad_C is set increases by another birthday bound.

$$|\Pr[\mathrm{G}_2] - \Pr[\mathrm{G}_1]| \leq \frac{2(\mathsf{q}_{\textsc{Pub}} + 6\mathsf{q}_{\textsc{Priv}})^2}{2^{hl}}.$$

The next step is the most subtle. In G_3 (Figure ??), we move the new computations of TKDF from the FIN oracle into PRIV. When PRIV is called with index i and input X, it still returns $\mathrm{RO}_i(X)$. First, however, it computes $\mathsf{TKDF}_i[\textsc{Pub}](X)$. It discards the result of this computation, so the behavior of the PRIV oracle does not change in the adversary's view.

However, queries to PRIV now run the simulator Sim. They can update its state and set the global bad_C flag. This has two consequences. First, the changed order of PUB queries may cause bad_C to be set in G_3 when it was not set in G_2, or vice versa. Second, queries to PRIV in G_3 can add entries to the reverse lookup table T. These new entries can be used to satisfy the conditions the simulator uses to check if a full execution of TKDF has been completed. Then the simulator in G_3 may program responses that were not programmed in G_2.

We claim that despite the changed order of the queries, G_3 and G_2 behave identically in the adversary's view except when one of them would set the bad_C flag, assuming that the same random coins are used in both games. Let E denote the event that bad_C is set either when $\mathscr{A}$ plays G_2 or when $\mathscr{A}$ plays G_3. Differences between the two games about when this flag is set are obviously irrelevant unless event E occurs.

The argument that PUB responses are identical in both games except when event E occurs is more subtle. Assume event E does not occur. There must be a first adversarial query to PUB that gives different responses in G_3 and G_2, all oracles behave identically in both games. We name this query Q. Both games sample the same random responses, so query Q has its response programmed by the simulator in at least one of the two games.

The simulator decides whether to program based on the entries of reverse lookup table T, so we consider the differences in this table between our two games. Let T_2 be the table in G_2 at the time when Query Q is made, and let T_3 be the table at the same point in G_4. Entries in the reverse lookup table are indexed by randomly sampled values y, so they cannot be overwritten by later queries unless event E occurs. Furthermore, until query Q is made, every $\textsc{Pub}$ query in G_2 that updates T gives the identical response in G_3, so every entry in T_2 is also an entry in T_3. Therefore any query which is programmed in G_2, up to and including query Q, will be programmed to the same response in G_3. The contrapositive statement says that any response which is randomly sampled in G_3 will be also be randomly sampled in G_2.

It follows that query Q must have a randomly sampled response in G_2 but be programmed in G_3. There must exist a sequence of entries in T_3 that correspond to a full execution of $\mathsf{TKDF}[\textsc{Pub}]$ on some input. We name the queries that created these entries $Q_1,\ldots,Q_i$. In each execution, our simulator either stores an entry in T, or it programs the response y, never both. Therefore queries $Q_1,\ldots Q_i$ have randomly sampled responses. By the definition of TKDF, the output of each query Q_j is contained in the input of the next query Q_{j+1}. The output of Q_i is contained in the input of Q, so we identify query Q with Q_{i+1}.

In G_2, one of the entries in the sequence is not present in T_2. Therefore one of the queries $Q_1,\ldots,Q_i$ is not made before query Q in G_2. This query, Q_j must have been one of the $\textsc{fin}$ queries of G_2 that were moved earlier in G_3. It will therefore be made in $\textsc{fin}$, after all of the other queries, including Q_{j+1}. The randomly sampled output of Q_j will collide with the input of earlier query Q_{j+1}, setting bad_C and causing event E to occur.

The difference in advantage in G_3 and G_2 is therefore bounded by the probability of event E. Both games make $\mathsf{q}_{\textsc{Pub}}+6\mathsf{q}_{\textsc{Priv}}$ queries to $\textsc{Pub}$, each of which sets bad_C is set with probability at most $\frac{2(\mathsf{q}_{\textsc{Pub}}+6\mathsf{q}_{\textsc{Priv}})}{2^{hl}}$. By a union bound,

$$|\Pr[\mathrm{G}_3]-\Pr[\mathrm{G}_2]| \leq \frac{4(\mathsf{q}_{\textsc{Pub}}+6\mathsf{q}_{\textsc{Priv}})^2}{2^{hl}}.$$

Pseudocode for the last three games is given in Figure ??. Now we adjust $\textsc{Priv}$ in G_4 to return the result of $\mathbf{C}[\textsc{Pub}]$ instead of querying RO. Unless bad_C is set, $\mathsf{TKDF}[\textsc{Pub}](r,X)=\mathrm{RO}_r(X)$. The function TKDF makes sequential queries to $\textsc{Pub}$ that are properly formatted, so our Sim will program the last query in the sequence for consistency with the appropriate RO. This programming occurs every time $\mathsf{TKDF}[\textsc{Pub}]$ is called, unless the last query is a repeated query.

In that case, it will be answered using table M instead of RO. However, if the queries in the sequence occur out of order, they will always cause bad_C to be set because the output of a later query will match the input to an earlier query. Then the adversary wins in G_4 with the same likelihood as G_3, unless bad_C is set. If bad_C is set, both games have a win probability of 0 thanks to the check in the FIN oracle, so

$$\Pr[G_4] = \Pr[G_3].$$

Starting with G_5, we stop returning 0 in FIN when bad_C is set. This increases the win probability by at most $\Pr[G_4 \text{ sets } \mathsf{bad}_C] \leq \frac{2(\mathsf{q}_{\text{PUB}}+6\mathsf{q}_{\text{PRIV}})^2}{2^{hl}}$, by the same birthday and union bounds over the $\mathsf{q}_{\text{PUB}} + 6\mathsf{q}_{\text{PRIV}}$ queries to PUB.

$$|\Pr[G_5] - \Pr[G_4]| \leq \frac{2(\mathsf{q}_{\text{PUB}}+6\mathsf{q}_{\text{PRIV}})^2}{2^{hl}}.$$

From G_4 onward, all queries to $\text{RO}_{\mathsf{HMAC}}$ are made by Sim. In G_6, therefore, we can inline the lazily sampled $\text{RO}_{\mathsf{HMAC}}$ oracle as part of the simulator. Repeated queries to Sim are cached, so the random oracle does not need to maintain its own lookup table. Now all responses from PUB are randomly sampled from $\{0,1\}^{hl}$, regardless of the contents of table T. The table and the conditional statements used to maintain it are now redundant bookkeeping, as is the unused bad_C flag after G_5. We eliminate all of this code from G_6 without detection by the adversary. Then

$$\Pr[G_6] = \Pr[G_5].$$

The remaining code of Sim just implements random oracles $\text{RO}_{\mathsf{HMAC}}$ and RO_{Th}. Consequently G_6 is identical to the ideal indifferentiability game for the TKDF construction. Collecting bounds proves the theorem.

■

We have now established that in order to give a (tight) security proof for TLS 1.3 PSK-only and PSK-(EC)DHE, it suffices to prove (tight) security of the protocol on the right-hand side of Figure 3.1.

Game G_0

```
INIT():
  $b \leftarrow 0$
  $\mathrm{RO} \leftarrow\!\!\$\ \mathsf{ES}$
  $state \leftarrow\!\!\$\ \varepsilon$

$\mathsf{Sim}(i, s, state)$:
  if $i = \mathsf{Th}$ then return $\mathrm{RO}_{\mathsf{Th}}, (s)$
  $T, M \leftarrow state$
  if $M[s] \neq \bot$
    then return $M[s]$
  $K, Y \leftarrow s$
  $y \leftarrow \mathsf{Sim}[\mathrm{RO}](K, Y, T)$
  $M[s] \leftarrow y$
  return $y$

$\mathsf{Sim}[\mathrm{RO}](K, Y, T)$:
    // Randomly sample a response
  $y \leftarrow\!\!\$\ \{0,1\}^{hl}$
  if $Y = 0$
    $T_{\mathrm{PSK}}[y] \leftarrow K$
  else if $K = 0$
    $T_{\mathrm{dHS}}[y] \leftarrow Y$
  else if $T_{fk_b/fk_C/fk_S}[K] \neq \bot$
    $\mathrm{ES} \leftarrow T_{\mathrm{ES}}[T_{\mathrm{BK/CHTS/SHTS}}[K]]$
    $\mathrm{PSK} \leftarrow T_{\mathrm{PSK}}[\mathrm{ES}]$
    if $\mathrm{PSK} \neq \bot$
      $y \leftarrow \mathrm{RO}_{binder}(\mathrm{PSK}, Y)$
      $\mathrm{HTS} \leftarrow T_{\mathrm{BK/CHTS/SHTS}}[K]$
    $(\ell', \mathrm{HS}, \mathsf{H}_2) \leftarrow T_{\mathrm{HS}/d}[\mathrm{HTS}]$
    $(\mathrm{dES}, \mathrm{DHE}) \leftarrow T_{\mathrm{dES/DHE}}[\mathrm{HS}]$
    $\mathrm{PSK} \leftarrow T_{\mathrm{PSK}}[T_{\mathrm{ES/HS}}[\mathrm{dES}]]$
    if $\mathrm{PSK} \neq \bot$
      $y \leftarrow \mathrm{RO}_{\ell'[1]}(\mathrm{PSK}, \mathrm{DHE}, \mathsf{H}_2, Y)[\mathscr{L}(\ell)]$
  else $T_{\mathrm{dES/DHE}}[y] \leftarrow (K, Y)$
  if $(Y[0 \dots 2] \neq hl)$
    $\vee Y[2] < 8) \vee (Y[2] > 18)$
    $\vee (Y[3 \dots 9] \neq$ "tls13")
    $\vee (Y[|Y| - 1] \neq 1)$
      // This query does not match HKDF.Expand
  formatting.
      return $y$
    // Parse the Expand formatting to find the
  label.
  $\mathrm{len}_\ell \leftarrow Y[2]$
  $\ell \leftarrow Y[3 \dots (3 + \mathrm{len}_\ell)]$
  $d \leftarrow Y[(3 + \mathrm{len}_\ell) \dots |Y|]$
  ...  // continued in next column

$\mathsf{Sim}[\mathrm{RO}](K, Y, T)$  // ...continued:
  if $\ell = \ell_{binder}$ and $d = \mathsf{H}("")$
    $T_{\mathrm{ES}}[y] \leftarrow K$
  else if $\ell = \ell_{\mathrm{dES/dHS}}$ and $d = \mathsf{H}("")$
    $T_{\mathrm{ES/HS}}[y] \leftarrow K$
  else if $\ell \in \{\ell_{\mathrm{CHTS}}, \ell_{\mathrm{SHTS}}\}$
    $T_{\mathrm{HS}/d}[y] \leftarrow (\mathscr{L}(\ell), K, d)$
  else if $\exists k \in \{\mathrm{ETS}, \mathrm{EEMS}\}$ with $\ell = \ell_k$ and $T_{\mathrm{PSK}}[K] \neq \bot$
    $y \leftarrow \mathrm{RO}_k(T_{\mathrm{PSK}}[K], d)$
  else if $\exists k \in \{\mathrm{CATS}, \mathrm{SATS}, \mathrm{EMS}, \mathrm{RMS}\}$ with $\ell = \ell_k$
    $(\mathrm{dES}, \mathrm{DHE}) \leftarrow T_{\mathrm{dES/DHE}}[T_{\mathrm{ES/HS}}[T_{\mathrm{dHS}}[K])]]$
    $\mathrm{PSK} \leftarrow T_{\mathrm{PSK}}[T_{\mathrm{ES/HS}}[\mathrm{dES}]]$
    if $\mathrm{PSK} \neq \bot$
      $y \leftarrow \mathrm{RO}_k(\mathrm{PSK}, \mathrm{DHE}, d)$
  else if $\ell = \ell_{fk}$ and $d = ""$
    $T_{\mathrm{BK/CHTS/SHTS}}[y] \leftarrow K$
  else if $\ell \in \{$"tls13 key", "tls13 iv"$\}$
    and $d = \mathsf{H}("")$
    $(\ell', \mathrm{HS}, \mathsf{H}_2) \leftarrow T_{\mathrm{HS}/d}[K]$
    $(\mathrm{dES}, \mathrm{DHE}) \leftarrow T_{\mathrm{dES/DHE}}[\mathrm{HS}]$
    $\mathrm{PSK} \leftarrow T_{\mathrm{PSK}}[T_{\mathrm{ES/HS}}[\mathrm{dES}]]$
    if $\mathrm{PSK} \neq \bot$
      $y \leftarrow \mathrm{RO}_{\ell'[0]}(\mathrm{PSK}, \mathrm{DHE}, \mathsf{H}_2)[\mathscr{L}(\ell)]$

  return $y$

PUB$(i, s)$:
  $(z, state) \leftarrow \mathsf{Sim}(i, s, state)$
  return $z$

PRIV$(r, X)$:
  return $\mathrm{RO}_r(X)$

FIN$(b')$:
  return $b'$
```

Figure 3.7. Indiff game instantiated with simulator Sim, also Game G_0 in the proof of Lemma 3.

Games G_1

```
$\mathsf{Sim}(i, s, state)$:
  if $i = \mathsf{Th}$ then return $\mathrm{RO}_{\mathsf{Th}}(s)$
  $T, M, \mathbb{L} \leftarrow state$
  if $M[s] \neq \bot$
      then return $M[s]$
  $K, Y \leftarrow s$
  $y \leftarrow \mathsf{Sim}[\mathrm{RO}](K, Y, T, \mathbb{L})$
  $M[s] \leftarrow y$
  $L \leftarrow L \cup \{y, s\}$
  return $y$
```

```
$\mathsf{Sim}[\mathrm{RO}](K, Y, T, \mathbb{L})$:
  $y \leftarrow\!\!\$\ \{0,1\}^{hl}$
  if $y \in L$ or $\exists t \in L$ such that $y \in t$
      $\mathsf{bad}_C \leftarrow \mathsf{true}$
  ...

$\textsc{fin}(b')$:
  if $\mathsf{bad}_C$ then return 0
  return $b'$
```

Figure 3.8. Game G_1 in the proof of Lemma 3.

Game G_2

```
$\textsc{Priv}(r, X)$:
  $Q \leftarrow Q \bigcup \{(r, X)\}$
  return $\mathrm{RO}_r(X)$

$\textsc{fin}(b')$:
  for $(r, X) \in Q$ do
      $z \leftarrow \mathsf{TKDF}_r[\textsc{Pub}](X)$
  if $\mathsf{bad}_C$ then return 0
  return $b'$
```

Game G_3

```
$\textsc{Priv}(r, X)$:
  $z \leftarrow \mathsf{TKDF}_r[\textsc{Pub}](X)$
  return $\mathrm{RO}_r(X)$

$\textsc{fin}(b')$:
  if $\mathsf{bad}_C$ then return 0
  return $b'$
```

Figure 3.9. Games G_2 and G_3 in the proof of Lemma 3.

3.5 Modularizing Handshake Encryption

Next will argue that using "internal" keys to encrypt handshake messages on the TLS 1.3 record-layer does not impact the security of other keys established by the handshake.

Theorem 10 below formulates our argument in a general way, applicable to any multi-stage key exchange protocol, so that future analyses of similar protocols might take advantage of this modularity as well.

Intuitively, we argue as follows. Let KE_2 be a protocol that provides multiple different stages with different external keys (i.e., none of the keys is used in the protocol, e.g., to encrypt messages), and let KE_1 be the same protocol, except that some keys are "internal" and used, e.g., to encrypt certain protocol messages. We argue that either using "internal" keys in KE_1 does not harm the security of *other* keys of KE_1, or KE_2 cannot be secure in the first place. This will establish that we can prove security of a variant TLS 1.3 *without* handshake encryption (in an accordingly simpler model), and then lift this result to the actual TLS 1.3 protocol *with* handshake encryption and the handshake traffic keys treated as "internal" keys.

```
Games $\boxed{G_4}$, $G_5$

PRIV$(r,X)$:
  $z \leftarrow \mathsf{TKDF}_r[\textsc{Pub}](X)$
  return $z$

FIN$(b')$:
  $\boxed{\text{if } \mathsf{bad}_C \text{ then return } 0}$
  return $b'$
```

```
Game $G_6$

$\mathsf{Sim}[\mathrm{RO}](i,s,T)$:
  $y \leftarrow\!\!\$\ \{0,1\}^{hl}$
  return $y$
```

Figure 3.10. Games G_4, G_5, and G_6 in the proof of Lemma 3.

Theorem 9. *Let* KE_1 *be the TLS 1.3 PSK-only resp. PSK-(EC)DHE mode* with *handshake encryption (i.e., with internal stages* $\mathsf{KE}_1.\mathsf{INT} = \{3,4\}$*) as specified on the right-hand side in Figure 3.1. Let* KE_2 *be the same mode* without *handshake encryption (i.e.,* $\mathsf{KE}_1.\mathsf{INT} = \emptyset$ *and AEAD-encryption/decryption of messages is omitted). Let* $\mathsf{Transform}_{\mathsf{Send}}$ *and* $\mathsf{Transform}_{\mathsf{Recv}}$ *be the AEAD encryption resp. decryption algorithms deployed in TLS 1.3 and* $\mathsf{K}_{\mathsf{Transform}} = \mathsf{KE}_1.\mathsf{INT} = \{3,4\}$*. Then we have*

$$\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_1}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) \leq \mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_2}(t + t_{\mathrm{AEAD}} \cdot q_{\mathrm{S}}, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}} + q_{\mathrm{S}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}),$$

where t_{AEAD} *is the maximum time required to execute AEAD encryption or decryption of TLS 1.3 messages.*

For TLS 1.3 this means that we will not consider any security guarantees provided by the additional encryption of handshake messages. We consider this as reasonable for PSK-mode ciphersuites, because the main purposes of handshake message encryption in TLS 1.3 is to hide the identities of communicating parties, e.g., in digital certificates, cf. [16]. In PSK mode there are no such identities. The *pskid* might be viewed as a string that could identify communicating parties, but it is sent unencrypted in the `ClientHello` message, anyway; the encryption of subsequent handshake messages would not contribute to its protection.

3.5.1 Handshake Encryption as a Modular Transformation

Formally, let $\mathsf{KE}_2 = (\mathsf{KGen}, \mathsf{Activate}, \mathsf{Run})$ be a key exchange protocol with no internal keys. We define another key exchange protocol KE_1 which is parameterized by two functions $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$ and a list $\mathsf{K}_{\mathsf{Transform}} \subseteq \{1, \ldots, \mathsf{STAGES}\}$, where STAGES is the number of stages of KE_2. KE_1 inherits its key generation and activation algorithms from KE_2.

In its KE_1.Run algorithm, described in Figure 3.11, it essentially applies $\mathsf{Transform}_{\mathsf{Recv}}$ to a message before calling KE_2.Run, and then $\mathsf{Transform}_{\mathsf{Send}}$ to the returned message, to transform the protocol messages as they pass over a wire. This transformation may be, for instance, the encryption and decryption of messages of KE_2 using an internal key.

In addition to the messages, both algorithms take as input the list of stages that have been accepted by the current session, its role (initiator or responder) in the protocol, and a list of the keys from all stages in $\mathsf{K}_{\mathsf{Transform}}$. In the security game for KE_1, the stages in $\mathsf{K}_{\mathsf{Transform}}$ will produce internal keys; all other keys remain external.

Although $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$ change the messages as they pass over the wire, the way that the messages are processed after receipt by KE_2.Run must not change. In particular, KE_2.Run, internally run within KE_1.Run, still expects messages of the same format and content; also, KE_1 defines its session and contributive identifiers, as well as all other session-specific information in the same way as KE_2.

Correctness.

Not all choices of $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$ are "good choices". For example, if mauling overwrites critical pieces of the protocol messages, then no honest session would ever accept a key. The resulting key exchange KE_2 would be vacuously "secure" because it would be unusable.

For our perspective to be meaningful, we therefore need a correctness property that guarantees that two honest parties executing KE_1 with no adversarial interference will accept at all stages. Informally, we wish that if two sessions honestly executing KE_2 will accept keys for stage s with probability p, then two sessions honestly executing KE_1 will accept keys for stage s with probability close to p. This property only needs to hold when the protocol messages are relayed honestly, with no changes or delivery failures beyond those caused by the application of $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$.

We do not give a formal definition or proof of correctness for TLS 1.3, but we note that in TLS 1.3, the transformation algorithms are AEAD encryption and decryption. Since decryption failures cannot occur in the standardized AEAD algorithms if messages are honestly relayed (due to their perfect correctness), received messages will always match their corresponding sent message, and correctness of $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$ follows.

Security.

We wish KE_1 to be secure if KE_2 is secure. This should be independent of $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$, i.e., should hold even if $\mathsf{Transform}_{\mathsf{Send}}$ leaks its keys and fully overwrites all protocol messages. The following theorem established this result, using that the keys used for the transformation are internal and $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$ have no access to other privileged information. Therefore, their behavior can be mimicked by a reduction to the security of KE_2 as long as KE_2 has "public session matching" for the stages in $\mathsf{K}_{\mathsf{Transform}}$ of KE_1, i.e., session partnering (or matching) for those stages is decidable from the publicly exchanged messages.[8]

Theorem 10. *Let KE_2 be a key exchange protocol with STAGES stages, $\mathsf{KE}_2.\mathsf{INT}$ being empty, and public session matching. Let $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$ be algorithms as above and $\mathsf{K}_{\mathsf{Transform}} \subseteq \{1,\ldots,\mathsf{STAGES}\}$. Define key exchange KE_1 such that $\mathsf{KE}_1.\mathsf{Run}$ is described in Figure 3.11, $\mathsf{KE}_1.\mathsf{INT} = \mathsf{K}_{\mathsf{Transform}}$, and all other attributes of KE_1 are identical to those of KE_2.*

Let $\mathscr{A}$ be an adversary with running time t against the multi-stage key exchange security of KE_1, making q_{S} queries to the Send *oracle. Then there exists an adversary $\mathscr{B}$ with running time $\approx t + q_{\mathrm{S}} m$, where m is the maximum running time of $\mathsf{Transform}_{\mathsf{Send}}$ and $\mathsf{Transform}_{\mathsf{Recv}}$, such that*

$$\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_1}(\mathscr{A}) \leq \mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_2}(\mathscr{B}).$$

$\mathscr{B}$ makes at most q_{S} queries to RevSessionKey *in addition to queries made by $\mathscr{A}$ and the same number of queries as $\mathscr{A}$ to all other oracles in the* $\mathsf{KE\text{-}SEC}$ *game.*

Proof: Adversary $\mathscr{B}$ runs adversary $\mathscr{A}$ and relays all of its queries to the appropriate oracles in its own $\mathsf{KE\text{-}SEC}$ game, except for Send queries. It maintains the time time of the $\mathsf{KE\text{-}SEC}$ game itself, incrementing it once per query. For each session π_u^i, it maintains a list $keys_u^i$ that is initially empty and a list acc_u^i in which $acc_u^i[stage]$ is initially false for each $stage \in \mathsf{K}_{\mathsf{Transform}}$.

When $\mathscr{A}$ makes a query $\textsc{Send}(u,i,m)$, $\mathscr{B}$ first checks for each $stage \in \mathsf{K}_{\mathsf{Transform}}$ with $acc_u^i[stage] = \mathsf{false}$ whether $\pi_u^i.\mathsf{accepted}[stage] \neq \infty$. For each $stage$ which satisfies this condition, $\mathscr{B}$ checks

[8] The property of "public session matching" has already already come up when considering the composition of (regular or multi-stage) key exchange protocols with subsequent symmetric-key protocols [65, 93, 94, 113].

whether $\pi_u^i.\mathsf{tested}[stage]$ or $\pi_u^i.\mathsf{revealed}[stage]$ is true and if π_u^i has a partnered session (matching $sid[stage]$) which has been tested or revealed. (The latter check for partnering is possible because KE_1 has public session matching.) If any of these conditions is true, then $\mathscr{B}$ knows $\pi_u^i.skey[stage]$. Otherwise, it makes an extra query $\textsc{RevSessionKey}(u,i,stage)$ and adds the response to $keys_u^i$. Then it marks $acc_u^i[stage] \leftarrow \mathsf{true}$ and computes $\tilde{m} \leftarrow \mathsf{Transform}_{\mathsf{Recv}}(keys_u^i, \pi_u^i.role, acc_u^i, m)$. It queries its own $\textsc{Send}$ oracle on the tuple $(u,i,\tilde{m})$ and captures the response $\tilde{m'}$. Then it returns $m' \leftarrow \mathsf{Transform}_{\mathsf{Send}}(keys_u^i, \pi_u^i.role, acc_u^i, \tilde{m}')$ to $\mathscr{A}$.

$\mathscr{B}$ perfectly simulates KE_1 for $\mathscr{A}$, so we wish that if $\mathscr{A}$ wins its simulated game, $\mathscr{B}$ should also win its game. $\mathscr{A}$ can win the $\mathsf{KE\text{-}SEC}$ game in one of three ways: it can violate the Sound predicate, it can violate the $\mathsf{ExplicitAuth}$ predicate, or it can satisfy the Fresh predicate and guess the secret bit b. All of the variables tracked by the $\mathsf{ExplicitAuth}$ and Sound predicates are maintained by the $\mathsf{KE\text{-}SEC}$ game for KE_1, not by $\mathscr{B}$. Therefore $\mathscr{A}$ wins the simulated game by violating Sound or $\mathsf{ExplicitAuth}$ only if Sound or $\mathsf{ExplicitAuth}$ is violated in the $\mathsf{KE\text{-}SEC}$ game for KE_2. In this case, $\mathscr{B}$ also wins.

If $\mathscr{A}$ wins by guessing the secret bit b, the story is more complicated. The bit b is chosen by the $\mathsf{KE\text{-}SEC}$ game, so if $\mathscr{A}$ guesses correctly, then so will $\mathscr{B}$. However, a correct guess only matters if the queries do not violate the Fresh predicate. Even if $\mathscr{A}$ did not violate the Fresh predicate, $\mathscr{B}$ makes up to q_{S} additional $\textsc{RevSessionKey}$ queries. Each of these could cause Fresh to be set to false. We claim that none of these queries violate the Fresh predicate.

The Fresh predicate requires that no session be both tested and revealed. $\mathscr{B}$ only reveals keys that have not already been tested, so the only worry is that $\mathscr{A}$ will test this key later. However, all keys that $\mathscr{B}$ reveals are in $\mathsf{K}_{\mathsf{Transform}}$, which is a subset of $\mathsf{KE}_1.\mathsf{INT}$, meaning they are internal keys. These keys cannot be tested if any session which has accepted it has moved on with the protocol. Since $\mathscr{B}$ only reveals a key when a session has both accepted that key and received the next protocol message, it will have moved on and $\mathscr{A}$ can not make any later $\textsc{Test}$ queries on a key that $\mathscr{B}$ has revealed.

The next condition of Fresh is that a tested session's partner cannot be tested or revealed. $\mathscr{B}$ ensures that such a $\textsc{Test}$ query does not occur before the $\textsc{RevSessionKey}$ query. Again, the

Test query cannot happen after the RevSessionKey query because the session whose key was revealed has moved on with the protocol. Since all the revealed keys are internal in the simulated game, $\mathscr{A}$ cannot test them after this point.

The remaining three conditions of the Fresh predicate establish different levels of forward secrecy. They check for the existence of a contributive partner. We want to exclude the situation that a contributive partner exists in $\mathscr{A}$'s simulated game, but not in $\mathscr{B}$'s game. However, contributive identifiers are defined identically in KE_1 and KE_2. Therefore if two sessions π_u^i and π_v^j have matching contributive identifiers in the simulated game for KE_2, they will also have matching identfiers in the game for KE_1.

It is therefore not possible for $\mathscr{A}$ to win its simulated $\mathsf{KE\text{-}SEC}$ game unless $\mathscr{B}$ also wins its $\mathsf{KE\text{-}SEC}$ game, and the theorem follows. ▮

3.6 Tight Security of the TLS 1.3 PSK Modes

In this section, we apply the insights gained in Sections 3.4 and 3.5 to obtain tight security bounds for both the PSK-only and the PSK-(EC)DHE mode of TLS 1.3. To that end, we first present the protocol-specific properties of the TLS 1.3 PSK-only and PSK-(EC)DHE modes such that they can be viewed as multi-stage key exchange (MSKE) protocols as defined in Section 3.3. Then, we prove tight security bounds in the MSKE model in Theorem 11 for the TLS 1.3 PSK-(EC)DHE mode and in Theorem 12 for the TLS 1.3 PSK-only mode.

3.6.1 TLS 1.3 PSK-only/PSK-(EC)DHE as a MSKE Protocol

We begin by capturing the TLS 1.3 PSK-only and PSK-(EC)DHE modes, specified in Figure 3.1, formally as MSKE protocols. To this end, we must explicitly define the variables discussed in Section 3.3. In particular, we have to define the stages themselves, which stages are internal and which replayable, the session and contributive identifiers, when stages receive explicit 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, CATS, SATS, 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 stage$s$ 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:[9]

$$
\begin{aligned}
sid[1] &= \left(\text{“ETS”}, (\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK})\right),\\
sid[2] &= \left(\text{“EEMS”}, (\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK})\right),\\
sid[3] &= \left(\text{“}htk_C\text{”}, (\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK}, \mathtt{SH}, \mathtt{SKS}^\dagger, \mathtt{SPSK})\right),\\
sid[4] &= \left(\text{“}htk_S\text{”}, (\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK}, \mathtt{SH}, \mathtt{SKS}^\dagger, \mathtt{SPSK})\right),\\
sid[5] &= \left(\text{“CATS”}, (\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK}, \mathtt{SH}, \mathtt{SKS}^\dagger, \mathtt{SPSK}, \mathtt{EE}, \mathtt{SF})\right),\\
sid[6] &= \left(\text{“SATS”}, (\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK}, \mathtt{SH}, \mathtt{SKS}^\dagger, \mathtt{SPSK}, \mathtt{EE}, \mathtt{SF})\right),\\
sid[7] &= \left(\text{“EMS”}, (\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK}, \mathtt{SH}, \mathtt{SKS}^\dagger, \mathtt{SPSK}, \mathtt{EE}, \mathtt{SF})\right), \text{ and}\\
sid[8] &= \left(\text{“RMS”}, (\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK}, \mathtt{SH}, \mathtt{SKS}^\dagger, \mathtt{SPSK}, \mathtt{EE}, \mathtt{SF}, \mathtt{CF})\right).
\end{aligned}
$$

To make sure that a server that received `ClientHello`, `ClientKeyShare`†, and `ClientPreSharedKey` 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. receving) the messages $(\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK})$ set

$$
\begin{aligned}
cid_{\mathsf{responder}}[3] &= \left(\text{“}htk_C\text{”}, (\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK})\right) \text{ and}\\
cid_{\mathsf{responder}}[4] &= \left(\text{“}htk_S\text{”}, (\mathtt{CH}, \mathtt{CKS}^\dagger, \mathtt{CPSK})\right).
\end{aligned}
$$

[9] Components marked with † are only part of the TLS 1.3 PSK-(EC)DHE handshake.

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).

Explicit authentication.

For initiator sessions, all stages achieve explicit authentication when the `ServerFinished` message is verified successfully. This happens right before stage 5 (i.e., CATS) 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 $s \in \{1, \ldots, 8\}$.

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 $s \in \{1, \ldots, 8\}$.

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 $s \in \{3, \ldots, 8\}$ 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 $s \in \{3, \ldots, 8\}$. 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 $s \in \{1,2\}$.

3.6.2 Tight Security Analysis of TLS 1.3 PSK-(EC)DHE

We now come to the tight MSKE security result for the TLS 1.3 PSK-(EC)DHE handshake.

Theorem 11. *Let* $\texttt{TLS1.3-PSK-(EC)DHE}$ *be the TLS 1.3 PSK-(EC)DHE handshake protocol (with optional 0-RTT) as specified on the right-hand side in Figure 3.1 without handshake encryption. Let* $\mathbb{G}$ *be the Diffie–Hellman group of order* p*. Let* nl *be the length in bits of the nonce, let* hl *be the output length in bits of* $\mathbf{H}$*, and let the pre-shared key space be* $\mathsf{KE.PSKS} = \{0,1\}^{hl}$*. We model the functions* $\mathbf{H}$ *and* TKDF_x *for each* $x \in \{binder, \ldots, \mathrm{RMS}\}$ *as* 12 *independent random oracles* $\mathrm{RO}_{\mathsf{Th}}, \mathrm{RO}_{binder}, \ldots, \mathrm{RO}_{\mathsf{RMS}}$*. Then,*

$$\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\texttt{TLS1.3-PSK-(EC)DHE}}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) \leq \frac{2q_{\mathrm{S}}^2}{2^{nl} \cdot p}$$
$$+ \frac{(q_{\mathrm{RO}} + q_{\mathrm{S}})^2 + q_{\mathrm{NS}}^2 + (q_{\mathrm{RO}} + 6q_{\mathrm{S}})^2 + q_{\mathrm{RO}} \cdot q_{\mathrm{NS}} + q_{\mathrm{S}}}{2^{hl}} + \frac{4(t + 4\log(p) \cdot q_{\mathrm{RO}})^2}{p}.$$

Remark 1. Our MSKE model from Section 3.3 assumes pre-shared keys to be uniformly random sampled from $\mathsf{KE.PSKS}$, where here $\mathsf{KE.PSKS} = \{0,1\}^{hl}$. This matches how pre-shared keys are derived for session resumption, as well as our analysis of domain separation, which assumes pre-shared keys to be of length hl.

Remark 2. Our bound is easily adapted to any distribution on $\{0,1\}^{hl}$ in order to accommodate out-of-band pre-shared keys that satisfy the length requirement but do not have full entropy. Expectedly, 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.

Proof: To prove our bound, we make an incremental series of changes to the key exchange security game $G_{\texttt{TLS1.3-PSK-(EC)DHE},\mathscr{A}}$. We divide the proof into three phases reflecting the three ways of the adversary to win the security game.

1. We establish that the adversary cannot violate the predicate Sound.

2. We establish the same for the predicate $\mathsf{ExplicitAuth}$.

3. Finally, we ensure that all TEST queries return uniformly random keys independent of the challenge bit b if predicate Fresh is not violated.

We can then conclude that the adversary cannot do better than random guessing to win the game, i.e., its advantage is 0.

Game 0 (Initial game). The initial game $G_0^{\mathscr{A}}$ is the key exchange security game $G_{\mathsf{TLS1.3\text{-}PSK\text{-}(EC)DHE},\mathscr{A}}$ played for the TLS 1.3 PSK-(EC)DHE handshake (with optional 0-RTT) as specified in Figure 3.1 (right), but without handshake encryption. Note that the functions $\mathbf{H}$ and TKDF_x for $x \in \{binder, \ldots, \mathrm{RMS}\}$ are modeled as 12 independent random oracles $\mathrm{RO}_{\mathsf{Th}}, \mathrm{RO}_{binder}, \ldots, \mathrm{RO}_{\mathsf{RMS}}$. We implement random oracle RO_x by a look-up table ROList_x assigning inputs to outputs. We assume that every look-up table implementing a random oracle is stored in a data structure that enables constant time access when indexed either by random oracle inputs or by random oracle outputs, using two hash tables, for instance. By definition, we have

$$\Pr[G_0^{\mathscr{A}} \Rightarrow 1] = \mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{TLS1.3\text{-}PSK\text{-}(EC)DHE}}(\mathscr{A}).$$

Phase 1: Ensuring Predicate Sound cannot be violated

Game 1 (Exclude collisions of nonces and group elements). In $G_1^{\mathscr{A}}$, we eliminate collisions among nonces and group elements computed by honest sessions via two new flags:

- bad_C is set when two honest sessions choose the same nonce and group element, and
- bad_O is set when an honest responder samples some nonce and group element that have already been received by another session. We view this nonce and group element as having been chosen by an adversarial session.

If either bad_C or bad_O is set, the game returns 0 from FIN.

By the well-known identical-until-bad-lemma [41, Lemma 2], we get

$$\begin{aligned}\Pr[G_0^{\mathscr{A}} \Rightarrow 1] \le \Pr[G_1^{\mathscr{A}} \Rightarrow 1] + \Pr[G_1^{\mathscr{A}} \text{ sets } \mathsf{bad}_C] \\ + \Pr[G_1^{\mathscr{A}} \text{ sets } \mathsf{bad}_O]. \qquad (3.1)\end{aligned}$$

Let us separately analyze the probabilities that $G_1^{\mathscr{A}}$ sets the flags bad_C and bad_O. Each Send query causes at most one session to uniformly and independently sample a nonce $r \xleftarrow{\$} \{0,1\}^{nl}$

and a group element $g \xleftarrow{\$} \mathbb{G}$. If the bad_C flag is set, we have that there exists some SEND query that creates a session π_u^i using $\mathsf{Activate}$. This new session samples nonce and group element(r, g) which were previously sampled by another session $\pi_{u'}^{i'}$. That is, the probability for bad_C to be set is the probability of a collision among the (up to) q_S pairs of uniformly and independently sampled nonces and group elements; we can use the birthday bound to bound the probability of setting bad_C by

$$\Pr[\mathrm{G}_1^{\mathscr{A}} \text{ sets } \mathsf{bad}_C] \leq \frac{q_\mathrm{S}^2}{2^{nl} \cdot p}. \tag{3.2}$$

where q_S is the number of SEND queries.

Next, if the game sets bad_O, we have that there is a SEND query which creates a new session π_v^j. This session samples a nonce $r_S \xleftarrow{\$} \{0,1\}^{nl}$ and a group element $Y \xleftarrow{\$} \mathbb{G}$, which were already received by another session π_u^i. There are at most q_S sessions, so there are no more than q_S received pairs which which (r_S, Y) can collide. Since π_v^j samples its nonce and group element uniformly and independently at random from $\{0,1\}^{nl} \times \mathbb{G}$, we get by the union bound that the probability that π_v^j samples one of the already received pairs is bounded from above by $q_\mathrm{S}/(2^{nl} \cdot p)$. Overall, we again get by the union bound that there is such a collision for any π_v^j with probability

$$\Pr[\mathrm{G}_1^{\mathscr{A}} \text{ sets } \mathsf{bad}_O] \leq q_\mathrm{S} \cdot \frac{q_\mathrm{S}}{2^{nl} \cdot p} = \frac{q_\mathrm{S}^2}{2^{nl} \cdot p}. \tag{3.3}$$

Combining Equations (3.1)–(3.3), we get

$$\Pr[\mathrm{G}_0^{\mathscr{A}^*} \Rightarrow 1] \leq \Pr[\mathrm{G}_1^{\mathscr{A}^*} \Rightarrow 1] + \frac{2q_\mathrm{S}^2}{2^{nl} \cdot p}. \tag{3.4}$$

GAME 2 (Exclude binder collisions). In game $\mathrm{G}_2^{\mathscr{A}}$, we let the adversary lose 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 return 0 from FIN.

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 $b = \mathrm{RO}_{binder}(\mathsf{PSK}, ctxt)$, we log $\mathsf{CollList}_{binder}[b] \leftarrow (\mathsf{PSK}, ctxt)$. 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 $s' = (\mathsf{PSK}, ctxt)$ with $\mathrm{RO}_{binder}(psk, ctxt) = b$ queried before and we have found a collision if $s \neq s'$. In this case we set bad_{binder}.

Again by the identical-until-bad-lemma,

$$\Pr[\mathrm{G}_1^{\mathscr{A}} \Rightarrow 1] \leq \Pr[\mathrm{G}_2^{\mathscr{A}} \Rightarrow 1] + \Pr[\mathrm{G}_2^{\mathscr{A}} \text{ sets } \mathsf{bad}_{binder}].$$

To bound the probability that the game sets flag bad_{binder}, we construct a reduction $\mathscr{B}_1$ to the collision-resistance of RO_{binder}. The reduction $\mathscr{B}_1$ simulates Game 2 for adversary $\mathscr{A}$. It implements all oracles itself except for RO_{binder}. $\mathscr{B}_1$ will need to query its own oracle RO_{binder} at most once per RO query and once per SEND query, so it makes $q_{\mathrm{RO}} + q_{\mathrm{S}}$ queries in total. If the flag bad_{binder} would be set in Game 2, which can be checked efficiently using $\mathsf{CollList}_{binder}$ as described before, then the reduction has found a collision (s, s') with $s \neq s'$ such that $\mathrm{RO}_{binder}(s) = \mathrm{RO}_{binder}(s')$. Reduction $\mathscr{B}_1$ then outputs (s, s') and wins the collision-resistance game.

Therefore, we have that

$$\Pr[\mathrm{G}_1^{\mathscr{A}} \Rightarrow 1] \leq \Pr[\mathrm{G}_2^{\mathscr{A}} \Rightarrow 1] + \mathbf{Adv}^{\mathsf{CR}}_{\mathrm{RO}_{binder}}(q_{\mathrm{RO}} + q_{\mathrm{S}}). \tag{3.5}$$

GAME 3 (Exclude collisions of pre-shared keys). In game $\mathrm{G}_3^{\mathscr{A}}$, we set a flag bad_{PC} and return 0 from FIN 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 $\mathsf{pskeys}[(u, v, pskid)] = \mathsf{pskeys}[(u', v', pskid')]$. By the identical-until-bad-lemma,

$$\Pr[\mathrm{G}_2^{\mathscr{A}} \Rightarrow 1] \leq \Pr[\mathrm{G}_3^{\mathscr{A}} \Rightarrow 1] + \Pr[\mathrm{G}_3^{\mathscr{A}} \text{ sets } \mathsf{bad}_{PC}].$$

Since the pre-shared keys are uniformly distributed[10] on $\{0,1\}^{hl}$, by the birthday bound

$$\Pr[\mathrm{G}_3^{\mathscr{A}} \text{ sets } \mathsf{bad}_{PC}] \le \frac{q_{\mathrm{NS}}^2}{2^{hl}}.$$

Conclusion of Phase 1.

At this point, we argue that in Game 3 and any subsequent games, adversary $\mathscr{A}$ cannot violate the Sound predicate without also causing FIN to return 0. If any Sound check fails, one of the checks we have added to the FIN oracle will also fail. According to the definition of the MSKE game, there are six events that cause the predicate Sound to be violated (see Figure 3.4). 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.

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 $s \ge 3$. Recall that session identifiers sid for all stages $s \ge 3$ 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 3.6.1). Every session's sid therefore contains its own randomly sampled nonce-group element pair. For three sessions to accept the same $sid[s]$ for $s \ge 3$, 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 FIN to return 0.

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 $s \ge 3$ as defined by TLS 1.3 (see Section 3.6.1) 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.

[10] As mentioned in Remark 2, this term has to be adapted for a different distribution on $\{0,1\}^{hl}$, i.e., for any distribution $\mathscr{D}$ on $\{0,1\}^{hl}$, the denominator would change to 2^{α}, where α is the min-entropy of $\mathscr{D}$.

3. *There are two sessions with the same session identifier in some stage that do not share the same contributive identifier in that stage.*

   Once a session holds both a contributive identifier and a session identifier for the same stage, both are equal by our definition (see Section 3.6.1) 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 that FIN returns 0 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 FIN returns 0 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 DHE must include both Diffie–Hellman shares X and Y (contained in `CKS` and `SKS`). These shares uniquely determine DHE. 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 $\mathsf{ExplicitAuth}$ cannot be violated

GAME 4 (Exclude transcript hash collisions). In $G_4^{\mathscr{A}}$, we let the adversary lose if two distinct queries to $\mathrm{RO}_{\mathsf{Th}}$ lead to colliding outputs. This ensures that each transcript has a unique hash. When such a collision occurs, we set a new flag $\mathsf{bad}_{\mathsf{H}}$ and let the game return 0 from FIN.

As in Game 2, we introduce a table $\mathsf{CollList}_{\mathsf{Th}}$ to random oracle $\mathrm{RO}_{\mathsf{Th}}$. Whenever it computes a hash $d = \mathrm{RO}_{\mathsf{Th}}(s)$ for some string s, we log $\mathsf{CollList}_{\mathsf{Th}}[d] \leftarrow s$. This table then is used to set $\mathsf{bad}_{\mathsf{Th}}$ as in Game 2.

Analogously to Game 2, we can construct a reduction $\mathscr{B}_2$ to the collision-resistance of $\mathrm{RO}_{\mathsf{Th}}$. As it simulates Game 4, the adversary $\mathscr{B}_2$ will need to make one query to its $\mathrm{RO}_{\mathsf{Th}}$ oracle for each $\mathrm{RO}_{\mathsf{Th}}$ query of $\mathscr{A}$ and up to 6 $\mathrm{RO}_{\mathsf{Th}}$ queries for the up to 6 *distinct* transcript hash values computed in a protocol step per SEND query of $\mathscr{A}$; in total $q_{\mathrm{RO}} + 6q_{\mathrm{S}}$ queries.

Therefore, we have that

$$\Pr[G_4^{\mathscr{A}} \text{sets } \mathsf{bad}_{\mathsf{H}}] \leq \mathbf{Adv}^{\mathsf{CR}}_{\mathrm{RO}_{\mathsf{Th}}}(q_{\mathrm{RO}} + 6q_{\mathrm{S}})$$

and it follows that

$$\Pr[G_3^{\mathscr{A}} \Rightarrow 1] \leq \Pr[G_4^{\mathscr{A}} \Rightarrow 1] + \mathbf{Adv}^{\mathsf{CR}}_{\mathrm{RO}_{\mathsf{Th}}}(q_{\mathrm{RO}} + 6q_{\mathrm{S}}).$$

GAME 5 (Abort if adversary guess a uncorrupted PSK). In $G_5^{\mathscr{A}}$, we make the adversary lose when it 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 $\mathsf{PSKList}$ that is maintained by the NEWSECRET oracle. When $\text{NEWSECRET}(u, v, pskid)$ samples a fresh pre-shared key PSK, we log the tuple under index PSK as $\mathsf{PSKList}[\mathsf{PSK}] \leftarrow (u, v, pskid)$. Note that the pre-shared keys might repeat, so we may have multiple entries in $\mathsf{PSKList}$ 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 $\mathsf{T}[\mathsf{PSK}] \leftarrow \mathsf{time}$, where time is the counter maintained by the key exchange experiment, unless $\mathsf{T}[\mathsf{PSK}]$ already exists.

The actual check whether the adversary queries any random oracle with a PSK before it was corrupted is performed by the FIN oracle. We set a flag $\mathsf{bad}_{\mathsf{PSK}}$ if $\mathsf{T}(\mathsf{PSK}) \leq \mathsf{revpsk}_{(u,v,pskid)}$ for any $\mathsf{PSK} \in \mathsf{T}$ and $(u,v,pskid) \in P[\mathsf{PSK}]$. If the $\mathsf{bad}_{\mathsf{PSK}}$ flag was set during this process, the FIN oracle returns 0.

Next, let us analyze the probability that the game is lost due to flag $\mathsf{bad}_{\mathsf{PSK}}$ being set. Each random oracle query could hit one out of q_{NS} many pre-shared keys. Before a given pre-shared key is corrupted or queried to a random oracle, the adversary knows nothing about its value. Since we assume that pre-shared keys are sampled uniformly at random from $\{0,1\}^{hl}$, the probability to hit a specific one is at most 2^{-hl}.[11] 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 $q_{\mathrm{NS}} \cdot 2^{-hl}$. Thus, the probability that $\mathsf{bad}_{\mathsf{PSK}}$ is set in response to any of the q_{RO} many random oracle queries overall is limited by $q_{\mathrm{RO}} \cdot q_{\mathrm{NS}} \cdot 2^{-hl}$. This follows again by applying the union bound.

Hence, we get by the identical-until-bad lemma,

$$\begin{aligned}\Pr[\mathrm{G}_4^{\mathscr{A}} \Rightarrow 1] &\leq \Pr[\mathrm{G}_5^{\mathscr{A}} \Rightarrow 1] + \Pr[\mathrm{G}_5^{\mathscr{A}} \text{ sets } \mathsf{bad}_{\mathsf{PSK}}] \\ &\leq \Pr[\mathrm{G}_5^{\mathscr{A}} \Rightarrow 1] + \frac{q_{\mathrm{RO}} \cdot q_{\mathrm{NS}}}{2^{hl}}.\end{aligned}$$

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

[11] Note that at this point, we use that the pre-shared key distribution is uniform. As already mentioned before, for any distribution $\mathscr{D}$ on $\{0,1\}^{hl}$, the probability would be $2^{-\alpha}$, where α is the min-entropy of $\mathscr{D}$.

security proof techniques of Cohn-Gordon et al. [73].

Game 6 (Log session keys and MAC tags). 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.

Honest sessions π_u^i in the initiator role 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.,

$$\mathsf{SKEYS}[\pi_u^i.sid[s]] \leftarrow \pi_u^i.skey[s]$$

for $s \in \{1,2,8\}$. Honest responder sessions π_v^j will derive the keys htk_S, htk_C, CATS, SATS, and EMS before their partners. These sessions also log their keys in S under the appropriate session identifier:

$$\mathsf{SKEYS}[\pi_v^j.sid[s]] \leftarrow \pi_v^j.skey[s]$$

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 $(\mathsf{PSK}, \mathsf{DHE}, d_1, d_2)$ to RO_{fin_S}, we log

$$\mathsf{TAGS}[fin_S, \mathsf{PSK}, \mathsf{DHE}, d_1, d_2] \leftarrow \mathrm{RO}_{fin_S}(\mathsf{PSK}, \mathsf{DHE}, d_1, d_2).$$

Since Game 6 only introduces book-keeping steps, we have that

$$\Pr[G_5^{\mathscr{A}} \Rightarrow 1] = \Pr[G_6^{\mathscr{A}} \Rightarrow 1].$$

GAME 7 (Copy session keys and MAC tags from partnered session). In this game, 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 $(\mathtt{CH}, \mathtt{CKS}, \mathtt{CPSK})$, 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 $\mathtt{CF}$, 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 $(\mathtt{SH}, \mathtt{SKS}, \mathtt{SPSK})$, sets its session identifiers for stages 3–7, which identify the keys htk_S, htk_C, CATS, SATS 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 (Property 6) 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

$$\Pr[\mathrm{G}_6^{\mathscr{A}} \Rightarrow 1] = \Pr[\mathrm{G}_7^{\mathscr{A}} \Rightarrow 1].$$

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 (Postpone PSK sampling until after corruption). In this game, we postpone the sampling of pre-shared keys from the NewSecret oracle to the RevLongTermKey oracle (if the pre-shared key gets corrupted) or the fin oracle (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 instead use the tuple $(u, v, pskid)$ 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 $(\star, \mathsf{DHE}, ctxt)$ (where DHE might be $\perp$) 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

$$\mathsf{PrgList}_x[(u, v, pskid), \mathsf{DHE}, ctxt] \leftarrow k$$

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 $\mathsf{PSK} \xleftarrow{\$} \mathsf{KE.PSKS}$ and update pskeys, i.e., set $\mathsf{pskeys}[(u, v, pskid)] \leftarrow \mathsf{PSK}$. Next, we need to reprogram the random oracles using the lists R_x to ensure consistency. Thus, for all x we update the random oracle tables ROList_x for RO_x using $\mathsf{PrgList}_x$. For every entry $\mathsf{PrgList}_x[((u, v, pskid), \mathrm{DHE}, ctxt)] = k$, we set

$$\mathsf{ROList}_x[\mathsf{PSK}, \mathrm{DHE}, ctxt] \leftarrow k$$

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 $\mathsf{PSKList}[\mathsf{PSK}]$.

To be able to still set $\mathsf{bad}_{\mathrm{PSK}}$, we also make sure that in the FIN procedure 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 $(u, v, pskid) \in P[\star]$, setting $\mathsf{pskeys}[(u, v, pskid)] \xleftarrow{\$} \mathsf{KE.PSKS}$, and update the reverse look-up table $\mathsf{PSKList}$ accordingly. As a result, also uncorrupted sessions now have a pre-shared key defined and we can check the condition for $\mathsf{bad}_{\mathrm{PSK}}$ 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 $\mathsf{bad}_{\mathrm{PSK}}$ 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 change to the FIN procedure 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,

$$\Pr[G_7^{\mathscr{A}} \Rightarrow 1] = \Pr[G_8^{\mathscr{A}} \Rightarrow 1].$$

GAME 9 (Exclude that honest sessions accept without a partner). In this game, we set a flag $\mathsf{bad}_{\mathrm{MAC}}$ and return 0 from FIN 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 $\pi_u^i.\mathsf{t}_{\mathsf{acc}}[s] < \mathsf{revpsk}_{(u,v,\pi_u^i.pskid)}$ with $v = \pi_u^i.\textit{peerid}$ 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 $\Pr[G_9^{\mathscr{A}}$ sets $\mathsf{bad}_{\mathrm{MAC}}]$. 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 and the received `ServerFinished` message will match this tag with probability no more than 2^{-hl}.

Thus the probability that π_u^i triggers the flag $\mathsf{bad}_{\mathrm{MAC}}$ is bounded by 2^{-hl}. A union bound over all sessions gives

$$\Pr[\mathrm{G}_9^{\mathscr{A}} \text{ sets } \mathsf{bad}_{\mathrm{MAC}}] \leq \frac{q_{\mathrm{S}}}{2^{hl}}.$$

Overall, we get by the identical-until-bad-lemma

$$\begin{aligned}\Pr[\mathrm{G}_8^{\mathscr{A}} \Rightarrow 1] &\leq \Pr[\mathrm{G}_9^{\mathscr{A}} \Rightarrow 1] + \Pr[\mathrm{G}_9^{\mathscr{A}} \text{ sets } \mathsf{bad}_{\mathrm{MAC}}] \\ &\leq \Pr[\mathrm{G}_9^{\mathscr{A}} \Rightarrow 1] + \frac{q_{\mathrm{S}}}{2^{hl}}.\end{aligned}$$

Conclusion of Phase 2.

At this point, we argue that in Game 9 and any subsequent games, adversary $\mathscr{A}$ cannot violate the $\mathsf{ExplicitAuth}$ predicate without also causing FIN to return 0. To this end, we argue that $\mathsf{ExplicitAuth} = \mathsf{true}$ holds with certainty from Game 9 on.

The predicate $\mathsf{ExplicitAuth}$ 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 $s' \geq s$ 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. Property 5 of the Sound predicate then ensures that all partnered sessions agree on the peer identity and the pre-shared key identity, so ExplicitAuth is not violated by session π_u^i. The same property holds for initiator sessions accepting stages $s \leq 5$. So ExplicitAuth 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 (CATS) and 7 (SATS) 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 ExplicitAuth 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 ExplicitAuth predicate.

Phase 3: Ensuring that the Challenge Bit is Idependently 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 return 0 from FIN when it is set. Flag $\mathsf{bad}_{\mathrm{DHE}}$ is set if the adversary ever queries a random oracle

$$\mathrm{RO}_x(\mathrm{PSK}, \mathrm{DHE}, \mathrm{RO}_{\mathsf{Th}}(sid[s]))$$

for $(x,s) \in \{(htk_C,3),(htk_S,4),(fin_S,5)(\mathrm{CATS},5),(\mathrm{SATS},6),(\mathrm{EMS},7),(fin_C,8),(\mathrm{RMS},8)\}$ such that

- PSK is corrupted, i.e., the adversary made a prior query $\textsc{RevLongTermKey}(u,v,pskid)$ with $\mathsf{pskeys}[(u,v,pskid)] = \mathrm{PSK}$,
- there are honest sessions π_u^i and π_v^j that are contributively partnered in stage s with $\pi_v^j.cid_{\pi_u^i.role}[s] = \pi_u^i.cid_{\pi_u^i.role}[s] = (\mathtt{CH},\mathtt{CKS},\mathtt{CPSK},\mathtt{SH},\mathtt{SKS},\mathtt{SPSK},\dots)$, and
- $\mathrm{DHE} = g^{xy}$ such that $\mathtt{CKS} = g^x$ and $\mathtt{SKS} = g^y$.[12]

We bound the probability of flag $\mathsf{bad}_{\mathrm{DHE}}$ being set via a reduction $\mathscr{B}_{\mathrm{DHE}}$ to the strong Diffie–Hellman assumption in group $\mathbb{G}$. Reduction $\mathscr{B}_{\mathrm{DHE}}$ simulates Game 10 for $\mathscr{A}$, and it wins the strong Diffie–Hellman whenever the simulated game would set the $\mathsf{bad}_{\mathrm{DHE}}$ flag.

Definition 12. Let $\mathbb{G}$ be a group of order p generated by g. We define

$$\mathbf{Adv}_{\mathbb{G}}^{\mathsf{stDH}}(t_{\mathscr{B}_{\mathrm{DHE}}}, 2q_{\mathrm{RO}}) := \Pr\left[g^{ab} \leftarrow_{\$} \mathscr{B}_{\mathrm{DHE}}^{\mathsf{stDH}_a(\cdot,\cdot)}(g^a, g^b) : a,b \leftarrow_{\$} \mathbb{Z}_p\right]$$

where stDH_a is a special "fixed-exponent DDH oracle" that on input (B,C) returns 1 if and only if $C = B^a$.

Construction of reduction $\mathscr{B}_{\mathrm{DHE}}$.

The reduction $\mathscr{B}_{\mathrm{DHE}}$ gets as input a strong DH challenge $(A = g^a, B = g^b)$ as well as access to the oracle stDH_a for the Decisional Diffie–Hellman problem with the first argument fixed. Adversary $\mathscr{B}_{\mathrm{DHE}}$ then honestly executes the INIT, $\textsc{RevSessionKey}$, $\textsc{Test}$, and $\textsc{NewSecret}$ oracles as Game 10 would, managing all game variables itself. We explain in more detail how $\mathscr{B}_{\mathrm{DHE}}$ answers $\textsc{Send}$, $\textsc{RevLongTermKey}$, and random oracle queries.

[12] Note that the game knows the exponents x and y used by the sessions, but the reduction constructed in the remainder will not.

When $\mathscr{A}$ makes a query to the SEND oracle, $\mathscr{B}_{\mathsf{DHE}}$ 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, $\mathscr{B}_{\mathsf{DHE}}$ embeds its strong DH challenges into the key shares of every initiator session and every partnered responder session. When $\mathsf{bad}_{\mathsf{DHE}}$ is triggered, $\mathscr{B}_{\mathsf{DHE}}$ learns the Diffie–Hellman secret DHE associated with two of these embedded key shares, and it can extract a solution to the strong DH challenge using some basic algebra. However, $\mathscr{B}_{\mathsf{DHE}}$ 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 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 $\mathscr{B}_{\mathsf{DHE}}$ 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 $RO_x(\mathsf{PSK}, Z, d)$, 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 stDH challenge. See the remainder of the proof for details.

Implementation of honest server sessions.

Consider any server session π_v^j.

1. Upon receiving $(\texttt{CH}, \texttt{CKS},$

   $\mathit{CPSKtls})$, the reduction $\mathscr{B}_{\mathrm{DHE}}$ first checks whether π_v^j has an honest partner in stages 1 (ETS) and 2 (EEMS) by checking for entries indexed by $\pi_v^j.\mathit{sid}[1]$ and $\pi_v^j.\mathit{sid}[2]$ in the look-up table SKEYS introduced in Game 6. If no such entries exist, then $\mathscr{B}_{\mathrm{DHE}}$ 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 $\texttt{SKS}$ by randomizing the challenge key share B. Namely, it chooses a randomizer $\tau_v^j \xleftarrow{\$} \mathbb{Z}_p$ uniformly at random and sets $Y \leftarrow B \cdot g^{\tau_v^j}$. Then, it logs τ_v^j under index (r_S, Y) in the look-up table KSRnd.

2. Before π_v^j outputs $(\texttt{SH}, \texttt{SKS}, \texttt{SPSK})$, it computes the keys htk_C and htk_S. By Game 8, these keys are sampled randomly when PSK is uncorrupted and computed using $\mathrm{RO}_{\mathit{htk}_C}$, resp. $\mathrm{RO}_{\mathit{htk}_S}$ otherwise. In both cases, $\mathscr{B}_{\mathrm{DHE}}$ needs to know the Diffie–Hellman secret DHE to log in table $\mathsf{PrgList}_x$ or to query RO_x for $x \in \{\mathit{htk}_C, \mathit{htk}_S\}$. However, $\mathscr{B}_{\mathrm{DHE}}$ cannot compute DHE because it does not know the discrete logarithms of either $\texttt{CKS}$ or $\texttt{SKS}$.

   Therefore, $\mathscr{B}_{\mathrm{DHE}}$ needs to compute the keys without knowing the DHE key using the control over the random oracles.

   If the pre-shared key has been corrupted, the adversary could potentially have already queried the random oracle $\mathrm{RO}_{\mathit{htk}_C}$ with the query π_v^j should make. To that end, $\mathscr{B}_{\mathrm{DHE}}$ first checks whether the corresponding query for htk_C was already made to $\mathrm{RO}_{\mathit{htk}_C}$. Concretely, $\mathscr{B}_{\mathrm{DHE}}$ computes the context hash $d = \mathrm{RO}_{\mathsf{Th}}(\texttt{CH} \| \cdots \| \texttt{SPSK})$ and checks for a suitable $\mathrm{RO}_{\mathit{htk}_C}$ query using the look-up table $\mathsf{DHEList}_{\mathit{htk}_C}[\mathsf{PSK}, d]$ maintained in $\mathrm{RO}_{\mathit{htk}_C}$ (see above for the definition). Reduction $\mathscr{B}_{\mathrm{DHE}}$ queries $\mathsf{stDH}_a(Y, Z \cdot Y^{-\tau_u^i})$ for all $Z \in \mathsf{DHEList}_{\mathit{htk}_C}[\mathsf{PSK}, d]$, 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. (Although this may cause several stDH_a queries in response to a single SEND query, $\mathscr{B}_{\mathrm{DHE}}$ 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 $\mathsf{DHEList}_{\mathit{htk}_C}[\mathsf{PSK}, d]$ will be checked at most twice over the course of the entire reduction.)

If any one of these queries is answered positively, we have by the definition of stDH_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 $\mathtt{CH}$ message received by π_v^j. This exactly is the DHE 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 $\mathscr{B}_{\mathrm{DHE}}$ submit the value

$$Z \cdot Y^{-\tau_u^i} \cdot A^{-\tau_v^j} = Y^a \cdot A^{-\tau_v^j} = (g^a)^{b+\tau_v^j} \cdot (g^a)^{-\tau_v^j} = g^{ab}$$

to the FIN oracle 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 stDH_a queries will be answered positively. Thus, $\mathscr{B}_{\mathrm{DHE}}$ will win if $\mathsf{bad}_{\mathrm{DHE}}$ 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., $\mathsf{bad}_{\mathrm{DHE}}$ will also not be set, then π_v^j samples the key $htk_C \xleftarrow{\$} \mathsf{KE.KS}[3]$ itself and logs the following information so that future RO queries can be answered appropriately:

$$\mathsf{RndList}_{htk_C}(\mathrm{PSK}, d = \mathsf{H}(\mathtt{CH} \,\|\, \cdots \,\|\, \mathtt{SPSK})) \leftarrow \left(\tau_u^i, \tau_v^j, (\mathtt{CH} \,\|\, \cdots \,\|\, \mathtt{SPSK}), \bot\right).$$

Again, we do the same for htk_S.

If PSK is not corrupted, then $\mathsf{bad}_{\mathrm{DHE}}$ 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 DHE 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 DHE; instead, it logs

$$\mathsf{RndList}_x[((u, v, pskid), d = \mathsf{H}(\mathtt{CH} \,\|\, \cdots \,\|\, \mathtt{SPSK}))] \leftarrow \left(\tau_u^i, \tau_v^j, (\mathtt{CH} \,\|\, \cdots \,\|\, \mathtt{SPSK}), \bot\right).$$

for $x \in \{htk_C, htk_S\}$. This will allow $\mathscr{B}_{\text{DHE}}$ to win if a later RevLongTermKey or random oracle query triggers $\mathsf{bad}_{\text{DHE}}$.

3. To compute the `ServerFinished` message $\mathscr{B}_{\text{DHE}}$ proceeds exactly as in Step 2 except that it uses the random oracle RO_{fin_S} and context $\texttt{CH} \| \cdots \| \texttt{EE}$ through the `EncryptedExtensions`. Also, the `ServerFinished` message is computed first by the server, so $\mathscr{B}_{\text{DHE}}$ does not check table SKEYS or TAGS for any entries. Reduction $\mathscr{B}_{\text{DHE}}$ 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 DHE. Instead, it logs the derived value of fin_S in table TAGS and replaces DHE in the index of TAGS by $\left(\tau_u^i, \tau_v^j, (\texttt{CH} \| \cdots \| \texttt{EE})\right)$. That is, if it computes fin_S for inputs PSK, d_1, and d_2, it logs
$$\mathsf{TAGS}[fin_S, \text{PSK}, (\tau_u^i, \tau_v^j, (\texttt{CH} \| \cdots \| \texttt{EE})), d_1, d_2] \leftarrow fin_S.$$
That way, it is possible to identify DHE 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 $\mathscr{B}_{\text{DHE}}$ proceeds exactly as for fin_S above, except that we again use different random oracles and the context $cid_{\text{CATS}} = \texttt{CH} \| \cdots \| \texttt{SF} = cid_{\text{SATS}} = cid_{\text{EMS}}$, where cid_x denotes transcript contained in the contributive identifier which is prefixed by "x", and thus the hash $d = \text{RO}_{\mathsf{Th}}(\texttt{CH} \| \cdots \| \texttt{SF})$. With respect to random oracles, we have RO_{CATS} for CATS, RO_{SATS} for SATS and RO_{EMS} for EMS, respectively. Reduction $\mathscr{B}_{\text{DHE}}$ logs the keys in table SKEYS under their respective session identifiers, which do not contain DHE or any unknown values. After this is done, π_v^j outputs $(\texttt{EE}, \texttt{SF})$.

5. Upon receiving `CF`, $\mathscr{B}_{\text{DHE}}$ looks for a suitable entry for fin_C in TAGS. If there is a value fin_C consistent with π_v^j's view, $\mathscr{B}_{\text{DHE}}$ terminates the session as specified if `CF` does not match the looked-up value of fin_C. Otherwise, $\mathscr{B}_{\text{DHE}}$ continues to compute RMS. To this end, $\mathscr{B}_{\text{DHE}}$ 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 $\mathsf{bad}_{\mathsf{MAC}}$ introduced in Game 9. Therefore, the adversary also would be allowed to query $\mathrm{RO}_{\mathsf{RMS}}$ to compute **RMS**. Thus, $\mathscr{B}_{\mathrm{DHE}}$ 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, $\mathscr{B}_{\mathrm{DHE}}$ 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 $\mathscr{B}_{\mathrm{DHE}}$ can simply implement π_v^j as specified in Game 10.

In case there is an honest contributive stage-3 partner, then $\mathscr{B}_{\mathrm{DHE}}$ proceeds as described in Step 2 for oracle $\mathrm{RO}_{\mathsf{RMS}}$ and context hash $d = \mathrm{RO}_{\mathtt{Th}}(cid_{\mathsf{RMS}}) = \mathrm{RO}_{\mathtt{Th}}(\mathtt{CH} \| \cdots \| \mathtt{CF})$ to check whether the adversary already solved the stDH challenge for $\mathscr{B}_{\mathrm{DHE}}$. Note that the stage-3 session identifier uniquely defines the **DHE** key, thus if there is an honest partner and there is a respective $\mathrm{RO}_{\mathsf{RMS}}$ query, the adversary has to break stDH to submit the query.

Implementation of honest client sessions.

Consider any client session π_u^i.

1. The reduction $\mathscr{B}_4$ proceeds exactly as in Game 10 until the session chooses its key share. Instead of choosing a fresh exponent as specified in Figure 3.1, it chooses a value $\tau_u^i \xleftarrow{\$} \mathbb{Z}_p$ uniformly at random and sets $X \leftarrow A \cdot g^{\tau_u^i}$ 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 $(\mathtt{CH}, \mathtt{CKS},$
$CPSKtls)$.

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, $\mathscr{B}_{\mathrm{DHE}}$ 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, $\mathscr{B}_{\text{DHE}}$ needs to use the stDH_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 $d = \text{RO}_{\mathsf{Th}}(\mathtt{CH} \| \ldots \| \mathtt{SPSK})$ and then query the stDH_a oracle for all $Z \in \mathsf{DHEList}_x[\text{PSK}, d]$, where PSK is the pre-shared key used by π_u^i, as

$$\mathsf{stDH}_a(Y, Z \cdot Y^{-\tau_u^i}) = 1 \iff Z = Y^a,$$

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 $\text{RO}_x(\text{PSK}, Z, d)$. If there is no Z that results in a positive query, let $key \xleftarrow{\$} \mathsf{KE.KS}[x]$ be sampled at random, and $\mathscr{B}_{\text{DHE}}$ logs the value for possible later reprogramming of the random oracle RO_x, i.e.,

$$\mathsf{RndList}_x[(\text{PSK}, d = \text{RO}_{\mathsf{Th}}(\mathtt{CH} \| \cdots \| \mathtt{SPSK}))] \leftarrow \left(\tau_u^i, \perp, (\mathtt{CH} \| \cdots \| \mathtt{SPSK}), key\right).$$

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

$$\mathsf{RndList}_x[(\text{PSK}, d = \text{RO}_{\mathsf{Th}}(\mathtt{CH} \| \cdots \| \mathtt{SPSK}))] \leftarrow \left(\tau_u^i, \perp, (\mathtt{CH} \| \cdots \| \mathtt{SPSK}), key\right).$$

$\mathsf{PrgList}_x$ is not updated as in Game 10, because DHE is unknown.

3. Upon receiving $(\texttt{EE}, \texttt{SF})$, similar to the previous step, π_v^j checks whether there is an entry in SKEYS and TAGS (to verify $\texttt{SF}$) 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 $\texttt{EE}$ output by an honest session and then compute a new $\texttt{SF}$ message for the changed transcript. Hence, there is an honest stage-3 partner, but no stage-5 partner. In this case, $\mathscr{B}_{\mathrm{DHE}}$ again applies the approach from above (see implementation of server session, Step 2) for the random oracles RO_x for $x \in \{\mathrm{CATS}, \mathrm{SATS}, \mathrm{EMS}\}$ and the context $d = \mathrm{RO}_{\mathsf{Th}}(\texttt{CH} \| \cdots \| \texttt{SF})$ checking whether the random oracles received already a correct query which set the keys CATS, SATS and EMS. If this is the case and since there was a stage-3 partner, $\mathscr{B}_{\mathrm{DHE}}$ 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, $\texttt{SKS}$ was chosen by the adversary. Hence, $\mathscr{B}_{\mathrm{DHE}}$ needs to apply the procedure described in the previous step (Step 2) and use the oracle stDH_a to check the random oracles RO_x for $x \in \{\mathrm{CATS}, \mathrm{SATS}, \mathrm{EMS}\}$ whether they already set the keys. The important difference here is that a positive answer of the stDH_a oracle does not solve stDH, as $\texttt{SKS}$ was chosen by the adversary. Note that $\mathscr{B}_{\mathrm{DHE}}$ 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 $\mathsf{DHEList}_{\mathit{fin}_C}[\mathrm{PSK}, d]$ 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 calls FIN 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 $\mathsf{RndList}_{\mathit{fin}_C}$ 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 DHE independent of there being a honest stage-3 partner or not, $\mathscr{B}_{\text{DHE}}$ 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 $d = \text{RO}_{\mathsf{Th}}(\mathtt{CH} \| \cdots \| \mathtt{CF})$. The only difference is that in case there was a stage-3 partner, FIN is queried when the stDH oracle returns true, and if there is no stage-3 partner, RMS is only programmed. Then, π_u^i outputs $\mathtt{CF}$.

Besides changing the implementation of the session oracles, we also need to adapt the random oracles RO_x for $x \in \{htk_C, \dots, \text{RMS}\}$ to make sure (1) $\mathscr{B}_{\text{DHE}}$ programs the random oracle retroactively if the random oracle receives the right query and (2) to check whether the adversary computed DHE for $\mathscr{B}_{\text{DHE}}$ 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 $\mathsf{DHEList}_k[\text{PSK}, d]$. If $\mathsf{RndList}_k[\text{PSK}, d] \neq \perp$, then there already was a session using PSK and context hash d trying to compute a key without knowing the correct DHE secret. Therefore, $\mathscr{B}_{\text{DHE}}$ uses the stDH_a oracle to check whether Z is that secret. Let $(\tau_u^i, \tau_v^j, ctxt, key)$ be the entry of $\mathsf{RndList}_k[\text{PSK}, d]$, where τ_u^i and τ_v^j denote the randomness used by the client and the server to randomize the stDH challenge, respectively, $ctxt = \mathtt{CH} \| \mathtt{CKS} \|$
$CPSKtls \| \mathtt{SH} \| \mathtt{SKS} \| \mathtt{SPSK} \| \cdots$ denotes the context such that $d = \text{RO}_{\mathsf{Th}}(ctxt)$ 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 $\mathsf{stDH}_a(Y, Z \cdot Y^{-\tau_u^i})$. If this query is answered positively, $\mathscr{B}_{\text{DHE}}$ knows that the right DH value Z was queried. If $\tau_u^j = \perp$, i.e., the log in $\mathsf{RndList}_k$ was set by a client without an honestly partnered server, $\mathscr{B}_{\text{DHE}}$ needs to program the random oracle to be consistent. That is, $\mathsf{ROList}_k[\text{PSK}, Z, d] \leftarrow key$. Otherwise, $\mathscr{B}_{\text{DHE}}$ knows that the $\mathsf{PrgList}_x$ entry was set by an honestly partnered session, and thus Z is a randomized solution to the stDH challenge. Thus, $\mathscr{B}_{\text{DHE}}$ submits the solution $Z \cdot Y^{-\tau_u^i} \cdot A^{-\tau_v^j}$ to its stDH FIN oracle.

Unless $\mathscr{B}_{\mathrm{DHE}}$ solved the stDH challenge, the oracle outputs $\mathsf{ROList}_x[\mathsf{PSK}, Z, d]$.

Implementation of corruption oracle RevLongTermKey.

Finally, $\mathscr{B}_{\mathrm{DHE}}$ 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 $\mathscr{B}_{\mathrm{DHE}}$ 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 $((u, v, pskid), d)$. $\mathscr{B}_{\mathrm{DHE}}$ 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 $\mathsf{bad}_{\mathsf{PSK}}$ flag and cause the game to abort. $\mathscr{B}_{\mathrm{DHE}}$ also cannot program future queries because we still do not know DHE. Instead, $\mathscr{B}_{\mathrm{DHE}}$ 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 $\mathscr{B}_{\mathrm{DHE}}$.

By the considerations above, we have that if $\mathsf{bad}_{\mathsf{DHE}}$ is set the $\mathscr{B}_{\mathrm{DHE}}$ wins the strong DH challenge. The identical-until-bad-lemma gives us that

$$\begin{aligned}\Pr[\mathrm{G}_9^{\mathscr{A}} \Rightarrow 1] &\leq \Pr[\mathrm{G}_{10}^{\mathscr{A}} \Rightarrow 1] + \Pr[\mathsf{bad}_{\mathsf{DHE}}] \\ &\leq \Pr[\mathrm{G}_{10}^{\mathscr{A}} \Rightarrow 1] + \mathbf{Adv}_{\mathbb{G}}^{\mathsf{stDH}}(t_{\mathscr{B}_{\mathrm{DHE}}}, 2q_{\mathrm{RO}}), \end{aligned} \tag{3.6}$$

where the number of stDH_a oracle queries is no greater than $2q_{\mathrm{RO}}$, since $\mathscr{B}_{\mathrm{DHE}}$ will query the oracle at most twice (once for each partner) for every random oracle query issued by the adversary, and $t_{\mathscr{B}_{\mathrm{DHE}}}$ with $t_{\mathscr{B}_{\mathrm{DHE}}} \approx t + 4\log(p) \cdot q_{\mathrm{RO}}$ is the running time of $\mathscr{B}_{\mathrm{DHE}}$. Note that for every stDH_a query, $\mathscr{B}_{\mathrm{DHE}}$ needs to perform one group operation and one exponentiation in $\mathbb{G}$, the latter can be done in $2\log(p)$ many group operations using, e.g., the square-and-multiply algorithm. Thus, the time to answer a single stDH_a query take approximately time $2\log(p)$ and taking this together with the bound on the number of stDH_a yields the approximate runtime $t_{\mathscr{B}_{\mathrm{DHE}}}$.

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 FIN to return 0. 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 ExplicitAuth never beeing 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 in Game 10 cannot gain any information on the challenge bit b without violating any of the predicates Sound, ExplicitAuth, or Fresh. Thus, the probability that FIN and thus Game 10 returns 1 is no greater than 1/2. Formally,

$$\Pr[\mathrm{G}_{10}^{\mathscr{A}} \Rightarrow 1] \leq \frac{1}{2}.$$

Collecting all the terms, we get the final bound

$$\begin{aligned}\mathbf{Adv}^{\textsf{KE-SEC}}_{\textsf{TLS1.3-PSK-(EC)DHE}}&(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}})\\ &\leq \frac{2q_{\mathrm{S}}^2}{2^{nl} \cdot p} + \mathbf{Adv}^{\textsf{CR}}_{\mathrm{RO}_{binder}}(q_{\mathrm{RO}} + q_{\mathrm{S}}) + \frac{q_{\mathrm{NS}}^2}{2^{hl}} + \mathbf{Adv}^{\textsf{CR}}_{\mathrm{RO}_{\textsf{Th}}}(q_{\mathrm{RO}} + 6q_{\mathrm{S}})\\ &\quad + \frac{q_{\mathrm{RO}} \cdot q_{\mathrm{NS}}}{2^{hl}} + \frac{q_{\mathrm{S}}}{2^{hl}} + \mathbf{Adv}^{\textsf{stDH}}_{\mathbb{G}}(t_{\mathscr{B}_{\mathrm{DHE}}}, 2q_{\mathrm{RO}})\end{aligned}$$

Applying the result of Appendix **??**, we can make the collision resistance terms explicit

$$\begin{aligned}\mathbf{Adv}^{\textsf{KE-SEC}}_{\textsf{TLS1.3-PSK-(EC)DHE}}&(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}})\\ &\leq \frac{2q_{\mathrm{S}}^2}{2^{nl} \cdot p} + \frac{(q_{\mathrm{RO}} + q_{\mathrm{S}})^2}{2^{hl}} + \frac{q_{\mathrm{NS}}^2}{2^{hl}} + \frac{(q_{\mathrm{RO}} + 6q_{\mathrm{S}})^2}{2^{hl}} + \frac{q_{\mathrm{RO}} \cdot q_{\mathrm{NS}}}{2^{hl}} + \frac{q_{\mathrm{S}}}{2^{hl}}\\ &\quad + \mathbf{Adv}^{\textsf{stDH}}_{\mathbb{G}}(t_{\mathscr{B}_{\mathrm{DHE}}}, 2q_{\mathrm{RO}})\end{aligned}$$

Further, applying the GGM bound for the strong Diffie–Hellman problem proven by Davis and Günther in [82], we get the final result

$$
\begin{aligned}
&\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathtt{TLS1.3\text{-}PSK\text{-}(EC)DHE}}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) \\
&\quad \leq \frac{2q_{\mathrm{S}}^2}{2^{nl} \cdot p} + \frac{(q_{\mathrm{RO}} + q_{\mathrm{S}})^2}{2^{hl}} + \frac{q_{\mathrm{NS}}^2}{2^{hl}} + \frac{(q_{\mathrm{RO}} + 6q_{\mathrm{S}})^2}{2^{hl}} + \frac{q_{\mathrm{RO}} \cdot q_{\mathrm{NS}}}{2^{hl}} + \frac{q_{\mathrm{S}}}{2^{hl}} \\
&\qquad + \frac{4(t + 4\log(p) \cdot q_{\mathrm{RO}})^2}{p} \\
&\quad = \frac{2q_{\mathrm{S}}^2}{2^{nl} \cdot p} + \frac{(q_{\mathrm{RO}} + q_{\mathrm{S}})^2 + q_{\mathrm{NS}}^2 + (q_{\mathrm{RO}} + 6q_{\mathrm{S}})^2 + q_{\mathrm{RO}} \cdot q_{\mathrm{NS}} + q_{\mathrm{S}}}{2^{hl}} \\
&\qquad + \frac{4(t + 4\log(p) \cdot q_{\mathrm{RO}})^2}{p}
\end{aligned}
$$

∎

3.6.3 Full Security Bound for TLS 1.3 PSK-(EC)DHE and PSK-only

We can finally combine the results of Sections 3.4, 3.5, and our key exchange bound above to produce fully concrete bounds for the TLS 1.3 PSK-(EC)DHE and PSK-only handshake protocols as specified on the left-hand side of Figure 1. This bound applies to the protocol *with handshake traffic encryption* and *internal keys* when *only modeling as random oracle* RO_{H} the hash function $\mathbf{H}$.

First, we define three variants of the TLS 1.3 PSK handshake:

- KE_0, as defined in Theorem 8 with handshake traffic encryption and one random oracle RO_{H}. (This is the variant we want to obtain our overall result for.)

- KE_1, as defined in Theorem 8 with handshake traffic encryption and 12 random oracles $\mathrm{RO}_{\mathsf{Th}}$, RO_{binder}, ..., $\mathrm{RO}_{\mathsf{RMS}}$.

- KE_2: as defined in Theorem 9, with no handshake traffic encryption and 12 random oracles $\mathrm{RO}_{\mathsf{Th}}$, RO_{binder}, ..., $\mathrm{RO}_{\mathsf{RMS}}$.

Theorem 8 grants that

$$\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_0}(t,q_{\mathrm{NS}},q_{\mathrm{S}},q_{\mathrm{RS}},q_{\mathrm{RL}},q_{\mathrm{T}},q_{\mathrm{RO}}) \leq \mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_1}(t,q_{\mathrm{NS}},q_{\mathrm{S}},q_{\mathrm{RS}},q_{\mathrm{RL}},q_{\mathrm{T}},q_{\mathrm{RO}}) + \frac{2(12q_{\mathrm{S}}+q_{\mathrm{RO}})^2}{2^{hl}} + \frac{2q_{\mathrm{RO}}^2}{2^{hl}} + \frac{8(q_{\mathrm{RO}}+36q_{\mathrm{S}})^2}{2^{hl}}.$$

Next, we apply Theorem 9, yielding the bound

$$\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_1}(t,q_{\mathrm{NS}},q_{\mathrm{S}},q_{\mathrm{RS}},q_{\mathrm{RL}},q_{\mathrm{T}},q_{\mathrm{RO}}) \leq \mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_2}(t+t_{\mathrm{AEAD}}\cdot q_{\mathrm{S}},q_{\mathrm{NS}},q_{\mathrm{S}},q_{\mathrm{RS}}+q_{\mathrm{S}},q_{\mathrm{RL}},q_{\mathrm{T}},q_{\mathrm{RO}}),$$

where t_{AEAD} is the maximum time required to execute AEAD encryption or decryption of TLS 1.3 messages.

Theorem 11 then finally and entirely bounds the advantage against the $\mathsf{KE\text{-}SEC}$ security of KE_2. Collecting these bounds gives

$$\begin{aligned}
\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_0}(t,q_{\mathrm{NS}},q_{\mathrm{S}},q_{\mathrm{RS}},q_{\mathrm{RL}},q_{\mathrm{T}},q_{\mathrm{RO}}) &\leq \mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_1}(t,q_{\mathrm{NS}},q_{\mathrm{S}},q_{\mathrm{RS}},q_{\mathrm{RL}},q_{\mathrm{T}},q_{\mathrm{RO}}) \\
&\quad + \frac{2(12q_{\mathrm{S}}+q_{\mathrm{RO}})^2}{2^{hl}} + \frac{2q_{\mathrm{RO}}^2}{2^{hl}} + \frac{8(q_{\mathrm{RO}}+36q_{\mathrm{S}})^2}{2^{hl}} \\
&\leq \mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{KE}_2}(t+t_{\mathrm{AEAD}}\cdot q_{\mathrm{S}},q_{\mathrm{NS}},q_{\mathrm{S}},q_{\mathrm{RS}}+q_{\mathrm{S}},q_{\mathrm{RL}},q_{\mathrm{T}},q_{\mathrm{RO}}) \\
&\quad + \frac{2(12q_{\mathrm{S}}+q_{\mathrm{RO}})^2+2q_{\mathrm{RO}}^2+8(q_{\mathrm{RO}}+36q_{\mathrm{S}})^2}{2^{hl}} \\
&\leq \frac{2q_{\mathrm{S}}^2}{2^{nl}\cdot p} + \frac{(q_{\mathrm{RO}}+q_{\mathrm{S}})^2+q_{\mathrm{NS}}^2+(q_{\mathrm{RO}}+6q_{\mathrm{S}})^2+q_{\mathrm{RO}}\cdot q_{\mathrm{NS}}+q_{\mathrm{S}}}{2^{hl}} \\
&\quad + \frac{4(t+t_{\mathrm{AEAD}}\cdot q_{\mathrm{S}}+4\log(p)\cdot q_{\mathrm{RO}})^2}{p} \\
&\quad + \frac{2(12q_{\mathrm{S}}+q_{\mathrm{RO}})^2+2q_{\mathrm{RO}}^2+8(q_{\mathrm{RO}}+36q_{\mathrm{S}})^2}{2^{hl}}.
\end{aligned}$$

This yields the following overall result for the $\mathsf{KE\text{-}SEC}$ security of the TLS 1.3 PSK-(EC)DHE handshake protocol.

Corollary 1. *Let* `TLS1.3-PSK-(EC)DHE` *be the TLS 1.3 PSK-(EC)DHE handshake protocol as specified on the left-hand side in Figure 3.1. Let* $\mathbb{G}$ *be the Diffie–Hellman group of order* p*. Let* nl *be the length in bits of the nonce, let* hl *be the output length in bits of* $\mathbf{H}$*, and let the pre-shared* *key space be* $\mathsf{KE.PSKS} = \{0,1\}^{hl}$. *Let* $\mathbf{H}$ *be modeled as a random oracle* RO_{H}. *Then,*

$$\begin{aligned}
&\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathtt{TLS1.3\text{-}PSK\text{-}(EC)DHE}}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}}) \\
&\quad \leq \frac{2q_{\mathrm{S}}^2}{2^{nl} \cdot p} + \frac{(q_{\mathrm{RO}} + q_{\mathrm{S}})^2 + q_{\mathrm{NS}}^2 + (q_{\mathrm{RO}} + 6q_{\mathrm{S}})^2 + q_{\mathrm{RO}} \cdot q_{\mathrm{NS}} + q_{\mathrm{S}}}{2^{hl}} \\
&\qquad + \frac{4(t + t_{\mathrm{AEAD}} \cdot q_{\mathrm{S}} + 4\log(p) \cdot q_{\mathrm{RO}})^2}{p} \\
&\qquad + \frac{2(12q_{\mathrm{S}} + q_{\mathrm{RO}})^2 + 2q_{\mathrm{RO}}^2 + 8(q_{\mathrm{RO}} + 36q_{\mathrm{S}})^2}{2^{hl}}.
\end{aligned}$$

Our tight security proof for the TLS 1.3 PSK-(EC)DHE handshake given in Section 3.6.2 can be adapted to the PSK-only handshake. The structure and resulting bounds are largely the same between the two modes, with a couple of significant changes. Naturally, we have no Diffie–Hellman group, no key shares in the `ClientHello` or `ServerHello` messages, and no reduction to the strong Diffie–Hellman problem. Without the reduction to stDH, we cannot achieve forward secrecy for any key: an adversary in possession of the pre-shared key can compute all session keys.

The security proof for the TLS 1.3 PSK-only handshake uses the same sequence of games G_0 to G_9 (excluding the reduction to the strong Diffie–Hellman problem in G_{10}). There only is a difference in G_1, in which we exclude collisions of nonces and group elements sampled by honest session to compute there `Hello` messages. Since we do not have any key shares in the PSK-only mode, the session will consequently also not sample a group elements. Thus, the bound for G_0 changes to

$$\Pr[\mathrm{G}_0 \Rightarrow 1] \leq \Pr[\mathrm{G}_1 \Rightarrow 1] + \frac{2q_{\mathrm{S}}^2}{2^{nl}}.$$

The rest of the arguments follow similarly as given in Section 3.6.2. We obtain the following result.

Theorem 12. *Let* `TLS1.3-PSK` *be the TLS 1.3 PSK-only handshake protocol as specified on the right-hand side in Figure 3.1 without handshake encryption. Let functions* $\mathbf{H}$ *and* TKDF_x *for each* $x \in \{binder, \ldots, \mathrm{RMS}\}$ *be modeled as* 12 *independent random oracles* $\mathrm{RO}_{\mathsf{Th}}, \mathrm{RO}_{binder}, \ldots, \mathrm{RO}_{\mathsf{RMS}}$. *Let* nl *be the length in bits of the nonce, let* hl *be the output length in bits of* $\mathbf{H}$, *and let the* *pre-shared key space* $\mathsf{KE.PSKS}$ *be the set* $\{0,1\}^{hl}$*. Then,*

$$\begin{aligned}&\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{TLS1.3\text{-}PSK}}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}})\\ &\quad \leq \frac{2q_{\mathrm{S}}^2}{2^{nl}} + \frac{(q_{\mathrm{RO}}+q_{\mathrm{S}})^2 + q_{\mathrm{NS}}^2 + (q_{\mathrm{RO}}+6q_{\mathrm{S}})^2 + q_{\mathrm{RO}} \cdot q_{\mathrm{NS}} + q_{\mathrm{S}}}{2^{hl}}\end{aligned}$$

From this we obtain the following overall result for the TLS 1.3 PSK-only mode via the same series of arguments as in Section 3.6.3.

Corollary 2. *Let* $\mathtt{TLS1.3\text{-}PSK}$ *be the TLS 1.3 PSK-only handshake protocol as specified on the left-hand side in Figure 3.1. Let* nl *be the length in bits of the nonce, let* hl *be the output length in bits of* $\mathbf{H}$*, and let the pre-shared key space be* $\mathsf{KE.PSKS} = \{0,1\}^{hl}$*. Let* $\mathbf{H}$ *be modeled as a random oracle* RO_{H}*. Then,*

$$\begin{aligned}&\mathbf{Adv}^{\mathsf{KE\text{-}SEC}}_{\mathsf{TLS1.3\text{-}PSK}}(t, q_{\mathrm{NS}}, q_{\mathrm{S}}, q_{\mathrm{RS}}, q_{\mathrm{RL}}, q_{\mathrm{T}}, q_{\mathrm{RO}})\\ &\quad \leq \frac{2q_{\mathrm{S}}^2}{2^{nl}} + \frac{(q_{\mathrm{RO}}+q_{\mathrm{S}})^2 + q_{\mathrm{NS}}^2 + (q_{\mathrm{RO}}+6q_{\mathrm{S}})^2 + q_{\mathrm{RO}} \cdot q_{\mathrm{NS}} + q_{\mathrm{S}}}{2^{hl}}\\ &\qquad + \frac{2(12q_{\mathrm{S}}+q_{\mathrm{RO}})^2 + 2q_{\mathrm{RO}}^2 + 8(q_{\mathrm{RO}}+36q_{\mathrm{S}})^2}{2^{hl}}.\end{aligned}$$

3.7 Evaluation

Asymptotically, our tighter security bounds improve on prior analysis of TLS 1.3 by a quadratic factor. We evaluate ours and prior bounds over a wide range of fully concrete resource parameters, following the approach of Davis and Günther [82]. The full range of evaluated parameters is given in Tables 3.2 and 3.3 below, along with reasoning for how we chose the various ranges of resource parameters. The tables show that while the prior PSK-(EC)DHE bound by Dowling et al. [92] meets the target security goals in a number of configurations, there are at least some settings for all elliptic-curve groups in which the targeted security is not met. Our bounds do significantly better than the target in all configurations we considered. The gap for the PSK-only handshake is less significant as the loosest prior reduction for TLS 1.3 was to the Diffie–Hellman problem.

Overall, our bounds improve on previous analyses of the PSK-only handshake by 15 to 53 bits of security, and those of the PSK-(EC)DHE handshake by 60 to 131 bits of security, across all our parameters evaluated.

3.7.1 Evaluation Details

In the following, we will briefly explain the reasoning behind each of our specific resource parameter estimates. An adversary in the MSKE game (cf. Definition 10) is limited in its runtime t, the number of pre-shared keys $\#N$, and distinct protocol sessions $\#S$ it can observe or interact with, and the number of random oracle queries $\#RO$ it can make. This last quantity captures offline work the adversary spends on computing the hash function H, which in our analysis we model as random oracle. The choice of ciphersuite enters the bound through the length of the symmetric session keys and pre-shared keys. For the PSK-(EC)DHE handshake, the bound also depends on the underlying Diffie–Hellman group.

Runtime $t \in \{2^{40}, 2^{60}, 2^{80}\}$.

We consider a range of adversarial runtimes from easily achievable (2^{40} operations) to state-scaled computational power (2^{80} operations).

Random oracle queries $\#RO \in \{2^{40}, 2^{60}, 2^{80}\}$.

The number of random oracle queries models the number of hash function computations an adversary is capable of computing. Accordingly, we scale the number of RO queries with the runtime, always setting $\#RO = t/2^{10}$.

Number of pre-shared keys $\#N \in \{2^{25}, 2^{35}\}$.

The world's largest certificate authority Let's Encrypt reports $\approx 2^{27.5}$ active certificates for fully-qualified domains.[13] While not every *user* of TLS 1.3 will perform resumption, our model counts the number of *pre-shared keys*, where typically users may hold many pre-shared keys, with servers regularly issuing several PSKs per full-handshake connection for later resumption. We hence estimate that the number of pre-shared keys accessible to a globally-scaled adversary may well exceed the reported number of (server) certificates.

[13] https://letsencrypt.org/stats/

Number of sessions $\#S \in \{2^{35}, 2^{45}, 2^{55}\}$.

We use the same estimates as Davis and Günther [82], based on Google's and Firefox's usage reports.[14] With a daily browser user base of 2 billion ($\approx 2^{31}$) and an HTTPS traffic encryption rate in the range of 76–98%, we estimate an adversary could encounter up to 2^{55} distinct sessions over an extended time period. Note that although the PSK handshakes are less commonly used by browsers than the full TLS 1.3 handshake, they are frequently used by embedded and low-powered devices which do not appear in these reports. Naturally, we do not allow the number of sessions to exceed the adversary's runtime t.

Diffie–Hellman groups.

There are ten Diffie–Hellman groups standardized for use with the PSK-(EC)DHE handshake: five elliptic-curve groups and five finite-field groups. We reduce to the security of the strong Diffie–Hellman assumption in each of these groups. Davis and Günther gave a proof of hardness in the generic group model (GGM) for the strong DH problem. This result is a good heuristic for elliptic-curve groups, but not for finite-field ones because they are vulnerable to index-calculus based attacks not covered by the GGM. The elliptic-curve groups are more efficient and more widely used than finite-field groups, so we restrict our analysis to these groups: `secp256r1`, `x25519`, `secp384r1`, `x448`, `secp521r1`. For each group, we give in Table 3.3 the order p and the expected security level b in bits. We use the security level b to determine the choice of hash function and the target security level for the entire PSK-(EC)DHE handshake.

Ciphersuite and symmetric lengths.

Our bounds reduce to the collision resistance of the random oracle $\mathrm{RO}_{\mathtt{Th}}$, which models the handshake's hash function. The choice of hash function also determines the length of the session and resumption keys. TLS 1.3 has five ciphersuites, all of which set the hash function to be either `SHA256` or `SHA384`. For PSK-(EC)DHE mode, we select `SHA256` as the hash function whenever a curve with 128-bit security is used and we select `SHA384` for higher-security curves. As our results of Section 3.4 only apply to PSK-only mode when `SHA256` is the hash function, we always use `SHA256` and a target-security level of 128 bits.

[14] https://transparencyreport.google.com/, https://telemetry.mozilla.org/

Adversary resources					PSK-only	
t	$\#N$	$\#S$	$\#RO$	Target $t/2^b$	DFGS [92]	Us (Cor. 2)
2^{40}	2^{25}	2^{35}	2^{30}	2^{-88}	$\approx 2^{-158}$	$\approx 2^{-173}$
2^{40}	2^{35}	2^{35}	2^{30}	2^{-88}	$\approx 2^{-150}$	$\approx 2^{-173}$
2^{60}	2^{25}	2^{35}	2^{50}	2^{-68}	$\approx 2^{-119}$	$\approx 2^{-152}$
2^{60}	2^{25}	2^{45}	2^{50}	2^{-68}	$\approx 2^{-109}$	$\approx 2^{-151}$
2^{60}	2^{25}	2^{55}	2^{50}	2^{-68}	$\approx 2^{-99}$	$\approx 2^{-133}$
2^{60}	2^{35}	2^{35}	2^{50}	2^{-68}	$\approx 2^{-119}$	$\approx 2^{-152}$
2^{60}	2^{35}	2^{45}	2^{50}	2^{-68}	$\approx 2^{-109}$	$\approx 2^{-151}$
2^{60}	2^{35}	2^{55}	2^{50}	2^{-68}	$\approx 2^{-99}$	$\approx 2^{-133}$
2^{80}	2^{25}	2^{35}	2^{70}	2^{-48}	$\approx 2^{-79}$	$\approx 2^{-112}$
2^{80}	2^{25}	2^{45}	2^{70}	2^{-48}	$\approx 2^{-69}$	$\approx 2^{-112}$
2^{80}	2^{25}	2^{55}	2^{70}	2^{-48}	$\approx 2^{-59}$	$\approx 2^{-112}$
2^{80}	2^{35}	2^{35}	2^{70}	2^{-48}	$\approx 2^{-79}$	$\approx 2^{-112}$
2^{80}	2^{35}	2^{45}	2^{70}	2^{-48}	$\approx 2^{-69}$	$\approx 2^{-112}$
2^{80}	2^{35}	2^{55}	2^{70}	2^{-48}	$\approx 2^{-59}$	$\approx 2^{-112}$

Table 3.2. Concrete advantages of a key exchange adversary with given resources t (running time), $\#N$ (number of pre-shared keys), $\#S$ (number of sessions), and $\#RO$ (number of random oracle queries) in breaking the security of the TLS 1.3 PSK-only handshake protocol with a ciphersuite targeting 128-bit security. Numbers based on the prior bounds by Dowling et al. [92] and our bound for PSK-only in Corollary 2. "Target" indicates the maximal advantage $t/2^b$ tolerable for a given bound on t when aiming for the bit security level $b = 128$; entries in green-shaded cells meet that target. We assume that the ciphersuite uses SHA256 as its hash function (see Appendix 3.8 for further explanation).

3.8 A Careful Discussion of Domain Separation

In our indifferentiability treatment of the TLS 1.3 key schedule (cf. Section 3.4), we change what we capture as random oracles in the key exchange model. We start with one random oracle, RO_{H}, used wherever the hash function **H** would be called in the protocol. We change this to classify queries to RO_{H} into two types:

Type 1 queries: *component hashes* (via function Ch) used within **Extract**, **Expand**, and **MAC** to compute $\mathsf{HKDF.Extract}$, $\mathsf{HKDF.Expand}$, resp. HMAC.

Type 2 queries: *transcript hashes* (via function Th) computing hash values of protocol transcripts (or empty strings).

We wish to model Ch and Th now as *two* independent random oracles: $\mathrm{RO}_{\mathsf{Ch}}$ resp. $\mathrm{RO}_{\mathsf{Th}}$.

To change the model, we can just change the pseudocode of the protocol to replace RO_{H} with whichever of $\mathrm{RO}_{\mathsf{Ch}}$ and $\mathrm{RO}_{\mathsf{Th}}$ seems more appropriate. However, we must define an explicit construction that performs this substitution in a systematic way in order to give a formal proof of security. This construction needs a Boolean condition to determine which of $\mathrm{RO}_{\mathsf{Ch}}$ and $\mathrm{RO}_{\mathsf{Th}}$

Adversary resources						PSK-(EC)DHE	
t	$\#N$	$\#S$	$\#RO$	Curve (bit security b, group order p)	Target $t/2^b$	DFGS [92]	Us (Cor. 1)
2^{40}	2^{25}	2^{35}	2^{30}	`secp256r1` ($b=128,\ p\approx 2^{256}$)	2^{-88}	$\approx 2^{-92}$	$\approx 2^{-167}$
2^{40}	2^{35}	2^{35}	2^{30}	`secp256r1` ($b=128,\ p\approx 2^{256}$)	2^{-88}	$\approx 2^{-82}$	$\approx 2^{-167}$
2^{40}	2^{25}	2^{35}	2^{30}	`x25519` ($b=128,\ p\approx 2^{252}$)	2^{-88}	$\approx 2^{-92}$	$\approx 2^{-163}$
2^{40}	2^{35}	2^{35}	2^{30}	`x25519` ($b=128,\ p\approx 2^{252}$)	2^{-88}	$\approx 2^{-82}$	$\approx 2^{-163}$
2^{40}	2^{25}	2^{35}	2^{30}	`secp384r1` ($b=192,\ p\approx 2^{384}$)	2^{-152}	$\approx 2^{-220}$	$\approx 2^{-294}$
2^{40}	2^{35}	2^{35}	2^{30}	`secp384r1` ($b=192,\ p\approx 2^{384}$)	2^{-152}	$\approx 2^{-210}$	$\approx 2^{-294}$
2^{40}	2^{25}	2^{35}	2^{30}	`x448` ($b=224,\ p\approx 2^{446}$)	2^{-184}	$\approx 2^{-220}$	$\approx 2^{-301}$
2^{40}	2^{35}	2^{35}	2^{30}	`x448` ($b=224,\ p\approx 2^{446}$)	2^{-184}	$\approx 2^{-210}$	$\approx 2^{-301}$
2^{40}	2^{25}	2^{35}	2^{30}	`secp521r1` ($b=256,\ p\approx 2^{521}$)	2^{-216}	$\approx 2^{-220}$	$\approx 2^{-301}$
2^{40}	2^{35}	2^{35}	2^{30}	`secp521r1` ($b=256,\ p\approx 2^{521}$)	2^{-216}	$\approx 2^{-210}$	$\approx 2^{-301}$
2^{60}	2^{25}	2^{35}	2^{50}	`secp256r1` ($b=128,\ p\approx 2^{256}$)	2^{-68}	$\approx 2^{-61}$	$\approx 2^{-132}$
2^{60}	2^{25}	2^{45}	2^{50}	`secp256r1` ($b=128,\ p\approx 2^{256}$)	2^{-68}	$\approx 2^{-40}$	$\approx 2^{-132}$
2^{60}	2^{25}	2^{55}	2^{50}	`secp256r1` ($b=128,\ p\approx 2^{256}$)	2^{-68}	$\approx 2^{-12}$	$\approx 2^{-127}$
2^{60}	2^{35}	2^{35}	2^{50}	`secp256r1` ($b=128,\ p\approx 2^{256}$)	2^{-68}	$\approx 2^{-60}$	$\approx 2^{-132}$
2^{60}	2^{35}	2^{45}	2^{50}	`secp256r1` ($b=128,\ p\approx 2^{256}$)	2^{-68}	$\approx 2^{-32}$	$\approx 2^{-132}$
2^{60}	2^{35}	2^{55}	2^{50}	`secp256r1` ($b=128,\ p\approx 2^{256}$)	2^{-68}	$\approx 2^{-2}$	$\approx 2^{-127}$
2^{60}	2^{25}	2^{35}	2^{50}	`x25519` ($b=128,\ p\approx 2^{252}$)	2^{-68}	$\approx 2^{-57}$	$\approx 2^{-128}$
2^{60}	2^{25}	2^{45}	2^{50}	`x25519` ($b=128,\ p\approx 2^{252}$)	2^{-68}	$\approx 2^{-37}$	$\approx 2^{-128}$
2^{60}	2^{25}	2^{55}	2^{50}	`x25519` ($b=128,\ p\approx 2^{252}$)	2^{-68}	$\approx 2^{-12}$	$\approx 2^{-123}$
2^{60}	2^{35}	2^{35}	2^{50}	`x25519` ($b=128,\ p\approx 2^{252}$)	2^{-68}	$\approx 2^{-57}$	$\approx 2^{-128}$
2^{60}	2^{35}	2^{45}	2^{50}	`x25519` ($b=128,\ p\approx 2^{252}$)	2^{-68}	$\approx 2^{-32}$	$\approx 2^{-128}$
2^{60}	2^{35}	2^{55}	2^{50}	`x25519` ($b=128,\ p\approx 2^{252}$)	2^{-68}	$\approx 2^{-2}$	$\approx 2^{-123}$
2^{60}	2^{25}	2^{35}	2^{50}	`secp384r1` ($b=192,\ p\approx 2^{384}$)	2^{-132}	$\approx 2^{-189}$	$\approx 2^{-259}$
2^{60}	2^{25}	2^{45}	2^{50}	`secp384r1` ($b=192,\ p\approx 2^{384}$)	2^{-132}	$\approx 2^{-168}$	$\approx 2^{-259}$
2^{60}	2^{25}	2^{55}	2^{50}	`secp384r1` ($b=192,\ p\approx 2^{384}$)	2^{-132}	$\approx 2^{-140}$	$\approx 2^{-254}$
2^{60}	2^{35}	2^{35}	2^{50}	`secp384r1` ($b=192,\ p\approx 2^{384}$)	2^{-132}	$\approx 2^{-188}$	$\approx 2^{-259}$
2^{60}	2^{35}	2^{45}	2^{50}	`secp384r1` ($b=192,\ p\approx 2^{384}$)	2^{-132}	$\approx 2^{-160}$	$\approx 2^{-259}$
2^{60}	2^{35}	2^{55}	2^{50}	`secp384r1` ($b=192,\ p\approx 2^{384}$)	2^{-132}	$\approx 2^{-130}$	$\approx 2^{-254}$
2^{60}	2^{25}	2^{35}	2^{50}	`x448` ($b=224,\ p\approx 2^{446}$)	2^{-164}	$\approx 2^{-200}$	$\approx 2^{-280}$
2^{60}	2^{25}	2^{45}	2^{50}	`x448` ($b=224,\ p\approx 2^{446}$)	2^{-164}	$\approx 2^{-170}$	$\approx 2^{-279}$
2^{60}	2^{25}	2^{55}	2^{50}	`x448` ($b=224,\ p\approx 2^{446}$)	2^{-164}	$\approx 2^{-140}$	$\approx 2^{-261}$
2^{60}	2^{35}	2^{35}	2^{50}	`x448` ($b=224,\ p\approx 2^{446}$)	2^{-164}	$\approx 2^{-190}$	$\approx 2^{-280}$
2^{60}	2^{35}	2^{45}	2^{50}	`x448` ($b=224,\ p\approx 2^{446}$)	2^{-164}	$\approx 2^{-160}$	$\approx 2^{-279}$
2^{60}	2^{35}	2^{55}	2^{50}	`x448` ($b=224,\ p\approx 2^{446}$)	2^{-164}	$\approx 2^{-130}$	$\approx 2^{-261}$
2^{60}	2^{25}	2^{35}	2^{50}	`secp521r1` ($b=256,\ p\approx 2^{521}$)	2^{-196}	$\approx 2^{-200}$	$\approx 2^{-280}$
2^{60}	2^{25}	2^{45}	2^{50}	`secp521r1` ($b=256,\ p\approx 2^{521}$)	2^{-196}	$\approx 2^{-170}$	$\approx 2^{-279}$
2^{60}	2^{25}	2^{55}	2^{50}	`secp521r1` ($b=256,\ p\approx 2^{521}$)	2^{-196}	$\approx 2^{-140}$	$\approx 2^{-261}$
2^{60}	2^{35}	2^{35}	2^{50}	`secp521r1` ($b=256,\ p\approx 2^{521}$)	2^{-196}	$\approx 2^{-190}$	$\approx 2^{-280}$
2^{60}	2^{35}	2^{45}	2^{50}	`secp521r1` ($b=256,\ p\approx 2^{521}$)	2^{-196}	$\approx 2^{-160}$	$\approx 2^{-279}$
2^{60}	2^{35}	2^{55}	2^{50}	`secp521r1` ($b=256,\ p\approx 2^{521}$)	2^{-196}	$\approx 2^{-130}$	$\approx 2^{-261}$
2^{80}	2^{25}	2^{35}	2^{70}	`secp256r1` ($b=128,\ p\approx 2^{256}$)	2^{-48}	$\approx 2^{-21}$	$\approx 2^{-92}$
2^{80}	2^{25}	2^{45}	2^{70}	`secp256r1` ($b=128,\ p\approx 2^{256}$)	2^{-48}	$\approx 2^{-1}$	$\approx 2^{-92}$
2^{80}	2^{25}	2^{55}	2^{70}	`secp256r1` ($b=128,\ p\approx 2^{256}$)	2^{-48}	$\approx 2^{19}$	$\approx 2^{-92}$
2^{80}	2^{35}	2^{35}	2^{70}	`secp256r1` ($b=128,\ p\approx 2^{256}$)	2^{-48}	$\approx 2^{-21}$	$\approx 2^{-92}$
2^{80}	2^{35}	2^{45}	2^{70}	`secp256r1` ($b=128,\ p\approx 2^{256}$)	2^{-48}	$\approx 2^{-1}$	$\approx 2^{-92}$
2^{80}	2^{35}	2^{55}	2^{70}	`secp256r1` ($b=128,\ p\approx 2^{256}$)	2^{-48}	$\approx 2^{20}$	$\approx 2^{-92}$
2^{80}	2^{25}	2^{35}	2^{70}	`x25519` ($b=128,\ p\approx 2^{252}$)	2^{-48}	$\approx 2^{-17}$	$\approx 2^{-88}$
2^{80}	2^{25}	2^{45}	2^{70}	`x25519` ($b=128,\ p\approx 2^{252}$)	2^{-48}	$\approx 2^{3}$	$\approx 2^{-88}$
2^{80}	2^{25}	2^{55}	2^{70}	`x25519` ($b=128,\ p\approx 2^{252}$)	2^{-48}	$\approx 2^{23}$	$\approx 2^{-88}$
2^{80}	2^{35}	2^{35}	2^{70}	`x25519` ($b=128,\ p\approx 2^{252}$)	2^{-48}	$\approx 2^{-17}$	$\approx 2^{-88}$
2^{80}	2^{35}	2^{45}	2^{70}	`x25519` ($b=128,\ p\approx 2^{252}$)	2^{-48}	$\approx 2^{3}$	$\approx 2^{-88}$
2^{80}	2^{35}	2^{55}	2^{70}	`x25519` ($b=128,\ p\approx 2^{252}$)	2^{-48}	$\approx 2^{23}$	$\approx 2^{-88}$
2^{80}	2^{25}	2^{35}	2^{70}	`secp384r1` ($b=192,\ p\approx 2^{384}$)	2^{-112}	$\approx 2^{-149}$	$\approx 2^{-219}$
2^{80}	2^{25}	2^{45}	2^{70}	`secp384r1` ($b=192,\ p\approx 2^{384}$)	2^{-112}	$\approx 2^{-129}$	$\approx 2^{-219}$
2^{80}	2^{25}	2^{55}	2^{70}	`secp384r1` ($b=192,\ p\approx 2^{384}$)	2^{-112}	$\approx 2^{-109}$	$\approx 2^{-219}$
2^{80}	2^{35}	2^{35}	2^{70}	`secp384r1` ($b=192,\ p\approx 2^{384}$)	2^{-112}	$\approx 2^{-149}$	$\approx 2^{-219}$
2^{80}	2^{35}	2^{45}	2^{70}	`secp384r1` ($b=192,\ p\approx 2^{384}$)	2^{-112}	$\approx 2^{-129}$	$\approx 2^{-219}$
2^{80}	2^{35}	2^{55}	2^{70}	`secp384r1` ($b=192,\ p\approx 2^{384}$)	2^{-112}	$\approx 2^{-108}$	$\approx 2^{-219}$
2^{80}	2^{25}	2^{35}	2^{70}	`x448` ($b=224,\ p\approx 2^{446}$)	2^{-144}	$\approx 2^{-180}$	$\approx 2^{-240}$
2^{80}	2^{25}	2^{45}	2^{70}	`x448` ($b=224,\ p\approx 2^{446}$)	2^{-144}	$\approx 2^{-150}$	$\approx 2^{-240}$
2^{80}	2^{25}	2^{55}	2^{70}	`x448` ($b=224,\ p\approx 2^{446}$)	2^{-144}	$\approx 2^{-120}$	$\approx 2^{-240}$
2^{80}	2^{35}	2^{35}	2^{70}	`x448` ($b=224,\ p\approx 2^{446}$)	2^{-144}	$\approx 2^{-170}$	$\approx 2^{-240}$
2^{80}	2^{35}	2^{45}	2^{70}	`x448` ($b=224,\ p\approx 2^{446}$)	2^{-144}	$\approx 2^{-140}$	$\approx 2^{-240}$
2^{80}	2^{35}	2^{55}	2^{70}	`x448` ($b=224,\ p\approx 2^{446}$)	2^{-144}	$\approx 2^{-110}$	$\approx 2^{-240}$
2^{80}	2^{25}	2^{35}	2^{70}	`secp521r1` ($b=256,\ p\approx 2^{521}$)	2^{-176}	$\approx 2^{-180}$	$\approx 2^{-240}$
2^{80}	2^{25}	2^{45}	2^{70}	`secp521r1` ($b=256,\ p\approx 2^{521}$)	2^{-176}	$\approx 2^{-150}$	$\approx 2^{-240}$
2^{80}	2^{25}	2^{55}	2^{70}	`secp521r1` ($b=256,\ p\approx 2^{521}$)	2^{-176}	$\approx 2^{-120}$	$\approx 2^{-240}$
2^{80}	2^{35}	2^{35}	2^{70}	`secp521r1` ($b=256,\ p\approx 2^{521}$)	2^{-176}	$\approx 2^{-170}$	$\approx 2^{-240}$
2^{80}	2^{35}	2^{45}	2^{70}	`secp521r1` ($b=256,\ p\approx 2^{521}$)	2^{-176}	$\approx 2^{-140}$	$\approx 2^{-240}$
2^{80}	2^{35}	2^{55}	2^{70}	`secp521r1` ($b=256,\ p\approx 2^{521}$)	2^{-176}	$\approx 2^{-110}$	$\approx 2^{-240}$

Table 3.3. Concrete advantages of a key exchange adversary with given resources t (running time), $\#N$ (number of pre-shared keys), $\#S$ (number of sessions), and $\#RO$ (number of random oracle queries) in breaking the security of the TLS 1.3 PSK-(EC)DH handshake protocol. Numbers based on the prior bounds by Dowling et al. [92] and our bound for PSK-(EC)DHE in Corollary 1. "Target" indicates the maximal advantage $t/2^b$ tolerable for a given bound on t when aiming for the respective curve's bit security level b; entries in green-shaded cells meet that target. See Section 3.7 and Appendix 3.7.1 for further details.

should be queried, and this condition cannot be dependent on the higher-level context of the protocol's usage. Instead, we must define two disjoint sets $\mathsf{Dom}_{\mathsf{Ch}}$ and $\mathsf{Dom}_{\mathsf{Th}}$ such that honest executions of TLS 1.3 only query RO_{H} on inputs in $\mathsf{Dom}_{\mathsf{Ch}}$ when computing $\mathsf{HKDF.Extract}$, $\mathsf{HKDF.Expand}$, or HMAC, and it otherwise only queries RO_{H} on inputs in $\mathsf{Dom}_{\mathsf{Th}}$.

This separation must hold even when an honest session is responding to adversarially-chosen messages. We do make some assumptions about the way that honest sessions process incoming messages. We assume that a server receiving a first `ClientHello` message from a client will not respond or execute the protocol unless the message contains correct encodings of all of the mandatory parameters for TLS 1.3. If the client fails to specify a valid group and key share in PSK-(EC)DHE mode, or version number, mode, and pre-shared key in any mode, the server should abort. Of course, the `ClientHello` message may also contain invalid encodings of these values or even arbitrary data; we do not exclude this possibility. Note that our conditions apply only to random-oracle queries made by honest executions of the protocol. An adversary may of course call RO_{H} on any input it chooses in either $\mathsf{Dom}_{\mathsf{Ch}}$ or $\mathsf{Dom}_{\mathsf{Th}}$.

The TLS 1.3 handshake protocol does not provide any intentional domain separation between Type 1 and Type 2 queries. We therefore turn to the formatting of queries to RO_{H} in the hopes of finding some unintentional separation. We identify seven subtypes of query: five subtypes of Type 1 and two subtypes of Type 2. Queries of each subtype have some unique formatting: a fixed length, a byte with a particular value, an encoded label. These attributes are heavily dependent on the specific configuration of the TLS 1.3 protocol; we therefore analyze four separate cases: two modes of operation (PSK-(EC)DHE and PSK-only mode) and two ciphersuites defining RO_{H} as `SHA256` and `SHA384` respectively. Throughout, we will assume that any pre-shared-keys are the same length as the output length of RO_{H}, i.e., hl bits. This is true of resumption keys, but may not be true in general for pre-shared keys negotiated out-of-band. As TLS 1.3 fields length are given in (full) *bytes*, we will be talking about *byte lengths* if not otherwise stated in the following and use the shorthand $Hl := hl/8$ for the output length of RO_{H} in *bytes*. We also assume that if a Diffie–Hellman group is used, it is one of the standardized elliptic curve or finite field groups.

All Type 1 queries to RO_{H} are intermediate steps in the computation of HMAC, HKDF.Extract, and HKDF.Expand. They consequently share some formatting which we discuss here before addressing each subtype individually. HKDF.Extract and HMAC are two names for the same function. Given a key K and input s, $\mathsf{HKDF.Expand}(K,s)$ pads s with a single trailing counter byte with value `0x01`, then returns $\mathsf{HMAC}(K,s|||\mathtt{0x01})$. Therefore all Type 1 queries to RO_{H} arise in the computation of HMAC. $\mathsf{HMAC}[\mathrm{RO}_{\mathsf{H}}](K,s)$ takes a key K of length Hl bytes. It then pads this key with zeroes up to the block length Bl of its hash function. The block lengths of `SHA256` and `SHA384` are 64 and 128 bytes respectively. We call the padded key K'. Then $\mathsf{HMAC}[\mathrm{RO}_{\mathsf{H}}]$ makes two queries to RO_{H}:

1. $d \leftarrow \mathrm{RO}_{\mathsf{H}}(K' \oplus \mathsf{ipad}||s)$

2. $\mathrm{RO}_{\mathsf{H}}(K' \oplus \mathsf{opad}||d)$

ipad and opad are strings of Bl bytes. Each byte in ipad is fixed to `0x36`, and each byte in opad is fixed to `0x5c`. The padded key K' is Bl long, longer than K, so every Type 1 query has a segment of length $Bl - Hl$ bytes in which each byte equals one of `0x36` and `0x5c`.

Now we can present the seven subtypes of queries made by TLS 1.3. The first five types are Type 1 queries, and the last two (Empty and Transcript) are Type 2 queries.

The seven subtypes of queries are:

1. **Outer HMAC queries.** These queries are the second query made in the computation of HMAC. Its key has length Hl, and the digest d also has length Hl. In between these is a segment containing $Bl - Hl$ bytes `0x5c`. We will often refer to this segment as the "fixed region". When the hash function is `SHA256`, resp. `SHA384`, the fixed region is 32, resp 80 bytes long. The total query is 96, resp. 176 bytes long.

2. **Inner HMAC queries.** We divide the first RO_{H} query made by HMAC into several subtypes; this type includes only those where the input to HMAC is an arbitrary string of length Hl. This subtype is formatted identically to an outer HMAC query, except that the bytes of the fixed region are fixed to the value `0x36` instead of `0x5c`. TLS 1.3 makes inner HMAC queries while computing `Finished` and *binder* messages (where the input is a hashed transcript), the early and master secrets, and in PSK-only mode, also the handshake secret.

3. **Diffie–Hellman HMAC query.** In PSK-(EC)DHE mode, TLS 1.3 computes the handshake secret by calling HMAC on an encoded Diffie–Hellman key share. HMAC's first query is a Diffie–Hellman HMAC query. The formatting is the same as an inner HMAC hash except that the segment following the fixed region has a different length. The byte lengths ($|\mathbb{G}|/8$) of the encodings for each standardized Diffie–Hellman group can be found in Table 3.12.

4. **`Derive-Secret` hashes.** The `Derive-Secret` function is a component of the TLS key schedule [186, Section 7.1]. Its inputs are a key of length Hl, a label string of 2 to 12-bytes in length, and an input `Messages` string.

   `Derive-Secret` queries RO_{H} three times: once to hash the `Messages` string, and twice as part of HKDF.Expand. The first of these three queries is a transcript query, and the third is an Outer HMAC query. The second query we call a `Derive-Secret` query. The `Derive-Secret` query has the same formatting as Inner HMAC queries and Diffie–Hellman queries, but the segment following the fixed region contains a strictly formatted `HkdfLabel` struct [186, Section 7.1].

   This struct begins with a two-byte field encoding the integer value Hl. This is followed by a variable-length vector with a 1-byte length field containing the string `"tls13 "` followed by a label string with length between 2 and 12 bytes. Lastly comes a vector of length Hl, prefixed with a 1-byte field encoding its length. The last byte in the input contains the `0x01`. This byte is the counter mandated by the definition of HKDF.Expand; however since HKDF.Expand is never called on inputs longer than Hl, the counter never reaches a value higher than 1.

   The total length of a the label struct, including the counter byte, is at least $Hl + 13$ bytes and at most $Hl + 23$ bytes.

5. **`Finished` hash.** The `HKDF-Expand-Label` function is a subroutine of the `Derive-Secret` function, but also called during the computation of `Finished` messages and the *binder* value [186, Section 4.4.4]. `HKDF-Expand-Label` makes two calls to RO_{H}. The second is an Outer HMAC hash; we call the first a `Finished` hash. A `Finished` hash is identical to a `Derive-Secret` hash, except that the label string is fixed to `finished` and the final vector

has length 0. The counter byte is still present. In total, the label struct occupies 19 bytes.

6. **Empty hashes.** Occasionally in the key schedule, TLS 1.3 calls RO_{H} on the empty string.

7. **Transcript hashes.** The last use of RO_{H} is to condense partial transcripts. Each transcript includes at least a partial `ClientHello` message. We assume calling RO_{H}. on a transcript which includes at least a partial `ClientHello`. The minimum length of a partial `ClientHello` message in PSK-only mode is 69 bytes. This includes the following fields [186, Section 4.1.2]:

   - 2 bytes `legacy_version` fixed to `0x0303`
   - 32 bytes `random`
   - 1 byte `legacy_session_id` (for an empty vector with 1-byte length field)
   - 4 bytes `ciphersuites` (must include a 2-byte length field and the value, e.g., `0x1301`)
   - 2 bytes `legacy_compression_methods` (must include a 1-byte length field and the value `0x00`)
   - 2 bytes encoded length of `extensions` field
   - 7 bytes `supported_versions extension` extension [186, Section 4.2.1] (must start with `0x002b` and include `0x0304`)
   - 6 bytes `psk_key_exchange_modes` extension [186, Section 4.2.9] (must start with `0x002d` and include `0x00`)
   - 9 bytes `pre_shared_key` extension [186, Section 4.2.11] (partial: excluding the binder list; must come last, must start with `0x0029`)

   The first 43 bytes (through the `extensions`' length encoding), must appear in the order displayed, although the `legacy_session_id`, `ciphersuites`, and `legacy_compression_methods` fields can be longer than the lengths given above. We will occasionally refer to this segment as the "fixed preface" of a `ClientHello` because it must appear at the beginning of every well-formed `ClientHello` message. The extensions can be reordered arbitrarily (except for the `pre_shared_key` extension) and additional extensions and ciphersuites can be added or repeated, up to a maximum length of $2^{16}-1$ bytes of ciphersuites and $2^{16}-2$ bytes for

Type	Minimum length (bytes)	Maximum length (bytes)
Outer HMAC	96	96
Inner HMAC	96	96
`Derive-Secret`	109	119
`Finished`	83	83
Empty	0	0
Transcript	69	$2^{32}+324$

Table 3.4. Table showing input lengths for hash function calls made by TLS 1.3 in PSK-only mode with `SHA256`.

extensions. The overall maximum length of a `ClientHello` is then $2^{32}+289$ bytes. A full `ClientHello` in PSK-only mode, including the binder list, adds at least another $3+Hl$ bytes for a `binder` vector with a 3 bytes of encoded length. The `ClientHello` message thus contains a minimum of $72+Hl$ bytes and a maximum of $2^{32}+292+Hl$ bytes.

In PSK-(EC)DHE mode, two additional extensions are also mandatory: the `key_share` and `supported_groups` extensions [186, Section 9.2], so the minimum `ClientHello` length increases by at least $17+|\mathbb{G}|/8$ bytes, cf. Table 3.12. This increase occurs for both truncated and full `ClientHello` messages. In this mode, a truncated `ClientHello` message is at least $86+|\mathbb{G}|/8$ bytes long, and a full `ClientHello` is at least $89+|\mathbb{G}|/8$ bytes long.

3.8.1 PSK-only mode with `SHA256`

The block length of this hash function is 64 bytes, and the output length is 32 bytes. In Table 3.4, we give the minimum and maximum input lengths for each of the six call types. (Diffie–Hellman HMAC calls do not occur in this mode.)

In Table 3.4 we note the minimum and maximum input lengths of each type of message. For those types with overlapping length ranges, we must show they have separate domains by other means. Outer and Inner HMAC hashes have identical lengths; however each of them has a 32-byte fixed region. In outer HMAC hashes, the fixed region contains opad; in inner HMAC hashes, it contains ipad. These are distinct values, so no string can be both an outer and an inner HMAC hash.

Transcript hashes are not domain-separated by length from any hash except the empty hashes. We therefore turn to formatting to separate these from other types. In the following, we visually lay out each byte of potentially overlapping inputs.

For a string to be both a transcript and an HMAC hash (outer or inner), it must be 96 bytes (cf. Table 3.4) long. We diagram and compare a transcript hash containing a partial ClientHello[15] and an HMAC hash (outer or inner) in Figure 3.13.

We can see that the fixed preface of the transcript hash overlaps the fixed region of the HMAC hash that is fixed to either ipad or opad. Consequently, the legacy_session_id vector must begin within the fixed region (at byte 35). This is a variable-length vector preceded by a 1-byte length field, and its maximum length is 32 bytes [186, Section 4.1.2]. Therefore the maximum value of the length field is 0x20 and it cannot contain either byte 0x36 or 0x5c. Any string containing a valid partial ClientHello therefore cannot also be a correctly formatted HMAC hash.

The same argument applies to Finished and Derive-Secret hashes, both of which contain the same fixed region in the same location as inner HMAC hashes.

For this mode, we define the set $\mathsf{Dom}_{\mathsf{Th}}$ to include of the empty string and all strings of length greater than or equal to 69 bytes for which the 35th byte is not equal to ipad or opad. We let $\mathsf{Dom}_{\mathsf{Ch}}$ contain all other elements of $\{0,1\}^*$.

3.8.2 Pre-shared key with Diffie–Hellmann mode with SHA256

Again, we present the minimum and maximum lengths of each hash type; see Table 3.5. We now include Diffie–Hellman HMAC hashes, and transcript hashes include additional mandatory extensions for PSK-(EC)DHE mode.

In this mode, Diffie–Hellman HMAC hashes may collide with Inner HMAC or Derive-Secret hashes for certain choices of $\mathbb{G}$. This is not a failure of domain separation because these inputs to these three types will all belong to $\mathsf{Dom}_{\mathsf{Ch}}$. Transcript hashes now only have length overlaps with Diffie–Hellman HMAC and Derive-Secret hashes. In both cases, however, the same argument about the 35th byte containing the length of legacy_session_id applies, and no string can be two different types.

For this mode, the set $\mathsf{Dom}_{\mathsf{Th}}$ consists of the empty string and all strings of length greater

[15] A full ClientHello contains at least $72 + Hl \geq 104$ bytes, which is too long to be an HMAC hash.

```
$\mathsf{KE}_1.\mathsf{Run}(u, \pi_u^i, psk, m)$:
$keys \leftarrow (\pi_u^i.skey[stage]$ for $stage \in \mathsf{K}_{\mathsf{Transform}})$
$acc \leftarrow (\pi_u^i.\mathsf{t}_{\mathsf{acc}}[stage] \neq \infty$ for $stage$ in $[1 \ldots \mathsf{STAGES}])$
$\tilde{m} \leftarrow \mathsf{Transform}_{\mathsf{Recv}}(keys, \pi_u^i.role, acc, m)$
$(\pi_u^i, \tilde{m}') \leftarrow \mathsf{KE}_2.\mathsf{Run}(u, \pi_u^i, psk, \tilde{m})$
$keys \leftarrow (\pi_u^i.skey[stage]$ for $stage \in \mathsf{K}_{\mathsf{Transform}})$
$acc \leftarrow (\pi_u^i.\mathsf{t}_{\mathsf{acc}}[stage] \neq \infty$ for $stage$ in $[1 \ldots \mathsf{STAGES}])$
$m' \leftarrow \mathsf{Transform}_{\mathsf{Send}}(keys, \pi_u^i.role, acc, \tilde{m}')$
return $(\pi_u^i, m')$
```

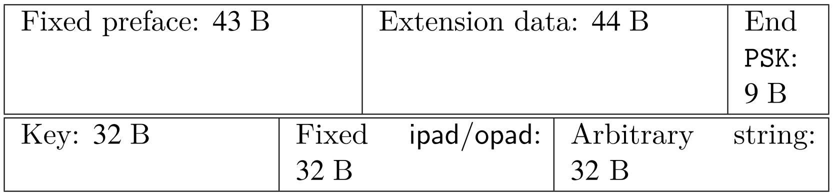
Figure 3.11. Key exchange KE_1 built by transforming protocol messages of KE_2.

Group name	`NamedGroup` enum value	Encoding	length
		$\lvert\mathbb{G}\rvert/8$	
secp256r1 [173]	0x0017	32	
secp384r1 [173]	0x0018	48	
secp521r1 [173]	0x0019	66	
x25519 [150]	0x001d	32	
x448 [150]	0x001E	56	
ffdhe2048 [107]	0x0100	128	
ffdhe3072 [107]	0x0101	192	
ffdhe4096 [107]	0x0102	256	
ffdhe6144 [107]	0x0103	384	
ffdhe8192 [107]	0x0104	512	

Figure 3.12. 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.

Type	Minimum length (bytes)	Maximum length (bytes)
Outer HMAC	96	96
Inner HMAC	96	96
Diffie–Hellman HMAC	$64 + \lvert\mathbb{G}\rvert/8$	$64 + \lvert\mathbb{G}\rvert/8$
`Derive-Secret`	109	119
`Finished`	83	83
Empty	0	0
Transcript	$86 + \lvert\mathbb{G}\rvert/8$	$2^{32} + 324$

Table 3.5. Table showing input lengths for hash function calls made by TLS 1.3 in PSK-(EC)DHE mode with `SHA256`. For transcript hashes, the encoding lengths $\lvert\mathbb{G}\rvert/8$ can be found in Table 3.12.



Fixed preface: 43 B	Extension data: 44 B	End PSK: 9 B

Key: 32 B	Fixed ipad/opad: 32 B	Arbitrary string: 32 B

Figure 3.13. Domain separation in PSK-only mode with `SHA256`: Transcript hash containing a partial `ClientHello` (top) vs. (outer or inner) HMAC hash (bottom). “End `PSK`” is the end of the `pre_shared_key` extension.

Type	Minimum length (bytes)	Maximum length (bytes)
Outer HMAC	176	176
Inner HMAC	176	176
Diffie–Hellman HMAC	$128+\lvert\mathbb{G}\rvert/8$	$128+\lvert\mathbb{G}\rvert/8$
`Derive-Secret`	189	199
`Finished`	147	147
Empty	0	0
Transcript	$86+\lvert\mathbb{G}\rvert/8$	$2^{32}+324$

Table 3.6. Table showing input lengths for hash function calls made by TLS 1.3 in PSK-(EC)DHE mode with `SHA384`.

than or equal to $86+|\mathbb{G}|$ bytes for which the 35th byte is not equal to ipad or opad. $\mathsf{Dom}_{\mathsf{Ch}}$ contains all other elements of $\{0,1\}^*$.

3.8.3 Pre-shared key with Diffie–Hellmann mode with SHA384

Table 3.6 shows the minimum and maximum lengths of each hash type for this configuration. The hash function `SHA384` has 48-byte output and 128-byte block length, so the fixed region in HMAC, `Finished`, and `Derive-Secret` hashes will be 80 bytes long.

Because 48 byte HMAC keys are longer than the 43 byte fixed preface of a `ClientHello`, we cannot rely on the distinction between `legacy_session_id` and the fixed region for domain separation. Instead, we consider whether a minimum-length `ClientHello` can accommodate the mandatory extensions for this mode.

We worry only about possible collisions between transcript hashes and the other types: `Finished`, HMAC, and `Derive-Secret`. We diagram a transcript hash of 176 bytes together with an outer HMAC hash as a demonstration of the domain-separation argument in Figure 3.14, but the same argument applies to all.

There are no obvious conflicts here: the fixed preface of a `ClientHello` message is covered

Fixed preface: 43 B	Extension data: 124 B	End PSK: 9 B

Key: 48 B	Fixed region (`opad`): 80 B	Arbitrary string: 48 B

Figure 3.14. Domain separation in PSK-(EC)DHE mode with `SHA384`: Transcript hash of 176 bytes (top) vs. outer HMAC hash (bottom). "End `PSK`" is the end of the `pre_shared_key` extension.

by the key section of the HMAC hash, and the `pre_shared_key` extension is covered by the arbitrary string at the end. However, notice that of the 124 bytes available for extension data in the `ClientHello`, 80 of them must be fixed to `opad` to allow a collision. Even including the 5 bytes immediately after the fixed preface and 9 bytes reservedf or the `pre_shared_key` extension, this leaves only 58 bytes. In PSK-(EC)DHE mode, five extensions are mandatory even for truncated `ClientHello` messages. They are `supported_versions` [186, Section 4.2.1] (minimum 7 bytes), `supported_groups` [186, Section 4.2.7] (minimum 7 bytes), `key_share` [186, Section 4.2.8] (minimum $16 + |\mathbb{G}|/8$ bytes), `psk_key_exchange_modes` [186, Section 4.2.9] (minimum 6 bytes), and `pre_shared_key` [186, Section 4.2.11] (minimum 13 bytes). Even for the smallest choice of $\mathbb{G}$, at least 71 bytes are required to contain these extensions. At least one of the extensions must overlap with the fixed field, and will differ from `opad` in at least one byte.

Any valid transcript hash will need at least $92 + |\mathbb{G}|/8$ bytes outside the fixed region: 43 bytes for the preface and $49 + |\mathbb{G}|/8$ for the mandatory extensions. An outer HMAC hash has only 124 unfixed bytes and cannot meet this threshold. This is true also for inner HMAC hashes (96 unfixed bytes), and Diffie–Hellman HMAC hashes, which have $48 + |\mathbb{G}|/8$ unfixed bytes. It is true for `Finished` hashes, which have 48 unfixed bytes. And it is true for `Derive-Secret` hashes, which have at most 119 unfixed bytes.

Let us be even more clear about why this overlap means no collision is possible. We cannot fit all of the extensions in the $48 + |\mathbb{G}|$ bytes after the fixed region. Therefore one of the extensions must start either in the fixed region, or before the fixed region. None of these extensions can start in the fixed region because they all begin with an extension type different from `ipad` or `opad`. Therefore one of them must start before the fixed region and continue into the fixed region. We call this the "first extension". The `pre_shared_key` extension must be the last extension, so it cannot be the first extension. Therefore the first extension is one of `key_share`, `supported_groups`, and `psk_key_exchange_modes`, and `supported_versions`.

All extensions start with a 4 byte encoding of their type and length. This means that the first extension may contain only one arbitrary byte of data before 80 bytes of ipad or opad. All four possible extensions consist of variable-length vectors. TLS encodes all variable-length vectors with a 1 or 2 byte prefix encoding their length. Consequently, the entries of the vector fall in or after the fixed region.

Each of the vector entries in the four possible first extensions begins with an element from an enum: either the `NamedGroup`, `ProtocolVersion`, or `PskKeyExchangeMode` enums. Luckily, none of these enums contain the bytes `0x36` or `0x5c`. To demonstrate this, we present the `NamedGroup` values in Table 3.12 [186, Section 4.2.7]. The `ProtocolVersion` encoding for TLS 1.3 is `0x0304` [186, Section 4.2.1], and the elements of the `PskKeyExchangeMode` enums are `0x00`, `0x01`, and `0xff` [186, Section 4.2.9]. Of course, a `ClientHello` message can contain badly formed extensions. We assume, however, that each of the mandatory extensions must contain one correctly formatted vector entry. Without these entries, communication partners will not be able to select the correct version, group, or mode to execute the protocol; we assume that in this case they would abort. Because the fixed region contains no valid enum elements, this correctly formatted vector entry must begin after the fixed region. Therefore the first extension uses at most 1 byte of the fixed region to encode meaningful data (a possible second byte of the vector length encoding). The mandatory extensions must occupy no more than 5 bytes before the fixed region, 1 byte in the fixed region, and either 71 bytes after the fixed region (for the longest possible `Derive-Secret` hash) or $|\mathbb{G}|/8$ bytes after (for an inner HMAC hash). But summing their minimum lengths gives $49 + |\mathbb{G}|/8$ bytes. Even for the smallest possible $|\mathbb{G}|/8 = 32$, the extensions just do not fit in the given space. It is therefore impossible to construct a valid `ClientHello` message, truncated or otherwise, that collides with a possible HMAC, `Derive-Secret`, or `Finished` hash.

Consequently we can set $\mathsf{Dom}_{\mathsf{Th}}$ to contain the empty string and all strings of at least 86 bytes for which at least one of bytes 48 through 128 does not equal either ipad or opad. As usual, we set $\mathsf{Dom}_{\mathsf{Ch}}$ to be all other elements of $\{0,1\}^*$.

3.8.4 PSK-only mode with `SHA384`

In this mode/hash function combination, the transcript hash can collide with outer HMAC hashes. There are other collisions as well, but one is sufficient to demonstrate the lack of domain separation. We illustrate this via a 176-byte transcript hash (containing a truncated `ClientHello`) and an outer HMAC hash, shown in Figure 3.15.

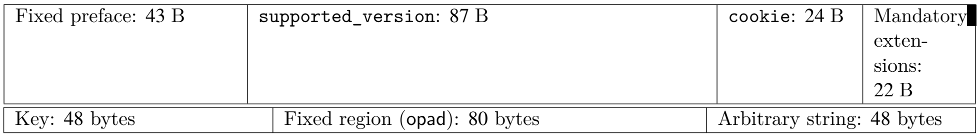

Fixed preface: 43 B
supported_version: 87 B
cookie: 24 B
Mandatory exten- sions: 22 B
Key: 48 bytes
Fixed region (opad): 80 bytes
Arbitrary string: 48 bytes

Figure 3.15. Failing domain separation in PSK-only mode with `SHA384`: Transcript hash of 176 bytes, containing a truncated `ClientHello`, (top) vs. outer HMAC hash (bottom). “End PSK” is the end of the `pre_shared_key` extension.

We construct the following message, which is both a truncated `ClientHello` (and therefore a transcript hash) and an outer HMAC hash. We let the first extension be the `supported_versions` extension. This extension will contain 41 protocol versions. The first 40 versions will be two bytes of `opad`: `0x5c5c`; the last will be the real version number `0x0304`. These extra version numbers match the HMAC key padding, and the real version number lies in the last 48 bytes, which are unrestricted by the formatting of an HMAC hash.

We place the remaining mandatory extensions at the end of the content section. In PSK-only mode, these are only `psk_key_exchange_modes`, and (the truncated) `pre_shared_key`, and they take up 22 bytes. This leaves 24 bytes unaccounted for between the end of the `supported_versions` extension and the start of `psk_key_exchange_modes`. We can fill these with a `cookie` [186, Section 4.2.2] extension with arbitrary content. (We can also fill these bytes without including additional extensions.)

This type of collision is unavoidable, so there are no disjoint sets $\mathsf{Dom}_{\mathsf{Th}}$ and $\mathsf{Dom}_{\mathsf{Ch}}$ that capture the way TLS 1.3 calls `SHA384` in pre-shared key only mode. Consequently the indifferentiability step of Section 3.4.1 does not apply to this mode.

3.8.5 Repairing domain separation for TLS 1.3-like protocols

The above analysis demonstrates that complete domain separation is nontrivial to achieve for a protocol like TLS 1.3 which uses a hash function for multiple purposes and at multiple levels of abstraction. We would like to present our suggestions for how this could be achieved most simply and efficiently in future iterations of TLS and other schemes. As discussed by Bellare et al. [?], the most well-known method of domain separation is the inclusion of distinct labels into each hash function call; this is precisely the method adopted by TLS 1.3 to distinguish calls to its `Derive-Secret` function. Ideally, a future scheme could specify a unique label string for each purpose: not only the various derived secrets, but also each time the transcript is hashed and each internal call made by HMAC, HKDF.Extract, and HKDF.Expand.

Unfortunately, this ideal method is not compatible with the existing specifications of HMAC and HKDF. Both of these functions make "Outer HMAC queries" as discussed above; these calls have a fixed input length of $Bl + Hl$ bytes and this input does not include a label. A protocol could avoid this roadblock by using a custom implementation of HMAC or HKDF whose underlying hash function prepends an HMAC-specific label to its input. This approach would be both standard-compliant and efficient, but we don't recommend it because existing cryptographic libraries already have trustworthy HMAC and HKDF functionality and encouraging custom implementations for every new protocol increases the probability of accidental errors in these new implementations. Instead, we suggest making no adjustments to the internal execution of HMAC or HKDF and instead altering direct hash function calls (the other six subtypes we discuss) to avoid collisions.

In practice, this means that under our recommendation, all hash function calls which are not outer HMAC queries should obey two simple rules: first, they should end with a unique label and second, that their input must not be $Bl + Hl$ long. To conform with the first rule, TLS 1.3 would need to make the following changes.

1. Add distinct labels to the end of each transcript before hashing; for clarity we suggest using the names of the last message in the transcript; i.e. "`PartialClientHello`", "`ClientHello`", "`ServerHello`", etc. If HKDF is used, we would also recommend that these labels should not end with the byte `0x01`.

2. Add distinct labels to the end of the input each time HMAC is called; this would include for Inner HMAC queries, Diffie–Hellman HMAC queries, `Finished` queries, and `Derive-Secret`

queries. Note that the labels should be postpended to the HMAC payload and not the key. The labels used by `Derive-Secret` could then be omitted, although this is not necessary.

3. Ensure that none of the labels used is a suffix of another; this can introduce collisions even if the labels are distinct.

We encourage using suffixes for domain separation, although prefixes are more commonly-used, because they are easier to use in conjunction with HMAC and HKDF. Although we are not applying labels to outer HMAC queries, we would still like to use them to domain separate inner HMAC queries (and the other subtypes). The inputs to these queries begin with the HMAC key, which undergoes an XOR operation with ipad before it is hashed. So prefixed labels would need to remain unique and prefix-free after this XOR operation; this introduces some confusion that we prefer to avoid. Additionally, the second step of our indifferentiability proof relies crucially on the fact that HMAC uses fixed-length keys shorter than Bl; prefixed labels would therefore need to share a fixed length shorter than $Bl - Hl$ bytes. With suffixes, we still need to contend with the counter byte that HKDF.Expand appends to its input, but in TLS 1.3 where this byte is always `0x01`, this presents less of a restriction.

To conform with the second rule, TLS 1.3 would need to enforce that it never hashes a string of $Bl + Hl$ except as an Outer HMAC query. The easiest and least error-prone way to do this would be to pad every non-empty hash function call and input to HMAC and HKDF with exactly $Bl + Hl$ bytes (before the suffixed labels); all calls would strictly longer than $Bl + Hl$. This method adds two additional compression function calls to each hash function execution. There are some ways to lessen this requirement without impacting the effectiveness of the length-based domain separation. Calls which already have input longer than $Bl + Hl$ bytes can omit the padding; so can calls which have strictly shorter input. It would also be possible to use only as much padding is needed to make input at least $Bl + Hl + 1$ bytes long. However, non-uniform padding should be done carefully so that, for example, two previously distinct `ClientHello` messages do not collide after being padded.

Chapter 4

Derive-then-Derandomize: Stronger Security Proofs for EdDSA Signatures

4.1 Introduction

In designing schemes, and proving them secure, theoreticians implicitly assume certain things, such as on-demand fresh randomness and correct implementation. In practice, these assumptions can fail. Weaknesses in system random-number generators are common and have catastrophic consequences. (An example relevant to this paper is the well-known key-recovery attack on Schnorr signatures when signing reuses randomness. Another striking example are Ps and Qs attacks [118, 152].) Meanwhile, implementation errors can be exploited, as shown by Bleichenbacher's attack on RSA signatures [55].

In light of this, practitioners may try to "harden" theoretical schemes before standardization and usage. A prominent and highly successful instance is EdDSA, a hardening of the Schnorr signature scheme proposed by Bernstein, Duif, Lange, Schwabe, and Yang (BDLSY) [50]. It incorporates explicit, simple key-derivation, makes signing deterministic, adds protection against sidechannel attacks via "clamping," and for simplicity confines itself to a single hash function, namely `SHA512`. The scheme is widely standardized [174, 133] and used [123].

There is however a subtle danger here, namely that the hardening attempt introduces new vulnerabilities. In other words, hardening needs to be done right; if not, it may even "soften" the scheme! Thus it is crucial that the hardened scheme be vetted via a proof of security. This is of particular importance for EdDSA given its widespread deployment. In that regard, Brendel, Cremers, Jackson and Zhao (BCJZ) [?] showed that EdDSA is secure if the Discrete-Log (DL) problem is hard and the hash function is modeled as a random oracle. This is significant as a first step but has at least two important limitations: (1) Due to the extension attack, a random oracle is not an appropriate model for the SHA512 hash function EdDSA actually uses, and (2) the reduction is so loose that there is no security guarantee for group sizes in use today.

Extrapolating EdDSA, the first part of this paper defines a general hardening transform on signature schemes called Derive-then-Derandomize (**DR**), and proves its soundness. Next we prove the indifferentiability of a general class of constructions, that we call shrink-MD; it includes the well-studied chop-MD construction [74] and also the modulo-a-prime construction arising in EdDSA. Armed with these results, the second part of the paper returns to give new proofs for EdDSA that in particular fill the above gaps. We begin with some background.

Respecting hash structure in proofs. Recall that the MD-transform [162, 77] defines a hash function $\mathsf{HH} = \mathbf{MD}[\mathsf{h}]\colon \{0,1\}^* \to \{0,1\}^{2k}$ by iterating an underlying compression function $\mathsf{h}\colon \{0,1\}^{b+2k} \to \{0,1\}^{2k}$. (See Section 4.2 for details.) SHA256 and SHA512 are obtained in this way, with (b,k) being $(512,128)$ and $(1024,256)$, respectively. This structure gives rise to attacks, of which the most well known is the extension attack. The latter allows an attacker given $t \leftarrow \mathbf{MD}[\mathsf{h}](e_2\|M)$, where e_2 is a secret unknown to the attacker and $M \in \{0,1\}^*$ is public, to compute compute $t' = \mathbf{MD}[\mathsf{h}](e_2\|M')$, for some $M' \in \{0,1\}^*$ of its choice. This has been exploited to violate the UF-security of the so-called prefix message authentication code $\mathsf{pfMAC}_{e_2}(M) = \mathsf{HH}(e_2\|M)$ when HH is an MD-hash function; HMAC [30] was designed to overcome this.

A proof of security of a scheme (such as EdDSA) that uses a hash function HH will often model HH as a random oracle [39], in what we'll call the $(\mathsf{HH},\mathsf{HH})$-model: scheme algorithms, and the adversary, both have oracle access to the same random HH. However the presence of the above-discussed structure in "real" hash functions led Dodis, Ristenpart and Shrimpton (DRS) [89] to argue that the "right" model in which to prove security of a scheme that uses $\mathsf{HH} = \mathbf{MD}[\mathsf{h}]$ is to model the compression function h —rather than the hash function $\mathsf{HH} = \mathbf{MD}[\mathsf{h}]$— as a random oracle. We'll call this the $(\mathbf{MD}[\mathsf{h}],\mathsf{h})$-model: the adversary has oracle access to a random h, with scheme algorithms having access to $\mathbf{MD}[\mathsf{h}]$. There is now widespread agreement with the DRS thesis that proofs of security of MD-hash-using schemes should use the $(\mathbf{MD}[\mathsf{h}],\mathsf{h})$ model.

Giving from-scratch proofs in the $(\mathbf{MD}[\mathsf{h}], \mathsf{h})$ model is, however, difficult. Maurer, Renner and Holenstein (MRH) [156] show that if a construction $\mathbf{F}$ is indifferentiable (abbreviated indiff) and a scheme is secure in the $(\mathsf{HH}, \mathsf{HH})$ model, then it remains secure in the $(\mathbf{F}[\mathsf{h}], \mathsf{h})$ model. (This requires the game defining security of the scheme to be single-stage [187], which is true for the relevant ones here.) Unfortunately, $\mathbf{F} = \mathbf{MD}$ is provably *not* indiff [74], due exactly to the extension attack. So the MRH result does not help with $\mathbf{MD}$. This led to a search for indiff variants. DRS [89] and YMO [199] (independently) offer public-indiff and show that it suffices to prove security, in the $(\mathbf{MD}[\mathsf{h}], \mathsf{h})$ model, of schemes that use $\mathbf{MD}$ in some restricted way. However, EdDSA does not obey these restrictions. Thus, other means are needed.

THE EdDSA SCHEME. The Edwards curve Digital Signature Algorithm (EdDSA) is a Schnorr-based signature scheme introduced by Bernstein, Duif, Lange, Schwabe and Yang [50]. Ed25519, which uses the Curve25519 Edwards curve and SHA512 as the hash function, is its most popular instance. The scheme is standardized by NIST [174] and the IETF [133]. It is used in TLS 1.3, OpenSSH, OpenSSL, Tor, GnuPGP, Signal and WhatsApp. It is also the preferred signature scheme of the Corda, Tezos, Stellar and Libra blockchain systems. Overall, IANIX [123] reports over 200 uses of Ed25519. Proving security of this scheme is accordingly of high importance.

Figure 4.4 shows EdDSA on the right, and, on the left, the classic Schnorr scheme [191] on which EdDSA is based. The schemes are over a cyclic, additively-written group $\mathbb{G}$ of prime order p with generator $\mathtt{B}$. The public verification key is $\mathtt{A}$. The Schnorr hash function has range $\mathbb{Z}_{\mathsf{p}} = \{0, \ldots, \mathsf{p}-1\}$, while, for EdDSA, function HH_1 has range $\{0,1\}^{2k}$ where k, the bit-length of p, is 256 for Ed25519. Functions $\mathsf{HH}_2, \mathsf{HH}_3$ have range $\mathbb{Z}_{\mathsf{p}}$.

EdDSA differs from Schnorr in significant ways. While the Schnorr secret key s is in $\mathbb{Z}_{\mathsf{p}}$, the EdDSA secret key sk is a k-bit string. This is hashed and the $2k$-bit result is split into k-bit halves $e_1 \| e_2$. A Schnorr secret-key s is derived by applying to e_1 a clamping function $\mathbf{CF}$ that zeroes out the three least significant bits of e_1. (Note: This means s is *not* uniformly distributed over $\mathbb{Z}_{\mathsf{p}}$.) Clamping increases resistance to side-channel attacks [50]. Signing is made deterministic by a standard de-randomization technique [109, 169, 36, 44], namely obtaining the Schnorr randomness r by hashing the message M with a secret-key dependent string e_2. We note that all of $\mathsf{HH}_1, \mathsf{HH}_2, \mathsf{HH}_3$ are instantiated via the same hash function, namely SHA512.

PRIOR WORK AND OUR QUESTIONS. Recall that the security goal for a signature scheme is UF (UnForgeability under Chosen-Message Attack) [110]. Schnorr is well studied, and proven UF under DL (Discrete Log in $\mathbb{G}$) when HH is a random oracle [184, 3]. The provable security of EdDSA, however, received surprisingly little attention until the work of Brendel, Cremers, Jackson and Zhao (BCJZ) [?]. They take the path also used for Schnorr and other identification-based signature schemes [184, 3], seeing EdDSA as the result of the Fiat-Shamir transform on an underlying identification scheme EdID that they define, proving security of the latter under DL, and concluding UF of EdDSA under DL when HH is a random oracle. This is an important step forward, but the BCJZ proof [?] remains in the $(\mathsf{HH}, \mathsf{HH})$ model. We ask and address the following two questions.

1. Can we prove security in the $(\mathbf{MD}[\mathsf{h}], \mathsf{h})$ model? The NIST standard [174] mandates that Ed25519 uses SHA512, which is an MD-hash function. Accordingly, as explained above, the BCJZ proof [?], being in the $(\mathsf{HH}, \mathsf{HH})$ model, does not guarantee security; to do the latter, we need a proof in the $(\mathbf{MD}[\mathsf{h}], \mathsf{h})$ model.

The gap is more than cosmetic. As we saw above with the example of the prefix MAC, a scheme could be secure in the $(\mathsf{HH}, \mathsf{HH})$ model, yet totally insecure in the more realistic $(\mathbf{MD}[\mathsf{h}], \mathsf{h})$ model, and thus also in practice. And EdDSA skirts close to the edge: line 14 is using the prefix-MAC that the extension attack breaks, and overlaps in inputs across the three uses of HH could lead to failures. Intuitively what prevents attacks is that the MAC outputs are taken modulo p, and inputs to HH in two of the three uses involve secrets. Thus, we'd expect that the scheme is indeed secure in the $(\mathbf{MD}[\mathsf{h}], \mathsf{h})$ model.

Proving this, however, is another matter. We already know that **MD** is not indiff. It is public indiff [89, 199], but this will not suffice for EdDSA because $\mathsf{HH}_1, \mathsf{HH}_2$ are being called on secrets. We ask, first, can EdDSA be proved secure in the $(\mathbf{MD}[\mathsf{h}], \mathsf{h})$ model, and second, can this be done in some modular way, rather than from scratch?

2. Can we improve reduction tightness? The reduction of BCJZ [?] is so loose that, in the 256-bit curve over which Ed25519 is implemented, it guarantees little security. Let's elaborate. Given an adversary A_{UF} violating the UF-security of EdDSA with probability $\varepsilon_{\mathrm{UF}}$, the reduction builds an adversary A_{DL} breaking DL with probability $\varepsilon_{\mathrm{DL}} = \varepsilon_{\mathrm{UF}}^2 / q_h$ where q_h is the number of HH-queries of A_{UF} and the two adversaries have about the same running time t. (The square arises from the use of rewinding, analyzed via the Reset Lemma of [35].) In an order p elliptic curve group, $\varepsilon_{\mathrm{DL}} \approx t^2/p$ so we get $\varepsilon_{\mathrm{UF}} = t \cdot \sqrt{q_h/p}$. Ed25519 has $p \approx 2^{256}$. Say $t = q_h = 2^{70}$, which (as shown by BitCoin mining capability) is not far from attacker reach. Then $\varepsilon_{\mathrm{DL}} = 2^{-116}$ is small but $\varepsilon_{\mathrm{UF}} = 2^{70} \cdot 2^{-(256-70)/2} = 2^{-23}$ is in comparison quite high.

Now, one might say that one would not expect better because the same reduction loss is present for Schnorr. The classical reductions for Schnorr [184, 3] did indeed display the above loss, but that has changed: recent advances for Schnorr include a tighter reduction from DL [?], an almost-tight reduction from the MBDL problem [32] and a tight reduction from DL in the Algebraic Group Model [?]. We'd like to put EdDSA on par with the state of the art for Schnorr. We ask, first, is this possible, and second, is there a modular way to do it that leverages, rather than repeats, the (many, complex) just-cited proofs for Schnorr?

Contributions for EdDSA. We simultaneously simplify and strengthen the security proofs for EdDSA as follows.

1. Reduction from Schnorr. Rather than, as in prior work, give a reduction from DL or some other algebraic problem, we give a simple, direct reduction from Schnorr itself. That is, we show that if the Schnorr signature scheme is UF-secure, then so is EdDSA. Furthermore, the reduction is *tight* up to a constant factor. This allows us to leverage prior work [?, 32, ?] to obtain tight proofs for EdDSA under various algebraic assumptions and justify security for group sizes in actual use. But there are two further dividends. First, Schnorr [191] is over 30 years old and has withstood the tests of time and cryptanalysis, so our proof that EdDSA is just as secure as Schnorr allows the former to inherit, and benefit from, this confidence. Second, our result formalizes and proves what was the intuition and belief in the first place [50], namely that, despite the algorithmic differences, EdDSA is a sound hardening of Schnorr.

2. Accurate modeling of the hash function. As noted above, BCJZ [?] assume the hash function HH is a random oracle, but this, due to the extension attack, is not an accurate model for the MD-hash function SHA512 used by EdDSA. We fill this gap by instead proving security in the $(\mathbf{MD}[\mathsf{h}], \mathsf{h})$ model, where $\mathsf{HH} = \mathbf{MD}[\mathsf{hh}]$ is derived via the MD-transform [162, 77] and the compression function hh is a random oracle.

Approach and broader contributions. The above-mentioned results on EdDSA are obtained as a consequence of more general ones.

3. The DR transform and its soundness. We extend the hardening technique used in EdDSA to define a general transform that we call Derive-then-Derandomize (**DR**). It takes an *arbitrary* signature scheme DS, and with the aid of a PRG HH_1 and a PRF HH_2, constructs a hardened signature scheme $\overline{\mathsf{DS}}$. We provide (Theorem 13) a strong and general validation of **DR**, showing that $\overline{\mathsf{DS}}$ is UF-secure assuming DS is UF-secure. Moreover *the reduction is tight* and the proof is simple. This shows that the EdDSA hardening method is generically sound.

4. Indifferentiability of Shrink-MD. It is well-known that **MD** is not indifferentiable [156] from a random oracle, but that the **Chop-MD** [74], which truncates the output of an an **MD** hash by some number of bits, is indifferentiable. Unfortunately, we identified gaps in two prominent proofs of indifferentiability of **Chop-MD** [74, 165]. EdDSA uses a similar construction that reduces the **MD** hash output modulo a prime p sufficiently smaller than the size of the range of **MD**, due to which we refer to this construction as **Mod-MD**. The **Mod-MD** construction has not been proven indifferentiable. We simultaneously give new proofs of indifferentiability for **Chop-MD** and **Mod-MD** as part of a more general class of constructions that we call **Shrink-MD** functors. These are constructions of the form $\mathsf{Out}(\mathbf{MD})$ where Out is some output-processing function, and we prove indifferentiability under certain “shrinking” conditions on Out.

5. Application to EdDSA. EdDSA is obtained as the result $\overline{\mathsf{DS}}$ of the **DR** transform applied to the $\mathsf{DS} = \mathsf{Schnorr}$ signature scheme, and with the PRG and PRF defined via **MD**, specifically $\mathsf{HH}_1(sk) = \mathbf{MD}[\mathsf{hh}](sk)$ and $\mathsf{HH}_2(e_2, M) = \mathbf{MD}[\mathsf{hh}](e_2 \| M) \bmod \mathsf{p}$ where p is the prime order of the underlying group. Additionally, the hash function used in Schnorr is also $\mathsf{HH}_3(X) = \mathbf{MD}[\mathsf{hh}](X) \bmod \mathsf{p}$. Due to Theorem 13 validating **DR**, we are left to show the PRG security of HH_1, the PRF security of HH_2 and the UF-security of Schnorr, all with hh modeled as a random oracle. We do the first directly. We obtain the second as a consequence of the indifferentiability of **Mod-MD**. (In principle it follows from the PRF security of AMAC [29], but we found it difficult to extract precise bounds via this route.) For the third, we again exploit indifferentiability of **Mod-MD**, together with a technique from BCJZ [**?**] to handle clamping, to reduce to the UF security of regular Schnorr, where the hash function is modeled as a random oracle. Putting all this carefully together yields our above-mentioned results for EdDSA. We note that one delicate and important point is that the idealized compression function hh is *the same* across $\mathsf{HH}_1, \mathsf{HH}_2$ and HH_3, meaning these are not independent. This is handled through the building blocks in Theorem 13 being functors [**?**] rather than functions.

Discussion and related work. Both BCJZ [**?**] and CGN [69] note that there are a few versions of EdDSA out there, the differences being in their verification algorithms. What Figure 4.4 shows is the most basic version of the scheme, but we will be able to cover the variants too, in a modular way, by reducing from Schnorr with the same verification algorithm.

BBT [29] define the function AMAC[h] to take a key e_2 and message M, and return $\mathbf{MD}[\mathsf{h}](e_2 \| M) \bmod \mathsf{p}$. This is the HH_2 in EdDSA. We could exploit their results to conclude PRF security of HH_2, but it requires putting together many different pieces from their work, and it is easier and more direct to establish PRF security of HH_2 by using our lemma on the indifferentiability of **Mod-MD**.

In the Generic Group Model (GGM) [195], it is possible to prove UF-security of Schnorr under standard (rather than random oracle) model assumptions on the hash functions [175, **?**]. But use of the GGM means the result applies to a limited class of adversaries. Our results, following the classical proofs for identification-based signatures [184, 179, 3, 137], instead use the standard model for the group, while modeling the hash function (in our case, the compression function) as a random oracle.

In an earlier version of this paper, our proofs had relied on a variant of indifferentiability that we had introduced. At the suggestion of a Crypto 2022 reviewer, this has been dropped in favor of a direct proof based on PRG and PRF assumptions on $\mathsf{HH}_1, \mathsf{HH}_2$. We thank the (anonymous) reviewer for this suggestion.

Theorem 13 is in the standard model if the PRG, PRF and starting signature scheme DS are standard-model, hence can be viewed as a standard-model justification of the hardening template underlying EdDSA. However, when we want to justify EdDSA itself, we need to consider the specific, $\mathbf{MD}$-based instantiations of the PRG, PRF and $\mathsf{Schnorr}$ hash function, and for these we use the model where the compression function is ideal.

Several works study de-randomization of signing by deriving the coins via a PRF applied to the message, considering different ways to key the PRF [109, 169, 36, 44]. We use their techniques in the proof of Theorem 13.

One might ask how to view the UF-security of $\mathsf{Schnorr}$ signatures as an assumption. What is relevant is not its form (it is interactive) but that (1) it can be seen as a hub from where one can bridge to other assumptions that imply it, such as DL (non-tightly) [184, 3] or MBDL (tightly) [32], and (2) it is validated by decades of cryptanalysis.

Our results have been stated for UF but extend to SUF (Strong unforgeability), meaning our proofs also show SUF-security of EdDSA in the $(\mathbf{MD}[\mathsf{h}], \mathsf{h})$ model assuming SUF security of $\mathsf{Schnorr}$, with a tight (up to the usual constant factor) reduction.

EdDSA could be used with other hash functions such as $\mathsf{SHAKE256}$. The extension attack does not apply to the latter, so the proof of BCJZ [**?**] applies, but gives a loose reduction from DL; our results still add something, namely a tight reduction from $\mathsf{Schnorr}$ and thus improved tightness in several ways as discussed above.

4.2 Preliminaries

<u>Notation.</u> If n is a positive integer, then $\mathbb{Z}_n$ denotes the set $\{0,\ldots,n-1\}$ and $[n]$ or $[1..n]$ denote the set $\{1,\ldots,n\}$. If $\mathbf{x}$ is a vector then $|\mathbf{x}|$ is its length (the number of its coordinates), $\mathbf{x}[i]$ is its i-th coordinate and $[\mathbf{x}] = \{\mathbf{x}[i] : 1 \leq i \leq |\mathbf{x}|\}$ is the set of all its coordinates. A string is identified with a vector over $\{0,1\}$, so that if x is a string then $x[i]$ is its i-th bit and $|x|$ is its length. We denote $x[i..j]$ the i-th bit to the j-th bit of string x. By ε we denote the empty vector or string. The size of a set S is denoted $|S|$. For sets D,R let $\mathrm{FUNC}((,D),R)$ denote the set of all functions $f\colon D \to R$. If $f\colon D \to R$ is a function then $\mathsf{Img}(f) = \{f(x) : x \in D\} \subseteq R$ is its image. We say that f is *regular* if every $y \in \mathsf{Img}(f)$ has the same number of pre-images under f. By $\{0,1\}^{\leq L}$ we denote the set of all strings of length at most L. For any variables a and b, the expression $[[a = b]]$ denotes the Boolean value true when a and b contain the same value and false otherwise.

Let S be a finite set. We let $x \leftarrow\!\!\$\, S$ denote sampling an element uniformly at random from S

and assigning it to x. We let $y \leftarrow A[\mathrm{O}_1,\ldots](x_1,\ldots;r)$ denote executing algorithm A on inputs $x_1,\ldots$ and coins r with access to oracles $\mathrm{O}_1,\ldots$ and letting y be the result. We let $y \leftarrow\!\!\$\, A[\mathrm{O}_1,\ldots](x_1,\ldots)$ be the resulting of picking r at random and letting $y \leftarrow A[\mathrm{O}_1,\ldots](x_1,\ldots;r)$ be the equivalent. We let $\mathsf{OUT}(A[\mathrm{O}_1,\ldots](x_1,\ldots)])$ denote the set of all possible outputs of A when invoked with inputs $x_1,\ldots$ and oracles $\mathrm{O}_1,\ldots$. Algorithms are randomized unless otherwise indicated. Running time is worst case.

GAMES. We use the code-based game playing framework of [42]. (See Fig. 1 for an example.) Games have procedures, also called oracles. Among the oracles are INIT and a FIN. In executing an adversary $\mathscr{A}$ with a game G, the adversary may query the oracles at will. We require that the adversary's first oracle query be to INIT and its last to FIN and it query these oracles at most once. The value return by the FIN procedure is taken as the game output. By $\mathrm{G}(\mathscr{A}) \Rightarrow y$ we denote the event that the execution of game G with adversary $\mathscr{A}$ results in output y. We write $\Pr[\mathrm{G}(\mathscr{A})]$ as shorthand for $\Pr[\mathrm{G}(\mathscr{A}) \Rightarrow \mathsf{true}]$, the probability that the game returns true.

In writing game or adversary pseudocode, it is assumed that Boolean variables are initialized to false, integer variables are initialized to 0 and set-valued variables are initialized to the empty set $\emptyset$.

We adopt the convention that the running time of an adversary is the time for the execution of the game with the adversary, so that the time for oracles to respond to queries is included. In counting the number of queries to an oracle O, we have two metrics. We let $\mathrm{Q}_{\mathrm{O}}^{\mathscr{A}}$ denote the number of queries made to O in the execution of the game with $\mathscr{A}$. (This includes not just queries made directly by $\mathscr{A}$ but also those made by game oracles, the latter usually arising from game executions of scheme algorithms that use O.) In particular, under this metric, the number of queries to a random oracle FO includes those made by scheme algorithms executed by game procedures. With $\mathsf{q}_{\mathrm{O}}^{\mathscr{A}}$ we count only queries made directly by $\mathscr{A}$ to O, not by other game oracles or scheme algorithms. These counts are all worst case.

GROUPS. Throughout the paper, we fix integers k and b, an odd prime p, and a positive integer f such that $2^{\mathsf{f}} < \mathsf{p}$. We then fix two groups: $\mathbb{G}$, a group of order $\mathsf{p} \cdot 2^{\mathsf{f}}$ whose elements are k-bit strings, and its cyclic subgroup $\mathbb{G}_{\mathsf{p}}$ of order p. We prove in Appendix 4.7 that this subgroup is unique, and that it has an efficient membership test. We also assume an efficient membership test for $\mathbb{G}$. We will use additive notation for the group operation, and we let $0_{\mathbb{G}}$ denote the identity element of $\mathbb{G}$. We let $\mathbb{G}_{\mathsf{p}}^* = \mathbb{G} \setminus \{0_{\mathbb{G}}\}$ denote the set of non-identity elements of $\mathbb{G}_{\mathsf{p}}$, which is its set of generators. We fix a distinguished generator $\mathtt{B} \in \mathbb{G}_{\mathsf{p}}^*$. Then for any $X \in \mathbb{G}^*$, the discrete logarithm base $\mathtt{B}$ of X is denoted $\mathsf{DL}_{\mathbb{G},\mathtt{B}}(X)$, and it is in the set $\mathbb{Z}_{|\mathbb{G}|}$. The instantiation of $\mathbb{G}$ used in Ed25519 is described in Section **??**.

4.3 Functor framework

Our treatment relies on the notion of functors [**?**], which are functions that access an idealized primitive. We give relevant definitions, starting with signature schemes whose security is measured relative to a functor. Then we extend the notions of PRGs and PRFs to functors.

FUNCTION SPACES. In using the random oracle model [39], works in the literature sometimes omit to say what exactly are the domain and range of the underlying functions, and, when multiple functions are present, whether or not they are independent. (Yet, implicitly their proofs rely on certain choices.) For greater precision, we use the language of function spaces of [**?**], which we now recall.

A *function space* FS is a set of tuples $\mathsf{HH} = (\mathsf{HH}_1, \ldots, \mathsf{HH}_n)$ of functions. The integer n is called the arity of the function space, and can be recovered as $\mathsf{FS.arity}$. We view HH as taking an input X that it parses as (i,x) to return $\mathsf{HH}_i(x)$.

FUNCTORS. Following [**?**], we use the term functor for a transform that constructs one function from another. A functor $\mathbf{F}$: $\mathsf{SS} \to \mathsf{ES}$ takes as oracle a function hh from a starting function space SS and returns a function $\mathbf{F}[\mathsf{hh}]$ in the ending function space ES. (The term is inspired by category theory, where a functor maps from one category into another. In our case, the categories are function spaces.) If ES has arity n, then we also refer to n as the arity of $\mathbf{F}$, and write $\mathbf{F}_i$ for the functor which returns the i-th component of $\mathbf{F}$. That is, $\mathbf{F}_i[\mathsf{hh}]$ lets $\mathsf{HH} \leftarrow \mathbf{F}[\mathsf{hh}]$ and returns HH_i.

MD FUNCTOR. We are interested in the Merkle-Damgrard [162, 77] transform. This transform constructs a hash function with domain $\{0,1\}^*$ from a compression function hh: $\{0,1\}^{b+2k} \to \{0,1\}^{2k}$ for some integers b and k. The compression function takes a $2k$-bit chaining variable y and a b-bit block B to return a $2k$ bit output $\mathsf{hh}(y\|B)$. In the case of SHA512, the hash function used in EdDSA, the compression function sha512 has $b = 1024$ and $k = 256$ (so the chaining variable is 512 bits and a block is 1024 bits), while $b = 512$ and $k = 128$ for SHA256. In our language, the Merkle-Damgrard transform is a functor **MD**: $\mathrm{FUNC}((,\{0,1\})^{b+2k},\{0,1\}^{2k}) \rightarrow \mathrm{FUNC}((,\{0,1\})^{*}, \{0,1\}^{2k})$. It is parameterized by a padding function pad that takes the length ℓ of an input to the hash function and returns a padding string such that $\ell + |\mathsf{pad}(\ell)|$ is a multiple of b. Specifically, $\mathsf{pad}(\ell)$ returns $10^*\langle\ell\rangle$ where $\langle\ell\rangle$ is a 64-bit, resp. 128-bit encoding of ℓ for SHA256 resp. SHA512, and 0^* indicates the minimum number p of 0s needed to make $\ell + 1 + p + 64$, resp. $\ell + 1 + p + 128$ a multiple of b. We also fix an "initialization vector" $IV \in \{0,1\}^{2k}$. Given oracle hh, the functor defines hash function $\mathsf{HH} = \mathbf{MD}[\mathsf{hh}]\colon \{0,1\}^* \rightarrow \{0,1\}^{2k}$ as follows:

```
Functor $\mathbf{MD}[\mathsf{hh}](X)$
$y[0] \leftarrow IV$
$P \leftarrow \mathsf{pad}(|X|)$ ; $X'[1]\ldots X'[m] \leftarrow X\|P$    // Split $X\|P$ into $b$-bit blocks
For $i = 1,\ldots,m$ do $y[i] \leftarrow \mathsf{hh}(y[i-1]\|X'[i])$
Return $y[m]$
```

Strictly speaking, the domain is only strings of length less than 2^{64} resp. 2^{128}, but since this is huge in practice, we view the domain as $\{0,1\}^*$.

Signature scheme syntax. We give an enhanced, flexible syntax for a signature scheme DS. We want to cover ROM schemes, which means scheme algorithms have oracle access to a function HH, but of what range and domain? Since these can vary from scheme to scheme, we have the scheme begin by naming the function space DS.FS from which HH is drawn. We see the key-generation algorithm DS.Kg as first picking a signing key $sk \leftarrow\!\!\$\, \mathsf{DS.SK}$ via a signing-key generation algorithm DS.SK, then obtaining the public verification key $pk \leftarrow \mathsf{DS.PK}[\mathsf{HH}](sk)$ by applying a deterministic verification-key generation algorithm DS.PK, and finally returning (pk, sk). (For simplicity, DS.SK, unlike other scheme algorithms, does not have access to HH.) We break it up like this because we may need to explicitly refer to the sub-algorithms in constructions. Continuing, via $\sigma \leftarrow \mathsf{DS.Sign}[\mathsf{HH}](sk, pk, M; r)$ the signing algorithm takes sk, pk, a message $M \in \{0,1\}^*$, and randomness r from the randomness space DS.SR of the algorithm, to return a signature σ. As usual, $\sigma \leftarrow\!\!\$\, \mathsf{DS.Sign}[\mathsf{HH}](sk, pk, M)$ is shorthand for picking $r \leftarrow\!\!\$\, \mathsf{DS.SR}$ and

```
Game $\mathbf{G}^{\mathrm{uf}}_{\mathsf{DS},\mathbf{FF}}$

INIT:
 1 $\mathsf{hh} \leftarrow\!\!\$\, \mathsf{SS}$ ; $\mathsf{HH} \leftarrow \mathbf{FF}[\mathrm{FO}]$ ; $(pk,sk) \leftarrow\!\!\$\, \mathsf{DS.Kg}[\mathsf{HH}]$ ; Return $pk$

SIGN($M$):
 2 $\sigma \leftarrow\!\!\$\, \mathsf{DS.Sign}[\mathsf{HH}](sk,pk,M)$ ; $S \leftarrow S \cup \{M\}$ ; Return $\sigma$

FO($X$):
 3 Return $\mathsf{hh}(X)$

FIN($M_*,\sigma_*$):
 4 If $(M_* \in S)$ then return $\mathsf{false}$
 5 Return $\mathsf{DS.Vf}[\mathsf{HH}](pk,M_*,\sigma_*)$
```

```
Game $\mathbf{G}^{\mathrm{prg}}_{\mathbf{P}}$

INIT:
 1 $\mathsf{hh} \leftarrow\!\!\$\, \mathsf{SS}$ ; $c \leftarrow\!\!\$\, \{0,1\}$
 2 $s \leftarrow\!\!\$\, \{0,1\}^k$ ; $y_1 \leftarrow \mathbf{P}[\mathrm{FO}](s)$
 3 $y_0 \leftarrow\!\!\$\, \{0,1\}^\ell$
 4 Return $y_c$

FO($X$):
 5 Return $\mathsf{hh}(X)$

FIN($c'$):
 6 Return $(c = c')$
```

```
Game $\mathbf{G}^{\mathrm{prf}}_{\mathbf{F}}$

INIT:
 1 $\mathsf{hh} \leftarrow\!\!\$\, \mathsf{SS}$ ; $c \leftarrow\!\!\$\, \{0,1\}$ ; $K \leftarrow\!\!\$\, \{0,1\}^k$

FN($X$):
 2 If $\mathbf{YT}[X] \neq \bot$ then
 3     If $(c = 1)$ then $\mathbf{YT}[X] \leftarrow \mathbf{F}[\mathrm{FO}](K,X)$
 4     Else $\mathbf{YT}[X] \leftarrow\!\!\$\, R$
 5 Return $\mathbf{YT}[X]$

FO($X$):
 6 Return $\mathsf{hh}(X)$

FIN($c'$):
 7 Return $(c = c')$
```

Figure 4.1. Top: Game defining UF security of signature scheme DS relative to functor $\mathbf{FF}\colon \mathsf{SS} \to \mathsf{DS.FS}$. Bottom Left: Game defining PRG security of functor $\mathbf{P}\colon \mathsf{SS} \to \mathsf{FUNC}((,\{0,1\})^k,\{0,1\}^\ell)$. Bottom Right: Game defining PRF security of functor $\mathbf{F}\colon \mathsf{SS} \to \mathsf{FUNC}((,\{0,1\})^k \times \{0,1\}^*,R)$.

returning $\sigma \leftarrow \mathsf{DS.Sign}[\mathsf{HH}](sk,pk,M;r)$. Via $b \leftarrow \mathsf{DS.Vf}[\mathsf{HH}](pk,M,\sigma)$, the verification algorithm obtains a boolean decision $b \in \{\mathsf{true},\mathsf{false}\}$ about the validity of the signature. The correctness requirement is that for all $\mathsf{HH} \in \mathsf{DS.FS}$, all $(pk,sk) \in \mathsf{OUT}(\mathsf{DS.Kg}[\mathsf{HH}])$, all $M \in \{0,1\}^*$ and all $\sigma \in \mathsf{OUT}(\mathsf{DS.Sign}[\mathsf{HH}](sk,pk,M))$ we have $\mathsf{DS.Vf}[\mathsf{HH}](pk,M,\sigma) = \mathsf{true}$.

UF SECURITY. We want to discuss security of a signature scheme DS under different ways in which the functions in $\mathsf{DS.FS}$ are chosen or built. Game $\mathbf{G}^{\mathrm{uf}}_{\mathsf{DS},\mathbf{FF}}$ in Fig. 4.1 is thus parameterized by a functor $\mathbf{FF}\colon \mathsf{SS} \to \mathsf{DS.FS}$. At line 1, a starting function hh is chosen from the starting space of the functor, and then the function $\mathsf{HH} \in \mathsf{DS.FS}$ that the scheme algorithms (key-generation, signing and verification) get as oracle is determined as $\mathsf{HH} \leftarrow \mathbf{FF}[\mathsf{hh}]$. The adversary, however, via oracle FO, gets access to hh, which here is the random oracle. The rest is as per the usual

```
$\overline{\mathsf{DS}}.\mathsf{SK}$:
    $\overline{sk} \leftarrow\!\!\$\, \{0,1\}^k$ ; Return $\overline{sk}$
$\overline{\mathsf{DS}}.\mathsf{PK}[\mathsf{HH}](\overline{sk})$:
    $e_1 \| e_2 \leftarrow \mathsf{HH}_1(\overline{sk})$ ; $sk \leftarrow \mathtt{CF}(e_1)$
    $pk \leftarrow \mathsf{DS}.\mathsf{PK}[\mathsf{HH}_3](sk)$
    Return $pk$
$\overline{\mathsf{DS}}.\mathsf{Sign}[\mathsf{HH}](\overline{sk}, pk, M)$:
    $e_1 \| e_2 \leftarrow \mathsf{HH}_1(\overline{sk})$ ; $sk \leftarrow \mathtt{CF}(e_1)$
    $r \leftarrow \mathsf{HH}_2(e_2, M)$
    $\sigma \leftarrow \mathsf{DS}.\mathsf{Sign}[\mathsf{HH}_3](sk, pk, M; r)$
    Return $\sigma$
$\overline{\mathsf{DS}}.\mathsf{Vf}[\mathsf{HH}](pk, M, \sigma)$:
    Return $\mathsf{DS}.\mathsf{Vf}[\mathsf{HH}_3](pk, M, \sigma)$
```

```
$\mathsf{DS}^*.\mathsf{SK}$:
    $\overline{sk} \leftarrow\!\!\$\, \{0,1\}^k$ ; Return $\overline{sk}$
$\mathsf{DS}^*.\mathsf{PK}[\mathsf{G}](\overline{sk})$:
    $sk \leftarrow \mathtt{CF}(\overline{sk})$
    $pk \leftarrow \mathsf{DS}.\mathsf{PK}[\mathsf{G}](sk)$
    Return $pk$
$\mathsf{DS}^*.\mathsf{Sign}[\mathsf{G}](\overline{sk}, pk, M)$:
    $sk \leftarrow \mathtt{CF}(\overline{sk})$
    $\sigma \leftarrow\!\!\$\, \mathsf{DS}.\mathsf{Sign}[\mathsf{G}](sk, pk, M)$
    Return $\sigma$
$\mathsf{DS}^*.\mathsf{Vf}[\mathsf{G}](pk, M, \sigma)$:
    Return $\mathsf{DS}.\mathsf{Vf}[\mathsf{G}](pk, M, \sigma)$
```

Figure 4.2. Left: The signature scheme $\overline{\mathsf{DS}} = \mathbf{DR}[\mathsf{DS}, \mathtt{CF}]$ constructed by the **DR** transform applied to signature scheme DS and clamping function $\mathtt{CF}\colon \{0,1\}^k \to \mathsf{OUT}(\mathsf{DS}.\mathsf{SK})$. **Right:** The signature scheme $\overline{\mathsf{DS}} = \mathbf{JCl}[\mathsf{DS}, \mathtt{CF}]$ constructed by the **JCl** transform.

unforgeability definition. (Given in the standard model in [110] and extended to the ROM in [39].) We define the UF advantage of adversary $\mathscr{A}$ as $\mathbf{Adv}^{\mathrm{uf}}_{\mathsf{DS},\mathbf{FF}}(\mathscr{A}) = \Pr[\mathbf{G}^{\mathrm{uf}}_{\mathsf{DS},\mathbf{FF}}(\mathscr{A})]$.

PRGs and PRFs. The usual definition of a PRGs is for a function; we define it instead for a functor $\mathbf{P}$. The game $\mathbf{G}^{\mathrm{prg}}_{\mathbf{P}}$ is in Figure 4.1. It picks a function hh from the starting space SS of the functor. The functor now determines a function $\mathbf{P}[\mathsf{hh}]\colon \{0,1\}^k \to \{0,1\}^\ell$. The game then follows the usual PRG one for this function, additionally giving the adversary oracle access to hh via oracle FO. We let $\mathbf{Adv}^{\mathrm{prg}}_{\mathbf{P}}(\mathscr{A}) = 2\Pr[\mathbf{G}^{\mathrm{prg}}_{\mathbf{P}}(\mathscr{A})] - 1$.

Similarly we extend the usual definition of PRG security to a functor $\mathbf{F}$, via game $\mathbf{G}^{\mathrm{prf}}_{\mathbf{F}}$ of Figure 4.1. Here, for hh in the starting space SS of the functor, the defined function maps as $\mathbf{F}[\mathsf{hh}]\colon \{0,1\}^k \times \{0,1\}^* \to R$ for some k and range set R. We let $\mathbf{Adv}^{\mathrm{prf}}_{\mathbf{F}}(\mathscr{A}) = 2\Pr[\mathbf{G}^{\mathrm{prf}}_{\mathbf{F}}(\mathscr{A})] - 1$.

4.4 The soundness of Derive-then-Derandomize

We specify a general signature-hardening transform that we call Derive-then-Derandomize (**DR**) and prove that it preserves the security of the starting signature scheme.

The **DR** transform. Let DS be a given signature scheme that we call the base signature scheme. It will be the (general) Schnorr scheme in our application. Assume for simplicity that its function space $\mathsf{DS}.\mathsf{FS}$ has arity 1.

The **DR** (derive then de-randomize) transform constructs a signature scheme $\overline{\mathsf{DS}} =$

$\mathbf{DR}[\mathsf{DS},\mathtt{CF}]$ based on DS and a function $\mathtt{CF}\colon \{0,1\}^k \to \mathsf{OUT}(\mathsf{DS.SK})$, called the clamping function, that turns a k-bit string into a signing key for DS. The algorithms of $\overline{\mathsf{DS}}$ are shown in Figure 4.2. They have access to oracle HH that specifies sub-functions $\mathsf{HH}_1, \mathsf{HH}_2, \mathsf{HH}_3$. Function $\mathsf{HH}_1\colon \{0,1\}^k \to \{0,1\}^{2k}$ expands the signing key $\overline{sk}$ of $\overline{\mathsf{DS}}$ into sub-keys ε_1 and ε_2. The clamping function is applied to ε_1 to get a signing key for the base scheme, and its associated verification key is returned as the one for the new scheme at line 4. At line 6, function $\mathsf{HH}_2\colon \{0,1\}^k \times \{0,1\}^* \to \mathsf{DS.SR}$ is applied to the second sub-key ε_2 and the message M to determine signing randomness r for the line 5 invocation of the base signing algorithm. Finally, $\mathsf{HH}_3 \in \mathsf{DS.FS}$ is an oracle for the algorithms of DS. Formally the oracle space $\overline{\mathsf{DS}}.\mathsf{FS}$ of $\overline{\mathsf{DS}}$ is the arity 3 space consisting of all $\mathsf{HH} = (\mathsf{HH}_1, \mathsf{HH}_2, \mathsf{HH}_3)$ that map as above.

Viewing the PRG HH_1, PRF HH_2 and oracle HH_3 for the base scheme as specified in the function space is convenient for our application to EdDSA, where they are all based on $\mathbf{MD}$ with the *same* underlying idealized compression function.

Just clamp. Given a signature scheme DS and a clamping function $\mathtt{CF}\colon \{0,1\}^k \to \mathsf{OUT}(\mathsf{DS.SK})$, it is useful to also consider the signature scheme $\mathsf{DS}^* = \mathbf{JCl}[\mathsf{DS},\mathtt{CF}]$ that does just the clamping. The scheme is shown in Figure 4.2. Its oracle space is the same as that of DS and is assumed to have arity 1. On the right of Figure 4.2 the function drawn from it is denoted G; it will be the same as HH_3 on the left.

Security of **DR**. We study the security of the scheme $\overline{\mathsf{DS}} = \mathbf{DR}[\mathsf{DS},\mathtt{CF}]$ obtained via the **DR** transform.

When we prove security of $\overline{\mathsf{DS}}$, it will be with respect to a functor **FF** that constructs all of $\mathsf{HH}_1, \mathsf{HH}_2, \mathsf{HH}_3$. This means that these three functions could all depend on the same starting function that **FF** uses, and in particular not be independent of each other. An important element of the following theorem is that it holds even in this case, managing to reduce security to conditions on the individual functors despite their using related (in fact, the same) underlying starting function.

Theorem 13. *Let* DS *be a signature scheme. Let* $\mathtt{CF}\colon \{0,1\}^k \to \mathsf{OUT}(\mathsf{DS.SK})$ *be a clamping function. Let* $\overline{\mathsf{DS}} = \mathbf{DR}[\mathsf{DS},\mathtt{CF}]$ *and* $\mathsf{DS}^* = \mathbf{JCl}[\mathsf{DS},\mathtt{CF}]$ *be the signature schemes obtained by* *the above transforms. Let* $\mathbf{FF}\colon \mathsf{SS} \to \overline{\mathsf{DS}}.\mathsf{FS}$ *be a functor that constructs the function* HH *that algorithms of* $\overline{\mathsf{DS}}$ *use as an oracle. Let* $\mathscr{A}$ *be an adversary attacking the* $\mathbf{G}^{\mathrm{uf}}$ *security of* $\overline{\mathsf{DS}}$*. Then there are adversaries* $\mathscr{A}_1, \mathscr{A}_2, \mathscr{A}_3$ *such that*

$$\mathbf{Adv}^{\mathrm{uf}}_{\overline{\mathsf{DS}},\mathbf{FF}}(\mathscr{A}) \leq \mathbf{Adv}^{\mathrm{prg}}_{\mathbf{FF}_1}(\mathscr{A}_1) + \mathbf{Adv}^{\mathrm{prf}}_{\mathbf{FF}_2}(\mathscr{A}_2) + \mathbf{Adv}^{\mathrm{uf}}_{\mathsf{DS}^*,\mathbf{FF}_3}(\mathscr{A}_3)\,.$$

The constructed adversaries have $\mathrm{Q}^{\mathscr{A}_i}_{\mathrm{FO}} = \mathrm{Q}^{\mathscr{A}}_{\mathrm{FO}}$ $(i = 1,2,3)$ *and approximately the same running time as* $\mathscr{A}$*. Adversary* $\mathscr{A}_2$ *makes* $\mathrm{Q}^{\mathscr{A}}_{\mathrm{SIGN}}$ *queries to* FN*. Adversary* $\mathscr{A}_3$ *makes* $\mathrm{Q}^{\mathscr{A}}_{\mathrm{SIGN}}$ *queries to* Sign*.*

Recall that $\mathrm{Q}^{\mathscr{B}}_{\mathrm{O}}$ means the number of queries made to oracle O in the execution of the game with adversary $\mathscr{B}$, so queries made by scheme algorithms, run in the game in response to $\mathscr{B}$'s queries, are included. The theorem says the number of queries to FO is preserved under this metric. The number of direct queries to FO is not necessarily preserved. Thus $\mathrm{q}^{\mathscr{A}_i}_{\mathrm{FO}}$ could be more than $\mathrm{q}^{\mathscr{A}}_{\mathrm{FO}}$. For example $\mathrm{q}^{\mathscr{A}_1}_{\mathrm{FO}}$ is $\mathrm{q}^{\mathscr{A}}_{\mathrm{FO}}$ plus the number of queries to FO made by the calls to $\mathbf{FF}_3[\mathrm{FO}]$, the latter calls in turn made by the execution of $\mathsf{DS}.\mathsf{Sign}[\mathbf{FF}_3[\mathrm{FO}]]$ across the different queries to Sign. Accounting precisely for this is involved, whence a preference where possible for the game-inclusive query metric $\mathrm{Q}^{\cdot}_{\cdot}$.

Proof of Theorem 13: The proof uses code-based game playing [42]. Consider the games of Figure 4.3. Let $\varepsilon_i = \Pr[\mathrm{G}_i(\mathscr{A})]$ for $i = 0,1,2$.

Game G_0 is the $\mathbf{G}^{\mathrm{uf}}$ game for $\overline{\mathsf{DS}}$ except that the signature of M is stored in table $\mathbf{ST}$ at line 8, and, at line 5, if a signature for M already exists, it is returned directly. Since signing in $\overline{\mathsf{DS}}$ is deterministic, meaning the signature is always the same for a given message and signing key, this does not change what Sign returns, and thus

$$\begin{aligned}\mathbf{Adv}^{\mathrm{uf}}_{\overline{\mathsf{DS}},\mathbf{FF}}(\mathscr{A}) &= \varepsilon_0 \\ &= (\varepsilon_0 - \varepsilon_1) + (\varepsilon_1 - \varepsilon_2) + \varepsilon_2\,.\end{aligned}$$

We bound each of the three terms above in turn.

The change in moving to game G_1 is at line 3, where we sample $\mathrm{e}_1 \| \mathrm{e}_2$ uniformly from the set

```
Games G_0, G_1, G_2

INIT:
1  $\mathsf{hh} \leftarrow\!\!\$\ \mathsf{SS}$
2  $\overline{sk} \leftarrow\!\!\$\ \{0,1\}^k$ ; $e_1 \| e_2 \leftarrow \mathbf{FF}_1[\mathrm{FO}](\overline{sk})$   // Game $\mathrm{G}_0$
3  $e_1 \| e_2 \leftarrow\!\!\$\ \{0,1\}^{2k}$   // Games $\mathrm{G}_1, \mathrm{G}_2$
4  $sk \leftarrow \mathtt{CF}(e_1)$ ; $pk \leftarrow \mathsf{DS.PK}[\mathbf{FF}_3[\mathrm{FO}]](sk)$ ; Return $pk$

SIGN($M$):
5  If $\mathsf{ST}[M] \neq \bot$ then return $\mathsf{ST}[M]$
6  $r \leftarrow \mathbf{FF}_2[\mathrm{FO}](\varepsilon_2, M)$   // Games $\mathrm{G}_0, \mathrm{G}_1$
7  $r \leftarrow\!\!\$\ \mathsf{DS.SR}$   // Game $\mathrm{G}_2$
8  $\mathsf{ST}[M] \leftarrow \mathsf{DS.Sign}[\mathbf{FF}_3[\mathrm{FO}]](sk, pk, M; r)$ ; Return $\mathsf{ST}[M]$

FO($X$):
9  Return $\mathsf{hh}(X)$

FIN($M_*, \sigma_*$):
10 If $(\mathsf{ST}[M_*] \neq \bot)$ then return false
11 Return $\mathsf{DS.Vf}[\mathbf{FF}_3[\mathrm{FO}]](pk, M_*, \sigma_*)$
```

Figure 4.3. Games for proof of Theorem 13. A line annotated with names of games is included only in those games.

$\{0,1\}^{2k}$ rather than obtaining it via $\mathbf{FF}_1[\mathrm{FO}]$ as in game G_0. We build PRG adversary $\mathscr{A}_1$ such that

$$\varepsilon_0 - \varepsilon_1 \leq \mathbf{Adv}^{\mathrm{prg}}_{\mathbf{FF}_1}(\mathscr{A}_1)\,. \tag{4.1}$$

Adversary $\mathscr{A}_1$ is playing game $\mathbf{G}^{\mathrm{prg}}_{\mathbf{FF}_1}$. It gets its challenge via $e_1 \| e_2 \leftarrow \mathbf{G}^{\mathrm{prg}}_{\mathbf{FF}_1}.\textsc{init}$. It lets $sk \leftarrow \mathtt{CF}(e_1)$ and $vk \leftarrow \mathsf{DS.PK}[\mathbf{FF}_3[\mathbf{G}^{\mathrm{prg}}_{\mathbf{FF}_1}.\mathrm{FO}]](sk)$ where $\mathbf{G}^{\mathrm{prg}}_{\mathbf{FF}_1}.\mathrm{FO}$ is the oracle provided in its own game. It runs $\mathscr{A}$, returning vk in response to $\mathscr{A}$'s $\textsc{init}$ query. It answers $\textsc{Sign}$ queries as do $\mathrm{G}_0, \mathrm{G}_1$ except that it uses $\mathbf{G}^{\mathrm{prg}}_{\mathbf{FF}_1}.\mathrm{FO}$ in place of FO at lines 6,8. As part of this simulation, it maintains table ST. It answers FO queries via $\mathbf{G}^{\mathrm{prg}}_{\mathbf{FF}_1}.\mathrm{FO}$. When $\mathscr{A}$ calls $\textsc{fin}(M_*, \sigma_*)$, adversary $\mathscr{A}_1$ lets $c' \leftarrow 1$ if $\mathsf{DS.Vf}[\mathbf{FF}_3[\mathbf{G}^{\mathrm{prg}}_{\mathbf{FF}_1}.\mathrm{FO}]](pk, M_*, \sigma_*)$ is true and $\mathsf{ST}[M_*] = \bot$, and otherwise lets $c' \leftarrow 0$. It then calls $\mathbf{G}^{\mathrm{prg}}_{\mathbf{FF}_1}.\textsc{fin}(c')$. When the challenge bit c in game $\mathbf{G}^{\mathrm{prg}}_{\mathbf{FF}_1}$ is $c = 1$, the view of $\mathscr{A}$ is as in G_0, and when $c = 0$ it is as in G_1, which explains Eq. (4.1).

Moving to G_2, the change is that line 6 is replaced by line 7, meaning signing coins are now chosen at random from the randomness space $\mathsf{DS.SR}$ of DS. We build PRF adversary $\mathscr{A}_2$ such that

$$\varepsilon_1 - \varepsilon_2 \leq \mathbf{Adv}^{\mathrm{prf}}_{\mathbf{FF}_2}(\mathscr{A}_2)\ . \tag{4.2}$$

Adversary $\mathscr{A}_2$ is playing game $\mathbf{G}^{\mathrm{prf}}_{\mathbf{FF}_2}$. It picks $e_1 \| e_2 \leftarrow\!\!\$\ \{0,1\}^{2k}$. It lets $sk \leftarrow \mathtt{CF}(e_1)$ and $vk \leftarrow \mathsf{DS.PK}[\mathbf{FF}_3[\mathbf{G}^{\mathrm{prf}}_{\mathbf{FF}_2}.\mathrm{FO}]](sk)$ where $\mathbf{G}^{\mathrm{prg}}_{\mathbf{FF}_2}.\mathrm{FO}$ is the oracle provided in its own game. It runs $\mathscr{A}$, returning vk in response to $\mathscr{A}$'s INIT query. It answers SIGN queries as does G_1 except that it uses $\mathbf{G}^{\mathrm{prf}}_{\mathbf{FF}_2}.\mathrm{FN}$ in place of $\mathbf{FF}_2[\mathrm{FO}]$ at line 6 and $\mathbf{G}^{\mathrm{prf}}_{\mathbf{FF}_2}.\mathrm{FO}$ in place of FO in line 8. As part of this simulation, it maintains table ST. It answers FO queries via $\mathbf{G}^{\mathrm{prf}}_{\mathbf{FF}_2}.\mathrm{FO}$. When $\mathscr{A}$ calls $\mathrm{FIN}(M_*, \sigma_*)$, adversary $\mathscr{A}_2$ lets $c' \leftarrow 1$ if $\mathsf{DS.Vf}[\mathbf{FF}_3[\mathbf{G}^{\mathrm{prf}}_{\mathbf{FF}_2}.\mathrm{FO}]](pk, M_*, \sigma_*)$ is true and $\mathrm{ST}[M_*] = \bot$, and otherwise lets $c' \leftarrow 0$. It then calls $\mathbf{G}^{\mathrm{prf}}_{\mathbf{FF}_2}.\mathrm{FIN}(c')$. When the challenge bit c in game $\mathbf{G}^{\mathrm{prf}}_{\mathbf{FF}_2}$ is $c = 1$, the view of $\mathscr{A}$ is as in G_1, and when $c = 0$ it is as in G_2, which explains Eq. (4.2).

Finally we build adversary $\mathscr{A}_3$ such that

$$\varepsilon_2 \leq \mathbf{Adv}^{\mathrm{uf}}_{\mathsf{DS}^*, \mathbf{FF}_3}(\mathscr{A}_3)\ . \tag{4.3}$$

Adversary $\mathscr{A}_3$ is playing game $\mathbf{G}^{\mathrm{uf}}_{\mathsf{DS}^*, \mathbf{FF}_3}$. It lets $vk \leftarrow \mathbf{G}^{\mathrm{uf}}_{\mathsf{DS}^*, \mathbf{FF}_3}.\mathrm{INIT}$. It runs $\mathscr{A}$, returning vk in response to $\mathscr{A}$'s INIT query. When $\mathscr{A}$ makes query M to SIGN, it answers as per the following:

If $\mathrm{ST}[M] \neq \bot$ then return $\mathrm{ST}[M]$

$\mathrm{ST}[M] \leftarrow\!\!\$\ \mathbf{G}^{\mathrm{uf}}_{\mathsf{DS}^*, \mathbf{FF}_3}.\mathrm{SIGN}(M)$; Return $\mathrm{ST}[M]$

Note that memoizing signatures in ST is important here to ensure that the SIGN queries of $\mathscr{A}$ are correctly simulated. It answers FO queries via $\mathbf{G}^{\mathrm{uf}}_{\mathsf{DS}^*, \mathbf{FF}_3}.\mathrm{FO}$. When $\mathscr{A}$ calls $\mathrm{FIN}(M_*, \sigma_*)$, adversary $\mathscr{A}_2$ calls $\mathbf{G}^{\mathrm{uf}}_{\mathsf{DS}^*, \mathbf{FF}_3}.\mathrm{FIN}(M_*, \sigma_*)$. The distribution of signatures that $\mathscr{A}$ is given, and of the keys underlying them, is as in G_2, which explains Eq. (4.3).

Note that the constructed adversaries having access to oracle FO in their games is important to their ability to simulate $\mathscr{A}$ faithfully.

With regard to the costs (number of queries, running time) of the constructed adversaries, recall that we have defined these as the costs in the execution of the adversary with the game that the adversary is playing, so for example the number of queries to FO includes the ones made by algorithms executed in the game. When this is taken into account, queries to FO are preserved, and the other claims are direct. ▌

Security of **JCl**. We have now reduced the security of $\overline{\mathsf{DS}}$ to that of DS^*. To further reduce the security of DS^* to that of DS, we give a general result on clamping. Let $\mathscr{K} = \mathsf{OUT}(\mathsf{DS.SK})$ and let $\mathtt{CF}\colon \{0,1\}^k \to \mathscr{K}$ be a clamping function. As per terminology in Section 4.2, recall that $\mathsf{Img}(\mathtt{CF}) = \{\mathtt{CF}(\overline{sk}) : |\overline{sk}| = k\} \subseteq \mathscr{K}$ is the image of the clamping function, and $\mathtt{CF}$ is regular if every $y \in \mathsf{Img}(\mathtt{CF})$ has the same number of pre-images under $\mathtt{CF}$.

Theorem 14. *Let* DS *be a signature scheme such that* $\mathsf{DS.SK}$ *draws its signing key* $sk \leftarrow\!\!\$\, \mathscr{K}$ *at random from a set* $\mathscr{K}$*. Let* $\mathtt{CF}\colon \{0,1\}^k \to \mathscr{K}$ *be a regular clamping function. Let* $\delta = |\mathsf{Img}(\mathtt{CF})|/|\mathscr{K}| > 0$*. Let* $\mathsf{DS}^* = \mathbf{JCl}[\mathsf{DS},\mathtt{CF}]$ *be the signature scheme obtained by the just-clamp transform. Let* $\mathbf{FF}\colon \mathsf{SS} \to \mathsf{DS.FS}$ *be any functor. Let* $\mathscr{B}$ *be an adversary attacking the* $\mathbf{G}^{\mathrm{uf}}$ *security of* DS^**. Then*

$$\mathbf{Adv}^{\mathrm{uf}}_{\mathsf{DS}^*,\mathbf{FF}}(\mathscr{B}) \leq (1/\delta) \cdot \mathbf{Adv}^{\mathrm{uf}}_{\mathsf{DS},\mathbf{FF}}(\mathscr{B}) .$$

Proof of Theorem 14: We consider running $\mathscr{B}$ in game $\mathbf{G}^{\mathrm{uf}}_{\mathsf{DS},\mathbf{FF}}$, where the signing key is $sk \leftarrow\!\!\$\, \mathscr{K}$. With probability δ we have $sk \in \mathsf{Img}(\mathtt{CF})$. Due to the regularity of $\mathtt{CF}$, key sk now has the same distribution as a key $\mathtt{CF}(\overline{sk})$ for $\overline{sk} \leftarrow\!\!\$\, \{0,1\}^k$ drawn in game $\mathbf{G}^{\mathrm{uf}}_{\mathsf{DS}^*,\mathbf{FF}}$. Thus $\mathbf{Adv}^{\mathrm{uf}}_{\mathsf{DS},\mathbf{FF}}(\mathscr{B}) \geq \delta \cdot \mathbf{Adv}^{\mathrm{uf}}_{\mathsf{DS}^*,\mathbf{FF}}(\mathscr{B})$. ▌

4.5 Security of EdDSA

The Schnorr scheme. Let the prime-order group $\mathbb{G}_{\mathsf{p}}$ of k-bit strings with generator $\mathtt{B}$ be as described in Section 4.2. The algorithms of the Schnorr signature scheme $\mathsf{DS} = \mathsf{Sch}$ are shown on the left in Figure 4.4. The function space $\mathsf{DS.FS}$ is $\mathsf{FUNC}((,\{0,1\})^*, \mathbb{Z}_{\mathsf{p}})$. (Implementations may use a hash function that outputs a string and embed the result in $\mathbb{Z}_{\mathsf{p}}$ but following prior proofs [3] we view the hash function as directly mapping into $\mathbb{Z}_{\mathsf{p}}$.) Verification is parameterized by an algorithm **VF** to allow us to consider strict and permissive verification in a modular way.

```
$\mathsf{DS.SK}$:
  $s \leftarrow\!\!\$\, \mathbb{Z}_{\mathsf{p}}$
  Return $s$
$\mathsf{DS.PK}(s)$:
  $\mathtt{A} \leftarrow s \cdot \mathtt{B}$ ; Return $\mathtt{A}$
$\mathsf{DS.Sign}[\mathsf{HH}](s, \mathtt{A}, M)$:
  $r \leftarrow\!\!\$\, \mathbb{Z}_{\mathsf{p}}$ ; $\mathtt{R} \leftarrow r \cdot \mathtt{B}$
  $c \leftarrow \mathsf{HH}(\mathtt{R} \| \mathtt{A} \| M)$
  $z \leftarrow (sc + r) \mod \mathsf{p}$
  Return $(\mathtt{R}, z)$
$\mathsf{DS.Vf}[\mathsf{HH}](\mathtt{A}, M, \sigma)$:
  $(\mathtt{R}, z) \leftarrow \sigma$
  $c \leftarrow \mathsf{HH}(\mathtt{R} \| \mathtt{A} \| M)$
  Return $\mathrm{VF}(\mathtt{A}, \mathtt{R}, c, z)$
```

```
$\overline{\mathsf{DS}}.\mathsf{SK}$:
  $sk \leftarrow\!\!\$\, \{0,1\}^k$ ; Return $sk$
$\overline{\mathsf{DS}}.\mathsf{PK}(sk)$:
  $\mathsf{e}_1 \| \mathsf{e}_2 \leftarrow \mathsf{HH}_1(sk)$ ; $s \leftarrow \mathsf{CF}(\mathsf{e}_1)$
  $\mathtt{A} \leftarrow s \cdot \mathtt{B}$ ; Return $\mathtt{A}$
$\overline{\mathsf{DS}}.\mathsf{Sign}[\mathsf{HH}](sk, \mathtt{A}, M)$:
  $\mathsf{e}_1 \| \mathsf{e}_2 \leftarrow \mathsf{HH}_1(sk)$ ; $s \leftarrow \mathsf{CF}(\mathsf{e}_1)$
  $r \leftarrow \mathsf{HH}_2(\mathsf{e}_2, M)$ ; $\mathtt{R} \leftarrow r \cdot \mathtt{B}$
  $c \leftarrow \mathsf{HH}_3(\mathtt{R} \| \mathtt{A} \| M)$
  $z \leftarrow (sc + r) \mod \mathsf{p}$
  Return $(\mathtt{R}, z)$
$\overline{\mathsf{DS}}.\mathsf{Vf}[\mathsf{HH}](\mathtt{A}, M, \sigma)$:
  $(\mathtt{R}, z) \leftarrow \sigma$
  $c \leftarrow \mathsf{HH}_3(\mathtt{R} \| \mathtt{A} \| M) \mod \mathsf{p}$
  Return $\mathrm{VF}(\mathtt{A}, \mathtt{R}, c, z)$
```

```
$\mathtt{CF}(e)$   // $e \in \{0,1\}^k$:
  $t \leftarrow 2^{k-2}$
  for $i \in [4..k-2]$
      $t \leftarrow t + 2^{i-1} \cdot e[i]$
  $s \leftarrow t \mod \mathsf{p}$
  return $s$
```

```
$\mathrm{sVF}(\mathtt{A}, \mathtt{R}, c, z)$:
  Return $(z \cdot \mathtt{B} = c \cdot \mathtt{A} + \mathtt{R})$

$\mathrm{pVF}(\mathtt{A}, \mathtt{R}, c, z)$:
  Return $2^{\mathsf{f}}(z \cdot \mathtt{B}) = 2^{\mathsf{f}}(c \cdot \mathtt{A} + \mathtt{R})$
```

Figure 4.4. Top Left: the Schnorr scheme. **Top Right:** The EdDSA scheme. **Bottom Left:** EDDSA clamping function (generalized for any k; in the original definition, $k = 256$). **Bottom Right:** Strict and Permissive verification algorithms as choices for VF.

The corresponding choices of verification algorithms are at the bottom of Figure 4.4. The signing randomness space is $\mathsf{DS.SR} = \mathbb{Z}_{\mathsf{p}}$.

Schnorr signatures have a few variants that differ in details. In Schnorr's paper [191], the challenge is $c = \mathsf{HH}(\mathtt{R} \| M) \bmod \mathsf{p}$. Our inclusion of the public key in the input to HH follows Bernstein [45] and helps here because it is what EdDSA does. It doesn't affect security. (The security of the scheme that includes the public key in the hash input is implied by the security of the one that doesn't via a reduction that includes the public key in the message.) Also in [191], the signature is (c, z). The version we use, where it is $(\mathtt{R}, z)$, is from [3]. However, BBSS [20] shows that these versions have equivalent security.

The EdDSA scheme. Let the prime-order group $\mathbb{G}_{\mathsf{p}}$ of k-bit strings with generator $\mathtt{B}$ be as before and assume $2^{k-5} < \mathsf{p} < 2^k$. Let $\mathtt{CF}$: $\{0,1\}^k \to \mathbb{Z}_{\mathsf{p}}$ be the clamping function shown at the bottom of Figure 4.4. The algorithms of the scheme $\overline{\mathsf{DS}}$ are shown on the right side of Figure 4.4. The key length is k. As before, the verification algorithm VF is a parameter. The HH available to the algorithms defines three sub-functions. The first, $\mathsf{HH}_1\colon \{0,1\}^k \to \{0,1\}^{2k}$, is used at lines 2,4, where its output is parsed into k-bit halves. The second, $\mathsf{HH}_2\colon \{0,1\}^k \times \{0,1\}^* \to \mathbb{Z}_{\mathsf{p}}$, is used at line 5 for de-randomization. The third, $\mathsf{HH}_3\colon \{0,1\}^* \to \mathbb{Z}_{\mathsf{p}}$, plays the role of the function HH for the Schnorr schemes. Formally, $\overline{\mathsf{DS}}.\mathsf{FS}$ is the arity-3 function space consisting of all HH mapping as just indicated.

In [50, **?**], the output of the clamping is an integer that (in our notation) is in the range $2^{k-2}, \dots, 2^{k-1} - 8$. When used in the scheme, however, it is (implicitly) modulo p. It is convenient for our analysis, accordingly, to define $\mathtt{CF}$ to be the result modulo p of the actual clamping. Note that in EdDSA the prime p has magnitude a little more than 2^{k-4} and less than 2^{k-3}.

There are several versions of EdDSA depending on the choice for verification algorithms: strict, permissive or batch VF. We specify the first two choices in Figure 4.4. Our results hold for all choices of VF, meaning EdDSA is secure with respect to VF assuming $\mathsf{Schnorr}$ is secure with respect to VF. It is in order to make this general claim that we abstract out VF.

Security of EdDSA with independent ROs. As a warm-up, we show security of EdDSA when the three functions it uses are independent random oracles, the setting assumed by BCJZ [**?**]. However, while they assume hardness of DL, our result is more general, assuming only security of $\mathsf{Schnorr}$ with a monolithic random oracle. We can then use known results on $\mathsf{Schnorr}$ [184, 3] to recover the result of BCJZ [**?**], but the proof is simpler and more modular. Also, other known results on $\mathsf{Schnorr}$ [**?**, 32, **?**] can be applied to get better bounds. Following this, we will turn to the "real" case, where the three functions are all **MD** with a random compression function.

The Theorem below is for a general prime $\mathsf{p} > 2^{k-5}$ but in EdDSA the prime is $2^{k-4} < \mathsf{p} < 2^{k-3}$ so the value of δ below is $\delta = 2^{k-5}/\mathsf{p} > 2^{k-5}/2^{k-3} = 1/4$, so the factor $1/\delta$ is ≤ 4. We capture the three functions of EdDSA being independent random oracles by setting functor $\mathbf{P}$ below to the identity functor, and similarly capture $\mathsf{Schnorr}$ being with a monolithic random oracle by setting $\mathbf{F}_{\mathrm{id}}$ to be the identity functor.

Theorem 15. *Let $\mathsf{DS} = \mathsf{Sch}$ be the $\mathsf{Schnorr}$ signature scheme of Figure 4.4. Let $\mathtt{CF}\colon \{0,1\}^k \to \mathbb{Z}_{\mathsf{p}}$ be the clamping function of Figure 4.4. Assume $\mathsf{p} > 2^{k-5}$ and let $\delta = 2^{k-5}/\mathsf{p}$. Let $\overline{\mathsf{DS}} = \mathbf{DR}[\mathsf{DS}, \mathtt{CF}]$ be the EdDSA signature scheme. Let $\mathbf{F}_{\mathrm{id}}\colon \mathrm{FUNC}((,\{0,1\})^*, \mathbb{Z}_{\mathsf{p}}) \to \mathrm{FUNC}((,\{0,1\})^*, \mathbb{Z}_{\mathsf{p}})$ be the identity functor. Let $\mathbf{P}\colon \overline{\mathsf{DS}}.\mathsf{FS} \to \overline{\mathsf{DS}}.\mathsf{FS}$ be the identity functor. Let $\mathscr{A}$ be an adversary attacking*

```
Functor $\mathbf{S}_1[\mathsf{hh}](sk)$:   // $|sk| = k$
  $\varepsilon \leftarrow \mathbf{MD}[\mathsf{hh}](sk)$ ; Return $e$   // $|e| = 2k$
Functor $\mathbf{S}_2[\mathsf{hh}](e_2, M)$:   // $|e_2| = k$
  Return $\mathbf{MD}[\mathsf{hh}](e_2 \| M)$ mod $\mathsf{p}$
Functor $\mathbf{S}_3[\mathsf{hh}](X)$:   // also called $\mathbf{Mod\text{-}MD}$
  Return $\mathbf{MD}[\mathsf{hh}](X)$ mod $\mathsf{p}$
```

Figure 4.5. The arity-3 functor $\mathbf{S}$ for EdDSA. Here $\mathsf{hh}\colon \{0,1\}^{b+2k} \to \{0,1\}^{2k}$ is a compression function.

the $\mathbf{G}^{\mathrm{uf}}$ *security of* $\overline{\mathsf{DS}}$*. Then there is an adversary* $\mathscr{B}$ *such that*

$$\mathbf{Adv}^{\mathrm{uf}}_{\mathsf{DS},\mathbf{P}}(\mathscr{A}) \le (1/\delta) \cdot \mathbf{Adv}^{\mathrm{uf}}_{\mathsf{DS},\mathbf{F}_{\mathrm{id}}}(\mathscr{B}) + \frac{2 \cdot \mathrm{Q}^{\mathscr{A}}_{\mathrm{FO}}}{2^k} .$$

Adversary $\mathscr{B}$ *preserves the queries and running time of* $\mathscr{A}$*.*

Proof of Theorem 15: Let $\mathsf{DS}^* = \mathbf{JCl}[\mathsf{Sch}, \mathtt{CF}]$. By Theorem 13, we have

$$\mathbf{Adv}^{\mathrm{uf}}_{\mathsf{DS},\mathbf{P}}(\mathscr{A}) \le \mathbf{Adv}^{\mathrm{prg}}_{\mathbf{P}_1}(\mathscr{A}_1) + \mathbf{Adv}^{\mathrm{prf}}_{\mathbf{P}_2}(\mathscr{A}_2) + \mathbf{Adv}^{\mathrm{uf}}_{\mathsf{DS}^*,\mathbf{P}_3}(\mathscr{A}_3) .$$

It is easy to see that

$$\begin{aligned} \mathbf{Adv}^{\mathrm{prg}}_{\mathbf{P}_1}(\mathscr{A}_1) &\le \frac{\mathrm{q}^{\mathscr{A}_1}_{\mathrm{FO}}}{2^k} \le \frac{\mathrm{Q}^{\mathscr{A}}_{\mathrm{FO}}}{2^k} \\ \mathbf{Adv}^{\mathrm{prf}}_{\mathbf{P}_2}(\mathscr{A}_2) &\le \frac{\mathrm{q}^{\mathscr{A}_2}_{\mathrm{FO}}}{2^k} \le \frac{\mathrm{Q}^{\mathscr{A}}_{\mathrm{FO}}}{2^k} . \end{aligned}$$

Under the assumption $\mathsf{p} > 2^{k-5}$ made in the theorem, BCJZ [?] established that $|\mathsf{Img}(\mathtt{CF})| = 2^{k-5}$. So $|\mathsf{Img}(\mathtt{CF})|/|\mathbb{Z}_{\mathsf{p}}| = 2^{k-5}/\mathsf{p} = \delta$. Let $\mathscr{B} = \mathscr{A}_3$ and note that $\mathbf{P}_3 = \mathbf{F}_{\mathrm{id}}$. So by Theorem 14 we have

$$\mathbf{Adv}^{\mathrm{uf}}_{\mathsf{DS}^*,\mathbf{P}_3}(\mathscr{A}_3) \le (1/\delta) \cdot \mathbf{Adv}^{\mathrm{uf}}_{\mathsf{DS},\mathbf{F}_{\mathrm{id}}}(\mathscr{B}) . \tag{4.4}$$

Collecting terms, we obtain the claimed bound stated in Theorem 15. ▌

Analysis of the $\mathbf{S}$ functor. Let $\overline{\mathsf{DS}}$ be the result of the $\mathbf{DR}$ transform applied to Sch and a clamping function $\mathtt{CF}\colon \{0,1\}^k \to \mathbb{Z}_{\mathsf{p}}$. Security of EdDSA is captured as security in game $\mathbf{G}^{\mathrm{uf}}_{\mathsf{DS},\mathbf{S}}$ when $\mathbf{S}$ is the functor that builds the component hash functions in the way that EdDSA does, namely from a MD-hash function. To evaluate this security, we start by defining the

```
Games $\mathrm{G}_0$, $\boxed{\mathrm{G}_1}$

INIT:
  $sk \leftarrow\!\!\$\, \{0,1\}^k$ ; $e \leftarrow\!\!\$\, \{0,1\}^{2k}$
  Return $e$

$\mathrm{FO}(X)$:
  If $\mathrm{FT}[X] \neq \bot$ then return $\mathrm{FT}[X]$
  $Y \leftarrow\!\!\$\, \{0,1\}^{2k}$
  If $X = IV \| sk \| P$ then $\mathsf{bad} \leftarrow \mathsf{true}$ ; $\boxed{Y \leftarrow e}$
  $\mathrm{FT}[X] \leftarrow Y$ ; Return $\mathrm{FT}[X]$

FIN$(c')$:
  Return $(c' = 1)$
```

Figure 4.6. Games G_0 and G_1 in the proof of Lemma 4. Boxed code is only in G_1.

functor $\mathbf{S}$ in Figure 4.5. It is an arity-3 functor, and we separately specify $\mathbf{S}_1, \mathbf{S}_2, \mathbf{S}_3$. (Functor $\mathbf{S}_3$ will be called **Mod-MD** in later analyses.) The starting space, from which hh is drawn, is $\mathrm{FUNC}((,\{0,1\})^{b+2k},\{0,1\}^{2k})$, the set of compression functions. The prime p is as before, and is public.

We want to establish the three assumptions of Theorem 13. Namely: (1) $\mathbf{S}_1$ is PRG-secure (2) $\mathbf{S}_2$ is PRF secure and (3) security holds in game $\mathbf{G}^{\mathrm{uf}}_{\mathsf{Sch}^*,\mathbf{S}_3}$ where $\mathsf{Sch}^* = \mathbf{JCl}[\mathsf{Sch}, \mathtt{CF}]$. Bridging from Sch^* to Sch itself will use Theorem 14.

Lemma 4. *Let functor* $\mathbf{S}_1\colon \mathrm{FUNC}((,\{0,1\})^{b+2k},\{0,1\}^{2k}) \to \mathrm{FUNC}((,\{0,1\})^{k},\{0,1\}^{2k})$ *be defined as in Figure 4.5. Let* $\mathscr{A}_1$ *be an adversary. Then*

$$\mathbf{Adv}^{\mathrm{prg}}_{\mathbf{S}_1}(\mathscr{A}_1) \le \frac{\mathrm{q}^{\mathscr{A}_1}_{\mathrm{FO}}}{2^k} \le \frac{\mathrm{Q}^{\mathscr{A}_1}_{\mathrm{FO}}}{2^k}\,. \tag{4.5}$$

Proof of Lemma 4: Since the input sk to $\mathbf{S}_1[\mathsf{hh}]$ is k-bits long, the **MD** transform defined in Section 4.3 only iterates once and the output is $e = \mathsf{hh}(IV \| sk \| P)$, for padding $P \in \{0,1\}^{3k}$ and initialization vector $IV \in \{0,1\}^{2k}$ that are fixed and known. Now consider the games in Figure 4.6, where the boxed code is only in G_1. Then we have

$$\begin{aligned}
\mathbf{Adv}^{\mathrm{prg}}_{\mathbf{S}_1}(\mathscr{A}_1) &= \Pr[\mathrm{G}_1(\mathscr{A}_1)] - \Pr[\mathrm{G}_0(\mathscr{A}_1)] \\
&\le \Pr[\mathrm{G}_0(\mathscr{A}_1) \text{ sets } \mathsf{bad}] \\
&\le \frac{\mathrm{Q}^{\mathscr{A}_1}_{\mathrm{FO}}}{2^k}\,.
\end{aligned}$$

The second line above is by the Fundamental Lemma of Game Playing, which applies since $\mathrm{G}_0, \mathrm{G}_1$ are identical-until-bad. ▌

We turn to PRF security of the $\mathbf{S}_2$ functor. Note that the construction is what BRT called AMAC [29]. They proved its PRF security by a combination of standard-model and ROM results. First they showed AMAC is PRF-secure if the compression function hh is PRF-secure under leakage of a certain function of the key. Then they show that ideal compression functions have this PRF-under-leakage security. Putting this together implies PRF security of $\mathbf{S}_2$. However, we found it hard to put the steps and Lemmas in BRT together to get a good, concrete bound for the PRF security of $\mathbf{S}_2$. Instead we give a direct proof, with an explicit bound, using our result on the indifferentiability of **Mod-MD** from Theorem 17 together with the indifferentiability composition theorem [156].

Lemma 5. *Let functor* $\mathbf{S}_2\colon \mathsf{FUNC}((,\{0,1\})^{b+2k},\{0,1\}^{2k}) \to \mathsf{FUNC}((,\{0,1\})^{k} \times \{0,1\}^{*},\mathbb{Z}_{\mathsf{p}})$ *be defined as in Figure 4.5. Let ℓ be an integer such that all messages queried to* FO *are no more than $b \cdot (\ell - 1) - k$ bits long. Let $\mathscr{A}_2$ be an adversary. Then*

$$\mathbf{Adv}^{\mathrm{prf}}_{\mathbf{S}_2}(\mathscr{A}_2) \leq \frac{\mathsf{Q}^{\mathscr{A}_2}_{\mathrm{FO}}}{2^k} + \frac{2\mathsf{p}(\mathsf{q}^{\mathscr{A}_2}_{\mathrm{FO}} + \ell\mathsf{Q}^{\mathscr{A}_2}_{\mathrm{FN}})}{2^{2k}} + \frac{(\mathsf{q}^{\mathscr{A}_2}_{\mathrm{FO}} + \ell\mathsf{Q}^{\mathscr{A}_2}_{\mathrm{FN}})^2}{2^{2k}} + \frac{\mathsf{p}\mathsf{q}^{\mathscr{A}_2}_{\mathrm{FO}} \cdot \ell\mathsf{Q}^{\mathscr{A}_2}_{\mathrm{FN}}}{2^{2k}}.$$

Proof of Lemma 5: In Section 4.6, we prove the indifferentiability of functor $\mathbf{S}_3$ (c.f. Figure 4.5), which we also call **Mod-MD**. Define $\mathbf{R}\colon \mathsf{FUNC}((,\{0,1\})^{*},\mathbb{Z}_{\mathsf{p}}) \to \mathsf{FUNC}((,\{0,1\})^{k} \times \{0,1\}^{*},\mathbb{Z}_{\mathsf{p}})$ to be the identity functor such that $\mathbf{R}[\mathsf{HH}](x,y) = \mathsf{HH}(x \,\|\, y)$ for all x, y, HH in the appropriate domains. Notice that when $\mathbf{R}$ is given access to the **Mod-MD** functor as its oracle, the resulting functor is exactly $\mathbf{S}_2$. Using this property, we will reduce the PRF security of functor $\mathbf{S}_2$ to the indifferentiability of **Mod-MD**.

For any simulator algorithm Sim, the indifferentiability composition theorem [156] grants the existence of distinguisher $\mathscr{D}$ and adversary $\mathscr{A}_5$ such that

$$\mathbf{Adv}^{\mathrm{prf}}_{\mathbf{S}_2}(\mathscr{A}_2) \leq \mathbf{Adv}^{\mathrm{prf}}_{\mathbf{R}}(\mathscr{A}_5) + \mathbf{Adv}^{\mathsf{indiff}}_{\mathbf{Mod\text{-}MD},\mathsf{Sim}}(\mathscr{D}).$$

We let Sim be the simulator guaranteed by Theorem 17 and separately bound each of these terms. Adversary $\mathscr{A}_5$ simulates the PRF game for its challenger $\mathscr{A}_2$ by forwarding all FN queries to its own FN oracle and answering FO queries using the simulator, which has access to the FO oracle of $\mathscr{A}_5$. Since the simulator is efficient and makes at most one query to its oracle each time it is run, we can say the runtime of $\mathscr{A}_5$ is approximately the same as that of $\mathscr{A}_2$. $\mathscr{A}_5$ makes the same number of FN and FO queries as $\mathscr{A}_2$.

Next, we want to compute $\mathbf{Adv}^{\mathsf{prf}}_{\mathbf{R}}(\mathscr{A}_5)$. When $\mathbf{R}$ is evaluated with access to a random function hh, its outputs are random unless the adversary makes a relevant query involving the secret key. The adversary can only distinguish if the output of FN is randomly sampled or from $\mathbf{R}[\mathsf{hh}]$ if it queries FO on the k-bit secret key (e_2), which has probability $\frac{1}{2^k}$ for a single query. Taking a union bound over all FO queries, we have

$$\mathbf{Adv}^{\mathsf{prf}}_{\mathbf{R}}(\mathscr{A}_5) \leq \frac{\mathrm{Q}^{\mathscr{A}_2}_{\mathrm{FO}}}{2^k}.$$

Distinguisher $\mathscr{D}$ simulates the PRF game for $\mathscr{A}_2$, by replacing functor **Mod-MD** with its own Priv oracle within the FN oracle and forwarding $\mathscr{A}_2$'s direct FO queries to Pub. $\mathscr{D}$ hence makes $\mathrm{Q}^{\mathrm{FN}}_{\mathscr{A}_2}$ queries to Priv of maximum length $b\cdot(\ell-1)$ and $\mathrm{q}^{\mathrm{FO}}_{\mathscr{A}_2}$ to Pub. To bound the second term, we apply Theorem 17 on the indifferentiability of shrink-MD transforms. This theorem is parameterized by two numbers γ and ε; in Section 4.6, we show that **Mod-MD** belongs to the shrink-MD class for $\gamma = \lfloor \frac{2^{2k}}{\mathsf{p}} \rfloor$ and $\varepsilon = \frac{\mathsf{p}}{2^{2k}}$. Then the theorem gives

$$\mathbf{Adv}^{\mathsf{indiff}}_{\mathbf{Mod\text{-}MD},\mathsf{Sim}}(\mathscr{D}) \leq 2(\mathrm{Q}^{\mathscr{D}}_{\mathrm{PUB}} + \ell\mathrm{Q}^{\mathscr{D}}_{\mathrm{PRIV}})\varepsilon + \frac{(\mathrm{Q}^{\mathscr{D}}_{\mathrm{PUB}} + \ell\mathrm{Q}^{\mathscr{D}}_{\mathrm{PRIV}})^2}{2^{2k}} + \frac{\mathrm{Q}^{\mathscr{D}}_{\mathrm{PUB}} \cdot \ell\mathrm{Q}^{\mathscr{D}}_{\mathrm{PRIV}}}{\gamma}.$$

By substituting $\mathrm{Q}^{\mathscr{D}}_{\mathrm{PUB}} = \mathrm{q}^{\mathscr{A}_2}_{\mathrm{FO}}$ and $\mathrm{Q}^{\mathscr{D}}_{\mathrm{PRIV}} = \mathrm{Q}^{\mathscr{A}_2}_{\mathrm{FN}}$, we obtain the bound stated in the theorem.▌

Finally we turn to $\mathbf{S}_3$. The following considers the UF security of $\mathsf{DS}^* = \mathbf{JCl}[\mathsf{Sch}, \mathtt{CF}]$ with the hash function being an MD one, meaning with $\mathbf{S}_3$, and reduces this to the UF security of the same scheme with the hash function being a monolithic random oracle. Formally, the latter is captured by game $\mathbf{G}^{\mathsf{uf}}_{\mathsf{DS}^*,\mathbf{R}}$ where $\mathbf{R}$ is the identity functor. One route to this result is to exploit the public-indifferentiability of **MD** established by DRS [89]. However we found it simpler to give a direct proof and bound based on our Theorem 17.

Lemma 6. *Let functor* $\mathbf{S}_3$: $\mathsf{FUNC}((,\{0,1\})^{b+2k},\{0,1\}^{2k}) \to \mathsf{FUNC}((,\{0,1\})^*,\mathbb{Z}_\mathsf{p})$ *be defined as in Figure 4.5. Assume* $2^k > \mathsf{p}$. *Let* $\mathsf{DS}^* = \mathbf{JCl}[\mathsf{Sch},\mathtt{CF}]$ *where* $\mathtt{CF}\colon \{0,1\}^k \to \mathbb{Z}_p$ *is a clamping function. Let* $\mathbf{R}$: $\mathsf{FUNC}((,\{0,1\})^*,\mathbb{Z}_\mathsf{p}) \to \mathsf{FUNC}((,\{0,1\})^*,\mathbb{Z}_\mathsf{p})$ *be the identity functor, meaning* $\mathbf{R}[\mathsf{HH}] = \mathsf{HH}$. *Let* $\mathscr{A}_3$ *be a* $\mathbf{G}^{\mathrm{uf}}$ *adversary and let* ℓ *be an integer such that the maximum message length* $\mathscr{A}_3$ *queries to* Sign *is at most* $b\cdot(\ell-1)-2k$ *bits. Then we can construct adversary* $\mathscr{A}_4$ *such that*

$$\mathbf{Adv}^{\mathsf{uf}}_{\mathsf{DS}^*,\mathbf{S}_3}(\mathscr{A}_3) \le \mathbf{Adv}^{\mathsf{uf}}_{\mathsf{DS}^*,\mathbf{R}}(\mathscr{A}_4) + \frac{2\mathsf{p}(\mathrm{q}^{\mathscr{A}_3}_{\mathrm{FO}} + \ell \mathrm{Q}^{\mathscr{A}_3}_{\mathrm{SIGN}})}{2^{2k}} \tag{4.6}$$

$$+ \frac{(\mathrm{q}^{\mathscr{A}_3}_{\mathrm{FO}} + \ell \mathrm{Q}^{\mathscr{A}_3}_{\mathrm{SIGN}})^2}{2^{2k}} + \frac{\mathsf{p}\mathrm{q}^{\mathscr{A}_3}_{\mathrm{FO}} \cdot \ell \mathrm{Q}^{\mathscr{A}_3}_{\mathrm{SIGN}}}{2^{2k}}\,. \tag{4.7}$$

Adversary $\mathscr{A}_4$ *has approximately equal runtime and query complexity to* $\mathscr{A}_3$.

Proof of Lemma 6: Again, we rely on the indifferentiability of functor $\mathbf{S}_3 = \mathbf{Mod\text{-}MD}$, as shown in Section 4.6. The general indifferentiability composition theorem [156] states that for any simulator Sim and adversary $\mathscr{A}_3$, there exist distinguisher $\mathscr{D}$ and adversary $\mathscr{A}_4$ such that

$$\mathbf{Adv}^{\mathsf{uf}}_{\mathsf{DS}^*,\mathbf{S}_3}(\mathscr{A}_3) \le \mathbf{Adv}^{\mathsf{uf}}_{\mathsf{DS}^*,\mathbf{R}}(\mathscr{A}_4) + \mathbf{Adv}^{\mathsf{indiff}}_{\mathbf{S}_3,\mathsf{Sim}}(\mathscr{D}).$$

Let Sim be the simulator whose existence is implied by Theorem 17. The distinguisher runs the unforgeability game for its adversary, replacing $\mathbf{S}_3[\mathrm{FO}]$ in scheme algorithms and adversarial FO queries with its Priv and Pub oracles respectively. It makes $\mathrm{q}^{\mathscr{A}_3}_{\mathrm{FO}}$ queries to Pub and $\mathrm{Q}^{\mathscr{A}_3}_{\mathrm{SIGN}}$ queries to Priv, and the maximum length of any query to Priv is $b\cdot(\ell-1)$ bits because each element of group $\mathbb{G}_\mathsf{p}$ is a k-bit string (c.f. Section 4.2). We apply Theorem 17 to obtain the bound

$$\mathbf{Adv}^{\mathsf{indiff}}_{\mathbf{S}_3,\mathsf{Sim}}(\mathscr{D}) \le 2(\mathrm{q}^{\mathscr{A}_3}_{\mathrm{FO}} + \ell \mathrm{Q}^{\mathscr{A}_3}_{\mathrm{SIGN}})\varepsilon + \frac{(\mathrm{q}^{\mathscr{A}_3}_{\mathrm{FO}} + \ell \mathrm{Q}^{\mathscr{A}_3}_{\mathrm{SIGN}})^2}{2^{2k}} + \frac{\mathrm{q}^{\mathscr{A}_3}_{\mathrm{FO}} \cdot \ell \mathrm{Q}^{\mathscr{A}_3}_{\mathrm{SIGN}}}{\gamma}.$$

Adversary $\mathscr{A}_4$ is a wrapper for $\mathscr{A}_3$, which answers all of its queries to FO by running Sim with access to its own FO oracle; since the simulator runs in constant time and makes only one query to its oracle, the runtime and query complexity approximately equal those of $\mathscr{A}_3$.

Substituting $\frac{1}{\gamma} \ge \frac{\mathsf{p}}{2^{2k}}$ and $\varepsilon = \frac{\mathsf{p}}{2^{2k}}$ gives the bound.▮

Security of EdDSA with MD. We now want to conclude security of EdDSA, with an MD-hash function, assuming security of $\mathsf{Schnorr}$ with a monolithic random oracle. The Theorem is for a general prime p in the range $2^k > \mathsf{p} > 2^{k-5}$ but in EdDSA the prime is $2^{k-4} < \mathsf{p} < 2^{k-3}$ so the value of δ below is $\delta = 2^{k-5}/\mathsf{p} > 2^{k-5}/2^{k-3} = 1/4$, so the factor $1/\delta$ is ≤ 4. Again recall our convention that query counts of an adversary include those made by oracles in its game, implying for example that $\mathrm{Q}^{\mathscr{A}}_{\mathrm{FO}} \geq \mathrm{Q}^{\mathscr{A}}_{\mathrm{SIGN}}$.

Theorem 16. *Let* $\mathsf{DS} = \mathsf{Sch}$ *be the* $\mathsf{Schnorr}$ *signature scheme of Figure 4.4. Let* $\mathtt{CF}\colon \{0,1\}^k \to \mathbb{Z}_{\mathsf{p}}$ *be the clamping function of Figure 4.4. Assume* $2^k > \mathsf{p} > 2^{k-5}$ *and let* $\delta = 2^{k-5}/\mathsf{p}$*. Let* $\overline{\mathsf{DS}} = \mathbf{DR}[\mathsf{DS}, \mathtt{CF}]$ *be the* EdDSA *signature scheme. Let* $\mathbf{R}\colon \mathrm{FUNC}((,\{0,1\})^*, \mathbb{Z}_{\mathsf{p}}) \to \mathrm{FUNC}((,\{0,1\})^*, \mathbb{Z}_{\mathsf{p}})$ *be the identity functor. Let* $\mathbf{S}$ *be the functor of Figure 4.5. Let* $\mathscr{A}$ *be an adversary attacking the* $\mathbf{G}^{\mathrm{uf}}$ *security of* $\overline{\mathsf{DS}}$*. Again let* $b \cdot (\ell - 1) - 2k$ *be the maximum length in bits of a message input to* Sign*. Then there is an adversary* $\mathscr{B}$ *such that*

$$\mathbf{Adv}^{\mathrm{uf}}_{\overline{\mathsf{DS}},\mathbf{S}}(\mathscr{A}) \leq (1/\delta) \cdot \mathbf{Adv}^{\mathrm{uf}}_{\mathsf{DS},\mathbf{R}}(\mathscr{B}) + \frac{\mathrm{Q}^{\mathscr{A}}_{\mathrm{FO}}}{2^{k-1}} + \frac{\mathsf{p}(\mathsf{q}^{\mathscr{A}}_{\mathrm{FO}} + \ell \mathrm{Q}^{\mathscr{A}}_{\mathrm{SIGN}})}{2^{2k-2}}$$
$$+ \frac{(\mathsf{q}^{\mathscr{A}}_{\mathrm{FO}} + \ell \mathrm{Q}^{\mathscr{A}_2}_{\mathrm{SIGN}})^2}{2^{2k-1}} + \frac{\mathsf{p}\mathsf{q}^{\mathscr{A}}_{\mathrm{FO}} \cdot \ell \mathrm{Q}^{\mathscr{A}}_{\mathrm{SIGN}}}{2^{2k-1}} \,.$$

Adversary $\mathscr{B}$ *preserves the queries and running time of* $\mathscr{A}$*.*

Proof of Theorem 16: Let $\mathsf{DS}^* = \mathbf{JCl}[\mathsf{Sch}, \mathtt{CF}]$. By Theorem 13, we have

$$\mathbf{Adv}^{\mathrm{uf}}_{\overline{\mathsf{DS}},\mathbf{S}}(\mathscr{A}) \leq \mathbf{Adv}^{\mathrm{prg}}_{\mathbf{S}_1}(\mathscr{A}_1) + \mathbf{Adv}^{\mathrm{prf}}_{\mathbf{S}_2}(\mathscr{A}_2) + \mathbf{Adv}^{\mathrm{uf}}_{\mathsf{DS}^*,\mathbf{S}_3}(\mathscr{A}_3).$$

Now applying Lemma 4, we have

$$\mathbf{Adv}^{\mathrm{prg}}_{\mathbf{S}_1}(\mathscr{A}_1) \leq \frac{\mathrm{Q}^{\mathscr{A}}_{\mathrm{FO}}}{2^k} \,.$$

Applying Lemma 5, we have

$$\mathbf{Adv}^{\mathrm{prf}}_{\mathbf{S}_2}(\mathscr{A}_2) \leq \frac{\mathrm{Q}^{\mathscr{A}_2}_{\mathrm{FO}}}{2^k} + \frac{2\mathsf{p}(\mathsf{q}^{\mathscr{A}_2}_{\mathrm{FO}} + \ell \mathrm{Q}^{\mathscr{A}_2}_{\mathrm{FN}})}{2^{2k}} + \frac{(\mathsf{q}^{\mathscr{A}_2}_{\mathrm{FO}} + \ell \mathrm{Q}^{\mathscr{A}_2}_{\mathrm{FN}})^2}{2^{2k}} + \frac{\mathsf{p}\mathsf{q}^{\mathscr{A}_2}_{\mathrm{FO}} \cdot \ell \mathrm{Q}^{\mathscr{A}_2}_{\mathrm{FN}}}{2^{2k}}.$$

We substitute $\mathrm{Q}_{\mathrm{FO}}^{\mathscr{A}_2} = \mathrm{Q}_{\mathrm{FO}}^{\mathscr{A}}$, $\mathrm{q}_{\mathrm{FO}}^{\mathscr{A}_2} = \mathrm{q}_{\mathrm{FO}}^{\mathscr{A}}$ and $\mathrm{Q}_{\mathrm{FN}}^{\mathscr{A}_2} = \mathrm{Q}_{\mathrm{SIGN}}^{\mathscr{A}}$. By Lemma 6 we obtain

$$\begin{aligned}\mathbf{Adv}^{\mathsf{uf}}_{\mathsf{DS}^*,\mathbf{S}_3}(\mathscr{A}_3) \leq & \mathbf{Adv}^{\mathsf{uf}}_{\mathsf{DS}^*,\mathbf{R}}(\mathscr{B}) + \frac{2\mathsf{p}(\mathrm{Q}_{\mathrm{FO}}^{\mathscr{A}_3} + \ell \mathrm{Q}_{\mathrm{SIGN}}^{\mathscr{A}_3})}{2^{2k}} \\ & + \frac{(\mathrm{Q}_{\mathrm{FO}}^{\mathscr{A}_3} + \ell \mathrm{Q}_{\mathrm{SIGN}}^{\mathscr{A}_3})^2}{2^{2k}} + \frac{\mathsf{p}\mathrm{Q}_{\mathrm{FO}}^{\mathscr{A}_3} \cdot \ell \mathrm{Q}_{\mathrm{SIGN}}^{\mathscr{A}_3}}{2^{2k}}\,.\end{aligned}$$

Recall that adversary $\mathscr{A}_3$ has the same query complexity as $\mathscr{A}$.

Under the assumption $\mathsf{p} > 2^{k-5}$ made in the theorem, BCJZ [?] established that $|\mathsf{Img}(\mathtt{CF})| = 2^{k-5}$. So $|\mathsf{Img}(\mathtt{CF})|/|\mathbb{Z}_{\mathsf{p}}| = 2^{k-5}/\mathsf{p} = \delta$. So by Theorem 14 we have

$$\mathbf{Adv}^{\mathsf{uf}}_{\mathsf{DS}^*,\mathbf{R}}(\mathscr{B}) \leq (1/\delta) \cdot \mathbf{Adv}^{\mathsf{uf}}_{\mathsf{DS},\mathbf{R}}(\mathscr{B})\,. \tag{4.8}$$

By substituting with the number of queries made by $\mathscr{A}$ as in Theorem 13 and collecting terms, we obtain the claimed bound stated in Theorem 16. ▮

We can now obtain security of EdDSA under number-theoretic assumptions via known results on the security of Schnorr. Namely, we use the known results to bound $\mathbf{Adv}^{\mathsf{uf}}_{\mathsf{DS},\mathbf{R}}(\mathscr{B})$ above. From [184, 3] we can get a bound and proof based on the DL problems, and from [?] with a better bound. We can also get an almost tight bound under the MBDL assumption via [32] and a tight bound in the AGM via [?].

4.6 Indifferentiability of the shrink-MD class of functors

INDIFFERENTIABILITY We want the tuple of functions returned by a functor $\mathbf{F} : \mathsf{SS} \to \mathsf{ES}$ to be able to “replace" a tuple drawn directly from ES. Indifferentiability is a way of defining what this means. We adapt the original MRH definition of indifferentiability [156] to our game-based model in Figure 4.7. In this game, Sim is a simulator algorithm. The advantage of an adversary $\mathscr{A}$ against the indifferentiability of functor $\mathbf{F}$ with respect to simulator Sim is defined to be

$$\mathbf{Adv}^{\mathsf{indiff}}_{\mathbf{F},\mathsf{Sim}}(\mathscr{A}) := 2\Pr[\mathrm{G}^{\mathsf{indiff}}_{\mathbf{F},\mathsf{Sim}}(\mathscr{A}) \Rightarrow 1] - 1.$$

MODIFYING THE MERKLE-DAMGRARD TRANSFORM Coron et al. showed that the Merkle-

```
Game $\mathsf{G}^{\mathsf{indiff}}_{\mathbf{F},\mathsf{Sim}}$

INIT():
  $c \leftarrow\!\!\$\ \{0,1\}$
  $\mathsf{hh} \leftarrow\!\!\$\ \mathsf{SS}$
  $\mathsf{HH} \leftarrow\!\!\$\ \mathsf{ES}$

PUB($i, Y$):
  if $c = 0$ then
    return $\mathsf{Sim}[\mathsf{HH}](i, Y)$
  else return $\mathsf{hh}(i, Y)$

PRIV($i, X$):
  if $c = 0$ then return $\mathsf{HH}(i, X)$
  else return $\mathbf{F}[\mathsf{hh}](i, X)$

FIN($c'$):
  return $[[c = c']]$
```

Figure 4.7. The game $\mathsf{G}^{\mathsf{indiff}}_{\mathbf{F},\mathsf{Sim}}$ measuring indifferentiability of a functor $\mathbf{F}$ with respect to simulator Sim.

Damgrard transform is not indifferentiable with respect to any efficient simulator due to its susceptibility to length-extension attacks [74]. In the same work, they analysed the indifferentiability of several closely related indifferentiable constructions, including the "chop-MD" construction. Chop-MD is a functor with the same domain as the MD transform; it simply truncates a specified number of bits from the output of MD. The $\mathbf{S}_3$ functor of Figure 4.5 operates similarly to the chop-MD functor, except that $\mathbf{S}_3$ reduces the output modulo a prime p instead of truncating. This small change introduces some bias into the resulting construction that affects its indifferentiability due to the fact that the outputs of the MD transform, which are $2k$-bit strings, are not distributed uniformly over $\mathbb{Z}_{\mathsf{p}}$.

In this section, we establish indifferentiability for a general class of functors that includes both chop-MD and $\mathbf{S}_3$. We rely on the indifferentiability of $\mathbf{S}_3$ in Section 4.5 as a stepping-stone to the unforgeability of EdDSA; however, we think our proof for chop-MD is of independent interest and improves upon prior work.

The original analysis of the chop-MD construction [74] was set in the ideal cipher model and accounted for some of the structure of the underlying compression function. A later proof by Fischlin and Mittelbach [165] adapts the proof strategy to the simpler construction we address here and works in the random oracle model as we do. Both proofs, however, contain a subtle gap in the way they use their simulators.

At a high level, both proofs define stateful simulators Sim which simulate a random compression function by sampling uniform answers to some queries and programming others with the help of their random oracles. These simulators are not perfect, and fail with some probability that the proofs bound. In the ideal indifferentiability game, the PUB oracle answers queries using the simulator and the PRIV oracle answers queries using a random oracle. Both proofs at some point replace the random oracle HH in PRIV with **Chop-MD**$[\mathsf{Sim}]$ and claim that because **Chop-MD**$[\mathsf{Sim}[\mathsf{HH}]](X)$ will always return $\mathsf{HH}(X)$ if the simulator does not fail, the adversary cannot detect the change. This argument is not quite true, because the additional queries to Sim made by the PRIV oracle can affect its internal state and prevent the simulator from failing when it would have in the previous game. In our proof, we avoid this issue with a novel simulator with *two internal states* to enforce separation between PRIV and PUB queries that both run the simulator.

Our result establishes indifferentiability for all members of the **Shrink-MD** class of functors, which includes any functor built by composing of the MD transform with a function $\mathsf{Out}: \{0,1\}^{2k} \to S$ that satisfies three conditions, namely that for some $\gamma, \varepsilon \geq 0$,

1. For all $y \in S$, we can efficiently sample from the uniform distribution on the preimage set $\{\mathsf{Out}^{-1}(y)\}$. We permit the sampling algorithm to fail with probability at most ε, but require that upon failure the algorithm outputs a (not necessarily random) element of $\{\mathsf{Out}^{-1}(y)\}$.
2. For all $y \in S$, it holds that $\gamma \leq |\{\mathsf{Out}^{-1}(y)\}|$.
3. The statistical distance $\delta(D)$ between the distribution

$$D := z \leftarrow_{\$} \mathsf{Out}^{-1}(y) : y \leftarrow_{\$} S$$

   and the uniform distribution on $\{0,1\}^{2k}$ is bounded above by ε.

In principle, we wish γ to be large and ε to be small; if this is so, then the set S will be substantially smaller than $\{0,1\}^{2k}$ and the function Out "shrinks" its domain by mapping it onto a smaller set.

Both chop-MD and mod-MD are members of the **Shrink-MD** class of functors; we briefly show the functions that perform bit truncation and modular reduction by a prime satisfy our three conditions. Truncation by any number of bits trivially satisfies condition (1) with $\varepsilon = 0$.

Reduction modulo p also satisfies condition (1) because the following algorithm samples from the equivalence class of x modulo p with failure probability at most $\frac{\mathsf{p}}{2^{2k}}$. Let ℓ be the smallest integer such that $\ell > \frac{2^{2k}}{\mathsf{p}}$. Sample $w \leftarrow\!\!\$\, [0 \ldots \ell - 1]$ and output $w \cdot \mathsf{p} + x$, or x if $w \cdot \mathsf{p} + x > 2^{2k}$. We say this algorithm "fails" in the latter case, which occurs with probability at most $\frac{1}{\ell} < \frac{\mathsf{p}}{2^{2k}}$. In the event the algorithm does not fail, it outputs a uniform element of the equivalence class of x.

Bellare et al. showed that the truncation of n trailing bits satisfies condition (2) for $\gamma = 2^{2k-n}$ and reduction modulo prime p satisfies (2) for $\gamma = \lfloor 2^{2k}/\mathsf{p} \rfloor$. It is clear that sampling from the preimages of a random $2k - n$-bit string under n-bit truncation produces a uniform $2k$-bit string, so truncation satisfies condition (3) with $\varepsilon = 0$. Also from Bellare et al. [29], we have that the statistical distance between a uniform element of $\mathbb{Z}_{\mathsf{p}}$ and the modular reduction of a uniform $2k$-bit string is $\varepsilon = \frac{\mathsf{p}}{2^{2k}}$. The statistical distance of our distribution $z \leftarrow\!\!\$\, \mathsf{Out}^{-1}(Y)$ for uniform Y over S from the uniform distribution over $\{0,1\}^{2k}$ is bounded above by the same ε; hence condition (3) holds.

Given a set S and a function $\mathsf{Out} : \{0,1\}^{2k} \to S$, we define the functor $\mathbf{F}_{S,\mathsf{Out}}$ as the composition of Out with $\mathbf{MD}$. In other words, for any $x \in \{0,1\}^*$ and $\mathsf{hh} \in \mathrm{FUNC}((,\{0,1\})^{b+2k}, \{0,1\}^{2k})$, let $\mathbf{F}_{S,\mathsf{Out}}[\mathsf{hh}](x) := \mathsf{Out}(\mathbf{MD}[\mathsf{hh}](x))$.

Theorem 17. *Let k be an integer and S a set of bitstrings. Let $\mathsf{Out} : \{0,1\}^{2k} \to S$ be a function satisfying conditions (1), (2), and (3) above with respect to $\gamma, \varepsilon > 0$. Let $\mathbf{MD}$ be the Merkle-Damgrard functor(c.f. Section 4.2) $\mathbf{F}_{S,\mathsf{Out}} := \mathsf{Out} \circ \mathbf{MD}$ be the functor described in the prior paragraph. Let pad be the padding function used by $\mathbf{MD}$, and let unpad be the function that removes padding from its input (i.e., for all $X \in \{0,1\}^*$, it holds that $\mathsf{unpad}(X \,\|\, \mathsf{pad}(|X|)) = X$). Assume that unpad returns $\perp$ if its input is incorrectly padded and that unpad is injective on its support. Then there exists a simulator Sim such that for any adversary $\mathscr{A}$ making* Priv *queries of maximum length $b \cdot (\ell - 1)$ bits then*

$$\mathbf{Adv}^{\mathsf{indiff}}_{\mathbf{F},\mathsf{Sim}}(\mathscr{A}) \le 2(\mathrm{Q}^{\mathscr{A}}_{\mathrm{PUB}} + \ell \mathrm{Q}^{\mathscr{A}}_{\mathrm{PRIV}})\varepsilon + \frac{(\mathrm{Q}^{\mathscr{A}}_{\mathrm{PUB}} + \ell \mathrm{Q}^{\mathscr{A}}_{\mathrm{PRIV}})^2}{2^{2k}} + \frac{\mathrm{Q}^{\mathscr{A}}_{\mathrm{PUB}} \cdot \ell \mathrm{Q}^{\mathscr{A}}_{\mathrm{PRIV}}}{\gamma}.$$

Proof of Theorem 17: We first give a brief overview of our proof strategy and its differences from previous indifferentiability proofs for the chop-MD construction [74, 165].

```
Simulator $\mathsf{Sim}[\mathsf{HH}](Y,G)$ :

$(y,m) \leftarrow Y$
if $\exists z$ such that $(y,z,m) \in G$.edges
    return $z$
$M \leftarrow G$.FindPath$(IV, y)$
if $M \neq \perp$ and $\mathsf{unpad}(M \| m) \neq \perp$ then
    if $\mathrm{T}_{\mathsf{hh}}[Y,M] \neq \perp$ then $z \leftarrow \mathrm{T}_{\mathsf{hh}}[Y,M]$
    else $z \leftarrow\!\!\$\, \mathsf{Out}^{-1}(\mathsf{HH}(\mathsf{unpad}(M \| m)))$
        $\mathrm{T}_{\mathsf{hh}}[Y,M] \leftarrow z$
else if $\mathrm{T}_{\mathsf{hh}}[Y] \neq \perp$ then $z \leftarrow \mathrm{T}_{\mathsf{hh}}[Y]$
else $z \leftarrow\!\!\$\, \{0,1\}^{2k}$; $\mathrm{T}_{\mathsf{hh}}[Y] \leftarrow z$
add $(y,z,m)$ to $G$.edges
add $(y,z,m)$ to $G_{\mathtt{all}}$.edges
return $z$
```

```
Game $\mathrm{G}_0 := \mathsf{G}^{\mathsf{indiff}}_{\mathbf{F},\mathsf{Sim}} | b = 0$

INIT():
  $\mathsf{HH} \leftarrow\!\!\$\, \mathsf{FUNC}(\{0,1\}^*, S,)$
  $G_{\mathtt{all}}, G_{\mathtt{pub}} \leftarrow (IV)$
PRIV($X$):
  return $\mathsf{HH}(X)$
PUB($Y$):
  $z \leftarrow \mathsf{Sim}[\mathsf{HH}](Y, G_{\mathtt{pub}})$
  return $z$
FIN($c'$):
  return $c'$
```

Figure 4.8. Left: Indifferentiability simulator for the proof of Theorem 17. Right: The ideal game $\mathsf{G}^{\mathsf{indiff}}_{\mathbf{F},\mathsf{Sim}}$ measuring indifferentiability of a functor $\mathbf{F}$ with respect to simulator Sim

Our simulator, Sim, is defined in Figure 4.8. It is inspired by, but distinct from, that of Mittelbach and Fischlin's simulator for the chop-MD construction ([165] Figure 17.4.), which in turn adapts the simulator of Coron et al [74] from the ideal cipher model to the random oracle model. These simulators all present the interface of a random compression function hh and internally maintain a graph in which each edge represents an input-output pair under the simulated compression function. The intention is that each path through this graph will represent a possible evaluation of $\mathbf{F}_{S,\mathsf{Out}}[\mathsf{hh}]$. The fundamental difference between our simulator and previous ones is that we maintain two internal graphs instead of one: one graph for all queries, and one graph for public interface queries only. This novel method of using two graphs avoids the gap in prior proofs described above by tracking precisely which parts of the simulator's state are influenced by private and public interface queries respectively.

In the "ideal" indifferentiability game, PRIV queries are answered by random oracle $\mathsf{HH} \leftarrow\!\!\$\, \mathsf{FUNC}(\{0,1\}^*, S,)$ PUB queries are answered by the simulator Sim, which maintains the two graphs $G_{\mathtt{pub}}$ and $G_{\mathtt{all}}$. We present pseudocode for this game (G_0) in Figure 4.8. In each graph, the nodes and edges are labeled with $2k$-bit strings. An edge from node y to node z with label m is denoted (y,z,m), and represents a single value of the simulated compression function; namely, on $6k$-bit input $y \| m$, the simulated compression function should output z. Queries made in the process of evaluating

$\mathbf{MD}[S]$ will form a path that begins at the node labeled with the initialization vector IV; the path's edges will be labeled with the $4k$-bit blocks of $\mathsf{pad}(M)$.

Whenever the simulator receives a fresh query (y,m), it uses a pathfinding algorithm FindPath to check whether the query extends an existing path from IV and thus continues an existing evaluation of the MD transform. If so, it reads the message from the path's edge labels then appends the new block m to the end. If the result is a properly padded message, the simulator removes the padding and uses its oracle HH to compute the output of functor $\mathbf{F}$ on the original message. This output w is an element of S, and it should be consistent with Out when applied to the $2k$-bit simulator output. The simulator therefore samples its response from the preimages of w under Out. If any of these steps fail, then the query does not need to be programmed, so the simulator samples a uniformly random response z and updates its graph with the new edge from y. Because we are attempting to simulate a random function, the simulator must cache its responses to maintain consistency between repeated queries. It does this in two ways: via the graphs and via table T_{hh}. We require two forms of caching because the simulator may use two graphs and thus responses may not be cached consistently between private and public queries in the graphs alone.

Our G_0 differs from this ideal indifferentiability game only in the FIN oracle, which returns the adversary's challenge guess c'. Thus the probability that game G_0 returns 1 exactly equals $1-\Pr[\mathsf{G}^{\mathsf{indiff}}_{\mathbf{F},\mathsf{Sim}}(\mathscr{A})|c=0]$.

We move to G_1, where the PRIV oracle uses Sim to calculate the output of functor $\mathbf{F}$, then discards the result. We wish for the adversary's view of games G_0 and G_1 to be identical, so we must ensure that the additional queries to Sim do not influence its state or its responses to PUB queries. We therefore call the simulator with different graphs in the two oracles. It responds to public queries based only on the public graph, and queries made by PRIV are private and do not update the public graph. We do use shared table T_{hh} to cache outputs across all queries; in this sense a private query can affect a public query; however, we cache responses separately for each branch of the simulator, so our caching does not alter the simulator's branching behavior and the distribution of public queries' responses does not change. The adversary cannot detect at

```
Game $G_1$

INIT():
  1 $\mathsf{HH} \leftarrow\!\!\$\, \mathsf{FUNC}(\{0,1\}^*, S,)$
  2 $G_{\mathtt{all}}, G_{\mathsf{pub}} \leftarrow (IV)$

PUB($Y$):
  1 $z \leftarrow \mathsf{Sim}[\mathsf{HH}](Y, G_{\mathsf{pub}})$
  2 return $z$

PRIV($X$):
  1 $w \leftarrow \mathbf{F}[\mathsf{Sim}[\mathsf{HH}](\cdot, G_{\mathtt{all}})](X)$
  2 return $\mathsf{HH}(X)$

FIN($c'$):
  1 return $c'$
```

Figure 4.9. Game G_1 in the proof of Theorem 17. Highlighted code is changed from the previous game, and algorithms not shown are unchanged from the previous game.

what time a response z is first sampled, so its view does not change, and

$$\Pr[G_0] = \Pr[G_1].$$

In game G_2, we set a bad flag if the simulator if $G_{\mathtt{all}}$ contains any collisions, cycles, or "duplicate" edges: edges with the same starting node and label but different ending nodes.

Collisions and cycles are formed only when a new edge is created whose ending node is already present in the graph; we set bad in this case. The caching in line 2 prevents duplicate edges except when the PRIV and PUB oracles query the simulator on the same input (y,m), in that order. Even in this case, caching in table T_{hh} prevents duplicate edges unless one query detects a path that the other did not, or the two queries detect different paths.

If the PUB query detects a path to node y that did not exist during the previous PRIV query, or there are two distinct paths to y in $G_{\mathtt{all}}$, then $G_{\mathtt{all}}$ must contain a collision or a cycle, and the bad flag will be set when that is detected. Furthermore, G_{pub} is a subgraph of $G_{\mathtt{all}}$, so it cannot contain a path to y that $G_{\mathtt{all}}$ does not. To catch the formation of duplicate edges, it is therefore sufficient to set bad if $G_{\mathtt{all}}$ contains a path from IV to y that is not detected by the subsequent PUB query.

The bad flag is internal and does not affect the view of the game, so

$$\Pr[G_2] = \Pr[G_1]$$

In G_3, we force the adversary to lose when the bad flag is set. This strictly decreases their

```
Game $\mathrm{G}_2$, $\boxed{\mathrm{G}_3}$

FIN($c'$):
    $\boxed{\text{if bad then return 0}}$
    return $c'$

Game $\mathrm{G}_4$

PRIV($X$):
    $w \leftarrow \mathbf{F}[\mathsf{Sim}[\mathsf{HH}](\cdot, G_{\mathtt{all}})](X)$
    return $w$
```

```
$\mathsf{Sim}[\mathsf{HH}](Y, G)$:
    $(y, m) \leftarrow Y$
    if $\exists z$ such that $(y, z, m) \in G$.edges
        return $z$
    $M \leftarrow G$.FindPath$(IV, y)$
    $M_{\mathtt{all}} \leftarrow \mathrm{G}_{\mathtt{all}}$.FindPath$(IV, y)$
    if $M \neq \bot$ and $\mathsf{unpad}(M \| m) \neq \bot$ then
        if $\mathrm{T}_{\mathsf{hh}}[Y, M] \neq \bot$ then $z \leftarrow \mathrm{T}_{\mathsf{hh}}[Y, M]$
        else $z \leftarrow\!\!\$\; \mathsf{Out}^{-1}(\mathsf{HH}(\mathsf{unpad}(M \| m)))$
            $\mathrm{T}_{\mathsf{hh}}[Y, M] \leftarrow z$
    else if $\mathrm{T}_{\mathsf{hh}}[Y] \neq \bot$ then $z \leftarrow \mathrm{T}_{\mathsf{hh}}[Y]$
    else $z \leftarrow\!\!\$\; \{0,1\}^{2k}$; $\mathrm{T}_{\mathsf{hh}}[Y] \leftarrow z$
    if ($z \in G_{\mathtt{all}}$.nodes and $(y, z, m) \notin G_{\mathtt{all}}$.edges)
        or $M \neq M_{\mathtt{all}}$
            $\mathsf{bad} \leftarrow \mathsf{true}$
    add $(y, z, m)$ to $G$.edges
    add $(y, z, m)$ to $G_{\mathtt{all}}$.edges
    return $z$
```

Figure 4.10. Games G_2, G_3, and G_4 in the proof of Theorem 17. Highlighted code is changed from the previous game, and boxed code is present only in G_3 (and subsequent games). Algorithms not shown are unchanged from the previous game.

advantage, so

$$\Pr[\mathrm{G}_3] \leq \Pr[\mathrm{G}_2].$$

In our next game, we stop querying HH directly in the PRIV oracle and instead return w, the result of our functor on the query. We claim that in G_3, either $w = \mathsf{HH}(X)$ or $\mathsf{bad} = \mathsf{true}$; thus if the adversary wins G_3, then in all PRIV queries we have $w = \mathsf{HH}(X)$. From this claim, we can see that the change does not affect the view of the adversary and

$$\Pr[\mathrm{G}_4] = \Pr[\mathrm{G}_3].$$

To prove the claim, consider a query PRIV(X). Let $(X_1, \ldots, X_n)$ be the b-bit blocks of $X \| \mathsf{pad}(|X|)$. By the definition of the MD transform, PRIV makes n queries to Sim of the form $(y_i, X_i), G_{\mathtt{all}}$, where $y_1 = IV$ and $y_i = \mathsf{Sim}((y_{i-1}, X_{i-1}), G_{\mathtt{all}})$ for all $i > 1$. These may not be fresh queries, but they must be made in order or bad will be set: if query $\mathsf{Sim}((y_i, X_i))$ outputs y_{i+1} and this has already been the input of a prior query, then y_{i+1} is a node in $G_{\mathtt{all}}$; a collision has occurred and the query will set bad. Unless bad is set, there exists exactly one path in $G_{\mathtt{all}}$ from IV to y_i, and the labels on this path are $(X_1,\ldots,X_{i-1})$. This is trivially true for $i=1$; the path is the empty path. The query $\mathsf{Sim}((y_{i-1},X_{i-1}),G_{\mathtt{all}})$ creates the edge (y_{i-1},y_i,X_{i-1}) in $G_{\mathtt{all}}$. By induction on i, there is always a path from IV to y_i with labels $(X_1,\ldots,X_{i-1})$. If there exists more than one path from IV to y_i, then $G_{\mathtt{all}}$ must contain either a cycle or two edges with the same ending node; in either case the bad flag will be set.

Therefore, when PRIV first makes the query $\mathsf{Sim}((y_{n-1},X_n),G_{all})$, it will detect the path, compute $\mathsf{unpad}(M\,\|\,X_n)=X$ and output an element $z\in\mathsf{Out}^{-1}(\mathsf{HH}(X))$. By the definition of Out^{-1}, we have $w=\mathsf{Out}(z)=\mathsf{HH}(X)$, so the claim holds.

At this point, the adversary can no longer directly query random oracle HH, so we allow the simulator to lazily sample the function. Also in this game, the simulator queries HH on the path from IV to y in $G_{\mathtt{all}}$ for all queries, not just private queries. If the path in G is different from the path in G_{pub}, then the bad flag will be set and the adversary will lose anyway. Therefore the view in any winning game is unchanged, and

$$\Pr[\mathrm{G}_4]=\Pr[\mathrm{G}_5].$$

In our next game G_6, we replace the sampling of z from the preimages of a random point y with sampling a uniformly random $2k$-bit string. The sampling will never fail to be uniform, which means the adversary can distinguish the game if it were to fail in G_5; from condition (1) we have that the probability of failure was at most ε per query. Otherwise, we have from condition (3) on Out that the statistical distance of the distribution $(z\leftarrow\!\!\$\,\mathsf{Out}^{-1}(y):\ :y\leftarrow\!\!\$\,S)$ from the uniform distribution on $\{0,1\}^{2k}$ is at most ε. By a hybrid argument over the $\mathrm{Q}^{\mathscr{A}}_{\mathrm{PUB}}+\ell\mathrm{Q}^{\mathscr{A}}_{\mathrm{PRIV}}$ queries to the simulator, the probability that $\mathscr{A}$ can distinguish G_5 from G_6 is bounded above by $2(\mathrm{Q}^{\mathscr{A}}_{\mathrm{PUB}}+\ell\mathrm{Q}^{\mathscr{A}}_{\mathrm{PRIV}})\varepsilon$.

Now that we are caching z in table T_{HH} when the check of line 6 holdss true, it has become redundant to cache it in table T_{hh}, so we stop doing this caching. We must be careful since table T_{HH} is indexed by labels of the form $\mathsf{unpad}(M_{\mathtt{all}}\,\|\,m)$ where T_{hh} was indexed by tuples $(Y,M_{\mathtt{all}})$.

Game G_5

```
Sim$(Y,G)$:
  $(y,m) \leftarrow Y$
  if $\exists z$ such that $(y,z,m) \in G$.edges
    return $z$
  $M \leftarrow G$.FindPath$(IV,y)$
  $M_{\mathtt{all}} \leftarrow G_{\mathtt{all}}$.FindPath$(IV,y)$
  if $M_{\mathtt{all}} \neq \bot$
      and $\mathsf{unpad}(M_{\mathtt{all}} \| m) \neq \bot$ then
    if $\mathrm{T}_{\mathsf{hh}}[Y,M_{\mathtt{all}}] \neq \bot$ then
      $z \leftarrow \mathrm{T}_{\mathsf{hh}}[Y,M_{\mathtt{all}}]$
    else
      if $\mathrm{T}_{\mathsf{HH}}[\mathsf{unpad}(M_{\mathtt{all}} \| m)] \neq \bot$
        $y \leftarrow \mathrm{T}_{\mathsf{HH}}[\mathsf{unpad}(M_{\mathtt{all}} \| m)]$
      $\mathrm{T}_{\mathsf{HH}}[\mathsf{unpad}(M_{\mathtt{all}} \| m)] \leftarrow y$
      $z \leftarrow\!\!\$\, \mathsf{Out}^{-1}(y)$; $\mathrm{T}_{\mathsf{hh}}[Y,M_{\mathtt{all}}] \leftarrow z$
  else if $\mathrm{T}_{\mathsf{hh}}[Y] \neq \bot$ then $z \leftarrow \mathrm{T}_{\mathsf{hh}}[Y]$
  else $z \leftarrow\!\!\$\, \{0,1\}^{2k}$; $\mathrm{T}_{\mathsf{hh}}[Y] \leftarrow z$
  if $(z \in G_{\mathtt{all}}$.nodes and $(y,z,m) \notin G_{\mathtt{all}}$.edges$)$
    or $M \neq M_{\mathtt{all}}$
      bad $\leftarrow$ true
  add $(y,z,m)$ to $G$.edges
  add $(y,z,m)$ to $G_{\mathtt{all}}$.edges
  return $z$
```

Game G_6

```
Sim$(Y,G)$:
  $(y,m) \leftarrow Y$
  if $\exists z$ such that $(y,z,m) \in G$.edges
    return $z$
  $M \leftarrow G$.FindPath$(IV,y)$
  $M_{\mathtt{all}} \leftarrow G_{\mathtt{all}}$.FindPath$(IV,y)$
  if $M_{\mathtt{all}} \neq \bot$ and $\mathsf{unpad}(M_{\mathtt{all}} \| m) \neq \bot$ then
    $z \leftarrow\!\!\$\, \{0,1\}^{2k}$
    if $\mathrm{T}_{\mathsf{HH}}[\mathsf{unpad}(M_{\mathtt{all}} \| m)] \neq \bot$
      $z \leftarrow \mathrm{T}_{\mathsf{HH}}[\mathsf{unpad}(M_{\mathtt{all}} \| m)]$
    $\mathrm{T}_{\mathsf{HH}}[\mathsf{unpad}(M_{\mathtt{all}} \| m)] \leftarrow z$
  else if $\mathrm{T}_{\mathsf{hh}}[Y] \neq \bot$ then $z \leftarrow \mathrm{T}_{\mathsf{hh}}[Y]$
  else $z \leftarrow\!\!\$\, \{0,1\}^{2k}$; $\mathrm{T}_{\mathsf{hh}}[Y] \leftarrow z$
  if $(z \in G_{\mathtt{all}}$.nodes and $(y,z,m) \notin G_{\mathtt{all}}$.edges$)$
    or $M \neq M_{\mathtt{all}}$
      bad $\leftarrow$ true
  add $(y,z,m)$ to $G$.edges
  add $(y,z,m)$ to $G_{\mathtt{all}}$.edges
  return $z$
```

Figure 4.11. Left: Game G_5 in the proof of Theorem 17. Right: Game G_6 in the proof of Theorem 17. Highlighted code is changed from the previous game, and algorithms not shown are unchanged from the previous game.

Since $M_{\mathtt{all}}$ is a path from IV to y in a graph with no duplicate edges provided bad is not set, $M_{\mathtt{all}}$ uniquely determines its ending node y and $\mathsf{unpad}(M_{\mathtt{all}} \| m)$ uniquely determines a tuple $((y,m),M_{\mathtt{all}})$ because unpad is injective. Thus the entries of T_{HH} are in one-to-one correlation with the entries of T_{hh}, and we can safely retain only the former, and

$$\Pr[\mathrm{G}_6] \leq \Pr[\mathrm{G}_5] + 2(\mathrm{Q}^{\mathscr{A}}_{\mathrm{PUB}} + \ell \mathrm{Q}^{\mathscr{A}}_{\mathrm{PRIV}})\varepsilon$$

In G_7, all queries are sampled randomly from $\{0,1\}^{2k}$ and cached in table T_{hh} under the input Y, instead of some being cached under the message $\mathsf{unpad}(M_{\mathtt{all}} \| m)$. We claim that in G_6 if a query $\mathsf{Sim}(y,m)$ stores z in $\mathrm{T}_{\mathsf{HH}}[X]$, then a later query $\mathsf{Sim}(y',m')$ will return z if and only if $(y,m) = (y',m')$ or bad is set. The forward direction is trivial. If $\mathsf{Sim}(y',m')$ returns $\mathrm{T}_{\mathsf{HH}}[X]$, then either we have

$$X = \mathsf{unpad}(G_{\mathtt{all}}.\mathrm{FindPath}(IV, y') \,\|\, m') = \mathsf{unpad}(G_{\mathtt{all}}.\mathrm{FindPath}(IV, y) \,\|\, m),$$

or there was a bad-setting collision between $\mathrm{T}_{\mathsf{HH}}[X]$ and the randomly-sampled response z.

In the former case, the function unpad is injective, so we know $m = m'$, and the paths from IV to y' and y' respectively have the same sequence of edge labels. Unless bad is set, there are no duplicate edges, so a starting node and sequence of edge labels uniquely identify the ending node on the path; consequently $y = y'$ and the claim follows.

Queries in G_7 therefore hit a cache indexed by Y if and only if they would hit a cache indexed by X in G_6. We do not need to worry that the new entries in T_{hh} overlap with those created in line 11; if the check in line 6 holds true during some query, then it cannot have been false in an earlier query with the same Y unless bad would be set. Thus no queries are answered from table T_{hh} in G_7 that would not have been cached in earlier games, and

$$\Pr[\mathrm{G}_7] = \Pr[\mathrm{G}_6].$$

Notice that both branches of the simulator now identically sample $z \leftarrow\!\!\$\, \{0,1\}^{2k}$ uniformly, subject to caching in table T_{hh} under Y; in the next game we will eliminate the redundant check on M_{all} in line 6.

In our final game, G_8, we remove the bad flag and the internal variables used to set it. This increases the adversary's advantage, since it can now win even if the game would set bad. The probability of a collision among the $\mathrm{Q}^{\mathscr{A}}_{\mathrm{PUB}} + \ell \mathrm{Q}^{\mathscr{A}}_{\mathrm{PRIV}}$ randomly sampled nodes of $G_{\mathtt{all}}$ is at most $\frac{(\mathrm{Q}^{\mathscr{A}}_{\mathrm{PUB}} + \ell \mathrm{Q}^{\mathscr{A}}_{\mathrm{PRIV}})^2}{2^{2k}}$ by a birthday bound. The probability that $G_{\mathtt{all}}$ contains a path to y that G_{pub} does not is the probability that the adversary $\mathscr{A}$ queries $\textsc{Pub}$ on one of the ℓq_{PRIV} intermediate nodes on a path in $G_{\mathtt{all}}$, before it learns the label of that node from $\textsc{Pub}$. $\mathscr{A}$ may use $\textsc{Priv}$ to learn the output y of Out an intermediate node, but it does not learn anything about which of the equally likely preimages of y is the label; from condition (2) we have that there are at least γ such preimages to guess from. Then the probability that $\mathscr{A}$ sets bad with a single $\textsc{Pub}$ query is

```
Game G_7

Sim(Y, G):
  $(y,m) \leftarrow Y$
  if $\exists z$ such that $(y,z,m) \in G$.edges
      return $z$
  $M \leftarrow G$.FindPath$(IV, y)$
  $M_{\mathtt{all}} \leftarrow G_{\mathtt{all}}$.FindPath$(IV, y)$
  if $M_{\mathtt{all}} \neq \bot$ and unpad$(M_{\mathtt{all}} \| m) \neq \bot$ then
      $z \leftarrow\!\!\$\ \{0,1\}^{2k}$
      if $\mathrm{T}_{\mathsf{hh}}[Y] \neq \bot$ then $z \leftarrow \mathrm{T}_{\mathsf{hh}}[Y]$
      $\mathrm{T}_{\mathsf{hh}}[Y] \leftarrow z$
  else if $\mathrm{T}_{\mathsf{hh}}[Y] \neq \bot$ then $z \leftarrow \mathrm{T}_{\mathsf{hh}}[Y]$
  else $z \leftarrow\!\!\$\ \{0,1\}^{2k}$; $\mathrm{T}_{\mathsf{hh}}[Y] \leftarrow z$
  if $z \in G_{\mathtt{all}}$.nodes or $M \neq M_{\mathtt{all}}$
      bad $\leftarrow$ true
  add $(y,z,m)$ to $G$.edges
  add $(y,z,m)$ to $G_{\mathtt{all}}$.edges
  return $z$
```

```
Game G_8

Sim(Y):
  if $\mathrm{T}_{\mathsf{hh}}[Y] \neq \bot$ then $z \leftarrow \mathrm{T}_{\mathsf{hh}}[Y]$
  else $z \leftarrow\!\!\$\ \{0,1\}^{2k}$; $\mathrm{T}_{\mathsf{hh}}[Y] \leftarrow z$
  return $z$

FIN(c'):
  return $c'$
```

Figure 4.12. Left: Game G_7 in the proof of Theorem 17. Right: Game G_8 in the proof of Theorem 17. Highlighted code is changed from the previous game, and algorithms not shown are unchanged from the previous game.

at most $\frac{\ell \mathrm{Q}^{\mathscr{A}}_{\mathrm{PRIV}}}{\gamma}$; a union bound over all PUB queries gives that a path exists in $G_{\mathtt{all}}$ but not $G_{\mathtt{pub}}$ with probability no greater than $\frac{\mathrm{Q}^{\mathscr{A}}_{\mathrm{PUB}} \cdot \ell \mathrm{Q}^{\mathscr{A}}_{\mathrm{PRIV}}}{\gamma}$.

We also stop maintaining the graphs $G_{\mathtt{pub}}$ and $G_{\mathtt{all}}$, which are now only used to cache queries whose responses are already cached in table T_{hh}. This changes nothing about the view of the adversary, so

$$\Pr[\mathrm{G}_8] \leq \Pr[\mathrm{G}_7] + \frac{(\mathrm{Q}^{\mathscr{A}}_{\mathrm{PUB}} + \ell \mathrm{Q}^{\mathscr{A}}_{\mathrm{PRIV}})^2}{2^{2k}} + \frac{\mathrm{Q}^{\mathscr{A}}_{\mathrm{PUB}} \cdot \ell \mathrm{Q}^{\mathscr{A}}_{\mathrm{PRIV}}}{\gamma}.$$

If we look closely at G_8, we can see that the "simulator" is actually just a lazily-sampled random function with domain $\{0,1\}^{6k}$ and codomain $\{0,1\}^{2k}$. In fact, G_8 is identical to the "real" indifferentiability game for functor $\mathbf{F}$, save for its choice of challenge bit. Thus

$$\Pr[\mathrm{G}_8] = \Pr[\mathsf{G}^{\mathsf{indiff}}_{\mathbf{F},\mathsf{Sim}}(\mathscr{A}) | c = 1].$$

Collecting bounds across all gamehops gives the theorem.▌

4.7 The unique order-p subgroup of $\mathbb{G}$

Here, we briefly prove that our choice in Section 4.2 of $\mathbb{G}_\mathsf{p}$ as the unique subgroup of order p of group $\mathbb{G}$, which has order $\mathsf{p} \cdot 2^\mathsf{f}$, is well-defined. (We do not prove that $\mathbb{G}_\mathsf{p}$ is cyclic as this follows directly from the fact that its order is prime.) We also give an efficient test for membership in $\mathbb{G}_\mathsf{p}$.

Proposition 1. Let p be an odd prime, let $2^\mathsf{f} < \mathsf{p}$ be a positive integer, and let $\mathbb{G}$ be a group of order $2^\mathsf{f} \cdot \mathsf{p}$. Then (1) the group $\mathbb{G}$ has a unique subgroup of order p, and (2) For all $X \in \mathbb{G}$ it is the case that X is in this subgroup iff $\mathsf{p} \cdot X = 0_\mathbb{G}$.

(1) Let n be the number of p-order subgroups of $\mathbb{G}$. According to Sylow's theorem $n \equiv 1 \mod \mathsf{p}$. We now have two cases: either $n = 1$, or $n > 1$. We prove that $n = 1$ by contradiction; therefore we assume $n > 1$. It follows that $n \geq \mathsf{p} + 1$. Two distinct groups of prime order can intersect only at the identity, so each of the n subgroups of $\mathbb{G}$ contains $\mathsf{p} - 1$ unique elements. Consequently the order of $\mathbb{G}$ is at least $n(\mathsf{p}-1) \geq (\mathsf{p}+1)(\mathsf{p}-1) \geq \mathsf{p}(\mathsf{p}+1)$. Since we have already defined the order of $\mathbb{G}$ to be $2^\mathsf{f} \cdot \mathsf{p}$, we have that $2^\mathsf{f} \geq \mathsf{p} + 1$. This contradicts our initial assumption that $2^\mathsf{f} < \mathsf{p}$; thus our assumption that $n > 1$ must be false and $\mathbb{G}$ must have exactly one subgroup of order p. This subgroup is $\mathbb{G}_\mathsf{p}$.

(2) Let $X \in \mathbb{G}$ be a group element and assume that $\mathsf{p} \cdot X = 0_\mathbb{G}$. This implies that the order of X divides p. Since p is prime, either the order of X is 1 or it is p. In the first case, $x = 0_\mathbb{G}$. Otherwise, X generates a subgroup with order p, which by part (1) is the unique such subgroup $\mathbb{G}_\mathsf{p}$. Therefore X generates $\mathbb{G}_\mathsf{p}$ and must belong to it.

For the reverse direction, assume that X is in $\mathbb{G}^\mathsf{p}$. The order of X must divide the order of $\mathbb{G}_\mathsf{p}$; so X must either have order p or order 1. In either case, $\mathsf{p} \cdot X = 0_\mathbb{G}$.

Chapter 5

Verifiable Distributed Aggregation Functions

5.1 Introduction

Operating a complex software system, such as an operating system, web browser, or web service, often requires measuring the behavior of the system's users. When used for a specific purpose, such measurements are often only consumed in some aggregated form, e.g., $F(m_1, \ldots, m_{ct})$ for some specific function F, rather than the individual measurements $m_1, \ldots, m_{ct}$. But in conventional systems, the measurements are revealed to the operator as a matter of course, resulting in an increased capability to surveil users. Consider the following motivating examples:

1. *Identifying misbehaving or malicious origins.* To detect bugs or attack vectors, a browser vendor might want to know how often establishing a connection to a given origin or loading a given web page triggers a specific event [168]. But logging these events and aggregating them in the clear risks exposing browser history.

2. *Measuring ad conversion rates.* Today advertising is a significant revenue source for many web service providers. In order to accurately assess the value of an ad campaign, the service provider and advertiser might want to measure how many people who clicked on a given ad made a purchase [2].

3. *Classifying malicious client behavior.* Many operators benefit from the ability to classify (or predict) user behavior automatically, and in real-time. For example, anomaly detection systems use machine learning models, trained and validated on requests from real clients, to

classify fraudulent or otherwise malicious behavior [166].

These applications require only aggregates; by collecting individual measurements, the operator learns more information than is ultimately used for the intended purpose. One way out of this predicament is *multi-party computation (MPC)*, which allows computing some function of private inputs distributed across multiple parties, without revealing these private inputs. In this paper, we consider a class of MPC protocols in which the bulk of the computation is outsourced to a small set of non-colluding servers.

Recent attention from the MPC community on problems like these has yielded solutions that are practical enough for real-world deployment [111, 75, 58, 59, 9, 26]. Notable examples include Mozilla's Origin Telemetry project [168] and the COVID-19 Exposure Notification Private Analytics system developed jointly by Apple and Google [13]. The success of these projects spurred the formation of a working group within the Internet Engineering Task Force (IETF) whose objective is to standardize MPC for "Privacy-Preserving Measurement (PPM)" [1], thereby improving interoperability and providing a deployment roadmap for new schemes.

The primary goal of this paper is to lay some of the groundwork for the provable security analysis that will be needed to support this effort. We formalize a syntax and set of security definitions for a particular class of MPC protocols from the literature [75, 58, 59, 9] of interest to the working group. Our definitions unify previous ones into an explicit, game-based framework that accounts for practical matters not attended to in prior work.

We apply our definitional framework to two constructions. The first is a candidate for standardization based on the Prio scheme designed by [75]; we show that this protocol meets our security goals with only minor changes. Another candidate for standardization is the more recent Poplar scheme due to [59]; we introduce and analyze a variant of this protocol that has improved round complexity.

Overview

The PPM working group plans to develop multiple protocol standards, one of which is the focus of this work. The *Distributed Aggregation Protocol (DAP)* standard [104] centers around the execution of a particular class of MPC protocols, called *Verifiable Distributed Aggregation*

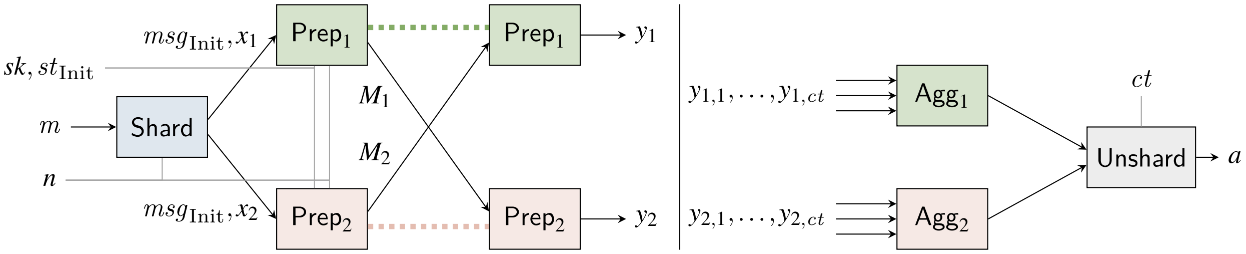


Figure 5.1. Illustration of (left) sharding and preparation of a single measurement and (right) aggregation and unsharding of a set of measurements. All parameters are defined in Section 5.3.

Functions (VDAFs) [25]. A VDAF is used to securely compute some **aggregation function** F over a set of measurements generated by the **clients**. To protect their privacy, the measurements are secret-shared and the computation of the aggregate is distributed amongst multiple, non-colluding aggregation servers (called **aggregators** hereafter). Execution of a VDAF involves four basic steps (illustrated in Figure 5.1):

- Shard: Each client shards its measurement m_i into **input shares** and sends one share to each aggregator. In this work, we sometimes refer to this sequence of input shares as the client's **report**.

- Prepare: After receiving a report from a client, the aggregators gossip amongst themselves in order to prepare their shares for aggregation. This involves refining the shares into an aggregatable form and verifying that the outputs are "well-formed", e.g., that they correspond to an integer in a given range, or correspond to a one-hot vector (a vector that is non-zero in at most one position). We call the outputs of this process the **refined shares**.

- Aggregate: Once an aggregator has recovered the desired number of refined shares, it combines them into its share of the aggregate result, called an **aggregate share**. It then sends this to the data consumer, known as the **collector**.

- Unshard: Finally, the collector combines each of the aggregate shares into $F(m_1, \dots, m_{ct})$.

WHY STANDARDIZE VDAFs? The case for standardizing this class of MPC protocols is made by the aforementioned deployments of Prio [168, 13], of which VDAFs are a natural generalization. The key feature that makes these protocols widely applicable and suited for Internet scale is that the expensive part of the computation (Shard/Prepare) is fully parallelizable across all reports being aggregated. This means that deployments can be scaled to such a degree that the time spent on executing the VDAF is primarily *network-bound* rather than *CPU-bound*. It is less clear (at least to those in the PPM working group) whether MPC techniques where the computations depend on all reports (e.g., oblivious sorting [198] or shuffling [12, 26]) would scale in the same way.

This feature also implies that VDAFs are only suitable for aggregation functions F that can be decomposed into f, g for which $F(m_1, \ldots, m_{ct}) = f(g(m_1), \ldots, g(m_{ct}))$, where g may be non-linear, but f must be affine. Indeed, the goal is not to encompass all possible MPC schemes, but a particular, useful, and highly parallelizable class of them. VDAFs can be used for a variety of aggregation tasks, including: simple statistics like sum, mean, standard deviation, quantile estimates, or linear regression [75]; a step of a gradient descent [124]; or heavy hitters (see below).

Security goals. The PPM working group's primary goal for VDAFs (cf. [104, Section 7]) is that they are **private** in the sense that the attacker learns nothing about the measurements $m_1, \ldots, m_{ct}$ beyond what it can infer from the aggregate result $F(m_1, \ldots, m_{ct})$. An active attacker who corrupts the collector and a fraction of the aggregators (typically all but one) and controls transmission of all messages in the protocol—except, of course, the input shares delivered to honest aggregators. Its corruptions are "static": the set of corrupt parties does not change over the course of the attack.

Another security consideration for VDAFs is that they are **robust** in the sense that the attacker cannot force the collector to compute anything other than the aggregate of honestly generated reports. Here the attacker is a set of malicious clients attempting to corrupt the aggregate result by sending malformed reports. For robustness we assume all of the aggregators execute the protocol correctly. Otherwise, a corrupt aggregator could trivially corrupt the result by sending the collector a malformed aggregate share.

We formalize these security notions in the game-playing paradigm [43]. First, in Section 5.3.2 we define privacy via an indistinguishability game $\mathsf{Exp}^{\mathrm{PRIV}}_{\Pi}(\mathscr{A})$ played by an attacker $\mathscr{A}$ against VDAF Π. The attacker interacts with the honest parties (i.e., the clients and uncorrupted aggregators) via a set of oracles. These oracles allow $\mathscr{A}$ to mount a kind of "chosen batch attack" in which the honest parties process one of two batches of measurements, and $\mathscr{A}$'s goal is to determine which was processed. This is analogous to the simulation-based definition of [75, Definition 1], which asks the the attacker to distinguish the protocol's execution from the view generated by a simulator.

We formalize robustness via a game $\mathsf{Exp}_{\Pi}^{\text{robust}}(\mathscr{A})$ (Section 5.3.2). Here the attacker $\mathscr{A}$—playing the role of a coalition of malicious clients—is given a single oracle that models the execution of the preparation step of VDAF execution on (invalid) reports. The attacker wins if an aggregator ever accepts an invalid share *or* if the aggregators compute refined shares that, when combined, do not correspond to a valid refined measurement. For natural VDAFs, robustness implies robustness in the sense of [75, Definition 6]: namely, the collector is guaranteed to correctly aggregate measurements uploaded by honest clients.

Note on the simulation paradigm. An alternative approach, and one that is more conventional for MPC, is to formulate security in the Universal Composability (UC) framework [66]. This methodology would begin by specifying the "ideal functionality" for computing an aggregation function such that, for any VDAF that securely realizes this functionality, any suitable notion of either privacy or robustness would follow from the UC composition theorem.

While this methodology is attractive, it creates the following difficulty in our setting. Many applications of VDAFs may be willing to tolerate a loose robustness bound (i.e., a non-negligible probability of accepting an invalid share) if doing so leads to better performance or communication. On the other hand, no application can accept a loose bound for privacy. In order to reason about this tradeoff, it is necessary to obtain explicit, concrete bounds for privacy and robustness *separately*. A theorem in the UC framework yields only a single bound, for the "UC-realizability" of the ideal functionality; applying this result directly would lead to parameter choices that might be more conservative than strictly necessary for the given application.

Another consideration is to make our results accessible to the target audience. Applying the UC framework, and interpreting its results, involves a number of subtleties that, based on our own observations, are often misunderstood when translated to practice.[1] One goal of our

[1] For a recent example, consider the standards for PAKEs ("Password-Authenticated Key Exchange") developed by the CFRG. Most of these standards are based on protocols with analysis in the UC framework. For one protocol [8], one question left open by that analysis was how to securely instantiate the "session identifier", one of the artifacts of the ideal functionality. The current draft offers recommendations for choosing the session identifier, but allows applications to ignore this entirely; a game-playing argument was used to justify this (cf [7, Section B]).

definitions is to make as explicit as possible all of the requirements an application like DAP [104] needs to meet in order to use VDAFs securely.

Previous definitions. Our definitions in Section 5.3 can be seen as a more precise (but not necessarily stronger) formulation of the informal definitions given in the original Prio paper [75, Section A]. While the authors mention the possibility of using a unified simulation-based security definition for privacy and robustness, they do not provide one.

For Poplar on the other hand, [59, Section A] provide a simulation-based definition for the end-to-end functionality. In order to capture the fact that a malicious server can influence the output of the protocol, they define a leakage function that allows the attacker to perturb the aggregate result with an arbitrary additive offset. While we believe this captures the robustness attacks that are possible for Poplar, it does not immediately generalize to the broader class of functionalities we consider as VDAFs. Also note that Bonet et al. [59] do not provide any proofs using their security definition. (The proofs they do provide are for definitions that are naturally captured by games, e.g., [59, Section D].) Finally, the simulation-based security definition of Poplar only considers a single security parameter, something that would need to be overcome to allow for separate security bounds for privacy and robustness.

Constructions

The starting point for our work is draft-irtf-cfrg-vdaf-03 [25], the current draft of the VDAF specification at the time of writing.

The first scheme described in draft-03, called Prio3, is based on Prio [75], but incorporates performance improvements from [58] (hereafter BBCG+19). Prio3 can be used to compute a wide variety of aggregation functions due to its use of *Fully Linear Proofs (FLPs)*. Briefly, an FLP is a special type of zero-knowledge proof that allows the client's input measurement to be validated by the aggregators (e.g., ensure that it is a number in some pre-determined range) who have only secret shares of the input and proof. The FLP designed by BBCG+19 (see [58, Theorem 4.3]) and adopted by the draft (with minor modifications; see [25, Section 7.3]) is expressed in terms of some arithmetic circuit C that takes in the prover's input x and a random string jr computed jointly by the prover and verifier. Computing this joint randomness, verifying the proof, and evaluating $C(x, jr)$ requires just one round of communication among the aggregators.

In Section 5.4, we prove Prio3 is both robust (Theorem 18) and private (Theorem 19) under the assumption that the underlying FLP is, respectively, *sound* and *honest-verifier zero-knowledge* as defined by BBCG+19. Our analysis unveiled a few subtle design issues in draft-03 that we address here.

The second scheme in draft-03 is called Poplar1 and is based on the recent Poplar protocol from [59] (BBCG+21). Poplar is designed to solve the private "heavy hitters" problem in which each client submits an arbitrary bitstring α and the collector wants to compute the set of unique strings that occurred at least T times. The key idea of BBCG+21 is an extension of *distributed point functions (DPFs)* [106], where two aggregators hold a share of a "DPF key" that concisely represents a *point function.* A point function evaluates to 0 on every input, except for the distinguished point α, where the function evaluates to some $\beta \neq 0$. By secret sharing the DPF keys generated by the clients, the aggregators can count *how many* clients submitted a particular candidate string without revealing *which* clients submitted it.

Poplar1 makes use of an enriched primitive called an *incremental DPF (IDPF).* IDPF keys can be queried not only at a given point, but a given *prefix.* That is, an *incremental point function* is one that evaluates to 0 on every input except for the set of strings that are a prefix of α. This new primitive gives rise to an efficient solution to the heavy hitters problem that involves running Poplar1 multiple times over the same set of IDPF keys, where each run begins with a set of candidate prefixes computed from the previous run.

To achieve robustness, Poplar1 uses a two-round multi-party computation in which the aggregators verify that the IDPF outputs are well-formed. That means that, compared to Prio3, the Poplar1 VDAF costs one additional round of communication, per report, during the preparation phase. The additional roundtrip is significant from an operational perspective.

In Section 5.5 we introduce *Doplar*, our modification to Poplar which achieves a one-round preparation. To achieve this, we combine FLPs and methods from distributed point functions in a novel way. We adopt a point-function verification method from De Castro and Polychroniadou [85]. We also introduce a new flavor of *delayed-input* FLPs, which may be of independent interest.

Related Work

Several works have considered private aggregate statistics, relying either on secret-sharing between non-colluding servers [78, 97, 99, 131, 148, 160], or on anonymization networks [185, 120, 64]. However, these works either do not provide privacy against malicious clients or rely on expensive zero-knowledge proofs.

A protocol for Secure Aggregation (SecAgg) in the single-server setting was presented by [56] and subsequently improved by [28, 27]. While SecAgg can provide security against malicious parties, it relies on multiple rounds of interaction between clients and server.

The VDAF abstraction was designed to encompass the architecture of Prio and Poplar in which the expensive portion of the MPC is fully parallelizable. Another example of a VDAF from the literature is the protocol of [9], which uses boolean (bit-wise) secret sharing instead of arithmetic circuit to improve communication cost from client to aggregator. However, this comes at a cost of weaker privacy, since their protocol does not protect against malicious servers.

There are also protocols that do not fit neatly into the VDAF framework as specified, but which might be adapted into VDAFs in the future. Masked LARK [124] is a proposal by Microsoft for training machine learning models on private data, using secret-sharing and MPC between a set of aggregators. AdScale [111] presents an aggregation system focused on private ads measurement. While designed for a single aggregation server, their construction appears to be amenable to our multi-server setting.

Other protocols in the literature share the same security goals of VDAFs, but do not have the same streaming architecture. One example is the recent “Oblivious Shuffling” protocol due to [12], which involves an MPC, assisted by a third-party, for unlinking each report from the client that sent it. The online processing for this procedure intrinsically involves all of the reports being shuffled; for VDAFs, all of the online processing is per-report. Similarly, [26] present a protocol for computing sparse histograms with two aggregators that is more efficient than DPFs for large domains, but reveals differentially private views to the aggregators. Again, the protocol crucially relies on shuffling contributions from multiple users. Vogue [129] is a protocol for computing private heavy hitters using three non-colluding servers. The protocol is secure against malicious servers and clients, but again relies on shuffling. Finally, the STAR protocol [80] uses an anonymizing proxy to ensure the collector only learns “popular” measurements, while any measurement that occurs less than a pre-determined threshold is not revealed to any party.

In recent concurrent work, [167] present another three-party, honest-majority protocol for computing heavy hitters. Their full protocol relies on a secure comparison protocol that is run after the aggregation phase, and thus doesn’t immediately fit our setting. However, we believe their input validation protocol can be adapted to obtain a VDAF for heavy hitters that has similar characteristics as our protocol in Section 5.5. (Indeed their core primitive, which they also call “Verifiable IDPF”, bears a striking resemblence to our own VIDPF abstraction.) Likewise, one could get robustness against malicious aggregators in the honest-majority setting by applying their "duplicate aggregator" technique to our protocols. We leave exploration of how to combine our results to future work.

Full version

This is the proceedings version of our paper. The full version [84] includes proofs of all theorems, a notion of “completeness” for VDAFs, and additional remarks and commentary.

5.2 Preliminaries

This section describes cryptographic primitives on which our constructions are based. We begin with a bit of non-standard notation.

Notation

Let $[i..j]$ denote the set of integers $\{i,\ldots,j\}$ and write $[i]$ as shorthand for the set $[1..i]$. If $\vec{v}$ is a vector, let $\vec{v}[i]$ denote the i-th element of $\vec{v}$. Let $(x,)$ denote the singleton vector with value x and $()$ the empty vector.

In our pseudocode, all variables that are undeclared implicitly have the value $\bot$. Let $y \leftarrow\!\!\$\, \mathscr{S}$ denote sampling y uniformly from a finite set $\mathscr{S}$; let $y \leftarrow\!\!\$\, A(x)$ denote execution of randomized algorithm A; and let $y \leftarrow A(x;r)$ denote execution of randomized algorithm A with coins r. If X is a random variable with support $\{0,1\}$ we let $\Pr[X]$ denote the probability that $X=1$.

A table T is a map from unique keys to values; we write $\mathrm{T}[K_1,\ldots]$ to denote the value corresponding to key $K_1,\ldots$. We sometimes write a dot "$\cdot$" in place of one of the elements of the key, e.g., "$\mathrm{T}[K_1,\cdot]$" instead of "$\mathrm{T}[K_1,K_2]$". We use this notation to denote the vector of values in the table that match the key pattern. For example, we write $\mathrm{T}[K_1,\cdot]$ for the vector $(\mathrm{T}[K_1,K_2^1],\ldots,\mathrm{T}[K_1,K_2^n])$ where $(K_1,K_2^1),\ldots,(K_1,K_2^n)$ are all of the keys in the table prefixed by K_1, in lexicographic order.

We measure an adversary's runtime by the time it takes to run its experiment to completion, including evaluating its queries.

Pseudorandom Generators

The VDAF spec [25, Section 6.2] calls for a particular type of object they call "pseudorandom generator (PRG)". Unlike the conventional PRGs, these objects are stateful. A PRG is comprised of the following algorithms:

- $\mathsf{PRG}.\textsc{init}(\mathit{seed} \in \{0,1\}^\kappa, \mathit{cntxt} \in \{0,1\}^*) \to \mathit{state} \in \mathsf{Q}$ takes a seed and context string to the initial PRG state. We call κ the **seed length**.
- $\mathsf{PRG}.\mathsf{Next}(\mathit{state} \in \mathsf{Q}, \ell \in \mathbb{N}) \to (\mathit{state}' \in \mathsf{Q}, \mathit{out} \in \{0,1\}^\ell)$ takes in the current PRG state and outputs a string of the desired length.

We also make use of an algorithm $\mathsf{Expand}[\mathsf{PRG}]$ that uses the given PRG to map a seed and context string to a vector of integers over the modular ring $\mathbb{Z}_p$ for the desired modulus p. We defer to [25, Section 6.2] for the full definition of $\mathsf{Expand}[\mathsf{PRG}]$.

In our security proofs, we model PRGs as random oracles [38]. In some cases, such as the distributed point functions (DPFs) in Section 5.5.1, constructions based on computational assumptions are known to be sufficient. We refer to [114, 115] for an overview of the state-of-the-art PRGs for DPFs and similar constructions.

Fully Linear Proof Systems

We recall the definition of FLP systems from BBCG+19 [58]. (Our formulation differs slightly, as we discuss below.) FLPs allow a prover to prove to a verifier, in zero-knowledge, that a secret-shared value has some property required by the application, e.g., the input is a number in the desired range, is a one-hot vector, etc. (The main construction of BBCG+19 allows the validity condition to be expressed in terms of an arithmetic circuit evaluated over the input, similar to more conventional zero-knowledge proof systems.) They are "fully linear" in the sense that verifying the proof involves computing a strictly linear function over both the input and proof. This allows verification to be performed on secret-shared data, leveraging its additive homomorphism property. (This is contrast to prior work on "linear PCPs" [17, 54, 125] in which the verifier has linear access to the proof, but arbitrary access to the input.)

An FLP with finite field $\mathbb{F}$, proof length m, verifier length v, prover randomness length pl, joint randomness length jl, and query randomness length ql is a triple of algorithms FLP defined as follows:

- $\mathsf{FLP.Prove}(x \in \mathbb{F}^n, jr \in \mathbb{F}^{jl}) \to \pi \in \mathbb{F}^m$ is the randomized **proof-generation** algorithm that takes in an input x and joint randomness jr and outputs a proof string $\pi \in \mathbb{F}^m$. We shall assume this algorithm generates random coins by sampling uniformly from $\mathbb{F}^{pl}$.
- $\mathsf{FLP.Query}(x \in \mathbb{F}^n, \pi \in \mathbb{F}^m, jr \in \mathbb{F}^{jl}) \to \sigma \in \mathbb{F}^v$ is the randomized **query-generation** algorithm that takes in an input x, proof string π, and joint randomness jr and outputs a verifier string σ. We shall assume the random coins are sampled uniformly from $\mathbb{F}^{ql}$.
- $\mathsf{FLP.Decide}(\sigma \in \mathbb{F}^v) \to acc \in \{0,1\}$ is the deterministic **decision predicate** that takes in a verifier string σ and outputs a bit acc indicating whether the input is valid.

We require the field $\mathbb{F}$ to have prime order; we occasionally denote its order by $\mathbb{F}.p$. We say that FLP is *fully linear* if the query-generation algorithm computes a linear function of the input and proof. That is, there exists a function Q whose output is a matrix in $\mathbb{F}^{v \times (n+m)}$ and, for all inputs x, proofs π, joint randomnesses jr, and query randomnesses qr, it holds that $\mathsf{Query}(x, \pi, jr;\, qr) = Q(jr;\, qr) \cdot (x \,\|\, \pi) \in \mathbb{F}^v$.

Associated with FLP is a language $\mathscr{L} \subseteq \mathbb{F}^n$. We say that FLP is **complete for $\mathscr{L}$** if the proof system outputs 1 whenever the input is in $\mathscr{L}$. That is, for all $x \in \mathscr{L}$ it holds that

$$\Pr\left[\,\mathsf{Decide}(\sigma) : jr \leftarrow\!\!\$\, \mathbb{F}^{jl}; \pi \leftarrow\!\!\$\, \mathsf{Prove}(x, jr); \sigma \leftarrow\!\!\$\, \mathsf{Query}(x, \pi, jr)\,\right] = 1\,.$$

```
Algorithm $\mathsf{View}_{\mathsf{FLP}}(x)$:
$jr \leftarrow\!\!\$\, \mathbb{F}^{jl}$; $qr \leftarrow\!\!\$\, \mathbb{F}^{ql}$
$\pi \leftarrow\!\!\$\, \mathsf{Prove}(x, jr)$
$\sigma \leftarrow \mathsf{Query}(x, \pi, jr; qr)$
ret $jr \,\|\, qr \,\|\, \sigma$
```

```
Algorithm $\mathsf{Err}_{\mathsf{FLP}}(P^*)$:
$(state_{P^*}, x) \leftarrow\!\!\$\, P^*()$; $jr \leftarrow\!\!\$\, \mathbb{F}^{jl}$
$\pi \leftarrow\!\!\$\, P^*(state_{P^*}, jr)$
$\sigma \leftarrow\!\!\$\, \mathsf{Query}(x, \pi, jl)$
ret $x \notin \mathscr{L} \wedge \mathsf{Decide}(\sigma)$
```

Figure 5.2. Procedures for defining security of FLPs.

We define soundness of FLP in terms of experiment $\mathsf{Err}_{\mathsf{FLP}}(P^*)$ shown in Figure 5.2 associated with a malicious prover P^*. In this experiment, the prover commits to an invalid input $x \in \mathbb{F}^n \setminus \mathscr{L}$. Next, joint randomness jr is generated and given to P^*, who then generates a proof π. Finally, the verifier is run on x, π, jr; the malicious prover "wins" if the verifier deems the input valid. We say FLP is **ε-sound for $\mathscr{L}$** if for all P^* it holds that $\Pr[\mathsf{Err}_{\mathsf{FLP}}(P^*)] \leq \varepsilon$.

Let $\mathsf{View}_{\mathsf{FLP}}(x)$ denote the procedure defined in Figure 5.2. We say FLP is **δ-statistical, strong, honest-verifier zero-knowledge**—or, simply, **δ-private**—if the verifier's view can be simulated without knowledge of the input. That is, there exists a randomized algorithm S such that for all $x \in \mathscr{L}$ it holds that

$$\sum_{\omega} \left| \Pr[\mathsf{View}_{\mathsf{FLP}}(x) = \omega] - \Pr[S() = \omega] \right| \leq \delta\,.$$

Comparison to [58]. Our syntax diverges slightly from BBCG+19 in two main respects. First, we have tailored the syntax to 1.5-round, public-coin IOP systems (cf. [58, Section 3.2]), as this is the only type of system considered in the VDAF specification [25]. Following the spec, we refer to the "random challenge" as the "joint randomness", as this allows us to more easily distinguish the challenge from the randomness consumed locally by the prover and verifier. Second, following the VDAF specification [25], we have adapted the syntax so that it describes explicitly the computations of the prover and verifier. Namely, our query-generation algorithm takes in the input and proof and outputs the verifier string consumed by the decision algorithm, whereas in BBCG+19, the query-generation algorithm outputs a description of the linear function used to compute the verifier string.

Our notion of FLP soundness differs slightly from BBCG+19 in that it explicitly requires the prover to "commit" to the invalid prior to the joint randomness being generated. This clarifies that the joint randomness needs to be independent of the input in order for soundness to be achievable.

Incremental Distributed Point Functions

A point function is a function that is 0 everywhere except on a special input α; an incremental point function is a function that is 0 everywhere except on *any prefix of* α. One can imagine arranging the co-domain of this function into a complete, binary tree in which the nodes are labeled with prefixes; and for each node labeled p, its children are labeled with $p \| 0$ and $p \| 1$. Each node on the path to the leaf node α is assigned a non-0 value, and all other nodes are assigned 0. (See [59, Figure 4] for an illustration.)

An incremental point function that gives output $\vec{\beta}[\ell]$ on the length-ℓ prefix of α is defined formally as:

$$f_{\alpha,\vec{\beta}}\left(pfx \in \{0,1\}^{\leq \eta}\right) = \begin{cases} \vec{\beta}\left[\ |pfx|\ \right] & \text{if } pfx \text{ is a prefix of } \alpha \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

An *Incremental Distributed Point Function (IDPF)* [59] is a concise secret sharing of an incremental point function. We recall the definition of an IDPF from [59] and restrict it slightly to suit the constructions of [25]. An IDPF's domain is the set of bitstrings of length at most η. For each input length ℓ, the IDPF generates outputs in the group $\mathbb{G}_\ell$. We present definitions only for the case of 2 parties, since leading constructions are specialized for that case. Let η, and κ be positive integers, let $\mathscr{M}$ be a set, and let $\mathbb{G}_\ell$ be a group for each $\ell \in [\eta]$. An IDPF is a pair of algorithms:

- $\mathsf{IDPF.Gen}(\alpha \in \{0,1\}^\eta, \vec{\beta} \in \mathbb{G}_1 \times \cdots \times \mathbb{G}_\eta) \to (\{0,1\}^\kappa)^2 \times \mathscr{M}$ is the **key generation** algorithm that takes a bitstring α and a vector $\vec{\beta}$ of point values, each of which is an element of the group $\mathbb{G}_\ell$ for the corresponding input length. It outputs a pair of key shares and a "public share" (an element of $\mathscr{M}$).

- $\mathsf{IDPF.Eval}(id \in \{1,2\}, key \in \{0,1\}^\kappa, pub \in \mathscr{M}, pfx \in \{0,1\}^\ell) \to \mathbb{G}_\ell$ is the **point-function eval-**

uation algorithm that takes in a shareholder index, an IDPF key share, a public share *pub*, and a prefix string of $\ell \leq \eta$ bits, then outputs a share of the IDPF output.

An IDPF is *correct* if for all $\alpha \in \{0,1\}^\eta$, all $\vec{\beta} \in \mathbb{G}_1 \times \cdots \times \mathbb{G}_\eta$, all $(key_1, key_2, pub) \in [\mathsf{IDPF.Gen}(\alpha, \vec{\beta})]$ and all strings *pfx* of length $\ell \leq \eta$:

$$f_{\alpha,\vec{\beta}}(pfx) = \sum_{\hat{j} \in \{1,2\}} \mathsf{IDPF.Eval}(\hat{j}, key_{\hat{j}}, pub, pfx)\,.$$

We define *privacy* for an IDPF later in Section 5.5.1.

5.3 Security Model

5.3.1 Syntax

As discussed in Section 5.1, a VDAF can be thought of as a protocol for evaluating an aggregation function F that takes as input the vector of measurements generated by the clients and outputs an aggregate result. In addition, the function may include an auxiliary "aggregation parameter" that allows the measurements to be "refined" to contain only the information of interest to the collector. Accordingly, prior to executing the VDAF, each aggregator's state is initialized with this aggregation parameter.

Recall that execution of a VDAF proceeds in four distinct phases. (See Figure 5.1 for an illustration.) We formalize the computation of the parties in each phase as the component algorithms of a VDAF:

- $\mathsf{Shard}(m \in \mathscr{I}, n \in N) \to (msg_{\mathrm{Init}} \in \mathscr{M}, \vec{x} \in X^s)$ is the randomized **sharding** algorithm run by the client. It takes in the client's input measurement m and a nonce n and returns an **initial message**[2] to be broadcasted to all aggregators and a sequence of **input shares**, one for each of the s aggregators.

- $\mathsf{Prep}(\hat{j} \in [s], sk \in \mathscr{SK}, state \in \mathsf{Q}, n \in N, \vec{M} \in \mathscr{M}^*, x \in X) \to (status \in \{\texttt{running}, \texttt{finished}, \texttt{failed}\}, out \in (\mathsf{Q} \times \mathscr{M}) \cup Y \cup \{\perp\}$ is the deterministic, interactive **preparation** algorithm run by each aggregator during the online preparation process. Its inputs are the share index $\hat{j}$, the

[2] This message is called the "public share" in the specification.

verification key shared by the aggregators sk, the current state $state$, the nonce n, the most recent round of **broadcast messages** $\vec{M}$ (or $(msg_{\mathrm{Init}},)$ if this is the first round), and the aggregator's input share x. The preparation algorithm returns an indication $status$ of whether the process is $\mathsf{running}$, `finished`, or `failed`. When the status is $\mathsf{running}$, the output includes the aggregator's next state and broadcast message ($(state, M) \in \mathsf{Q} \times \mathscr{M}$); and when the status is `finished`, the output includes the aggregator's **refined share** ($y \in Y$).

- $\mathsf{Agg}(\vec{y} \in Y^*) \rightarrow a \in \mathscr{A}$ is the deterministic *aggregation* algorithm run locally by each aggregator. It takes in a sequence of refined shares $\vec{y}$ and outputs an **aggregate share** a.

- $\mathsf{Unshard}(ct \in \mathbb{N}, \vec{a} \in \mathscr{A}^s) \rightarrow r \in \mathscr{O}$ is the deterministic *unsharding* algorithm used to compute the aggregate result r. Its inputs are the report count ct and aggregate shares $\vec{a}$.

The sets $\mathscr{I}$, N, $\mathscr{M}$, X, $\mathscr{SK}$, Q, Y, $\mathscr{A}$, and $\mathscr{O}$ must also be defined by the VDAF. (We typically do so only implicitly.) In addition to these sets, the VDAF specifies a set $\mathscr{Q}_{\mathrm{Init}} \subseteq \mathsf{Q}$ of possible **initial states**.

Our security definitions for VDAFs require three additional syntactic properties. The first is a property we call **refinement consistency**. Intuitively, this property insists that, for a given initial state, the VDAF defines the set of refined measurements with respect to which the validity of the refined shares is to be verified. For Doplar for example (Section 5.5), the set of measurements are fixed-length bitstrings, while the refined measurements are one-hot vectors over a finite field. Formally, refinement consistency requires the existence of functions refine and $\mathsf{refineFromShares}$ such that for all m, n and $st_{\mathrm{Init}} \in \mathscr{Q}_{\mathrm{Init}}$,

$$\begin{aligned}\Pr[\mathsf{refine}(st_{\mathrm{Init}}, m) = \mathsf{refineFromShares}(st_{\mathrm{Init}}, M, \vec{x}) :& \\ (M, \vec{x}) \leftarrow\!\!\$\, \mathsf{Shard}(m, n)] = 1\,.&\end{aligned}$$

Second, we require **aggregation consistency**, which means, roughly, that aggregating refined shares into aggregate shares, then unsharding, is equivalent to first unsharding the individual refined shares, then aggregating. To illustrate this idea, imagine arranging the refined shares into a matrix, where the rows correspond to aggregators and the columns to measurements.

Aggregation consistency means that one can either add up the columns, then the rows, or add up the rows, then the columns. Formally, we require the existence of a function $\mathsf{finishResult}$ such that for all refined shares $y^1_1, \ldots, y^1_{ct}, \ldots, y^s_1, \ldots, y^s_{ct} \in Y$, it holds that

$$\begin{aligned} &\mathsf{Unshard}(ct, (\mathsf{Agg}(y^1_1, \ldots, y^1_{ct}), \ldots, \mathsf{Agg}(y^s_1, \ldots, y^s_{ct}))) = \\ &\quad \mathsf{finishResult}(ct, \mathsf{Unshard}(1, (\mathsf{Agg}(y^1_1), \ldots, \mathsf{Agg}(y^s_1))), \\ &\qquad \ldots, \mathsf{Unshard}(1, (\mathsf{Agg}(y^1_{ct}), \ldots, \mathsf{Agg}(y^s_{ct})))) \,. \end{aligned}$$

We will see that these notions of refinement and aggregation consistency, while fairly technical in nature, are trivial to show for natural constructions (including Prio3 and Doplar).

Lastly, our privacy definition allows the VDAF to be executed multiple times over the same batch of measurements, each time beginning with a new initial state. (This accounts for the iterative nature of IDPFs.) Depending on the VDAF, it may be necessary for aggregators to restrict the sequence of initial states to prevent trivial leakage. Accordingly, we require each VDAF to specify an **allowed-state** algorithm $\mathsf{validSt}$ that takes in the sequence of previous initial states and the next initial state and returns a bit indicating whether the next initial state is allowed.

Remark 3. A notable feature of the VDAF syntax is the "verification key" shared by the aggregators. Looking ahead, this key is used to derive, from the nonce supplied by the client, shared randomness used for verifying refined shares. This is how the authors of the VDAF spec [25] chose to instantiate the "ideal coin-flipping functionality" used in the descriptions of protocols in the papers on which the spec is based [75, 58, 59]. As we will see in the next section, the details to how this functionality is instantiated are crucial to the privacy and robustness of VDAFs.

5.3.2 Security

Three definitions are given for VDAFs. The first, completeness, is used to specify correct evaluation of an aggregation function. The others, robustness and privacy, roughly correspond[3]

[3] We have not attempted to work out formal relationships between our definitions and those of Corrigan-Gibbs et al. [75]; whether our definitions, when restricted to the same class of protocols, are stronger, weaker, or equivalent is an open question.

to the notions of the same names from [75, Section A].

SECURITY CONSIDERATIONS FOR DAP [104]. Recall from the introduction that the DAP standard being developed by the PPM working group is designed to securely execute a VDAF in a real world network. Aspects of our security model can be thought of as abstracting away the functionality provided by DAP. As such, many of our modeling decisions here amount to requirements that the DAP protocol must fulfill. We will highlight some of these considerations throughout this section.

Completeness

We require that, when executed honestly, the VDAF evaluates its aggregation function F correctly. We formalize non-adversarial execution of Π via procedure Run_{Π} in Figure 5.3. Along with the VDAF Π, this procedure is parameterized by an initial state st_{Init} with which to configure the aggregators and a sequence of measurements and nonces to process into an aggregate result.

Algorithm Run processes the measurements as illustrated in Figure 5.1. First, each measurement is sharded into input shares by the submitting client (line 4), then refined into a set of refined shares by the aggregators (5–16). Next, the refined shares recovered by each aggregator are combined into an aggregate share (18). Finally, the aggregate shares are combined by the collector into the aggregate result (19).

Definition 13 (Completeness)**.** Let $F : \mathcal{Q}_{\text{Init}} \times \mathcal{I}^* \to \mathcal{O}$ be a function. We say that VDAF Π is *complete* for F if for all $\vec{m} \in \mathcal{I}^*$ and $\vec{n} \in N^*$ for which $|\vec{m}| = |\vec{n}|$ and $st_{\text{Init}} \in \mathcal{Q}_{\text{Init}}$ it holds that

$$\mathsf{Pr}\left[\,\mathsf{Run}_{\Pi}(st_{\text{Init}}, \vec{m}, \vec{n}) = F(st_{\text{Init}}, \vec{m})\,\right] = 1\,,$$

where the probability is over the randomness of Run and its subroutines. We say that Π is **complete** if it is complete for some function F.

Robustness

We say that VDAF Π is robust if, when all of the aggregators execute the protocol correctly, "valid" refined measurements are correctly aggregated, while any "invalid" measurements are filtered out by the aggregators (with high probability). This property is captured via the game

$\mathsf{Exp}^{\mathrm{robust}}_{\Pi}(\mathscr{A})$ defined in Figure 5.3. In this game the adversary, acting as a coalition of malicious clients, submits reports to the aggregators, eavesdrops on their communication, and observes the result of their computation. This functionality is modeled by the $\underline{\mathsf{Prep}}$ oracle, which the adversary may query any number of times. It controls the nonce and initial state for each trial, but its oracle queries are subject to the restriction that, for each distinct nonce, the sequence of initial states must be valid (according to the allowed-state algorithm $\mathsf{validSt}$).

Validity is defined in terms of the refinement-consistency algorithms (see Section 5.3.1). Let $\mathscr{V}_{st_{\mathrm{Init}}} = \{\mathsf{refine}_{st_{\mathrm{Init}}}(m) : m \in \mathscr{I}\}$ be the set of refined measurements for initial state st_{Init}. The adversary wins the robustness game if, when run on initial state st_{Init}, initial message msg_{Init}, and input shares $\vec{x}$, either: (1) an aggregator accepts a share of an invalid refined measurement, i.e., one of the aggregators ends in state `finished`, but the refined share y is not valid (i.e., not in the set $\mathscr{V}_{st_{\mathrm{Init}}}$, see line 15 in Figure 5.3); or (2) the refined shares computed by the aggregators do not match the expected refined measurement, i.e., unsharding the refined shares does not result in y (line 18).

Definition 14 (Robustness)**.** Define the advantage of $\mathscr{A}$ in defeating the robustness of VDAF Π as

$$\mathbf{Adv}\mathrm{robust}_{\Pi}(\mathscr{A}) = \mathsf{Pr}\left[\,\mathsf{Exp}^{\mathrm{robust}}_{\Pi}(\mathscr{A})\,\right].$$

Informally, we say that Π is **robust** if for every efficient adversary $\mathscr{A}$, the value of $\mathbf{Adv}\mathrm{robust}_{\Pi}(\mathscr{A})$ is small.

Remark 4. If a VDAF is robust in the sense of Definition 14 and aggregation-consistent, then the VDAF is also robust in the sense of [75, Definition 6]. Namely, as long as the aggregators execute the VDAF correctly, the collector is guaranteed to correctly aggregate measurements from honest clients (and reject the measurements from dishonest clients). The aggregation function that is computed is determined by the $\mathsf{finishResult}$ function implied by aggregation consistency, namely $F(st_{\mathrm{Init}}, m_1, \ldots, m_{ct}) = \mathsf{finishResult}(ct, (y_1, \ldots, y_{ct}))$, where $y_{\hat{k}}$ is the refined measurement obtained from refining $m_{\hat{k}}$ with st_{Init}.

```
Algorithm $\mathsf{Run}_{\Pi}(st_{\text{Init}}, \vec{m}, \vec{n})$:
$sk \leftarrow\!\!\$\; \mathscr{SK}$; $ct \leftarrow |\vec{m}|$
// Shard/Prepare
for $\hat{k} \in [ct]$:
  $(M, \vec{x}) \leftarrow \Pi.\mathsf{Shard}(\vec{m}[\hat{k}], \vec{n}[\hat{k}])$
  $\mathrm{Msg}[0,1] \leftarrow M$
  for $\hat{j} \in [s]$: $\mathrm{St}[\hat{j}] \leftarrow st_{\text{Init}}$
  for $\hat{\ell} \in [r+1]$:
    for $\hat{j} \in [s]$:
      $(status, out) \leftarrow \Pi.\mathsf{Prep}(\hat{j}, sk, \mathrm{St}[\hat{j}],$
                        $\vec{n}[\hat{k}], \mathrm{Msg}[\hat{\ell}\text{-}1, \cdot], \vec{x}[\hat{j}])$
      if $status = \mathsf{running}$:
        $(\mathrm{St}[\hat{j}], M) \leftarrow out$
        $\mathrm{Msg}[\hat{\ell}, \hat{j}] \leftarrow M$
      else if $status = \texttt{finished}$:
        $\mathrm{Out}[\hat{j}, \hat{k}] \leftarrow out$
      else if $status = \texttt{failed}$: ret $\perp$
// Aggregate/Unshard
for $\hat{j} \in [s]$: $\vec{a}[\hat{j}] \leftarrow \Pi.\mathsf{Agg}(\mathrm{Out}[\hat{j}, \cdot])$
ret $\Pi.\mathsf{Unshard}(ct, \vec{a})$
```

```
Game $\mathsf{Exp}^{\text{robust}}_{\Pi}(A)$:
$sk \leftarrow\!\!\$\; \mathscr{SK}$; $\mathsf{win} \leftarrow \mathsf{false}$; $A^{\underline{\mathsf{Prep}}}()$; ret $w$

$\underline{\mathsf{Prep}}(n \in N, \vec{x} \in X^s, msg_{\text{Init}} \in \mathscr{M}, st_{\text{Init}} \in \mathscr{Q}_{\text{Init}})$:
if not $\Pi.\mathsf{validSt}(\mathrm{Used}[n], st_{\text{Init}})$: ret $\perp$
$\mathrm{Used}[n] \leftarrow \mathrm{Used}[n] \,\|\, (st_{\text{Init}},)$
$\mathrm{Msg}[0,1] \leftarrow msg_{\text{Init}}$
$y \leftarrow \Pi.\mathsf{refineFromShares}(st_{\text{Init}}, msg_{\text{Init}}, \vec{x})$
for $\hat{j} \in [s]$: $\mathrm{St}[\hat{j}] \leftarrow st_{\text{Init}}$
for $\hat{\ell} \in [r+1]$:
  for $\hat{j} \in [s]$:
    $(status, out) \leftarrow \Pi.\mathsf{Prep}(\hat{j}, sk, \mathrm{St}[\hat{j}]$
                  $n, \mathrm{Msg}[\hat{\ell}\text{-}1, \cdot], \vec{x}[\hat{j}])$
    if $status = \mathsf{running}$:
      $(\mathrm{St}[\hat{j}], M) \leftarrow out$
      $\mathrm{Msg}[\hat{\ell}, \hat{j}] \leftarrow M$
    else if $status = \texttt{finished}$:
      $y_{\hat{j}} \leftarrow out$; $\tilde{\mathsf{win}} \leftarrow [y \notin \mathscr{V}_{st_{\text{Init}}}]$
    else if $status = \texttt{failed}$: pass
if not $\tilde{\mathsf{win}}$:
  $\tilde{\mathsf{win}} \leftarrow [y \neq \Pi.\mathsf{Unshard}(1, (\Pi.\mathsf{Agg}(y_{\hat{j}}))_{\hat{j} \in s}]$
$\mathsf{win} \leftarrow \mathsf{win} \bigvee \tilde{\mathsf{win}}$; ret $(\mathsf{win}, \mathrm{Msg})$
```

Figure 5.3. Left: Procedure for defining completeness of r-round, s-party VDAF Π. Right: Game for defining robustness of Π. Let $\mathscr{Q}_{\text{Init}} \subseteq \mathsf{Q}$ denote the set of valid initial states and, for each $st_{\text{Init}} \in \mathscr{Q}_{\text{Init}}$, let $\mathscr{V}_{st_{\text{Init}}} = \{\mathsf{refine}_{st_{\text{Init}}}(m) : m \in \mathscr{I}\}$.

Privacy

We formalize privacy via the indistinguishability game $\mathsf{Exp}^{\text{PRIV}}_{\Pi,t}(\mathscr{A})$ in the right panel of Figure 5.4. The game is associated with VDAF Π, adversary $\mathscr{A}$, and **corruption threshold** t. We consider an attacker that controls the collector and statically corrupts at most t aggregators (lines 1–2). Using its Prep oracle (lines 16–28), the adversary controls transmission of all messages in the protocol, *except* for the honestly generated input shares sent to honest (uncorrupted) aggregators. We assume that the adversary also controls setup (see the Setup oracle on lines 11–15), meaning that it can pick the verification keys for honest aggregators (1) and the initial state of each run of the preparation phase (14). This captures the real-world setting of the DAP protocol [104], where one of the aggregators (the "leader") effectively picks these values on behalf of the others (the "helpers"). Note that our game requires the secret key to be committed to prior to generating measurements: this is a deliberate restriction that was necessary to prove security of our constructions. (It is necessary for DAP to enforce this restriction.)

```
Game $\mathsf{Exp}^{\mathrm{PRIV}}_{\Pi,t}(\mathscr{A})$:
1  $(state_{\mathscr{A}}, \mathsf{V}, (sk_{\hat{j}})_{\hat{j}\in\mathsf{V}}) \leftarrow\!\!\$\ \mathscr{A}()$
2  if $|\mathsf{V}| + t \neq s$ return $\bot$
3  $b \leftarrow\!\!\$\ \{0,1\}$
4  $b' \leftarrow\!\!\$\ \mathscr{A}^{\underline{\mathsf{Shard}},\underline{\mathsf{Setup}},\underline{\mathsf{Prep}},\underline{\mathsf{Agg}}}(state_{\mathscr{A}})$
5  ret $b = b'$

$\underline{\mathsf{Shard}}(\hat{k} \in \mathbb{N}, m_0, m_1 \in \mathscr{I})$:
6  if $\mathrm{Used}[\hat{k}] \neq \bot$: ret $\bot$
7  $n \leftarrow\!\!\$\ N$
8  $(\mathrm{Pub}[\hat{k}], \mathrm{In}[\hat{k},\cdot]) \leftarrow\!\!\$\ \Pi.\mathsf{Shard}(m_b, n)$
9  $\mathrm{Used}[\hat{k}] \leftarrow (n, m_0, m_1)$
10 ret $(n, \mathrm{Pub}[\hat{k}], (\mathrm{In}[\hat{k},\hat{j}])_{\hat{j}\in T})$

$\underline{\mathsf{Setup}}(\hat{i} \in \mathbb{N}, \hat{j} \in \mathsf{V}, st_{\mathrm{Init}} \in \mathscr{Q}_{\mathrm{Init}})$:
11 if $\mathrm{Status}[\hat{i},\hat{j}] \neq \bot$
12     or not $\Pi.\mathsf{validSt}(\mathrm{Setup}[\cdot,\hat{j}], st_{\mathrm{Init}})$:
13   ret $\bot$
14 $\mathrm{Setup}[\hat{i},\hat{j}] \leftarrow st_{\mathrm{Init}}$
15 $\mathrm{Status}[\hat{i},\hat{j}] \leftarrow \mathsf{running}$

$\underline{\mathsf{Prep}}(\hat{i} \in \mathbb{N}, \hat{j} \in \mathsf{V}, \hat{k} \in \mathbb{N}, \vec{M} \in \mathscr{M}^*)$:
16 if $\mathrm{Status}[\hat{i},\hat{j}] \neq \mathsf{running}$ or $\mathrm{In}[\hat{k},\hat{j}] = \bot$: ret $\bot$
17 if $\mathrm{St}[\hat{i},\hat{j},\hat{k}] = \bot$:
18   $\mathrm{St}[\hat{i},\hat{j},\hat{k}] \leftarrow \mathrm{Setup}[\hat{i},\hat{j}]$; $\vec{M} \leftarrow (\mathrm{Pub}[\hat{k}],)$
19 $(n, m_0, m_1) \leftarrow \mathrm{Used}[\hat{k}]$
20 $(status, out) \leftarrow$
21     $\Pi.\mathsf{Prep}(\hat{j}, sk_{\hat{j}}, \mathrm{St}[\hat{i},\hat{j},\hat{k}], n, \vec{M}, \mathrm{In}[\hat{k},\hat{j}])$
22 if $status = \mathsf{running}$:
23   $(state, M) \leftarrow out$; $\mathrm{St}[\hat{i},\hat{j},\hat{k}] \leftarrow state$
24 else if $status = \mathtt{finished}$:
25   $\mathrm{St}[\hat{i},\hat{j},\hat{k}] \leftarrow \bot$; $\mathrm{Out}[\hat{i},\hat{j},\hat{k}] \leftarrow out$
26   $\mathrm{Batch}_0[\hat{i},\hat{j},\hat{k}] \leftarrow m_0$; $\mathrm{Batch}_1[\hat{i},\hat{j},\hat{k}] \leftarrow m_1$
27 else if $status = \mathtt{failed}$: $\mathrm{St}[\hat{i},\hat{j},\hat{k}] \leftarrow \bot$
28 ret $(status, M)$

$\underline{\mathsf{Agg}}(\hat{i} \in \mathbb{N}, \hat{j} \in \mathsf{V})$:
29 if $\mathrm{Status}[\hat{i},\hat{j}] \neq \mathsf{running}$: ret $\bot$
30 $(state_1, \ldots, state_s) \leftarrow \mathrm{Setup}[\hat{i},\cdot]$
31 if $F(state_{\hat{j}}, \mathrm{Batch}_0[\hat{i},\hat{j},\cdot]) \neq F(state_{\hat{j}}, \mathrm{Batch}_1[\hat{i},\hat{j},\cdot])$
32     and $(\forall j, j' \in \mathsf{V})\, state_j = state_{j'} \wedge sk_j = sk_{j'}$:
33   ret $\bot$
34 $\mathrm{Status}[\hat{i},\hat{j}] \leftarrow \mathtt{finished}$
35 ret $\Pi.\mathsf{Agg}(\mathrm{Out}[\hat{i},\hat{j},\cdot])$
```

Figure 5.4. Game for defining privacy of a complete, s-party VDAF Π for corruption threshold ≥ 0. Let F denote the aggregation function for which Π is complete and let $\mathscr{Q}_{\mathrm{Init}}$ its set of initial states. Let $T = [s] \setminus \mathsf{V}$.

The initial state for each run is subject to the restriction imposed by the allowed-state algorithm defined by the VDAF (lines 11–13). (Accordingly, it is necessary for honest aggregators to enforce this restriction in the DAP protocol.)

The game asks $\mathscr{A}$ to distinguish execution of the protocol on two sets of measurements of its choosing. To capture this, the attacker is given an oracle $\underline{\mathsf{Shard}}$ (lines 6–10) that models execution of the honest clients. This oracle takes in two measurements m_0, m_1 and shards m_b, where b is the challenge bit chosen at the start of the game, and returns the initial message and the input shares of the corrupted aggregators. The oracle chooses a nonce n from the nonce space N at random. (Accordingly, the DAP protocol must arrange for clients to choose their nonces at random.)

To model an attacker that controls the collector, the game allows the adversary to learn the aggregate shares computed by honest aggregators. This is captured by the $\underline{\mathsf{Agg}}$ oracle (lines 29–35). Queries to this oracle are subject to the restriction that the aggregate share does not trivially leak the challenge bit: namely, the aggregate of both batches of measurements specified by the adversary must be equal (31). (Tables $\mathrm{Batch}_0, \mathrm{Batch}_1$ keep track of the pairs of measurements m_0, m_1 passed to the $\underline{\mathsf{Shard}}$ for which a given aggregator has recovered a refined share for a given initial state.) This restriction is analogous to the "leakage function" provided to the simulator in previous simulation-style definitions. See [75, Section A] and [59, Section A]. We consider something slightly stronger: if the honest aggregators disagree either on the initial state or the verification key, then we do not impose the restriction (32). This amounts to demanding that the aggregate shares leak nothing in this case.

Definition 15 (Privacy)**.** Let Π be an s-party VDAF and let $t < s$ be a positive integer. Define the t-advantage of $\mathscr{A}$ in attacking the privacy of Π as

$$\mathbf{Adv}\mathrm{PRIV}_{\Pi,t}(\mathscr{A}) = 2 \cdot \Pr\big[\,\mathsf{Exp}^{\mathrm{PRIV}}_{\Pi,t}(\mathscr{A})\,\big] - 1\,.$$

Informally, we say that Π is t-**private** if for every efficient $\mathscr{A}$ the value of $\mathbf{Adv}\mathrm{PRIV}_{\Pi,t}(\mathscr{A})$ is small.

5.4 Prio3

In this section we present our security analysis for Prio3, one of the candidates for standardization specified in draft-irtf-cfrg-vdaf-03 [25]. The starting point for this VDAF is an FLP system (Section 5.2) that defines the set of valid measurements. Drawing on techniques from Boneh et al. [58], Prio3 exploits the full-linearity property to allow the aggregators to validate the secret shared input. However, in order for the resulting VDAF to be suitable for a particular aggregation function $F : \mathscr{I} \to \mathscr{O}$, we need the proof system to define how measurements ($\mathscr{I}$) are encoded as inputs to the prover and how refined shares are processed into the aggregate results ($\mathscr{O}$).

Definition 16 (Affine, aggregatable encodings [75, Sec. 5.])**.** Let $F : \mathscr{I} \to \mathscr{O}$ be a function. An FLP system FLP admits an *affine, aggregatable encoding for F* if it defines the following algorithms:

```
Algorithm $\mathsf{Shard}(m,n)$:
$inp \leftarrow \mathtt{Encode}(m)$
for $\hat{j} \in [2..s]$:
    $blind_{\hat{j}}, xseed_{\hat{j}}, pseed_{\hat{j}} \leftarrow\!\!\$\ \{0,1\}^{\kappa}$
    $\vec{x}[\hat{j}] \leftarrow \mathsf{RG}_2(xseed_{\hat{j}}, \hat{j})$
    $\overrightarrow{rseed}[\hat{j}] \leftarrow \mathsf{RG}_7(blind_{\hat{j}}, \hat{j} \,\|\, n \,\|\, \vec{x}[\hat{j}])$
$\vec{x}[1] \leftarrow inp - \sum_{\hat{j}=2}^{s} \vec{x}[\hat{j}]$
$blind_1 \leftarrow\!\!\$\ \{0,1\}^{\kappa}$
$\overrightarrow{rseed}[1] \leftarrow \mathsf{RG}_7(blind_1, 1 \,\|\, n \,\|\, \vec{x}[1])$
$jseed \leftarrow \mathsf{RG}_6(0^{\kappa}, \overrightarrow{rseed})$; $jr \leftarrow \mathsf{RG}_1(jseed, \varepsilon)$
$ps \leftarrow\!\!\$\ \{0,1\}^{\kappa}$; $pr \leftarrow \mathsf{RG}_4(ps, \varepsilon)$
$\vec{\pi}[1] \leftarrow \mathsf{Prove}(inp, jr\,; pr)$
$\vec{\pi}[1] \leftarrow \vec{\pi}[1] - \sum_{\hat{j}=2}^{s} \mathsf{RG}_3(pseed_{\hat{j}}, \hat{j})$
$\vec{x}[1] \leftarrow (\vec{x}[1], \vec{\pi}[1], blind_1)$
for $\hat{j} \in [2..s]$:
    $\vec{x}[\hat{j}] \leftarrow (xseed_{\hat{j}}, pseed_{\hat{j}}, blind_{\hat{j}})$
ret $(\overrightarrow{rseed}, \vec{x})$
```

```
Algorithm $\mathsf{Unpack}(\hat{j}, x)$:
if $\hat{j} = 1$: $(inp, \pi, blind) \leftarrow x$
else:
    $(xseed, pseed, blind) \leftarrow x$
    $inp \leftarrow \mathsf{RG}_2(xseed, \hat{j})$
    $\pi \leftarrow \mathsf{RG}_3(pseed, \hat{j})$
ret $(inp, \pi, blind)$
```

```
Algorithm $\mathsf{Prep}(\hat{j}, sk, state, n, \vec{M}, x)$:
if $state = \varepsilon$:  // Process initial message from client
    $(inp, \pi, blind) \leftarrow \mathsf{Unpack}(\hat{j}, x)$
    $(\overrightarrow{rseed},) \leftarrow \vec{M}$; $\overrightarrow{rseed}[\hat{j}] \leftarrow \mathsf{RG}_7(blind, \hat{j} \,\|\, n \,\|\, inp)$
    $jseed \leftarrow \mathsf{RG}_6(0^{\kappa}, \overrightarrow{rseed})$; $jr \leftarrow \mathsf{RG}_1(jseed, \varepsilon)$
    $qr \leftarrow \mathsf{RG}_5(sk, n)$
    $M \leftarrow (\mathsf{Query}(inp, \pi, jr; qr), \overrightarrow{rseed}[\hat{j}])$
    $state \leftarrow (jseed, \mathsf{Truncate}(inp))$
    ret $(\mathsf{running}, state, M)$
// Process broadcast messages from aggregators
$(jseed, y) \leftarrow state$; $(\overrightarrow{vfs}[\hat{j}], \overrightarrow{rseed}[\hat{j}])_{\hat{j} \in [s]} \leftarrow \vec{M}$
$acc \leftarrow \mathsf{Decide}(\sum_{\hat{j}=1}^{s} \overrightarrow{vfs}[\hat{j}])$
if $acc$ and $jseed = \mathsf{RG}_6(0^{\kappa}, \overrightarrow{rseed})$: ret $(\mathtt{finished}, y)$
else ret $(\mathtt{failed}, \perp)$
```

```
Algorithm $\mathsf{Agg}(\vec{y})$:
ret $\sum_{i=1}^{|\vec{y}|} \vec{y}[i]$
```

```
Algorithm $\mathsf{Unshard}(ct, \vec{a})$:
ret $\mathsf{Decode}(ct, \sum_{i=1}^{|\vec{a}|} \vec{a}[i])$
```

```
Algorithm $\mathsf{RG}_i(seed, cntxt)$:
$l \leftarrow (jl, n, m, pl, ql)$
if $i \le 5$: ret $\mathsf{Expand}[\mathsf{PRG}](seed, \ell_i \,\|\, cntxt, \mathbb{F}.p, l[i])$
else: ret $\mathsf{PRG.Next}(\mathsf{PRG.}\textsc{init}(seed, \ell_i \,\|\, cntxt), \kappa)$
```

Figure 5.5. Definition of 1-round, s-party VDAF $\mathsf{Prio3}[\mathsf{FLP}, \mathsf{PRG}]$. Let $\ell_1, \ldots, \ell_7$ be arbitrary, distinct bitstrings.

- $\mathsf{FLP}.\mathtt{Encode}(m \in \mathscr{I}) \to inp \in \mathbb{F}^n$ is an injective map from the domain of F to the input space $\mathbb{F}^n$ of FLP.
- $\mathsf{FLP}.\mathsf{Truncate}(inp \in \mathbb{F}^n) \to out \in \mathbb{F}^{ol}$ refines an FLP input into a format suitable for aggregation. We call ol the *output length*.
- $\mathsf{FLP}.\mathsf{Decode}(ct \in \mathbb{N}, out \in \mathbb{F}^{ol}) \to a \in \mathscr{O}$ converts a refined, aggregated output out to its final form a. This computation may depend on the number of measurements ct.

Correctness requires that for all $ct \geq 0$ and $\vec{m} \in \mathscr{I}^{ct}$ it holds that

$$F(\vec{m}) = \mathsf{Decode}\Big(ct, \sum_{i \in [ct]} \mathsf{Truncate}\left(\mathtt{Encode}\left(\vec{m}[i]\right)\right)\Big).$$

Let FLP be an FLP system that admits an affine, aggregatable encoding for F and let PRG

be a PRG. We specify the core algorithms of $\mathsf{Prio3}[\mathsf{FLP},\mathsf{PRG}]$ in Figure 5.5. (This version includes changes to draft-irtf-cfrg-vdaf-03 [25], as we discuss below.) The sharding algorithm begins by encoding the measurement as prescribed by the FLP. It then splits the encoded measurement inp into shares, generates a proof of inp's validity, and splits the proof into shares as well. The joint randomness jr passed to the proof generation algorithm is derived from the input shares following the Fiat-Shamir-style transform described—but not formally analyzed—in [58, Section 6.2.3]. During preparation, the aggregators collectively re-compute jr from their input shares. Each aggregator broadcasts a share of the verifier by running the FLP query-generation algorithm on its share of the input and proof. (The query randomness qr is derived from the shared verification key sk and the nonce n provided by the environment.) The FLP decision algorithm is run on the combined verifier shares.

The aggregators must derive the joint randomness prior to computing their verifier shares. In order to allow them to perform both computations in parallel in a single round, the client sends in its initial message the sequence $\vec{rseed}$ of "joint randomness parts" consisting of the intermediate values computed by the aggregators. This allows jr to be computed immediately on receipt of the input shares. To detect if a malicious client transmitted malformed parts, the aggregators also verify the joint randomness was computed properly in the same flow.

Allowed initial states

The set of initial states for Prio3 is simply $\mathcal{Q}_{\text{Init}} = \{\varepsilon\}$. In our security analysis, we assume honest aggregators process a batch at most once. Accordingly, the allowed-state algorithm $\mathsf{Prio3}[\mathsf{FLP},\mathsf{PRG}].\mathsf{validSt}$ accepts only if the batch was not aggregated previously.

Consistency

The set of refined measurements includes any output of the affine, aggregatable encoding for FLP. On input of $st_{\text{Init}} \in \{\varepsilon\}$ and $m \in \mathcal{I}$, the refinement algorithm $\mathsf{Prio3}[\mathsf{FLP},\mathsf{PRG}].\mathsf{refine}$ first encodes m, then truncates and decodes it as prescribed by FLP. The refine-from-shares algorithm, $\mathsf{Prio3}[\mathsf{FLP},\mathsf{PRG}].\mathsf{refineFromShares}$, unpacks each input share (see Unpack in Figure 5.5), extracts the shares of the FLP input, truncates them, adds them together, and decodes the result.

For aggregation consistency, we require the encoding scheme for FLP to be aggregation-consistent in a similar sense. Specifically, there must exist a function $\mathsf{finishResult}$ such that for all outputs $out_1, \ldots, out_{ct} \in \mathbb{F}^{ol}$ it holds that $\mathsf{Decode}(ct, \sum_{\hat{k}\in[ct]} out_{\hat{k}}) = \mathsf{finishResult}(ct, \mathsf{Decode}(1, out_1), \ldots, \mathsf{Decode}(1, out_{ct}))$.

Changes to the specification [25]. Figure 5.5 differs from draft-03 of the VDAF spec in three ways. The most important change is to incorporate the nonce provided by the environment into the joint randomness computation. This turns out to be crucial for a tight robustness bound; without this change, we must contend with cases in which joint randomness is reused across reports.

Second, we have revised the domain separation tags for the PRG invocations so that each RG_i in Figure 5.5 can be treated as an independent random oracle.

Lastly, we have moved the joint randomness parts from the input shares into the client's initial broadcast message. This change allowed us to simplify our proofs somewhat, but we do not believe it is essential for security. It also has the added benefit of reducing overall communication overhead for $s > 2$.

Security

Fix $s > 2$ and let $\Pi = \mathsf{Prio3}[\mathsf{FLP}, \mathsf{PRG}]$ be as specified above. Let N denote the nonce space for Π and let κ denote the seed length of PRG.

Theorem 18. *Modeling each RG_i in Figure 5.5 as a random oracle, if FLP is ε-sound (Section 5.2), then for every adversary $\mathscr{A}$ against the robustness of Π it holds that*

$$\mathbf{Adv}\mathrm{robust}_{\Pi}(\mathscr{A}) \le (q_{\mathsf{RG}} + q_{\mathsf{Prep}}) \cdot \varepsilon + \frac{q_{\mathsf{RG}} + {q_{\mathsf{Prep}}}^2}{2^{\kappa-1}},$$

where $\mathscr{A}$ makes q_{Prep} queries to $\underline{\mathsf{Prep}}$ and a total of q_{RG} queries to its random oracles.

For reasonable choices of the PRG seed size, the loosest term in this bound is $(q_{\mathsf{RG}} + q_{\mathsf{Prep}}) \cdot \varepsilon$. The multiplicative loss of $q_{\mathsf{RG}} + q_{\mathsf{Prep}}$ reflects the adversary's ability to partially control the randomness of the FLP insofar as it is able to use rejection sampling to obtain query and joint randomness with any property. The ε-soundness of FLP bounds the probability of violating soundness in a single interaction, but in a VDAF the attacker may interact with the underlying

FLP once in each of its q_{Prep} queries to $\underline{\mathsf{Prep}}$, and it can use its queries to RG_1 to bias these interactions' joint randomness.

Proof of Proof sketch: We sketch the security reduction here and defer the detailed proof to Section 5.9.1. Our goal is to construct from $\mathscr{A}$ a malicious prover P^* for the soundness of FLP. The overall idea is to run $\mathscr{A}$ in a simulation of the robustness game for Π in which P^*'s instance of the soundness experiment (Figure 5.2) is embedded in a random $\underline{\mathsf{Prep}}$ query so that P^* wins its game precisely when $\mathscr{A}$ sets $\mathsf{win} \leftarrow \mathsf{true}$ for the first time in that query. The main difficulty is that P^* must arrange to use the joint randomness it received as input in its own game. To provide a consistent simulation of RG_1, we need to arrange to extract the input to commit to from $\mathscr{A}$'s queries. This results in a union bound over all queries to RG_1, either by the simulation of $\underline{\mathsf{Prep}}$ or by $\mathscr{A}$ directly. ▮

Remark 5. For FLPs that do not make use of joint randomness (i.e., those for which $jl = 0$), queries to RG_1 can be disregarded, as this oracle is not used by Π. In particular, a similar reduction can be shown that results in a multiplicative loss of just q_{Prep}.

Remark 6. Although we have not addressed this explicitly in our specification, the extraction step of our security reduction relies on the encoding of the context string passed to each RG_i being invertible. (Similarly for Theorem 20.)

Theorem 19. *Modeling each RG_i in Figure 5.5 as a random oracle, if FLP is δ-private, then for all $0 < t < s$ and attackers $\mathscr{A}$ it holds that*

$$\mathbf{Adv}\mathrm{PRIV}_{\Pi,t}(\mathscr{A}) \leq 2q_{\mathsf{Shard}}\left(\delta + \frac{q_{\mathsf{RG}} + q_{\mathsf{Shard}}}{|N|} + \frac{s \cdot q_{\mathsf{RG}}}{2^{\kappa-1}}\right),$$

where $\mathscr{A}$ makes q_{Shard} queries to $\underline{\mathsf{Shard}}$ and a total of q_{RG} queries to the random oracles.

Proof of Proof sketch: The full proof is given in Section 5.9.2. The main idea is to arrange for $\mathscr{A}$'s queries to its oracles to be independent of the challenge bit. We do so via a game-playing argument in which we incrementally revise the game until the outcome of each oracle is independent of the current state of the game. The last step involves a hybrid argument, where in each hybrid world we replace one invocation of the proof- and query-generation algorithms of FLP (see Figure 5.2) with invocation of the simulator hypothesized by the δ-privacy of FLP. This accounts for the multiplicative loss of q_{Shard} in the bound. ▮

Remark 7. Instead of using separate seeds for the input share, proof share, and blind, it may be safe to reuse the same seed for all three purposes, similar to the seed in Doplar (Section 5.5). This may result in a slightly looser bound: such a change would enable the attacker to test guesses of the input share because the known joint randomness part would be derived from the same seed.

5.5 Doplar

In this section we describe and analyze Doplar, our round-reduced variant of Poplar1 [25]. Poplar1 is a candidate for standardization in draft-irtf-cfrg-vdaf-03; Doplar is introduced by our paper.

Poplar1 is designed to solve the "heavy hitters" problem (as described in Section 5.1) using an IDPF (Section 5.2) in the following way. Two aggregators hold shares of an IDPF key generated by the clients. Each evaluates its IDPF key at a number of equal-length candidate prefixes. They expect that the output is non-zero for at most one of these candidates; to verify this, they execute an MPC to determine if they hold shares of a one-hot vector, and that the non-zero value is in the desired range (i.e., equal to one or zero). If verification succeeds, then each adds its share of the vector together with the other verified shares. The result is a vector representing the number of measurements prefixed by each candidate.

The "secure sketch" MPC of Boneh et al. [59] requires two rounds of communication between the aggregators. (Computing and verifying this sketch occurs during the preparation phase of VDAF evaluation.) In this section we propose an alternative strategy that, leveraging techniques in Section 5.4, requires just one.

Our first step is to factor the validity check into two, parallelizable computations. The first computation is solely responsible for checking that the vector of IDPF outputs is one-hot. In Section 5.5.1 we extend IDPFs (Section 5.2) into *verifiable* IDPFs (VIDPFs), which preserve the same privacy properties as IDPFs, but additionally verify the one-hotness of the refined shares.

In Section 5.7 we show how to instantiate this primitive using a simple technique from DeCastro and Polychroniadou [85].

The second computation checks that the *sum* of the elements of the vector is in the desired range. Our first idea is to perform this range check using an FLP (Section 5.2). This does not work, however, since a standard FLP requires the prover to know the statement it is proving; in our case, it does not know the value of the sum computed by the aggregators, since it does not know the candidate prefixes. To overcome this, we show how to transform an FLP into one that is *delayed input* [151]. Such a proof system allows a proof to be generated for a *set* of potential inputs such that the honest verifier accepts the proof for any input in this set, but rejects otherwise (with high probability). We define delayed-input FLP in Section 5.5.2 and defer the construction to Section 5.8.

The result is the 1-round, 2-party VDAF presented in Section 5.5.3. The cost of this round reduction is a modest increase in overall communication cost and CPU time, at least for the current instantiations of the VIDPF and delayed-input FLP. We compare the cost of Doplar and Poplar1 at the end of this section.

5.5.1 Verifiable IDPF

A **verifiable** IDPF (VIDPF) allows the dealer to prove to the shareholders that their shares represent a one-hot vector. For our purposes, we define a **one-hot vector** as a vector that is nonzero in *at most* one component (i.e., the all-zeroes vector is also one-hot). Verifiable function secret sharings (of which VIDPF is a special case) were previously considered in [62, 85], and a construction specifically for VIDPF was given in [85].

A VIDPF has two algorithms in addition to the usual $\mathsf{Gen}, \mathsf{Eval}$:

- $\mathsf{VIDPF.VEval}(id \in \{1,2\}, key \in \{0,1\}^\kappa, pub \in \mathscr{M},$ $\vec{x} \in (\{0,1\}^\ell)^u) \to \{0,1\}^* \times (\mathbb{G}_\ell)^u$ takes as input an IDPF share (private and public parts), and a sequence of IDPF inputs. It outputs a **verification value** and a sequence of output shares.

- $\mathsf{VIDPF.Verify}(h_1, h_2) \to \{0,1\}$ takes as input two verification values and returns a boolean.

We also overload the syntax of the plaintext evaluation function to take a vector of inputs, i.e., we let

$$f_{\alpha,\vec{\beta}}(\vec{x}) = \Big(f_{\alpha,\vec{\beta}}(\vec{x}[1]), f_{\alpha,\vec{\beta}}(\vec{x}[2]), \dots\Big).$$

We say VIDPF is *correct* if, for all $\alpha \in \{0,1\}^\eta$, all $\vec{\beta} \in \mathbb{G}_1 \times \cdots \times \mathbb{G}_\eta$, all $\vec{x} \in (\{0,1\}^\ell)^*$, all $(key_1, key_2, pub) \in [\mathsf{Gen}(\alpha, \vec{\beta})]$, all $(h_1, \vec{y}_1) \in [\mathsf{VEval}(1, key_1, pub, \vec{x})]$, and all $(h_2, \vec{y}_2) \in [\mathsf{VEval}(2, key_2, pub, \vec{x})]$

- $\vec{y}_1 + \vec{y}_2 = f_{\alpha,\vec{\beta}}(\vec{x})$
- If $(\vec{y}_1 + \vec{y}_2)$ is a one-hot vector then $\mathsf{V.Verify}(h_1, h_2) = 1$

Theorem 20 requires VIDPF to be *extractable*. Intuitively, there should be an algorithm that can extract $\alpha, \vec{\beta}$ from adversarially generated VIDPF key shares. Then VEval must produce shares consistent with the incremental point function $f_{\alpha,\vec{\beta}}$, whenever Verify succeeds. (A similar property is formalized for IDPFs by BBCG+21.) This property implies, among other things, that if Verify succeeds, then shareholders are guaranteed to hold shares of a one-hot vector. We formalize this property below.

Definition 17 (Extractable VIDPF (cf. [59, Definition 7]))**.** Suppose that VIDPF is defined in terms of a random oracle with co-domain Y. Refer to the game in Figure 5.6 associated to VIDPF, **extractor** $\mathscr{E}$, and adversary $\mathscr{A}$. Define $\mathscr{A}$'s advantage in **fooling** $\mathscr{E}$ as $\mathbf{Adv}\mathrm{extract}_{\mathsf{VIDPF},\mathscr{E}}(\mathscr{A}) = 2 \cdot \Pr\big[\mathsf{Exp}^{\mathrm{extract}}_{\mathsf{VIDPF},\mathscr{E}}(\mathscr{A})\big] - 1$.

Finally, our privacy reduction for Doplar (Theorem 21) requires the underlying VIDPF to be *private*, in the sense that one shareholder's view—consisting of its share $key_{\hat{j}}$, the public share pub, and the other shareholder's verification value h—leaks nothing about the secrets α and β. Prior definitions of verifiable FSS—e.g., the one of DeCastro and Polychroniadou [85]—only define privacy with respect to a single vector of evaluation points and verification predicate, both of which are assumed to be known at the time of share generation. In our setting, shares are generated and only later is there a choice of evaluation points and verification predicates. The same shares may be evaluated many times, on different input vectors and with different verification predicates. This leads to a more interactive, and stronger, definition than in prior

```
Game $\mathsf{Exp}^{\text{extract}}_{\mathsf{VIDPF},\mathscr{E}}(\mathscr{A})$:
  $b \leftarrow\!\$ \{0,1\}$; $(key_1, key_2, pub, state_{\mathscr{A}}) \leftarrow\!\$ \mathscr{A}^{\text{RO}}()$
  if $b = 0$: $(\alpha, \vec{\beta}) \leftarrow\!\$ \mathscr{E}(key_1, key_2, pub, \text{Rand})$
  $b' \leftarrow\!\$ \mathscr{A}^{\text{RO},\underline{\mathsf{Eval}}}(state_{\mathscr{A}})$; ret $b = b'$

$\underline{\mathsf{Eval}}(\vec{x})$:
  $(h_1, \vec{y}_1) \leftarrow\!\$ \mathsf{VIDPF.VEval}^{\text{RO}}(1, key_1, pub, \vec{x})$
  $(h_2, \vec{y}_2) \leftarrow\!\$ \mathsf{VIDPF.VEval}^{\text{RO}}(2, key_2, pub, \vec{x})$
  if $b = 0$ and $\mathsf{VIDPF.Verify}^{\text{RO}}(h_1, h_2) = 1$: ret $f_{\alpha,\vec{\beta}}(\vec{x})$
  else: ret $\vec{y}_1 + \vec{y}_2$

$\text{RO}(inp)$:
  if $\text{Rand}[inp] = \perp$: $\text{Rand}[inp] \leftarrow\!\$ Y$
  ret $\text{Rand}[inp]$
```

```
Game $\mathsf{Exp}^{\text{PRIV}}_{\mathsf{VIDPF},\mathsf{Sim}}(\mathscr{A})$:
  $b \leftarrow\!\$ \{0,1\}$; $(state_{\mathscr{A}}, \alpha, \vec{\beta}, \hat{j}) \leftarrow \mathscr{A}()$
  if $b = 0$: $(key_{\hat{j}}, pub) \leftarrow\!\$ \mathsf{Sim}_1(\hat{j})$
  else: $(key_1, key_2, pub) \leftarrow\!\$ \mathsf{VIDPF.Gen}(\alpha, \vec{\beta})$
  $b' \leftarrow \mathscr{A}^{\underline{\mathsf{Sketch}}}(state_{\mathscr{A}}, key_{\hat{j}}, pub)$; ret $b = b'$

$\underline{\mathsf{Sketch}}(\vec{x})$:
  if $b = 0$: $h \leftarrow \mathsf{Sim}_2(\hat{j}, key_{\hat{j}}, pub, \vec{x})$
  else: $(h, \_) \leftarrow \mathsf{VIDPF.VEval}(3 - \hat{j}, key_{3-\hat{j}}, pub, \vec{x})$
  ret $h$
```

```
Game $\mathsf{Exp}^{\text{PRIV}}_{\mathsf{DFLP},\mathsf{Sim}}(A)$:
  $b \leftarrow\!\$ \{0,1\}$; $(X, st_A) \leftarrow A()$
  if $b = 0$: $(st_{\mathsf{Sim}}, jr, qr) \leftarrow \mathsf{Sim}_1(|X|)$
  else:
    $jr \leftarrow\!\$ \mathbb{F}^{jl}$; $qr \leftarrow\!\$ \mathbb{F}^{ql}$; $\Delta \leftarrow\!\$ \mathbb{F}^{el}$
    $\pi \leftarrow\!\$ \mathsf{DFLP.Prove}(X, \Delta, jr)$
  $(x, st_A) \leftarrow A(st_A, jr, qr)$; assert $x \in X$
  if $b = 0$: $\sigma \leftarrow \mathsf{Sim}(st_{\mathsf{Sim}})$
  else: $\sigma \leftarrow \mathsf{DFLP.Query}(\mathsf{DFLP.Encode}(\Delta, x), \Delta, \pi, jr; qr)$
  $b' \leftarrow A(st_A, \sigma)$; ret $b = b'$
```

Figure 5.6. Games for defining extractability (top-left), and privacy (bottom-left) of VIDPFs and privacy of delayed-input FLP (right).

works.[4]

Definition 18. Let $\mathsf{Exp}^{\text{PRIV}}_{\mathsf{VIDPF},\mathsf{Sim}}(A)$ be the privacy game for VIDPF, **simulator** $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$ and adversary A defined in Figure 5.6. Define the advantage of A in distinguishing Sim's simulation from its view of VIDPF's execution as $\mathbf{Adv}\text{PRIV}_{\mathsf{VIDPF},\mathsf{Sim}}(A) = 2 \cdot \Pr[\mathsf{Exp}^{\text{PRIV}}_{\mathsf{VIDPF},\mathsf{Sim}}(A)] - 1$.

If this privacy game withholds the $\underline{\mathsf{Sketch}}$ oracle from the adversary (shaded in Figure 5.6) then we obtain the privacy game for plain IDPFs, with the adversary's advantage defined analogously.

In Section 5.7 we describe a VIDPF construction that satisfies all the necessary security properties. The construction is heavily based on the verifiable DPF technique from [85].

[4] The game does not need to provide an oracle for $\mathsf{VIDPF.Verify}$ since it is a deterministic algorithm whose inputs are known to the adversary.

5.5.2 Delayed-Input FLPs

We introduce a new variant of fully linear proofs (FLPs), in which the prover does not know in advance which instance (i.e., input) will be used during verification. Instead, the proof is generated only knowing a set of possible instances; later, the proof is verified using one of those instances. For technical reasons, the proof and verification steps operate not on the instance, but on a *randomized encoding* of the instance. This extra randomness is useful in our eventual construction (Section 5.8).

We adopt the terminology of **delayed-input**, which is standard in the study of (interactive) zero-knowledge protocols. In an interactive protocol with delayed input, the instance and witness need not be known/chosen until some intermediate round (often the prover's final round). In our setting, the actual choice of instance/witness is not chosen until after the prover finishes "speaking". The protocol of Lapidot and Shamir [151] is often regarded as the first ZK protocol with delayed input, while Katz and Ostrovsky[134] were the first to explicitly rely on the delayed input property while using a ZK proof in an application.

Definition 19. A **delayed-input FLP** DFLP consists of the following algorithms:

- $\mathsf{DFLP}.\mathtt{Encode}(\Delta \in \mathbb{F}^{el}, x \in \mathbb{F}^{n}) \to e \in \mathbb{F}^{n'}$ takes as input encoding randomness Δ, and an input instance x. Returns an encoding of x; we let n' denote the length of the encoding. The function $\mathtt{Encode}(\Delta, \cdot)$ must be a linear function and invertible. We denote the inverse by Decode.

- $\mathsf{DFLP.Prove}(X \subseteq \mathbb{F}^{n}, \Delta \in \mathbb{F}^{el}, jr \in \mathbb{F}^{jl}) \to \pi \in \mathbb{F}^{m}$ takes as input a **set** of possible instances, encoding randomness Δ, and joint randomness jr. Produces output proof π.

- $\mathsf{DFLP.Query}(e \in \mathbb{F}^{n'}, \Delta \in \mathbb{F}^{el}, \pi \in \mathbb{F}^{m}, jr \in \mathbb{F}^{jl}; qr \in \mathbb{F}^{ql}) \to \sigma \in \mathbb{F}^{v}$ takes as input an encoded instance e, encoding randomness Δ, proof π, joint randomness jr, and query randomness qr. Returns a verifier σ. The function $\mathsf{Query}(\cdot, \cdot, \cdot, jr; qr)$ must be linear.

- $\mathsf{DFLP.Decide}(\sigma \in \mathbb{F}^{v}) \to acc \in \{0, 1\}$: Takes as input query responses σ and returns a boolean.

If Prove is restricted to sets X with $|X| = k$ then we call the construction a **delayed-k-input FLP**.

A delayed-input FLP should satisfy the following properties:

- **Completeness** (with respect to language $\mathscr{L}$): For all $X \subseteq \mathscr{L}$, all $x \in X$, and all Δ:

$$\mathsf{Pr}[\mathsf{Decide}(\sigma) : jr \leftarrow\!\!\$\, \mathbb{F}^{jl}; \pi \leftarrow\!\!\$\, \mathsf{Prove}(X, \Delta, jr);$$
$$\sigma \leftarrow\!\!\$\, \mathsf{Query}(\mathtt{Encode}(\Delta, x), \Delta, \pi, jr)] = 1\,.$$

- **Soundness** (with respect to $\mathscr{L}$): The scheme should be sound in the usual sense of FLPs, with respect to the language $\mathscr{L}^* = \{(\mathtt{Encode}(\Delta, x), \Delta) \mid x \in \mathscr{L}\}$. In other words, it is hard for a malicious prover to generate a proof that verifies with respect to $(e, \Delta) \notin \mathscr{L}^*$.

- **Privacy:** In Figure 5.6 we define a game for delayed-input FLPs, in which the proof is generated using some set X of candidates, and later verified with respect to a particular $x \in X$. A delayed-input FLP is δ-private if there exists a simulator Sim such that every A's advantage is $\mathbf{Adv}\textsc{priv}_{\mathsf{DFLP},\mathsf{Sim}}(A) \le \delta$, where

$$\mathbf{Adv}\textsc{priv}_{\mathsf{DFLP}}(A) = 2 \cdot \mathsf{Pr}[\mathsf{Exp}^{\textsc{priv}}_{\mathsf{DFLP},\mathsf{Sim}}(A)] - 1\,.$$

5.5.3 Construction

We specify our construction $\mathsf{Doplar}[\mathsf{VIDPF}, \mathsf{DFLP}, \mathsf{PRG}]$ in Figure 5.7. Its three components are: a verifiable IDPF VIDPF with input length η; a delayed-2-input FLP DFLP with input set $\{0,1\}$, proof length m, encoded input length n, encoding randomness length el, joint randomness length jl, and query randomness length ql; and a pseudorandom generator PRG (Section 5.2) with seed length κ. To be suitable for our construction, we must choose VIDPF and DFLP so that $\mathsf{VIDPF}.\mathbb{G}_\ell = \mathsf{DFLP}.\mathbb{F}^n$ for each $\ell \in [\eta]$.

To shard its measurement $\alpha \in \{0,1\}^\eta$, the client begins by running the VIDPF key generator on α. The initial state for Doplar encodes the "level" ℓ at which the VIDPF shares are to be evaluated; each candidate prefix must have length ℓ. (Recall from Section 5.2 that (V)IDPFs can be thought of as shares of values arranged in a binary tree with nodes labeled by prefixes.) For each level of the VIDPF tree, the client generates a delayed-input proof of the

```
Algorithm $\mathsf{Shard}(\alpha, n)$:
  // Construct the VIDPF key shares.
  $seed_1, seed_2 \leftarrow_\$ \{0,1\}^\kappa$
  for $\ell \in [\eta]$:
    $\vec{\Delta}[\ell] \leftarrow \mathsf{RG}_2(seed_1, n \| \ell \| 1)$
          $+ \mathsf{RG}_2(seed_2, n \| \ell \| 2)$
    $\vec{\beta}[\ell] \leftarrow \mathsf{DFLP.Encode}(\vec{\Delta}[\ell], 1)$
  $(key_1, key_2, pub) \leftarrow_\$ \mathsf{VIDPF.Gen}(\alpha, \vec{\beta})$
  // Prepare the joint randomness parts.
  $\vec{rseed}[1] \leftarrow \mathsf{RG}_5(seed_1, n \| 1 \| pub \| key_1)$
  $\vec{rseed}[2] \leftarrow \mathsf{RG}_5(seed_2, n \| 2 \| pub \| key_2)$
  // Generate the level proofs.
  for $\ell \in [\eta]$:
    $jseed \leftarrow \mathsf{RG}_6(0^\kappa, \ell \| \vec{rseed})$
    $jr \leftarrow \mathsf{RG}_1(jseed, n \| \ell)$
    $\pi \leftarrow_\$ \mathsf{DFLP.Prove}(\{0,1\}, \vec{\Delta}[\ell], jr)$
    $\vec{pf}[\ell] \leftarrow \pi - \mathsf{RG}_3(seed_2, n \| \ell)$
  // Prepare the initial message and input shares.
  $x_1 \leftarrow (key_1, seed_1, \vec{pf})$
  $x_2 \leftarrow (key_2, seed_2)$
  $M \leftarrow (pub, \vec{rseed})$
  ret $(M, x_1, x_2)$

Algorithm $\mathsf{Unpack}(\hat{j}, x, n, \ell)$:
  if $\hat{j} = 1$: $(key, seed, \vec{pf}) \leftarrow x$; $\pi \leftarrow \vec{pf}[\ell]$
  else: $(key, seed) \leftarrow x$; $\pi \leftarrow \mathsf{RG}_3(seed, n \| \ell)$
  ret $(key, seed, \pi)$

Algorithm $\mathsf{Prep}(\hat{j}, sk, state, n, M, x)$:
  if $state \in \mathcal{Q}_{\text{Init}}$: // Process initial message from client
    $(\ell, \vec{pfx}) \leftarrow state$; $u \leftarrow |\vec{pfx}|$
    $(pub, \vec{rseed}) \leftarrow M$; $(key, seed, \pi) \leftarrow \mathsf{Unpack}(\hat{j}, x, n, \ell)$
    $\Delta \leftarrow \mathsf{RG}_2(seed, n \| \ell \| \hat{j})$
    $\vec{rseed}[\hat{j}] \leftarrow \mathsf{RG}_5(seed, n \| \ell \| \hat{j} \| pub \| key)$
    $jseed \leftarrow \mathsf{RG}_6(0^\kappa, \vec{rseed})$
    $jr \leftarrow \mathsf{RG}_1(jseed, n \| \ell)$; $qr \leftarrow \mathsf{RG}_4(sk, n \| \ell)$
    $(h, \vec{y}) \leftarrow \mathsf{VIDPF.VEval}(\hat{j}, pub, key, \vec{pfx})$
    $inp \leftarrow \sum_{i \in [u]} \vec{y}[i]$
    $\sigma \leftarrow \mathsf{DFLP.Query}(inp, \Delta, \pi, jr; qr)$
    $M \leftarrow (\sigma, \vec{rseed}[\hat{j}], h)$; $state \leftarrow (jseed, (\mathsf{DFLP.Decode}(\vec{y}[i]))_{i \in [u]})$
    ret $(\mathsf{running}, state, M)$
  // Process broadcast messages from aggregators
  $(jseed, \vec{y}) \leftarrow state$; $\big((\sigma_1, rseed_1, h_1), (\sigma_2, rseed_2, h_2)\big) \leftarrow M$
  $acc \leftarrow \mathsf{DFLP.Decide}(\sigma_1 + \sigma_2)$
  if $acc$ and $jseed = \mathsf{RG}_6(0^\kappa, (rseed_1, rseed_2))$
      and $\mathsf{VIDPF.Verify}(h_1, h_2)$: ret $(\mathtt{finished}, \vec{y})$
  else: ret $(\mathtt{failed}, \bot)$

Algorithm $\mathsf{Agg}(\vec{y})$: ret $\sum_{i=1}^{|\vec{y}|} \vec{y}[i]$

Algorithm $\mathsf{Unshard}(\_, \vec{a})$: ret $\sum_{i=1}^{|\vec{a}|} \vec{a}[i]$

Algorithm $\mathsf{RG}_i(seed, cntxt)$:
  $l \leftarrow (jl, el, m, ql)$
  if $i \le 4$: ret $\mathsf{Expand}[\mathsf{PRG}](seed, \ell_i \| cntxt, \mathbb{F}.p, l[i])$
  else: ret $\mathsf{PRG.Next}(\mathsf{PRG.INIT}(seed, \ell_i \| cntxt), \kappa)$
```

Figure 5.7. Definition of 1-round, 2-party VDAF $\mathsf{Doplar}[\mathsf{VIDPF}, \mathsf{DFLP}, \mathsf{PRG}]$. Let $\ell_1, \ldots, \ell_6$ be arbitrary, distinct bitstrings.

refined shares' validity; just as for Prio3 (Section 5.4), the joint randomness used at each level is derived from the aggregator's input shares. The VIDPF output is programmed so that the sum of the output shares corresponds to an encoded input for the delayed-input FLP.

To prepare a report for aggregation, the aggregators evaluate their VIDPF key shares at the desired candidate prefixes, then interact in order to check that (1) the joint randomness was computed correctly, (2) their refined shares are one-hot, and (3) the sum of their refined shares is either one or zero.

Allowed initial states

An initial state is valid if it consists of a sequence of candidate prefixes all having the same length. Moreover, each of the prefixes must be distinct. An initial state is allowed for $\mathsf{Doplar}[\mathsf{VIDPF}, \mathsf{DFLP}, \mathsf{PRG}]$ if the prefix length is distinct from all previous states for the same report. That is, the allowed-state algorithm $\mathsf{validSt}$ only permits a new state $\mathit{state} = (\ell, \vec{pfx})$ if ℓ is distinct for all previous states and each of the prefixes $\vec{pfx}$ is distinct.

Remark 8. Although not addressed in Boneh et al. [59] explicitly, this restriction on the candidate prefixes is necessary for Poplar as well, as re-using the correlated randomness shared by the client would reveal information about the secret-shared vector.

Consistency

The set of refined measurements for Doplar are one-hot vectors over the field $\mathbb{F}$ for which the non-zero element is equal to 0 or 1. For a given initial state $(\ell, \vec{pfx})$, this can be computed from the VIDPF public share and key shares by evaluating the shares on each of the prefixes $\vec{pfx}$. Since the VIDPF is a point function and the prefixes are distinct, the vector of VIDPF outputs will contain at most one nonzero entry. Aggregation consistency for Doplar is similarly straight-forward, since the refined share space and aggregate share space are the same and both aggregation and unsharding are vector summation. When we let $\mathsf{finishResult}$ be vector summation as well, the desired property is trivially true.

Security

Let $\Pi = \mathsf{Doplar}[\mathsf{VIDPF}, \mathsf{DFLP}, \mathsf{PRG}]$ as specified above. Let N be the nonce space and let κ be the seed length for PRG.

Theorem 20. *Modeling each* RG_i *in Figure 5.7 as a random oracle, if* DFLP *is* ε*-sound, then for all* $t_{\mathscr{A}}$*-time adversaries* $\mathscr{A}$ *and* $t_{\mathscr{E}}$*-time extractors* $\mathscr{E}$ *there exists a* $O(t_{\mathscr{A}} + q_{\mathsf{Prep}} t_{\mathscr{E}})$*-time adversary* $\mathscr{B}$ *for which*

$$\mathbf{Adv}\mathrm{robust}_{\Pi}(\mathscr{A}) \le 2(q_{\mathsf{RG}} + q_{\mathsf{Prep}}) \cdot \varepsilon + \frac{(q_{\mathsf{RG}} + 3q_{\mathsf{Prep}})^2}{2^{\kappa}}$$
$$+ q_{\mathsf{Prep}} \cdot \mathbf{Adv}\mathrm{extract}_{\mathsf{VIDPF},\mathscr{E}}(\mathscr{B}),$$

where $\mathscr{A}$ makes q_{Prep} queries to $\underline{\mathsf{Prep}}$ *and a total of q_{RG} queries to its random oracles.*

Proof of Proof sketch: The proof has a similar structure to Theorem 18 in that the last step is a reduction to the soundness of DFLP. However in order to use this, we must first revise the game so that the challenge input issued by the malicious prover P^* was constructed from the sum of refined shares that are otherwise valid (i.e., one-hot). Using the extractability property of VIDPF, we can simplify the winning condition by extracting the the input measurement from the adversary's random oracle queries and use it to compute the refined measurement whenever the one-hotness check succeeds. Refer to Section 5.9.3 for the proof. ▮

Theorem 21. *For all $t_{\mathscr{A}}$-time adversaries $\mathscr{A}$ and t'-time simulators $\mathscr{S}, \mathscr{T}$ there exist $O(t_{\mathscr{A}} + q_{\mathsf{Shard}} t')$-time adversaries $\mathscr{B}, \mathscr{C}$ for which*

$$\mathbf{Adv}\text{PRIV}_{\Pi,1}(\mathscr{A}) \leq 2q_{\mathsf{Shard}}\Big(\mathbf{Adv}\text{PRIV}_{\mathsf{VIDPF},\mathscr{S}}(\mathscr{B}) + \eta \cdot \mathbf{Adv}\text{PRIV}_{\mathsf{DFLP},\mathscr{T}}(\mathscr{C}) + \frac{\eta q_{\mathsf{RG}} + q_{\mathsf{Shard}}}{|N|} + \frac{3q_{\mathsf{RG}}}{2^{\kappa-1}}\Big),$$

where each RG_i in Figure 5.7 is modeled as a random oracle, adversary $\mathscr{A}$ makes a total of q_{RG} queries to all of its random oracles and q_{Shard} queries to $\underline{\mathsf{Shard}}$.

Proof of Proof sketch: The reduction to DFLP privacy follows the same lines as Theorem 19 except there are $\eta \cdot q_{\mathsf{Shard}}$ different hybrid worlds in the last step. Privacy of VIDPF is used to ensure that the simulation of the boundary world can be carried out without access to the input measurement. Refer to Section 5.9.4 for the proof. ▮

5.5.4 Performance Evaluation

In this section we compare the cost of Doplar to Poplar1 in terms of communication (total bits written to the wire) and computation. The parameters chosen for Poplar1 by the specification [25] match those in the performance evaluation conducted by Boneh et al. [58]. We therefore take these parameters as our basis for comparison. In the following, we have instantiated VIDPF and DFLP as described in Section 5.7 and Section 5.8 respectively.

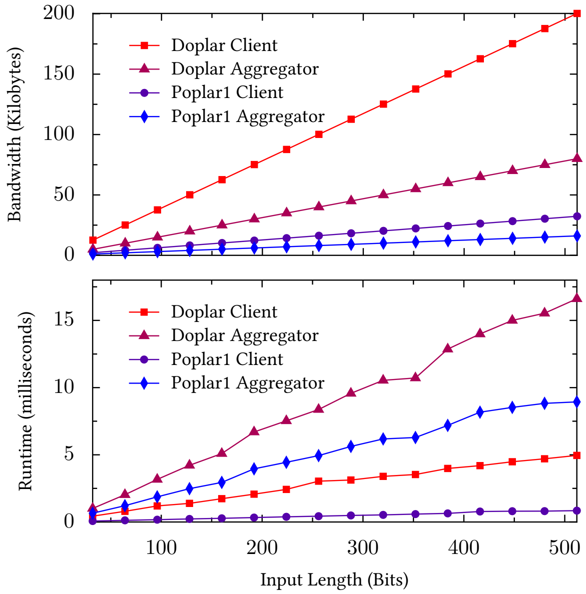


Figure 5.8. Bandwidth (top) and runtime (bottom) for Doplar and Poplar1.

Boneh et al. [58] claim a per-report robustness bound of roughly $2/|\mathbb{F}|$, where $\mathbb{F}$ is the field chosen for the inner nodes.[5] They choose a 62-bit field. In order to obtain the same robustness bound, while permitting the adversary at most 2^{64} queries to its random oracles, we need to use a 128-bit field for Doplar. For both constructions, we instantiate the PRG with AES-128 as described in [25, Section 6.2] (hence the seed length is $\kappa = 128$).

Communication overhead

In Figure 5.8 we plot the communication cost of Doplar and Poplar1 for various choices of the input length η. We plot the total number of kilobytes sent by each client. We also plot the total number of kilobytes sent by each aggregator, per report, over all η rounds of aggregation. As one would expect, the communication cost for Doplar scales linearly with the input length. However, the client's bandwidth is about 6 times that of Poplar1; and the Aggregator's bandwidth is about 5 times.

Computational overhead

To evaluate Doplar's computational overhead, we implement a prototype[6] and benchmark it against an existing implementation of Poplar1. The ISRG (Internet Security Research Group)

[5] Poplar1 uses a smaller field for the inner nodes of the IDPF tree than the leaf nodes.
[6] https://github.com/cloudflareresearch/doplar/tree/cjpatton/PoPETS-2023.4-Artifact

maintains Rust implementations of the current crop of VDAF standard candidates.[7] The code includes a work-in-progress version of Poplar1 (on a development branch, as of this writing) as well as the FLP and IDPF primitives we use in our own implementation of Doplar.

We use the Criterion framework for Rust.[8] All benchmarks reported below were run on a 2019 MacBook Pro (2.6 GHz 6-Core Intel Core i7) running rustc version 1.67.1 and cargo-criterion version 1.1.0. The default parameters were used, except the measurement time was set to 30 seconds for all benchmarks.

Microbenchmarks for sharding. To benchmark the client, we chose a random input string of the desired length, then measured the runtime of the sharding algorithm on that input. Figure 5.8 shows the runtimes for lengths ranging from 32 to 512 bits. From these data we see that sharding is about 6 times as expensive for Doplar as for Poplar1. However, sharding a 512-bit input takes only 5 milliseconds, which is still quite practical. (Moreover, there is more room for optimization of our prototype.)

Microbenchmarks for preparation. Due to the highly parallelizable nature of VDAFs, much of the time the aggregators spend on executing the protocol is network-bound. However, it is useful to assess the amount of CPU time spent on processing a single report. To do so, we report microbenchmarks for per-report preparation, specifically how much time it takes an aggregator to compute its (first) broadcast message from the initial state provided by the collector and the input share provided by the client. Let us call this "preparation initialization".

One complicating factor is that the runtime of IDPF evaluation depends intrinsically on the distribution of the batch of measurements and the heavy-hitters threshold used. (We refer the reader to Algorithm 3 in Boneh et al. [59] for details.) To address this, we generated a synthetic batch of measurements and computed the prefix tree (cf. [59, Section 5.1]) for the desired threshold, then ran preparation initialization on the longest paths of this tree.[9]

The following experiment was run 10 times. Following Boneh et al. [59], we sample random input strings from a Zipf distribution (with parameter 1.03 and support 128), then

[7] Source code for the `prio` crate: https://github.com/divviup/libprio-rs

[8] Criterion: https://docs.rs/criterion/latest/criterion/

[9] Note that IDPFs can be implemented with cross-aggregation cache, which amortizes longest-path evaluation over multiple aggregations.

compute the prefix tree with a heavy-hitters threshold of 10. We chose a batch size of 1000. For both Doplar and Poplar, run Criterion to measure the runtime of preparation for the longest paths of the tree.

Figure 5.8 shows the runtime averaged over all trials for lengths ranging from 32 to 512 bits. From these data we see that preparation is only about 1.75 times as expensive for Doplar as for Poplar1. This is not surprising, given that the runtime is dominated by IDPF evaluation, which in turn depends on the number of candidates.

Level skipping. One way to improve bandwidth for both schemes is to "skip" IDPF evaluation at certain levels. For example, if we descend the IDPF tree in τ-bit increments instead of 1-bit increments, then (1) our VIDPF construction requires one-hot check material only in every τ-th level, and (2) the Doplar construction requires DFLPs only at every τ-th level.[10] As a result, these major contributors to communication cost are reduced by a factor of τ. Additionally, the process of aggregating (traversing the tree of prefixes to find heavy hitters) requires fewer rounds by a factor of τ. The trade-off is that we consider more candidate prefixes at each level—i.e., at each step we consider the 2^τ descendants at depth τ from each candidate—but this cost is amortized over the batch.

Notably, the impact of this optimization is more significant for Doplar than for Poplar1. (For example, a "skip factor" of $\tau = 2$, i.e., skipping every other level, reduces the client's overhead from 6 to 5 times that of Poplar1 with the same optimization.) This is primarily due to the reduction in the number of delayed-input proofs, which make up the bulk of the first input share. (The second input share compresses its shares of the proofs into a single PRG seed.)

5.6 Conclusion and Future Work

The PPM working group's ambition is to preserve user privacy even as software systems rely increasingly on gaining insights into user behavior. Our work aims to help ensure that this effort rests on firm formal foundations. However, we leave open a number of directions for future work. We discuss two in the remainder.

[10] The underlying (non-verifiable) IDPF is still organized as a binary tree, so its cost is not affected.

Security analysis of DAP

The definitions in this paper apply to VDAFs, which are only a component of the DAP specification [104]. Thus, our work necessarily leaves open the security of the end-to-end protocol. There are two important questions. First, DAP is designed to inherit the security properties of VDAF, i.e., one would hope that whatever can be proven about the VDAF also holds when the VDAF is instantiated in the real-world environment in which DAP runs. One way to address this is to formulate the problem in terms of *indifferentiability* [181]: if DAP's execution can be shown to be indifferentiable from the execution of the VDAF in the idealized environment described here, then any attack against DAP can be translated into an attack against the underlying VDAF.

The other important question is whether DAP meets its own security goals, which, depending on the application, might go beyond what can be achieved with a VDAF alone. Consider that whether MPC-style definitions like ours are enough for privacy depends intrinsically on the nature of the measurements being collected and how they are aggregated. It is one thing to ensure that we securely compute the aggregate; it is another to ensure that the aggregate itself does not leak "too much" information about the measurements. In particular, in many applications it will be useful to achieve differential privacy (DP) [98] in addition to secure computation. There are definitions of DP that extend to the multi-party setting [163, 192], and a number of works have considered MPC protocols for aggregation functionalities that also guarantee differential privacy of the outputs [188, 122, 26]. We hope to see future work extend this investigation to specific VDAFs.

Doplar improvements

For some applications, it would be useful for Doplar (or Poplar1) if the leaf output could be "weighted", i.e., a number in range $\{a,\ldots,b\}$ rather than $\{0,1\}$. (Consider the ad-conversion use case from Section 5.1: it might be useful to know not only how many purchases were made per ad impression, but the total amount of money that was spent.) The delayed-k-input FLP paradigm may allow for this generalization, if schemes can be constructed for $k > 2$. (In this work, we only construct the delayed-2-input FLP needed for plain heavy hitters.)

There is also room for improvement of the communication cost. Despite the round reduction, the higher bandwidth may be prohibitive for some applications. However, we are optimistic that the bandwidth can be improved. Future work should focus on the delayed-2-input FLP. The current instantiation (Section 5.8), while simple, effectively doubles the proof size of the base FLP.

Acknowledgements

Thank you to the anonymous reviewers from the PETS 2023 program committee whose feedback helped us improve a number of technical aspects of our paper. Thank you as well to Christopher Wood who helped us position this work in the context of the ongoing standardization effort at IETF. Finally, thanks to Nikita Borisov, Sofía Celi, Tanya Verma, Tara Whalen, and Avani Wildani for editorial improvements.

Hannah and Mike carried out their work on this paper while visiting Cloudflare Research. This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

5.7 Instantiating VIDPF

In this section we present our proposed VIDPF construction.

De Castro-Polychroniadou technique.

De Castro & Polychroniadou [85] (hereafter DP22) proposed the following simple and elegant technique to verify that a vector is one-hot. Consider a vector $\vec{v}$ that is additively secret-shared $\vec{v} = \vec{v}_1 \oplus \vec{v}_2$. For simplicity, we describe the technique assuming that the sharing is with respect to XOR, since in that case the shares of zero are *identical strings.* The technique adapts readily to the more general case of additive shares over any group. Assume also that the parties have additive shares of a *binary* indicator vector $\vec{b} = \vec{b}_1 \oplus \vec{b}_2$, which is nonzero exactly in the same positions that $\vec{v}$ is.

First, observe that the parties can easily verify whether they hold shares of an all-zeroes vector, since this happens if and only if their shares (as strings) are identical. They can simply exchange and compare hashes of their share-vectors (although see our remark below for a disclaimer about this idea). The technique of DP22 is to adjust a one-hot vector into an all-zeroes vector, with the help of the dealer.

Define

$$\mathsf{adjust}(\vec{v}_i, \vec{b}_i, C) = \Big(H(1, \vec{v}_i[1]) \oplus \vec{b}_i[1] \cdot C, \quad H(2, \vec{v}_i[2]) \oplus \vec{b}_i[2] \cdot C, \ldots \Big)$$

If $\vec{v}$ and $\vec{b}$ are nonzero in (only) position i^*, then set $C^* = H(i^*, \vec{v}_1[i^*]) \oplus H(i^*, \vec{v}_2[i^*])$. Now consider the result of both shareholders applying $\mathsf{adjust}(\cdot, \cdot, C^*)$ to their shares:

- In positions $i \neq i^*$ where they share zero, we have $\vec{v}_1[i] = \vec{v}_2[i]$ and $\vec{b}_1[i] = \vec{b}_2[i]$. For these positions in the output of adjust, both parties will compute identical strings.
- In position i^*, the parties have $\vec{b}_1[i^*] \neq \vec{b}_2[i^*]$. By symmetry, suppose $\vec{b}_1[i^*] = 1$ and $\vec{b}_2[i^*] = 0$. Then the first party will compute

  $$\begin{aligned} & H(i^*, \vec{v}_1[i^*]) \oplus C^* \\ & \qquad = H(i^*, \vec{v}_1[i^*]) \oplus \big(H(i^*, \vec{v}_1[i^*]) \oplus H(i^*, \vec{v}_2[i^*]) \big) \\ & \qquad = H(i^*, \vec{v}_2[i^*]) \end{aligned}$$

  and the second party will compute $H(i^*, \vec{v}_2[i^*])$ as well.

In all cases, both parties will compute the same output of adjust, which they can check for equality by exchanging and comparing hashes. Hence, the dealer will compute the C^* value and include it in the parties' DPF keys. They can use C^* to perform their verification.

To see why the DP22 approach is sound, suppose the parties hold shares of a non-one-hot vector — i.e., it is nonzero at positions $i \neq i'$. Do both parties compute the same output of adjust? This can only happen if C^* value somehow corrects both positions i and i', and this happens only when

$$\begin{aligned} & H(i, \vec{v}_1[i]) \oplus H(i, \vec{v}_2[i]) = C^* = H(i', \vec{v}_1[i']) \oplus H(i', \vec{v}_2[i']) \\ \iff & H(i, \vec{v}_1[i]) \oplus H(i, \vec{v}_2[i]) \oplus H(i', \vec{v}_1[i']) \oplus H(i', \vec{v}_2[i']) = 0 \end{aligned}$$

The construction is therefore sound if it is hard to find "multi-collisions" of this form in H. In particular, if H is a random oracle with output length 4κ then an adversary making $q < 2^\kappa$ queries to H can find such a collision with probability bounded by $q^4/2^{4\kappa} \ll q/2^\kappa$.

Regarding privacy, there is one subtle issue that must be considered. Suppose party #1 holds its share $\vec{v}_1$ and the correction value $C^* = H(i^*, \vec{v}_1[i^*]) \oplus H(i^*, \vec{v}_2[i^*])$. Suppose this party has a guess for i^* and a guess for the nonzero value $v = \vec{v}_1[i^*] \oplus \vec{v}_2[i^*]$. Then she can verify this guess by checking whether $C^* = H(i^*, \vec{v}_1[i^*]) \oplus H(i^*, \vec{v}_1[i^*] \oplus v)$ — all values she knows. Hence, C^* exposes an offline dictionary attack on the secret values i^* and $\vec{v}[i^*]$. If $\vec{v}[i^*]$ is high entropy, then this is no vulnerability at all. But if $\vec{v}[i^*]$ is known to be a small value like 1 (as is the case in many applications), then this issue allows a corrupt shareholder to unilaterally learn i^*, violating privacy. We resolve this by simply ensuring that the dealer encodes a random element at the one-hot position (in addition to a potentially low-entropy desired value).[11]

Extending to incremental DPF

The technique of DP22 is well-suited for DPFs. In an incremental DPF (IDPF), we can apply their technique to each prefix-length. However, this guarantees only that each prefix-length corresponds to some point function. It does not necessarily guarantee that the point functions of the different prefix-lengths satisfy the prefix condition that is needed in an IDPF.

In our construction, we extend the DP22 technique to IDPFs. For each evaluation of the IDPF — say, at point x — we compute the adjustment strings using the DP22 technique, for x *and all of its prefixes.* This alone is not enough to guarantee the prefix property. To "tie different prefix lengths together," we ask the shareholders to compute the adjustment strings with respect to the *same sharings of the indicator bit*, for all the prefixes of x. We show that this forces the point functions at every prefix-length to be prefix-consistent.

[11] We have chosen to describe our VIDPF to use an underlying IDPF as a black-box. When this is the case, we must ensure that the IDPF outputs have sufficient entropy for the one-hotness check. If we were to instead to analyze our VIDPF (instantiated with a natural IDPF construction) as a *monolithic construction*, it is likely that the underlying IDPF would already have internal entropy available that could be used for the one-hotness check. I.e., we may be able to obtain smaller share sizes by exploiting internal properties of the underlying IDPF.

```
$\mathsf{VIDPF.Gen}(\alpha \in \{0,1\}^\eta, \vec{\beta} \in \mathbb{G}_1 \times \cdots \times \mathbb{G}_\eta)$:
for $\ell \in [\eta]$:
    $\vec{R}[\ell] \leftarrow\!\$\, \{0,1\}^\kappa$
    $\vec{\beta}^*[\ell] \leftarrow (1, \vec{\beta}[\ell], \vec{R}[\ell])$
$(key_1, key_2, pub) \leftarrow \mathsf{IDPF.Gen}(\alpha, \vec{\beta}^*)$
for $\ell \in [\eta]$:
    $pfx \leftarrow \alpha[1:\ell]$
    $(\_, data_1, R_1) \leftarrow \mathsf{IDPF.Eval}(1, key_1, pub, pfx)$
    $(\_, data_2, R_2) \leftarrow \mathsf{IDPF.Eval}(2, key_2, pub, pfx)$
    $\vec{C}[\ell] \leftarrow \mathsf{RG}(pfx \,\|\, -data_1 \,\|\, -R_1)$
            $\oplus \mathsf{RG}(pfx \,\|\, data_2 \,\|\, R_2)$
$pub^* \leftarrow (pub, \vec{C})$
ret $(key_1, key_2, pub^*)$

$\mathsf{VIDPF.VEval}(id, key, pub^*, \vec{x})$:
$(pub, \vec{C}) \leftarrow pub^*$
for $i \in [\ |\vec{x}|\ ]$
    $\vec{y}[i] \leftarrow \mathsf{IDPF.Eval}(id, key, pub, \vec{x}[i])$
    $(b, \vec{data}[i], R) \leftarrow \vec{y}[i]$
    $h \leftarrow h \,\|\, \mathsf{adjust}(id, key, pub^*, b, \vec{x}[i])$
ret $(h, \vec{data})$

$\mathsf{VIDPF.adjust}(id, key, pub^*, b, x)$:    // a helper procedure
$(pub, \vec{C}) \leftarrow pub^*$
if $|x| = 0$: ret $x$    // length of $x$ as a bit string
$(\_, d, R) \leftarrow \mathsf{IDPF.Eval}(id, key, pub, x)$
$prefix \leftarrow \mathsf{adjust}(id, key, pub^*, b, x[1:|x|-1])$
ret $prefix \,\|\, \Big(\mathsf{RG}(x \,\|\, (-1)^i dd \,\|\, (-1)^i dR) \oplus b \cdot \vec{C}[|x|]\Big)$

$\mathsf{VIDPF.Verify}(h_1, h_2)$:
    ret $h_1 == h_2$
```

Figure 5.9. VIDPF construction $\mathsf{VIDPF[IDPF]}$, based on any IDPF. If the VIDPF is to be instantiated with groups $\mathbb{G}_1, \ldots, \mathbb{G}_\eta$ then the underlying IPDF is instantiated with groups $\widetilde{\mathbb{G}}_1, \ldots, \widetilde{\mathbb{G}}_\eta$, where $\widetilde{\mathbb{G}}_\ell = \{0,1\} \times \mathbb{G}_\ell \times \{0,1\}^\kappa$.

Immediate Optimizations in an Implementation

Our construction evaluates the underlying IDPF on all prefixes of the given strings. Doing this naïvely would increase the computational costs by a factor of ℓ when evaluating on strings of length ℓ. However, these extra evaluations are essentially free in existing IDPFs — while evaluating at string x, these constructions already evaluate all prefixes of x along the way. A reasonable implementation of our VIDPF will take advantage of this fact.

The verification value h produced by VEval is a very long string, consisting of $\ell \cdot 4\kappa$ bits for each query point of length ℓ. If parties are to exchange these h values in an application of our VIDPF, it would account for a significant fraction of the total communication. However, the Verify algorithm that uses these h values merely checks them for equality. Therefore, it suffices for each party to send only a collision-resistant hash of their h value, which can have fixed length only 2κ. This optimization changes the concrete security bound for VIDPF soundness, by adding a term for the probability of finding a collision under the hash function.

Lemma 7. *Let* IDPF *be an IDPF and* RG *be a random oracle with outputs of length* 4κ*. Let* A *be an adversary making* q *queries to* RG*. There is a* $O(t_A)$*-time adversary* A' *such that the* *construction* $\mathsf{VIDPF}[\mathsf{IDPF}]$ *in Figure 5.9 satisfies the following:*

$$\mathbf{Adv}\text{extract}_{\mathsf{VIDPF[IDPF]},\mathscr{E}}(A) \leq (q^4+q^2)/2^{4\kappa}$$

$$\mathbf{Adv}\text{PRIV}_{\mathsf{VIDPF[IDPF]}}(A) \leq \mathbf{Adv}\text{PRIV}_{\mathsf{IDPF}}(A') + q/2^{\kappa}$$

Proof: Correctness of our construction follows from the discussion above, and is the same as in DP22.

Extractability: We begin with a few observations, which hold for all VIDPF keys, even adversarially generated ones:

Observation: If $\mathsf{adjust}(1, key_1, pub^*, b_1, x) = \mathsf{adjust}(2, key_2, pub^*, b_2, x)$, then $\mathsf{adjust}(1, key_1, pub^*, b_1, x') =$ $\mathsf{adjust}(2, key_2, pub^*, b_2, x')$ as well, for every prefix x' of x. This follows trivially by inspection and the recursive nature of adjust. Note that the same b_1, b_2 are used for both x and x'.

Observation: Let $pub^* = (pub, \vec{C})$. Suppose $\mathsf{adjust}(1, key_1, pub^*, b_1, x) = \mathsf{adjust}(2, key_2, pub^*, b_2, x)$, and $\mathsf{IDPF.Eval}(1, key_1, pub, x) = (\_, y_1, R_1)$, and $\mathsf{IDPF.Eval}(2, key_2, pub, x) = (\_, y_2, R_2)$. Then:

1. If $b_1 = b_2$ then $\mathsf{RG}(x \| -y_1 \| -R_1) = \mathsf{RG}(x \| y_2 \| R_2)$. This includes the case where $(y_1, R_1) + (y_2, R_2) = (0,0)$, making the two calls to RG identical. It also includes the case where these two calls to RG are a collision.

2. If $b_1 \neq b_2$ then $\vec{C}[|x|] = \mathsf{RG}(x \| -y_1 \| -R_1) \oplus \mathsf{RG}(x \| y_2 \| R_2)$.

This observation can be verified by inspection.

Let $\mathscr{E}_1$ denote the bad event that the adversary queries RG and observes a collision. If RG has outputs of length 4κ, and the adversary makes q oracle queries, then the probability of this bad event is bounded by $q^2/2^{4\kappa}$. When $\mathscr{E}_1$ does *not* happen, then in condition (1) above, only the case that $(y_1, R_1) + (y_2, R_2) = (0,0)$ is possible.

```
$\mathscr{E}(key_1, key_2, pub^*, \mathrm{Rand})$:
$(pub, \vec{C}) \leftarrow pub^*$
if $\mathscr{E}_1$ or $\mathscr{E}_2$: // defined in the text, here with respect to oracle queries listed in Rand
  abort
$\alpha \leftarrow$ empty string
for $\ell \in [\eta]$:
  if $\exists a \in \{0,1\}, y_1, R_1, y_2, R_2$ such that
      $\vec{C}[\ell] = \mathrm{Rand}[(\alpha \| a) \,\|\, {-y_1} \,\|\, {-R_1}] \oplus \mathrm{Rand}[(\alpha \| a) \,\|\, y_1 \,\|\, R_2]$
    $\alpha \leftarrow \alpha \,\|\, a$
    $\vec{\beta}[\ell] \leftarrow y_1 + y_2$
  else: $\alpha \leftarrow \alpha \,\|\, 0$; $\vec{\beta}[\ell] \leftarrow 0$
ret $(\alpha, \vec{\beta})$
```

Figure 5.10. Extractor for the proof of Lemma 7.

Let $\mathscr{E}_2$ denote the bad event that the adversary makes any four queries to RG that satisfy:

$$\mathsf{RG}(pfx \,\|\, {-d_1} \,\|\, {-R_1}) \oplus \mathsf{RG}(pfx \,\|\, d_2 \,\|\, R_2) =$$

$$\mathsf{RG}(pfx' \,\|\, {-d_1'} \,\|\, {-R_1'}) \oplus \mathsf{RG}(pfx' \,\|\, d_2' \,\|\, R_2')$$

for $pfx \neq pfx'$ and $d_1 + d_2 \neq 0$ and $d_1' + d_2' \neq 0$. (These conditions ensure that the four calls to RG must be on distinct inputs.) If RG has outputs of length 4κ, and the adversary makes q oracle queries, then the probability of this bad event is bounded by $q^4/2^{4\kappa}$. When $\mathscr{E}_2$ does *not* happen, then any value $C \in \{0,1\}^{4\kappa}$ uniquely determines *at most one* pair of queries satisfying $C = \mathsf{RG}(pfx \,\|\, {-d_1} \,\|\, {-R_1}) \oplus \mathsf{RG}(pfx \,\|\, d_2 \,\|\, R_2)$

We can apply the two observations inductively and obtain the following. If $\mathsf{adjust}(1, key_1, pub^*, b_1, x) = \mathsf{adjust}(2, key_2, \ pub^*, b_2, x)$ for $b_1 \neq b_2$, then every correction word $\vec{C}[\ell]$ must be of the form $\mathsf{RG}(x[1:\ell] \,\|\, \cdots) \oplus \mathsf{RG}(x[1:\ell] \,\|\, \cdots)$, for $\ell \leq |x|$. Then, provided that $\mathscr{E}_2$ does not happen, there is at most one x of length ℓ for which $\vec{C}$ can be written in this way.

Combining all of these observations, we can define the extractor as shown in Figure 5.10.

Conditioned on the event that $\mathscr{E}$ doesn't abort (which happens only with probability $(q^4+q^2)/2^{4\kappa}$), we claim that the adversary has no advantage in the extractability game.

Consider a query to $\underline{\mathsf{Eval}}(\vec{x})$ in the game, and assume the call to Verify succeeds. Then for every $\vec{x}[i]$, the corresponding calls to adjust produce identical output. If these calls to adjust have $b_1 = b_2$, and $\mathscr{E}_1$ has not happened, then the corresponding output $\vec{y}[i]$ must be 0. If these calls to adjust have $b_1 \neq b_2$, then $\vec{C}[\ell]$ must have the form $\mathsf{RG}(\vec{x}[i] \,\|\, \cdots) \oplus \mathsf{RG}(\vec{x}[i] \,\|\, \cdots)$. If $\mathscr{E}_2$ has not happened, then $\vec{x}[i]$ is in fact unique with this property, and therefore $\vec{x}[i]$ is a prefix of α computed by the extractor $\mathscr{E}$. One can easily check that $\mathscr{E}$ extracts $\vec{\beta}[\ell]$ that is equal to the VIDPF output $\vec{y}[i]$. In other words, $\vec{y}$ matches the output of $f_{\alpha,\vec{\beta}}$. Hence, the adversary's advantage is zero.

Privacy: Let $\mathsf{Sim}^{\mathsf{IDPF}}$ be the simulator for privacy for the underlying IDPF. The simulator for our construction is given in Figure 5.11. We prove privacy in a series of hybrids, also illustrated in Figure 5.11. Game G0 refers to the original experiment $\mathsf{Exp}^{\mathrm{PRIV}}_{\mathsf{VIDPF}}$, where we have inlined the definition of Sim_2 for convenience. The $b = 0$ and $b = 1$ branches of the $\underline{\mathsf{Sketch}}$ oracle differ only in whose shares are given as input to $\mathsf{VIDPF.VEval}$. By the correctness of the scheme, the distinction doesn't matter, so the $\underline{\mathsf{Sketch}}$ oracle is independent of b. Eliminating the conditional in the $\underline{\mathsf{Sketch}}$ oracle, we obtain G1, which is distributed identically to G0.

G2 is identical to G1, but we have inlined the definition of $\mathsf{VIDPF.Gen}$ for convenience. By the correctness of the underlying IDPF, outputs of $\mathsf{IDPF.Eval}(1,\cdot)$ and $\mathsf{IDPF.Eval}(2,\cdot)$ are secret-shares of the appropriate plaintext values. So it has no effect on the adversary's view to solve for the output of $\mathsf{IDPF.Eval}(3-\hat{j},\cdot)$ using the plaintext values and the output of $\mathsf{IDPF.Eval}(\hat{j},\cdot)$, instead of using $key_{3-\hat{j}}$. In doing so, we obtain G3 which is distributed identically to G2.

Now notice that in G3, the value $key_{3-\hat{j}}$ is never used. As such, we can replace line 26 (the call to $\mathsf{IDPF.Gen}$) with a corresponding call to the simulator $\mathsf{Sim}^{\mathsf{IDPF}}$, which generates a simulated $key_{\hat{j}}$ and pub. Call the result G4 (not pictured); this advantage in distinguishing G3 from G4 is at most $\varepsilon_{\mathrm{PRIV}}$.

In G4, the random values $\vec{R}[\ell]$ are used only to solve for $R_{3-\hat{j}}$, which is in turn used only as an argument to RG. Define a bad event that the adversary ever queries RG at an input of this form — i.e., of the form $\mathsf{RG}(\cdot \,\|\, \cdot \,\|\, \vec{R}[\ell] - R_{\hat{j}})$. The probability of the bad event is bounded by $q/2^{\kappa}$ since $\vec{R}[\ell]$ is uniformly random. Conditioned on this bad event not happening, the results of these queries to RG are freshly random, and the value that is assigned to $\vec{C}[\ell]$ is uniform. In that case, the behavior of the game is independent of the challenge bit because Sim_1 also assigns uniform values to $\vec{C}[\ell]$. The advantage in guessing the challenge bit is therefore bounded by the

```
$\underline{\mathsf{Sim}_1(\hat{j})}$:
  $(key, pub) \leftarrow \mathsf{Sim}^{\mathsf{IDPF}}()$
  for $\ell \in [\eta]$: $\vec{C}[\ell] \leftarrow\$ \{0,1\}^{4\kappa}$
  $pub^* \leftarrow (pub, \vec{C})$
  ret $(key, pub^*)$

$\underline{\mathsf{Sim}_2(\hat{j}, key, pub, \vec{x})}$:
  $(h, \_) \leftarrow \mathsf{VIDPF.VEval}(\hat{j}, key, pub, \vec{x})$
  ret $h$
```

```
Game $\colorbox{lightgray}{G0}$ $\boxed{\text{G1}}$:
  $b \leftarrow\$ \{0,1\}$
  $(state_{\mathscr{A}}, \alpha, \vec{\beta}, \hat{j}) \leftarrow \mathscr{A}()$
  if $b = 0$: $(key_{\hat{j}}, pub^*) \leftarrow\$ \mathsf{Sim}_1(\hat{j})$
  else: $(key_1, key_2, pub^*) \leftarrow\$ \mathsf{VIDPF.Gen}(\alpha, \vec{\beta})$
  $b^* \leftarrow \mathscr{A}^{\mathsf{Sketch}}(state_{\mathscr{A}}, key_{\hat{j}}, pub^*)$
  ret $b = b^*$

$\underline{\mathsf{Sketch}(\vec{x})}$:
  if $b = 0$:
    // $h \leftarrow \mathsf{Sim}_2(\hat{j}, key_{\hat{j}}, pub^*, \vec{x})$
    $(h, \_) \leftarrow \mathsf{VIDPF.VEval}(\hat{j}, key_{\hat{j}}, pub^*, \vec{x})$
  else: $(h, \_) \leftarrow \mathsf{VIDPF.VEval}(3 - \hat{j}, key_{3-\hat{j}}, pub^*, \vec{x})$
  ret $h$
```

```
Game $\colorbox{lightgray}{G2}$ $\boxed{\text{G3}}$:
  $b \leftarrow\$ \{0,1\}$
  $(state_{\mathscr{A}}, \alpha, \vec{\beta}, \hat{j}) \leftarrow \mathscr{A}()$
  if $b = 0$: $(key_{\hat{j}}, pub^*) \leftarrow\$ \mathsf{Sim}_1(\hat{j})$
  else:
    // $(key_1, key_2, pub^*) \leftarrow\$ \mathsf{VIDPF.Gen}(\alpha, \vec{\beta})$:
    for $\ell \in [\eta]$:
      $\vec{R}[\ell] \leftarrow\$ \{0,1\}^{\kappa}$
      $\vec{\beta}^*[\ell] \leftarrow (1, \vec{\beta}[\ell], \vec{R}[\ell])$
    $(key_1, key_2, pub) \leftarrow \mathsf{IDPF.Gen}(\alpha, \vec{\beta}^*)$
    for $\ell \in [\eta]$:
      $pfx \leftarrow \alpha[1:\ell]$
      $(b_1, data_1, R_1) \leftarrow \mathsf{IDPF.Eval}(1, key_1, pub, pfx)$
      $(b_2, data_2, R_2) \leftarrow \mathsf{IDPF.Eval}(2, key_2, pub, pfx)$
      $(b_{\hat{j}}, data_{\hat{j}}, R_{\hat{j}}) \leftarrow \mathsf{IDPF.Eval}(\hat{j}, key_{\hat{j}}, pub, pfx)$
      $(b_{3-\hat{j}}, data_{3-\hat{j}}, R_{3-\hat{j}})$
            $\leftarrow (1 \oplus b_{\hat{j}}, \vec{\beta}[\ell] - data_{\hat{j}}, \vec{R}[\ell] - R_{\hat{j}})$
      $\vec{C}[\ell] \leftarrow \mathsf{RG}(pfx \,\|\, -data_1 \,\|\, R_1)$
            $\oplus \mathsf{RG}(pfx \,\|\, data_2 \,\|\, R_2)$
    $pub^* \leftarrow (pub, \vec{C})$
  $b^* \leftarrow \mathscr{A}^{\mathsf{Sketch}}(state_{\mathscr{A}}, key_{\hat{j}}, pub^*)$
  ret $b = b^*$

$\underline{\mathsf{Sketch}(\vec{x})}$:
  // $h \leftarrow \mathsf{Sim}_2(\hat{j}, key_{\hat{j}}, pub^*, \vec{x})$
  $(h, \_) \leftarrow \mathsf{VIDPF.VEval}(\hat{j}, key_{\hat{j}}, pub^*, \vec{x})$
  ret $h$
```

Figure 5.11. Simulator and hybrids used in the proof of privacy for the VIDPF construction.

probability of the bad event. ∎

5.8 Instantiating Delayed-Input FLP

Our main result is to construct a delayed-2-input FLP for use in Doplar.

Lemma 8. *The construction in Figure 5.12 (when suitably instantiated) is a delayed-2-input FLP with perfect completeness, soundness* $4(n+2)/(|\mathbb{F}|-n-2)$*, and privacy* $1/|\mathbb{F}|$*, for* $\mathbb{F}$*-arithmetic circuits with n multiplication gates.*

The main idea of the construction is simple. The prover wishes to generate a proof that will work with either of two instances x_1 and x_2. She simply generates a separate FLP proof for both instances x_1 and x_2, and randomly permutes the two proofs. To verify the combined proof against some x, the verifier accepts iff either of the component proofs verifies against that x.

```
$\mathsf{DFLP}^*.\mathsf{Prove}(\{\vec{x}_1,\vec{x}_2\},\Delta,jr)$:
  $(jr_1,jr_2) \leftarrow jr$
  $\vec{e}_1 \leftarrow \mathsf{FLP}.\mathtt{Encode}(\Delta,\vec{x}_1)$
  $\vec{e}_2 \leftarrow \mathsf{FLP}.\mathtt{Encode}(\Delta,\vec{x}_2)$
  $b \leftarrow\!\!\$ \{1,2\}$
  $\pi_b \leftarrow\!\!\$ \mathsf{FLP}.\mathsf{Prove}(\vec{e}_1,\Delta,jr_b)$
  $\pi_{3-b} \leftarrow\!\!\$ \mathsf{FLP}.\mathsf{Prove}(\vec{e}_2,\Delta,jr_{3-b})$
  ret $(\pi_1,\pi_2)$

$\mathsf{DFLP}^*.\mathsf{Query}(\vec{e},\Delta,(\pi_1,\pi_2),jr;qr)$:
  $(jr_1,jr_2) \leftarrow jr$; $(qr_1,qr_2) \leftarrow qr$
  $\sigma_1 \leftarrow\!\!\$ \mathsf{FLP}.\mathsf{Query}(\vec{e},\Delta,\pi_1,jr_1;qr_1)$
  $\sigma_2 \leftarrow\!\!\$ \mathsf{FLP}.\mathsf{Query}(\vec{e},\Delta,\pi_2,jr_2;qr_2)$
  ret $(\sigma_1,\sigma_2)$

$\mathsf{DFLP}^*.\mathsf{Decide}(\sigma)$:
  $(\sigma_1,\sigma_2) \leftarrow \sigma$
  ret $\mathsf{FLP}.\mathsf{Decide}(\sigma_1)$
      $\vee\ \mathsf{FLP}.\mathsf{Decide}(\sigma_2)$

$\mathsf{DFLP}^*.\mathtt{Encode}(\Delta \in \mathbb{F},\vec{x} \in \mathbb{F}^n)$:
  for $i \in [n]$:
    $\vec{e}[i] \leftarrow \vec{x}[i]$
    $\vec{e}[i+n] \leftarrow \Delta \cdot \vec{x}[i]$
  ret $\vec{e}$

$\mathsf{DFLP}^*.\mathsf{Decode}(\vec{e} \in \mathbb{F}^{2n})$:
  ret $\vec{e}[1:n]$
```

Figure 5.12. Delayed-2-input FLP construction $\mathsf{DFLP}^*[\mathsf{FLP}]$. The construction should be instantiated where FLP is the FLP for arithmetic circuits from [58].

Completeness and soundness of this construction are relatively clear. However, the construction is not necessarily zero-knowledge. While verifying the combined proof, we expect to verify a component proof against a *proof that was generated for some other instance* — e.g., verify a proof generated for x_1 against x_2. The standard zero-knowledge property of the underlying FLP does not apply to this situation. Indeed, since the Query function is linear, the result of querying a “mismatched” instance-proof pair will reveal “how far away” the instance is from the correct one.

We show that, when the underlying FLP is that of Boneh et al. [58], and extra randomness is introduced into the statement by means of the $\mathtt{Encode}(\Delta,\cdot)$ function, even the queries to the “mismatched” instance+proof can be simulated. Intuitively, the extra uncertainty of Δ blinds the results of the problematic queries.

Proof of Proof of Lemma 8:

Figure 5.12 describes a delayed-input FLP that uses a basic FLP as a building block. Our claims in this proof rely on that FLP being instantiated using the construction of [58], also used in the VDAF draft specification (see [25, Section 7.3]). We recall the relevant aspects of that construction below, as needed.

In Doplar, we will use our DFLP construction for the language $\mathscr{L} = \{0,1\}$ — i.e., we use it to prove that a value is zero or one. In this case, we instantiate the underlying FLP with the circuit:

$$C((s,t,\mathit{\Delta}),r) = \big(r \cdot s(s-1) + r^2 \cdot (s \cdot \mathit{\Delta} - t)\big)^2, \tag{5.1}$$

where r denotes the joint randomness. This circuit recognizes the set of inputs $(s,t,\mathit{\Delta}) \in \{(0,0,\mathit{\Delta}),(1,\mathit{\Delta},\mathit{\Delta})\}$. Note that FLP has input length $n = 3$ and joint-randomness length $jl = 1$; its circuit has 3 multiplication gates ($s(s-1)$, $s\mathit{\Delta}$, and the outer square). In the more general case, FLP will be instantiated for the language $\{(s,t,\mathit{\Delta}) \mid s \in \mathscr{L} \wedge s\mathit{\Delta} = t\}$. If the circuit for membership in $\mathscr{L}$ has n multiplication gates, then FLP will be instantiated with a circuit with $n+2$ multiplication gates.

Completeness follows immediately from the perfect completeness of the underlying FLP. The FLP of [58] has soundness $2n'/(|\mathbb{F}| - n')$ when its circuit has n' multiplication gates. We instantiate that FLP with $n' = n+2$, and we also incur a factor 2 loss in soundness since our construction verifies two proofs in the underlying FLP. Hence, we obtain the soundness bound stated in the lemma.

The zero-knowledge simulator for our construction is given as Sim in Figure 5.13. To demonstrate privacy, we first consider the hybrid on the left of Figure 5.13. With the gray box included and white box excluded, the hybrid generates exactly the honest verifier's view. In this game, both proofs are queried on x_c, the adversary's choice. Note that proof $\pi_{b \oplus c}$ was generated with input x_c in mind, while $\pi_{b \oplus c \oplus 1}$ was not. Let $u = b \oplus c \oplus 1$, the index of the "mismatched" proof (i.e., π_u was generated with $x_{c \oplus 1}$ in mind, not x_c). By applying the linearity of $\mathtt{Encode}(\mathit{\Delta},\cdot)$ and $\mathsf{Query}(\cdot,\cdot,\cdot)$, we can write:

$$\begin{aligned}
&\mathsf{Query}(\mathtt{Encode}(\mathit{\Delta},x_c),\mathit{\Delta},\pi_u,jr_u;qr_u) = \\
&\qquad \mathsf{Query}(\mathtt{Encode}(\mathit{\Delta},x_{c\oplus 1}),\mathit{\Delta},\pi_u,jr_u;qr_u) + \mathsf{Query}(\mathtt{Encode}(\mathit{\Delta},x_c - x_{c\oplus 1}),0,\vec{0},jr_u;qr_u)
\end{aligned}$$

Making this change of notation in the game yields hybrid G1 (in Figure 5.13, gray boxes excluded and outlined box included). G1 is distributed identically to the original privacy game.

In each call to Query in G1 that involves a value π_i, we use the same input that was used to generate π_i. Hence, we can apply the zero-knowledge property of the underlying FLP to each such expression. In doing so, we obtain the hybrid G2 on the right of Figure 5.13. The underlying FLP of [58] has perfect zero-knowledge, so G2 is distributed identically to the original game.

To complete the proof, it suffices to show that σ_u is distributed pseudorandomly in $\mathbb{F}^3 \times \{0\}$, since the simulator samples σ_u uniformly from that set. In particular, when $\widetilde{\sigma}$ is distributed as in a simulated proof, Δ is random, and $d \neq 0$, what is the distribution on $\widetilde{\sigma} + \mathsf{Query}(\mathtt{Encode}(\Delta, d), \vec{0}, \vec{0}, \cdots)$?

To answer this question, we must use specific properties of the FLP from [58]. We first briefly review the main idea behind their proof. The prover defines two polynomials L and R such that, for each multiplication gate i in the verification circuit, the value on its left wire is $L(i)$ and its right wire $R(i)$. Additionally, $L(0)$ and $R(0)$ are chosen uniformly. Define the "gadget" polynomial $G = L \times R$ — then $G(i)$ is the value of the output wire of the ith gate.

The proof vector π then consists of $L(0)$, $R(0)$, and the coefficients of the G polynomial. With that in mind, the Query algorithm makes 4 linear queries to the input + proof vector:

1. Obtain evaluations of the polynomial L as follows:

    - $L(0)$ is part of the proof vector.
    - For $i > 0$, if the left input to gate i is an input to the circuit, then $L(i)$ is given as part of the proof input/instance, to which Query has access.
    - Otherwise, the left input to gate i is the output of some other multiplication gate j. This value can be obtained as $G(j)$, since the coefficients of G are included in the proof vector.

    Reconstruct L as the result of Lagrange interpolation over the points $\{(i, L(i))\}$. Evaluate this polynomial L at point qr (the query randomness).

2. Similarly, reconstruct R and evaluate it at point qr.

3. Evaluate the polynomial G at point qr.

4. Evaluate G at point m, where the output of verification circuit is the output wire of the m'th multiplication gate.

Suppose the results of these queries are (r,s,t,u); the Decide algorithm checks that $t = rs$ and $u = 0$. The zero-knowledge property is that the result of the queries is distributed as $(r,s,rs,0)$ for uniform $r,s \leftarrow_{\$} \mathbb{F}$.

With Query as above, we now consider the distribution of

$$(r,s,rs,0) + \mathsf{Query}(\mathtt{Encode}(\Delta,d),\vec{0},\vec{0},\cdots),$$

where Δ, r, s are uniform in $\mathbb{F}$.

- The first component of this expression is uniform due to r.

- With overwhelming probability $1 - 1/|\mathbb{F}|$ we have $r \neq 0$. Conditioned on $r \neq 0$, the third component of the expression is uniform, since it is masked with rs, and s is uniform (even conditioned on the first component).

- Let q_4 be the 4th component of Query's output in the above expression. By definition of Query, q_4 is the result of evaluating G at point qr. But in this expression, the "proof vector" argument to Query is all zeroes, hence Query evaluates the all-zeroes polynomial and outputs $q_4 = 0$. Hence the 4th component of the overall expression is zero.

- Let q_2 be the second output of Query in the above expression. We see that q_2 is the result of evaluating polynomial R at point qr, after reconstructing R as described above. Fix a position i in which $\vec{d}[i] \neq 0$. Then the $(n+i)$th position of $\vec{e} = \mathtt{Encode}(\Delta,\vec{d})$ is $\vec{e}[i] = \vec{d}[i]\Delta$, and therefore is uniformly distributed when Δ is uniformly distributed.

  The final multiplication gate in the verification circuit is the outermost square in (5.1). The input to this squaring operation is a linear combination that includes $\vec{e}[i]$. So as $\vec{e}[i]$ is uniformly distributed, the input to this multiplication gate is also uniformly distributed. Then the result of interpolating polynomial R (based on $\vec{e}[i]$ among other values) and evaluating R at qr is also uniformly distributed. In other words, q_2 is uniformly distributed over uniform choice of Δ, so the second component of the above expression is uniform.

Overall, we have shown that the distribution of σ_u in G2 of Figure 5.13 is statistical distance

```
$\mathsf{Exp}^{\mathrm{PRIV}}_{\mathsf{DFLP},\mathsf{Sim}}(A)$:  $\boxed{\mathrm{G1}(A)\text{:}}$
$b \leftarrow\!\!\$\ \{0,1\}$
$(\{x_0, x_1\}, st_A) \leftarrow A()$
if $b = 0$:
  $(st_{\mathsf{Sim}}, (jr_0, jr_1), (qr_0, qr_1)) \leftarrow \mathsf{Sim}_1()$
else:
  $jr_0, jr_1 \leftarrow\!\!\$\ \mathbb{F}^{jl};\ qr_0, qr_1 \leftarrow\!\!\$\ \mathbb{F}^{ql}$
  $\Delta \leftarrow\!\!\$\ \mathbb{F}$
  // Prove$(\{x_0, x_1\}, \Delta, jr)$
  $b \leftarrow\!\!\$\ \{0,1\}$
  $\pi_b \leftarrow \mathsf{Prove}(\mathsf{Encode}(\Delta, \vec{x}_0), \Delta, jr_b)$
  $\pi_{1\oplus b} \leftarrow \mathsf{Prove}(\mathsf{Encode}(\Delta, \vec{x}_1), \Delta, jr_{1\oplus b})$
$(c, st_A) \leftarrow A(st_A, (jr_0, jr_1), (qr_0, qr_1))$
if $b = 0$:  $(\sigma_0, \sigma_1) \leftarrow \mathsf{Sim}^*(st_{\mathsf{Sim}})$
else:  // Query(Encode$(\Delta, x_c), \Delta, \pi, jr;\ qr$)
  $\sigma_0 \leftarrow \mathsf{Query}(\mathsf{Encode}(\Delta, x_c), \Delta, \pi_0, jr_0;\ qr_0)$
  $\sigma_1 \leftarrow \mathsf{Query}(\mathsf{Encode}(\Delta, x_c), \Delta, \pi_1, jr_1;\ qr_1)$
  $\sigma_b \leftarrow \mathsf{Query}(\mathsf{Encode}(\Delta, x_0), \Delta, \pi_b, jr_b;\ qr_b)$
  $\sigma_{1\oplus b} \leftarrow \mathsf{Query}(\mathsf{Encode}(\Delta, x_1), \Delta, \pi_{1\oplus b}, jr_{1\oplus b};\ qr_{1\oplus}$ [redacted] )
  $u \leftarrow b \oplus c \oplus 1$  // index of "mismatched" proof
  $\sigma_u \leftarrow \sigma_u$
      $+\,\mathsf{Query}(\mathsf{Encode}(\Delta, x_c - x_{1\oplus c}), 0, \vec{0}, jr_u;\ qr_u)$
$b' \leftarrow A(st_A, (\sigma_0, \sigma_1))$
ret $b == b'$
```

```
G2$(A)$:
$b \leftarrow\!\!\$\ \{0,1\}$
$(\{x_0, x_1\}, st_A) \leftarrow A()$
if $b = 0$:
  $(st_{\mathsf{Sim}}, (jr_0, jr_1), (qr_0, qr_1)) \leftarrow \mathsf{Sim}_1()$
else:
  $\Delta \leftarrow\!\!\$\ \mathbb{F}$
  $b \leftarrow\!\!\$\ \{0,1\}$
  $(jr_0, qr_0, \sigma_0) \leftarrow \mathsf{Sim}_{\mathsf{FLP}}()$
  $(jr_1, qr_1, \sigma_1) \leftarrow \mathsf{Sim}_{\mathsf{FLP}}()$
$(c, st) \leftarrow A(st_A, (jr_0, jr_1), (qr_0, qr_1))$
if $b = 0$:  $(\sigma_0, \sigma_1) \leftarrow \mathsf{Sim}^*(st_{\mathsf{Sim}})$
else:
  $u \leftarrow b \oplus c \oplus 1$  // index of "mismatched" proof
  $\sigma_u \leftarrow \sigma_u$
      $+\,\mathsf{Query}(\mathsf{Encode}(\Delta, x_c - x_{1\oplus c}), 0, \vec{0}, jr_u;\ qr_u)$
$b' \leftarrow A(st_A, (\sigma_0, \sigma_1))$
ret $b == b'$
[redacted]

S[redacted]m$_1()$:
$(jr_0, qr_0, \sigma_0) \leftarrow \mathsf{Sim}()$
$(jr_1, qr_1, \sigma_1) \leftarrow \mathsf{Sim}()$
ret $\big(st_{\mathsf{Sim}} = (\sigma_0, \sigma_1), (jr_0, jr_1), (qr_0, qr_1)\big)$
$\mathsf{Sim}_2(\sigma_0, \sigma_1)$:
$b \leftarrow\!\!\$\ \{0,1\}$
$\sigma_b \leftarrow\!\!\$\ \mathbb{F}^3 \times \{0\}$  // overwrite $\sigma_b$
ret $(\sigma_0, \sigma_1)$
```

Figure 5.13. Hybrids for zero-knowledge property of the delayed-2-input FLP construction.

$1/|\mathbb{F}|$ from the simulator's distribution: uniform over $\mathbb{F}^3 \times \{0\}$. Hence in G2 the adversary has advantage bounded by $1/|\mathbb{F}|$ in G2. ▮

5.9 Proofs of Theorems

5.9.1 Prio3 Robustness (Theorem 18)

We begin by instantiating the robustness game for Π in Figure 5.14. We expand the Prep algorithm and make a few simplifications to the game's internal notation and bookkeeping. First, the game $\mathsf{Exp}^{\mathrm{robust}}$ calls for a VDAF with an arbitrary number of rounds, but Prio3 constructions has just one round. Second, we know that the Prep algorithm will be called exactly twice for each aggregator, and that the initial broadcast message and state are empty. We Therefore unroll the loop of lines 5–14 of Figure 5.3 and evaluate those if-statements whose

```
Game $\mathsf{Exp}^{\mathrm{robust}}_{\Pi}(A)$ $\boxed{\mathrm{G1}(A)}$ :
1 $\mathsf{win} \leftarrow \mathsf{false}$; $sk \leftarrow\!\!\$\, \{0,1\}^{\kappa}$
2 $A^{\mathrm{RO},\underline{\mathsf{Prep}}}()$; ret $w$

$\underline{\mathsf{Prep}}(n, \vec{x}, msg_{\mathrm{Init}}, st_{\mathrm{Init}})$:
3 if $\mathrm{Used}[n] \neq \bot$: ret $\bot$
4 $\mathrm{Used}[n] \leftarrow \top$
5 for $\hat{j} \in [s]$:
6     $(\vec{inp}[\hat{j}], \vec{\pi}[\hat{j}], blind) \leftarrow \mathsf{Unpack}(\hat{j}, \vec{x}[\hat{j}])$
7     $(\vec{\rho},) \leftarrow \vec{M}$; $\vec{\rho}[\hat{j}] \leftarrow \mathrm{RO}_7(blind, \hat{j} \| n \| \vec{inp}[\hat{j}])$
8     $\vec{rseed}[\hat{j}] \leftarrow \vec{\rho}[\hat{j}]$
9     $\vec{state}[\hat{j}] \leftarrow \mathrm{RO}_6(0^{\kappa}, \vec{\rho})$ // joint rand seed
10    $jr \leftarrow \mathrm{RO}_1(\vec{state}[\hat{j}], \varepsilon)$ $\boxed{jr \leftarrow \mathrm{RO}_1(\vec{state}[1], \varepsilon)}$
11    $qr \leftarrow \mathrm{RO}_5(sk, n)$
12    $\vec{vfs}[\hat{j}] \leftarrow \mathsf{Query}(\vec{inp}[\hat{j}], \vec{\pi}[\hat{j}], jr; qr)$
13 $vf \leftarrow \sum_{\hat{j}=1}^{s} \vec{vfs}[\hat{j}]$
14 $d \leftarrow \mathsf{FLP.Decide}(vf)$
15 for $\hat{j} \in [s]$:
16    $jseed_{\hat{j}} \leftarrow \vec{state}[\hat{j}]$; $jseed'_{\hat{j}} \leftarrow \mathrm{RO}_6(0^{\kappa}, \vec{rseed})$
17    $acc_{\hat{j}} \leftarrow d \wedge [[jseed_{\hat{j}} = jseed'_{\hat{j}}]]$
18    $\mathsf{win} \leftarrow (\mathsf{win} \bigvee [acc_{\hat{j}} \wedge \mathsf{refineFromShares}(\varepsilon, \vec{x}) \notin \mathscr{L}])$
19 ret $(\mathsf{win}, (msg_{\mathrm{Init}}, (\vec{vfs}[\hat{j}], \vec{rseed}[\hat{j}]))_{\hat{j} \in [s]})$
```

```
Adversary $\mathscr{B}^{\mathrm{RO},\underline{\mathsf{Prep}}}()$:
1 $sk' \leftarrow\!\!\$\, \{0,1\}^{\kappa}$
2 $A^{\mathrm{RO},\mathsf{PrepSim}}()$

$\mathsf{PrepSim}(n, \vec{x}, msg_{\mathrm{Init}}, st_{\mathrm{Init}})$:
3 if $\mathrm{Used}[n] \neq \bot$: ret $\bot$
4 $\mathrm{Used}[n] \leftarrow \top$; $fwd \leftarrow \mathsf{true}$
5 for $\hat{j} \in [s]$:
6     $(\vec{inp}[\hat{j}], \vec{\pi}[\hat{j}], blind) \leftarrow \mathsf{Unpack}(\hat{j}, \vec{x}[\hat{j}])$
7     $(\vec{\rho},) \leftarrow \vec{M}$; $\vec{\rho}[\hat{j}] \leftarrow \mathrm{RO}_7(blind, \hat{j} \| n \| \vec{inp}[\hat{j}])$
8     $\vec{rseed}[\hat{j}] \leftarrow \vec{\rho}[\hat{j}]$
9     $\vec{state}[\hat{j}] \leftarrow \mathrm{RO}_6(0^{\kappa}, \vec{\rho})$ // Joint rand seed
10    if $\vec{state}[\hat{j}] \neq \vec{state}[1]$: $fwd \leftarrow \mathsf{false}$
11    $jr \leftarrow \mathrm{RO}_1(\vec{state}[\hat{j}], \varepsilon)$
12    $qr \leftarrow \mathrm{RO}_5(sk', n)$
13    $\vec{vfs}[\hat{j}] \leftarrow \mathsf{Query}(\vec{inp}[\hat{j}], \vec{\pi}[\hat{j}], jr; qr)$
14 if $fwd$: return $\underline{\mathsf{Prep}}(n, \vec{x}, msg_{\mathrm{Init}}, st_{\mathrm{Init}})$
15 ret $(\mathsf{false}, (msg_{\mathrm{Init}}, (\vec{vfs}[\hat{j}], \vec{rseed}[\hat{j}]))_{\hat{j} \in [s]})$
```

Figure 5.14. Left: Definition of game G1 for the proof of Theorem 18. Also shown is the robustness game for Π and adversary A with some simplifications applied. Right: Adversary $\mathscr{B}$.

values are pre-determined. Third, we replace table St with a vector $\vec{state}$ and remove table Msg altogether. (The transcript output by the oracle is now constructed at the end on line 21 on the left-hand panel of Figure 5.14.) Fourth, we evaluate Prep in parallel for all aggregators instead of in sequence; the order of these operations does not affect their results because aggregators do not share state. Fifth, we perform the deterministic Decide operation only once since its result is the same for all aggregators. Finally, we replace each call to RG_i with a call to the corresponding random oracle RO_i. Let q_i denote the number of queries $\mathscr{A}$ makes to RO_i; note that $q_{\mathsf{RG}} = q_1 + \cdots + q_7$.

We have also dropped winning condition on line 16 of Figure 5.3. By definition, $\Pi.\mathsf{refineFromShares}(\varepsilon, \vec{x}) = \Pi.\mathsf{Unshard}(1,\ (\Pi.\mathsf{Agg}(\vec{inp}[1]), \ldots, \Pi.\mathsf{Agg}(\vec{inp}[s])))$, where $inp[\hat{j}]$ is the unpacked inner measurement share of input share $\vec{x}[\hat{j}]$ for each $\hat{j}$. Thus win can never be set by forcing the refined shares to mismatch the expected refined measurement.

Now we express the proof with a series of incrementally changed games, beginning with

G1 (c.f. Figure 5.14). The joint randomness for each aggregator $\hat{j}$ is derived in $\mathsf{Exp}_{\Pi}^{\text{robust}}$ from the seed $jseed_{\hat{j}}$ of that aggregator, which is also the state $\vec{state}[\hat{j}]$. In G1, we instead derive joint randomness from $\vec{state}[1]$ for all aggregators, thus ensuring that the joint randomness is the same for everyone.

We build a wrapper adversary $\mathscr{B}$ for which

$$\mathbf{Adv}\text{robust}_{\Pi}(\mathscr{A}) \leq \Pr[\text{G1}(\mathscr{B})] + \frac{q_5}{2^{\kappa}}\,. \tag{5.2}$$

Adversary $\mathscr{B}$ only makes queries to <u>Prep</u> that set $\vec{state}[\hat{j}] = \vec{state}[1]$ for all $\hat{j}$. It accomplishes this by calculating $\vec{state}[\hat{j}]$ for every aggregator and <u>Prep</u> query made by $\mathscr{A}$. If it finds that $\vec{state}[\hat{j}] = \vec{state}[1]$ for all aggregators, it forwards the query to its own <u>Prep</u> oracle. Otherwise, it runs <u>Prep</u> itself. $\mathscr{B}$ can perfectly simulate <u>Prep</u> except for line 9, because it does not know sk. Instead, $\mathscr{B}$ picks its own verification key sk' and uses sk' in line 9 where <u>Prep</u> would use sk. Adversary $\mathscr{A}$ can detect the substitution of sk' for sk in two cases: If $\mathscr{A}$ queries RO_5 on seed sk'; or if the <u>Prep</u> oracle and $\mathscr{B}$ query RO_5 on the same context string. The latter event does not occur because each query to RO_5 contains a unique nonce. The former occurs with probability at most $\frac{q_5}{2^{\kappa}}$, because sk' is a uniformly random κ-bit string. The queries that $\mathscr{B}$ simulates would always set $acc_{\hat{j}} \leftarrow 0$ in line 17. Thus any query that would set $\mathsf{win} \leftarrow \mathsf{true}$ is forwarded to the <u>Prep</u> oracle by $\mathscr{B}$, and $\mathscr{B}$ wins whenever $\mathscr{A}$ does. The claim follows.

Next, we use the full linearity of FLP to decompose $\mathsf{FLP.Query}$ into algorithm Q and a matrix multiplication operation, as shown in the left-hand panel of Figure 5.15. Q is a randomized algorithm, but it is executed deterministically with fixed input jr and coins qr. We may therefore call Q only once to eliminate redundancy. Finally, we sum the vectors $\vec{x}_{\hat{j}} \,\|\, \pi_{\hat{j}}$ before the multiplication instead of multiplying then summing the products. This preserves the output thanks to the associativity of matrix multiplication.

Full linearity is an information theoretic property that holds unconditionally for all inputs, proofs, and coins, so the new game computes the same verifier string vf. Thus

$$\Pr[\text{G1}(\mathscr{B})] = \Pr[\text{G2}(\mathscr{B})]\,. \tag{5.3}$$

```
Game $G1(\mathscr{B})$ $\boxed{G2(\mathscr{B})}$ :
1 $\mathsf{win} \leftarrow \mathsf{false}$; $sk \leftarrow\!\!\$\ \{0,1\}^\kappa$
2 $\mathscr{B}^{\mathrm{RO},\underline{\mathsf{Prep}}}()$; ret $w$

$\underline{\mathsf{Prep}}(n, \vec{x}, msg_{\mathrm{Init}}, st_{\mathrm{Init}})$:
3 if $\mathrm{Used}[n] \neq \bot$: ret $\bot$
4 $\mathrm{Used}[n] \leftarrow \top$
5 for $\hat{j} \in [s]$:
6     $(\vec{inp}[\hat{j}], \vec{\pi}[\hat{j}], blind) \leftarrow \mathsf{Unpack}(\hat{j}, \vec{x}[\hat{j}])$
7     $(\vec{\rho},) \leftarrow \vec{M}$; $\vec{\rho}[\hat{j}] \leftarrow \mathrm{RO}_7(blind, \hat{j} \| n \| \vec{inp}[\hat{j}])$
8     $\vec{rseed}[\hat{j}] \leftarrow \vec{\rho}[\hat{j}]$
9     $\vec{state}[\hat{j}] \leftarrow \mathrm{RO}_6(0^\kappa, \vec{\rho})$ // joint rand seed
10    $jr \leftarrow \mathrm{RO}_1(\vec{state}[1], \varepsilon)$; $qr \leftarrow \mathrm{RO}_5(sk, n)$
11    $\vec{vfs}[\hat{j}] \leftarrow \mathsf{Query}(\vec{inp}[\hat{j}], \vec{\pi}[\hat{j}], jr; qr)$
12 $vf \leftarrow \sum_{\hat{j}=1}^{s} \vec{vfs}[\hat{j}]$
13 $jr \leftarrow \mathrm{RO}_1(\vec{state}[1], \varepsilon)$; $qr \leftarrow \mathrm{RO}_5(sk, n)$
14 $Z \leftarrow Q(jr; qr)$
15 $vf \leftarrow Z \cdot \sum_{\hat{j}=1}^{s} \vec{inp}[\hat{j}] \| \vec{\pi}[\hat{j}]$
16 $d \leftarrow \mathsf{FLP.Decide}(vf)$
17 for $\hat{j} \in [s]$:
18    $jseed_{\hat{j}} \leftarrow \vec{state}[\hat{j}]$; $jseed'_{\hat{j}} \leftarrow \mathrm{RO}_6(0^\kappa, \vec{rseed})$
19    $acc_{\hat{j}} \leftarrow d \wedge [[jseed_{\hat{j}} = jseed'_{\hat{j}}]]$
20    $\mathsf{win} \leftarrow (\mathsf{win} \vee [acc_{\hat{j}} \wedge \mathsf{refineFromShares}(\varepsilon, \vec{x}) \notin \mathscr{L}])$
21 ret $(\mathsf{win}, (msg_{\mathrm{Init}}, (\vec{vfs}[\hat{j}], \vec{rseed}[\hat{j}]))_{\hat{j} \in [s]})$
```

```
Game $G2(\mathscr{B})$ $\boxed{G3(\mathscr{B})}$ :
1 $\mathsf{win} \leftarrow \mathsf{false}$; $sk \leftarrow\!\!\$\ \{0,1\}^\kappa$
2 $\mathscr{B}^{\mathrm{RO},\underline{\mathsf{Prep}}}()$; ret $w$

$\underline{\mathsf{Prep}}(n, \vec{x}, msg_{\mathrm{Init}}, st_{\mathrm{Init}})$:
3 if $\mathrm{Used}[n] \neq \bot$: ret $\bot$
4 $\mathrm{Used}[n] \leftarrow \top$
5 for $\hat{j} \in [s]$:
6     $(\vec{inp}[\hat{j}], \vec{\pi}[\hat{j}], blind) \leftarrow \mathsf{Unpack}(\hat{j}, \vec{x}[\hat{j}])$
7     $(\vec{\rho},) \leftarrow \vec{M}$; $\vec{\rho}[\hat{j}] \leftarrow \mathrm{RO}_7(blind, \hat{j} \| n \| \vec{inp}[\hat{j}])$
8     $\vec{rseed}[\hat{j}] \leftarrow \vec{\rho}[\hat{j}]$
9     $\vec{state}[\hat{j}] \leftarrow \mathrm{RO}_6(0^\kappa, \vec{\rho})$ // joint rand seed
10 $jr \leftarrow \mathrm{RO}_1(\vec{state}[1], \varepsilon)$
11 $qr \leftarrow \mathrm{RO}_5(sk, n)$
12 $Z \leftarrow Q(jr; qr)$
13 $vf \leftarrow Z \cdot \sum_{\hat{j}=1}^{s} \vec{inp}[\hat{j}] \| \vec{\pi}[\hat{j}]$
14 $inp \leftarrow \sum_{\hat{j}=1}^{s} \vec{inp}[\hat{j}]$; $\pi \leftarrow \sum_{\hat{j}=1}^{s} \vec{\pi}[\hat{j}]$
15 $vf \leftarrow \mathsf{FLP.Query}(inp, \pi, jr; qr)$
16 $d \leftarrow \mathsf{FLP.Decide}(vf)$
17 for $\hat{j} \in [s]$:
18    $jseed_{\hat{j}} \leftarrow \vec{state}[\hat{j}]$; $jseed'_{\hat{j}} \leftarrow \mathrm{RO}_6(0^\kappa, \vec{rseed})$
19    $acc_{\hat{j}} \leftarrow d \wedge [[jseed_{\hat{j}} = jseed'_{\hat{j}}]]$
20    $\mathsf{win} \leftarrow (\mathsf{win} \vee [acc_{\hat{j}} \wedge \mathsf{refineFromShares}(\varepsilon, \vec{x}) \notin \mathscr{L}])$
21    $\mathsf{win} \leftarrow (\mathsf{win} \vee [acc_{\hat{j}} \wedge inp \notin \mathscr{L}])$
22 ret $(\mathsf{win}, (msg_{\mathrm{Init}}, (\vec{vfs}[\hat{j}], \vec{rseed}[\hat{j}]))_{\hat{j} \in [s]})$
```

Figure 5.15. Game G2 (left) and game G3 (right) for the proof of Theorem 18.

We produce the next modified game, in the right-hand panel of Figure 5.15, to define variables $inp = \sum_{\hat{j}=1}^{s} \vec{inp}[\hat{j}]$ and $\pi = \sum_{\hat{j}=1}^{s} \vec{\pi}[\hat{j}]$ and invoke PRG.Query on inp, π directly. In addition, from Figure 5.5, we can see that $inp = \Pi.\mathsf{refineFromShares}(\varepsilon, \vec{x})$, so we substitute inp into line 21. By the full linearity of FLP, we have that $Q(jr; qr) \cdot (inp \| \pi) = \mathsf{FLP.Query}(inp, \pi, jr; qr)$. Again, these operations do not affect the adversary's view of $\underline{\mathsf{Prep}}$, and

$$\Pr[G2(\mathscr{B})] = \Pr[G3(\mathscr{B})]. \qquad (5.4)$$

In the next game, we replace the pseudorandom query randomness qr with a fresh random string that is implicitly sampled by Query. We bound the difference in advantage between games G3 and G4 via a reduction $\mathscr{B}'$ to the pseudorandomness of RG_5. The reduction honestly simulates G3 except in line 11, where it queries its challenge oracle on n and sets qr to the response.

```
Game $G3(\mathscr{B})$ $\boxed{G4(\mathscr{B})}$:
win $\leftarrow$ false; $sk \leftarrow\!\!\$ \{0,1\}^\kappa$
$\mathscr{B}^{\mathrm{RO},\underline{\mathsf{Prep}}}()$; ret $w$

$\underline{\mathsf{Prep}}(n, \vec{x}, msg_{\mathrm{Init}}, st_{\mathrm{Init}})$:
  if $\mathrm{Used}[n] \neq \bot$: ret $\bot$
  $\mathrm{Used}[n] \leftarrow \top$
  for $\hat{j} \in [s]$:
    $(\vec{inp}[\hat{j}], \vec{\pi}[\hat{j}], blind) \leftarrow \mathsf{Unpack}(\hat{j}, \vec{x}[\hat{j}])$
    $(\vec{\rho},) \leftarrow \vec{M}$; $\vec{\rho}[\hat{j}] \leftarrow \mathrm{RO}_7(blind, \hat{j} \| n \| \vec{inp}[\hat{j}])$
    $\vec{rseed}[\hat{j}] \leftarrow \vec{\rho}[\hat{j}]$
    $\vec{state}[\hat{j}] \leftarrow \mathrm{RO}_6(0^\kappa, \vec{\rho})$ //joint rand seed
  $jr \leftarrow \mathrm{RO}_1(\vec{state}[1], \varepsilon)$
  $qr \leftarrow \mathrm{RO}_5(sk, n)$
  $inp \leftarrow \sum_{\hat{j}=1}^{s} \vec{inp}[\hat{j}]$; $\pi \leftarrow \sum_{\hat{j}=1}^{s} \vec{\pi}[\hat{j}]$
  $vf \leftarrow \mathsf{FLP.Query}(inp, \pi, jr; qr)$
  $\boxed{vf \leftarrow\!\!\$ \mathsf{FLP.Query}(inp, \pi, jr)}$
  $d \leftarrow \mathsf{FLP.Decide}(vf)$
  for $\hat{j} \in [s]$:
    $jseed_{\hat{j}} \leftarrow \vec{state}[\hat{j}]$; $jseed'_{\hat{j}} \leftarrow \mathrm{RO}_6(0^\kappa, \vec{rseed})$
    $acc_{\hat{j}} \leftarrow d \wedge [[jseed_{\hat{j}} = jseed'_{\hat{j}}]]$
    win $\leftarrow$ (win$\bigvee[acc_{\hat{j}} \wedge inp \notin \mathscr{L}]$)
  ret (win, $(msg_{\mathrm{Init}}, (\vec{vfs}[\hat{j}], \vec{rseed}[\hat{j}]))_{\hat{j}\in[s]}$)
```

```
Game $G4(\mathscr{B})$ $\boxed{G5(\mathscr{B})}$:
win $\leftarrow$ false; $sk \leftarrow\!\!\$ \{0,1\}^\kappa$; $\boxed{\mathscr{J} \leftarrow \emptyset}$
$\mathscr{B}^{\mathrm{RO},\underline{\mathsf{Prep}}}()$; ret $w$

$\underline{\mathsf{Prep}}(n, \vec{x}, msg_{\mathrm{Init}}, st_{\mathrm{Init}})$:
  if $\mathrm{Used}[n] \neq \bot$: ret $\bot$
  $\mathrm{Used}[n] \leftarrow \top$
  for $\hat{j} \in [s]$:
    $(\vec{inp}[\hat{j}], \vec{\pi}[\hat{j}], blind) \leftarrow \mathsf{Unpack}(\hat{j}, \vec{x}[\hat{j}])$
    $(\vec{\rho},) \leftarrow \vec{M}$; $\vec{\rho}[\hat{j}] \leftarrow \mathrm{RO}_7(blind, \hat{j} \| n \| \vec{inp}[\hat{j}])$
    $\vec{rseed}[\hat{j}] \leftarrow \vec{\rho}[\hat{j}]$
    $\vec{state}[\hat{j}] \leftarrow \mathrm{RO}_6(0^\kappa, \vec{\rho})$ //joint rand seed
  $\boxed{\text{if } \vec{state}[1] \in \mathscr{J}\text{: bad} \leftarrow \text{true}}$
  $\boxed{\mathscr{J} \leftarrow \mathscr{J} \cup \{\vec{state}[1]\}}$
  $jr \leftarrow \mathrm{RO}_1(\vec{state}[1], \varepsilon)$
  $inp \leftarrow \sum_{\hat{j}=1}^{s} \vec{inp}[\hat{j}]$; $\pi \leftarrow \sum_{\hat{j}=1}^{s} \vec{\pi}[\hat{j}]$
  $vf \leftarrow\!\!\$ \mathsf{FLP.Query}(inp, \pi, jr)$
  $d \leftarrow \mathsf{FLP.Decide}(vf)$
  for $\hat{j} \in [s]$:
    $jseed_{\hat{j}} \leftarrow \vec{state}[\hat{j}]$; $jseed'_{\hat{j}} \leftarrow \mathrm{RO}_6(0^\kappa, \vec{rseed})$
    $acc_{\hat{j}} \leftarrow d \wedge [[jseed_{\hat{j}} = jseed'_{\hat{j}}]]$
    win $\leftarrow$ (win$\bigvee[\boxed{\neg\text{bad} \wedge}\ acc_{\hat{j}} \wedge inp \notin \mathscr{L}]$)
  ret (win, $(msg_{\mathrm{Init}}, (\vec{vfs}[\hat{j}], \vec{rseed}[\hat{j}]))_{\hat{j}\in[s]}$)
```

Figure 5.16. Fourth and fifth intermediate games for the proof of Theorem 18.

Because every nonce is unique, these queries are all distinct. When the challenge oracle is a random function, this is a perfect simulation of G4; otherwise it is a perfect simulation of G3. Adversary $\mathscr{B}'$ makes q_{Prep} queries to its challenge oracle; when RG_5 is modeled as a random oracle, there is a maximum of q_5 random oracle queries.

The generic PRF advantage for a (q_5, q_{Prep})-query attacker against a random oracle with domain $\{0,1\}^\kappa$ is bounded by the probability $\frac{q_5}{2^\kappa}$ that the attacker makes a random oracle query containing sk. Thus

$$\Pr[\mathrm{G3}(\mathscr{B})] \leq \Pr[\mathrm{G4}(\mathscr{B})] + \frac{q_5}{2^\kappa}. \tag{5.5}$$

Our next game (G5 defined in the right-hand panel of Figure 5.16) differs from G4 as follows. We set a bad flag and force the adversary to lose if it makes two queries to Prep which derive their joint randomness from the same seed. Each query to Prep derives its joint randomness seed from a unique nonce, so duplicate seeds require a collision between two queries to RO_6 or between two vectors of hints. Both seeds and hints are randomly sampled by random oracles RO_6 and RO_7 respectively, so we limit the probability of both types of collision with a birthday bound over the q_{Prep} queries to $\underline{\mathsf{Prep}}$:

$$\frac{{q_{\mathsf{Prep}}}^2}{2^{\kappa+1}} + \frac{{q_{\mathsf{Prep}}}^2}{2^{\kappa \cdot s+1}} < \frac{{q_{\mathsf{Prep}}}^2}{2^{\kappa}} \,.$$

Since the games are identical until bad gets set, we have

$$\Pr[\mathrm{G4}(\mathscr{B})] \leq \Pr[\mathrm{G5}(\mathscr{B})] + \frac{{q_{\mathsf{Prep}}}^2}{2^{\kappa}} \,. \tag{5.6}$$

We are now ready to reduce to FLP soundness. To do so, we construct a malicious prover P^* in Figure 5.17 from $\mathscr{B}$ whose advantage in the FLP soundness experiment is related to $\mathscr{B}$'s advantage in winning game G5. Recall from Figure 5.2 that the prover is called twice, first to choose an input and a second time to generate a proof. The prover is given joint randomness jr in this second call, after committing to the input. Thus, in our reduction we must extract this input from $\mathscr{B}$ random oracle queries, then program the random oracle with jr before proceeding.

The malicious prover P^* runs $\mathscr{B}$ in a simulation of G5. Its random oracle queries are answered by lazy-evaluating a table Rand; all oracle queries are handled the same way except for a distinguished query, which will be programmed using the jr string generated as part of the malicious prover's experiment. At the start of the simulation, the prover P^* samples $i^* \leftarrow\!\!\$\, [q_1 + q_{\mathsf{Prep}}]$. On the i^* unique invocation of RO_1 (see $\mathrm{ROE{\scriptsize XT}}_1$ in Figure 5.17), the prover checks the table Rand for a nonce n and input shares $inp_1, \ldots, inp_s$ that give rise to the seed $jseed$ provided as input. If successful, the prover outputs $inp_1 + \cdots + inp_s$ as its challenge input, awaits the response jr, and sets $\mathrm{Rand}[1, jseed, \varepsilon] \leftarrow jr$ (line 11). It also records $n^* \leftarrow n$ for use later on.

The simulation of $\underline{\mathsf{Prep}}$ queries is identical except after two events. First, the prover P^* halts and concedes if two $\underline{\mathsf{Prep}}$ queries generate the same joint randomness seed $\vec{state}[1]$. (Adversary $\mathscr{B}$ loses in this case.) Second, if $n = n^*$, then P^* immediately halts and outputs the proof π computed on line 30. If the simulation has reached this point, then the probability that P^* wins its game is at least the probability that the game sets $w \leftarrow \mathsf{true}$ on line 36. Conditioning on the probability that P^* guesses the winning query to RO_1, we have that

```
Adversary $P^*[\mathscr{B}]()$:
1  $\mathsf{win} \leftarrow \mathsf{false}$; $sk \leftarrow\!\!\$\, \{0,1\}^\kappa$; $\mathsf{bad} \leftarrow \mathsf{false}$; $\mathscr{J} \leftarrow \emptyset$
2  $ctr \leftarrow 0$; $n^* \leftarrow \perp$; $i^* \leftarrow\!\!\$\, [q_1 + q_{\mathsf{Prep}}]$
3  $\mathscr{B}^{\mathrm{ROExt}_1, \mathrm{RO}_2, \ldots, \mathrm{RO}_7 \mathsf{PrepSim}}()$; ret $w$

$\mathrm{ROExt}_1(seed, cntxt)$:
4  if $\mathrm{Rand}[1, seed, cntxt] \neq \perp$: ret $\mathrm{RO}_1(seed, cntxt)$
5  $ctr \leftarrow ctr + 1$
6  if $ctr = i^* \wedge (\exists n, (blind_{\hat{j}}, inp_{\hat{j}}, \rho_{\hat{j}})_{\hat{j} \in [s]})$
7      $(\forall \hat{j})\ \mathrm{Rand}[7, blind_{\hat{j}}, \hat{j} \| n \| inp_{\hat{j}}] = \rho_{\hat{j}}$
8        $\wedge\, \mathrm{Rand}[6, 0^\kappa, (\rho_1, \ldots, \rho_s)] = seed$:
9    output $inp_1 + \cdots + inp_s$ and wait for $jr$.
10   $n^* \leftarrow n$; $\mathrm{Rand}[1, seed, cntxt] \leftarrow jr$
11 ret $\mathrm{RO}_1(seed, cntxt)$

$\mathrm{RO}_i(seed, cntxt)$:
12 $l \leftarrow (jl, n, m, pl, ql)$
13 if $\mathrm{Rand}[i, seed, cntxt] = \perp$:
14   if $i \leq 5$: $\mathrm{Rand}[i, seed, cntxt] \leftarrow\!\!\$\, \mathbb{F}^{l[i]}$
15   else: $\mathrm{Rand}[i, seed, cntxt] \leftarrow\!\!\$\, \{0,1\}^\kappa$
16 ret $\mathrm{Rand}[i, seed, cntxt]$

$\mathsf{PrepSim}(n, \vec{x}, msg_{\mathrm{Init}}, st_{\mathrm{Init}})$:
17 if $\mathrm{Used}[n] \neq \perp$: ret $\perp$
18 $\mathrm{Used}[n] \leftarrow \top$
19 for $\hat{j} \in [s]$:
20   $(\vec{inp}[\hat{j}], \vec{\pi}[\hat{j}], blind) \leftarrow \mathsf{Unpack}(\hat{j}, \vec{x}[\hat{j}])$
21   $(\vec{\rho},) \leftarrow \vec{M}$; $\vec{\rho}[\hat{j}] \leftarrow \mathrm{RO}_7(blind, \hat{j} \| n \| \vec{inp}[\hat{j}])$
22   $\vec{rseed}[\hat{j}] \leftarrow \vec{\rho}[\hat{j}]$
23   $\vec{state}[\hat{j}] \leftarrow \mathrm{RO}_6(0^\kappa, \vec{\rho})$ // joint rand seed
24 if $\vec{state}[1] \in \mathscr{J}$: $\mathsf{bad} \leftarrow \mathsf{true}$; halt.
25 $\mathscr{J} \leftarrow \mathscr{J} \cup \{\vec{state}[1]\}$
26 $jr \leftarrow \mathrm{ROExt}_1(\vec{state}[1], \varepsilon)$
27 $inp \leftarrow \sum_{\hat{j}=1}^{s} \vec{inp}[\hat{j}]$; $\pi \leftarrow \sum_{\hat{j}=1}^{s} \vec{\pi}[\hat{j}]$
28 if $n = n^*$: output $\pi$ and halt.
29 $vf \leftarrow\!\!\$\, \mathsf{FLP.Query}(inp, \pi, jr)$
30 $d \leftarrow \mathsf{FLP.Decide}(vf)$
31 for $\hat{j} \in [s]$:
32   $jseed_{\hat{j}} \leftarrow \vec{state}[\hat{j}]$; $jseed'_{\hat{j}} \leftarrow \mathrm{RO}_6(0^\kappa, \vec{rseed})$
33   $acc_{\hat{j}} \leftarrow d \wedge [[jseed_{\hat{j}} = jseed'_{\hat{j}}]]$
34   $\mathsf{win} \leftarrow (\mathsf{win} \bigvee [\neg \mathsf{bad} \wedge acc_{\hat{j}} \wedge inp \notin \mathscr{L}])$
35 ret $(\mathsf{win}, (msg_{\mathrm{Init}}, (\vec{vfs}[\hat{j}], \vec{rseed}[\hat{j}]))_{\hat{j} \in [s]})$
```

Figure 5.17. Malicious prover P^* for the proof of Theorem 18. The lookup in the random oracle table Rand on lines 6–8 can performed efficiently by creating a reverse-lookup table; we omit the details for brevity.

$$\Pr\big[\,\mathrm{G5}(\mathscr{B})\,\big] \leq (q_1 + q_{\mathsf{Prep}}) \cdot \varepsilon\,. \tag{5.7}$$

The claimed bound follows by gathering up all of the bounds across the games and simplifying.

5.9.2 Prio3 Privacy (Theorem 19)

We begin by instantiating the privacy game $\mathsf{Exp}^{\mathrm{PRIV}}_{\Pi,t}$ for Prio3 VDAF Π. Game G0 in Figure 5.18 was constructed by inlining Π's constituent algorithms and cleaning up the control flow. In addition, calls to RG_i have been substituted with calls to a random oracle RO_i. Let q_i denote the number of queries $\mathscr{A}$ makes to RO_i; note that $q_{\mathsf{RG}} = q_1 + \cdots + q_7$.

In our first game hop, we modify $\underline{\mathsf{Shard}}$ oracle's behavior after setting flag bad_1 on line 7. In the new game, G1 (Figure 5.18), the nonce n is sampled without replacement, ensuring that each nonce is used is unique. Applying the Fundamental Lemma of Game Playing [43], and using a birthday bound for the probability of bad_1 getting set,

```
Game $\colorbox{lightgray}{G0($\mathscr{A}$)}$ $\boxed{\text{G1}(\mathscr{A})}$:
  $(state_{\mathscr{A}}, \mathsf{V}, (sk_{\hat{j}})_{\hat{j}\in\mathsf{V}}) \leftarrow\!\$\, \mathscr{A}^{\mathrm{RO}}()$; $T \leftarrow [s] \setminus \mathsf{V}$
  if $|\mathsf{V}| + t \neq s$ return $\bot$
  $b \leftarrow\!\$\, \{0,1\}$; $b' \leftarrow\!\$\, \mathscr{A}^{\mathrm{RO},\underline{\mathsf{Shard}},\underline{\mathsf{Setup}},\underline{\mathsf{Prep}},\underline{\mathsf{Agg}}}(state_{\mathscr{A}})$
  ret $b = b'$
```

```
$\underline{\mathsf{Shard}}(\hat{k} \in \mathbb{N}, m_0, m_1 \in \mathscr{I})$:
  if $\mathrm{Used}[\hat{k}] \neq \bot$: ret $\bot$
  $n \leftarrow\!\$\, N$
  if $n \in N^*$: $\mathsf{bad}_1 \leftarrow \mathsf{true}$; $\boxed{n \leftarrow\!\$\, N \setminus N^*}$
  $N^* \leftarrow N^* \cup \{n\}$
  $inp \leftarrow \mathtt{Encode}(m_b)$
  for $\hat{j} \in [2..s]$:
    $blind_{\hat{j}}, xseed_{\hat{j}}, pseed_{\hat{j}} \leftarrow\!\$\, \{0,1\}^{\kappa}$
    $\vec{x}[\hat{j}] \leftarrow \mathrm{RO}_2(xseed_{\hat{j}}, \hat{j})$
    $\vec{rseed}[\hat{j}] \leftarrow \mathrm{RO}_7(blind_{\hat{j}}, \hat{j} \,\|\, n \,\|\, \vec{x}[\hat{j}])$
  $\vec{x}[1] \leftarrow inp - \sum_{\hat{j}=2}^{s} \vec{x}[\hat{j}]$
  $blind_1 \leftarrow\!\$\, \{0,1\}^{\kappa}$; $ps \leftarrow\!\$\, \{0,1\}^{\kappa}$
  $\vec{rseed}[1] \leftarrow \mathrm{RO}_7(blind_1, 1 \,\|\, n \,\|\, \vec{x}[1])$
  $jseed \leftarrow \mathrm{RO}_6(0^{\kappa}, \vec{rseed})$; $jr \leftarrow \mathrm{RO}_1(jseed, \varepsilon)$
  $pr \leftarrow \mathrm{RO}_4(ps, \varepsilon)$
  $\vec{\pi}[1] \leftarrow \mathsf{Prove}(inp, jr\ pr)$
  $\vec{\pi}[1] \leftarrow \vec{\pi}[1] - \sum_{\hat{j}=2}^{s} \mathrm{RO}_3(pseed_{\hat{j}}, \hat{j})$
  $\vec{x}[1] \leftarrow (\vec{x}[1], \vec{\pi}[1], blind_1)$
  for $\hat{j} \in [2..s]$:
    $\vec{x}[\hat{j}] \leftarrow (xseed_{\hat{j}}, pseed_{\hat{j}}, blind_{\hat{j}})$
  $\mathrm{Pub}[\hat{k}] \leftarrow \vec{rseed}$; $\mathrm{In}[\hat{k}, \cdot] \leftarrow \vec{x}$
  $\mathrm{Used}[\hat{k}] \leftarrow (n, m_0, m_1)$
  ret $(n, \mathrm{Pub}[\hat{k}], (\mathrm{In}[\hat{k}, \hat{j}])_{\hat{j}\in T})$
```

```
$\underline{\mathsf{Setup}}(\hat{i} \in \mathbb{N}, \hat{j} \in \mathsf{V}, st_{\mathrm{Init}} \in \{\varepsilon\})$:
  if $\mathrm{Status}[\hat{i}, \hat{j}] \neq \bot$ or $|\mathrm{Setup}[\cdot, \hat{j}]| > 0$: ret $\bot$
  $\mathrm{Setup}[\hat{i}, \hat{j}] \leftarrow st_{\mathrm{Init}}$
  $\mathrm{Status}[\hat{i}, \hat{j}] \leftarrow \mathsf{running}$
```

```
$\underline{\mathsf{Prep}}(\hat{i} \in \mathbb{N}, \hat{j} \in \mathsf{V}, \hat{k} \in \mathbb{N}, \vec{M} \in \mathscr{M}^*)$:
  if $\mathrm{Status}[\hat{i}, \hat{j}] \neq \mathsf{running}$ or $\mathrm{In}[\hat{k}, \hat{j}] = \bot$:
    ret $\bot$
  if $\mathrm{St}[\hat{i}, \hat{j}, \hat{k}] = \bot$:
    $\mathrm{St}[\hat{i}, \hat{j}, \hat{k}] \leftarrow \mathrm{Setup}[\hat{i}, \hat{j}]$
    $\vec{M} \leftarrow (\mathrm{Pub}[\hat{k}],)$
  $(n, m_0, m_1) \leftarrow \mathrm{Used}[\hat{k}]$
  if $\mathrm{St}[\hat{i}, \hat{j}, \hat{k}] = \varepsilon$:  // Process initial message from client
    $(inp, \pi, blind) \leftarrow \mathsf{Unpack}(\hat{j}, \mathrm{In}[\hat{k}, \hat{j}])$
    $(\vec{rseed},) \leftarrow \vec{M}$
    $\vec{rseed}[\hat{j}] \leftarrow \mathrm{RO}_7(blind, \hat{j} \,\|\, n \,\|\, inp)$
    $jseed \leftarrow \mathrm{RO}_6(0^{\kappa}, \vec{rseed})$; $jr \leftarrow \mathrm{RO}_1(jseed, \varepsilon)$
    $qr \leftarrow \mathrm{RO}_5(sk_{\hat{j}}, n)$
    $M \leftarrow (\mathsf{Query}(inp, \pi, jr; qr), \vec{rseed}[\hat{j}])$
    $\mathrm{St}[\hat{i}, \hat{j}, \hat{k}] \leftarrow (jseed, \mathsf{Truncate}(inp))$
    ret $(\mathsf{running}, M)$
  // Process broadcast messages from aggregators
  $(jseed, y) \leftarrow \mathrm{St}[\hat{i}, \hat{j}, \hat{k}]$
  $(\vec{vfs}[\hat{j}], \vec{rseed}[\hat{j}])_{\hat{j}\in[s]} \leftarrow \vec{M}$
  $acc \leftarrow \mathsf{Decide}(\sum_{\hat{j}=1}^{s} \vec{vfs}[\hat{j}])$
  $\mathrm{St}[\hat{i}, \hat{j}, \hat{k}] \leftarrow \bot$
  if $acc = 0$ or $jseed \neq \mathrm{RO}_6(0^{\kappa}, \vec{rseed})$:
    ret $(\mathtt{failed}, \bot)$
  $\mathrm{Out}[\hat{i}, \hat{j}, \hat{k}] \leftarrow y$
  $\mathrm{Batch}_0[\hat{i}, \hat{j}, \hat{k}] \leftarrow m_0$
  $\mathrm{Batch}_1[\hat{i}, \hat{j}, \hat{k}] \leftarrow m_1$
  ret $(\mathtt{finished}, \bot)$
```

```
$\underline{\mathsf{Agg}}(\hat{i} \in \mathbb{N}, \hat{j} \in \mathsf{V})$:
  if $\mathrm{Status}[\hat{i}, \hat{j}] \neq \mathsf{running}$: ret $\bot$
  if $F(\mathrm{Batch}_0[\hat{i}, \hat{j}, \cdot]) \neq F(\mathrm{Batch}_1[\hat{i}, \hat{j}, \cdot])$
      and $(\forall j, j' \in \mathsf{V})\, sk_j = sk_{j'}$: ret $\bot$
  $\mathrm{Status}[\hat{i}, \hat{j}] \leftarrow \mathtt{finished}$
  $\vec{y} \leftarrow \mathrm{Out}[\hat{i}, \hat{j}, \cdot]$
  ret $\sum_{i=1}^{|\vec{y}|} \vec{y}[i]$
```

```
$\mathrm{RO}_i(seed, cntxt)$:
  $l \leftarrow (jl, n, m, pl, ql)$
  if $\mathrm{Rand}[i, seed, cntxt] = \bot$:
    if $i \leq 5$: $\mathrm{Rand}[i, seed, cntxt] \leftarrow\!\$\, \mathbb{F}^{l[i]}$
    else: $\mathrm{Rand}[i, seed, cntxt] \leftarrow\!\$\, \{0,1\}^{\kappa}$
  ret $\mathrm{Rand}[i, seed, cntxt]$
```

Figure 5.18. Games G0 and G1 for the proof of Theorem 19. This game is identical to the privacy game for Π, except the Shard, Prep, and Agg algorithms have been inlined. Algorithm Unpack is as defined in Figure 5.5. The random oracles RO_i are lazy-evaluated in a table Rand.

$$\Pr\big[\,\mathrm{G0}(\mathscr{A})\,\big] \le \Pr\big[\,\mathrm{G1}(\mathscr{A})\,\big] + \frac{{q_{\mathsf{Shard}}}^2}{|N|}\,. \tag{5.8}$$

Next we replace the adversary $\mathscr{A}$ with one that controls all but one aggregator. We construct such an adversary $\mathscr{B}$ as a wrapper around $\mathscr{A}$, and show that $\mathscr{B}$ wins with at least the probability of $\mathscr{A}$. The adversary $\mathscr{B}$, defined in Figure 5.19, presents four oracles $\mathsf{ShardSim}$, $\mathsf{SetupSim}$, $\mathsf{PrepSim}$, and AggSim to adversary $\mathscr{A}$, each emulating an oracle in game G1. Algorithms $\mathsf{SetupSim}$, $\mathsf{PrepSim}$, AggSim are computed by $\mathscr{B}$ just as the respective oracles in game G1 except that queries pertaining to aggregator z are forwarded to $\mathscr{B}$'s own oracles. Algorithm $\mathsf{ShardSim}$ forwards $\mathscr{A}$'s query to $\underline{\mathsf{Shard}}$ in the natural way, but returns the shares of the aggregators deemed honest by $\mathscr{A}$.

We claim that $\mathscr{B}$ perfectly simulates G1(A). This is obvious for $\mathsf{ShardSim}$ and for queries for which $\hat{j} = z$; in these cases, $\mathscr{B}$ simply forwards its queries to the appropriate oracles without changing their inputs. The only difference is that $\underline{\mathsf{Shard}}$ returns more input shares than $\mathscr{A}$ requests; $\mathscr{B}$ stores these extra input shares for its own use and does not reveal them. By construction, this is a subset of the input shares returned by the query.

When $\mathscr{A}$ makes queries to $\mathsf{SetupSim}$, $\mathsf{PrepSim}$, or AggSim with $\hat{j} \in \mathsf{V} \setminus \{z\}$, our wrapper adversary performs the operations of $\underline{\mathsf{Setup}}$, $\underline{\mathsf{Prep}}$, or $\underline{\mathsf{Agg}}$ respectively. Effectively, adversary $\mathscr{B}$ uses its stored input shares to fill in entries of tables In, Batch, Setup, Status, St, and Out exactly as the real privacy game would. Since each entry is disambiguated by its $\hat{j}$, there is no overlap with the tables maintained by the game; every table entry read by $\mathscr{B}$ must first have been written by $\mathscr{B}$ and thus all the information it needs to simulate the game perfectly is accessible. It follows that

$$\Pr\big[\,\mathrm{G1}(\mathscr{A})\,\big] = \Pr\big[\,\mathrm{G1}(\mathscr{B})\,\big]\,. \tag{5.9}$$

In the next game hop (Figure 5.20) we make some simplifying changes, including cleaning up the bad_1 flag and substituting $\{z\}$ for V and simplifying accordingly. (We do not highlight this change in Figure 5.20, as it is fairly straightforward.) We also make the following breaking change: In game G2, we program the table Rand with values chosen by the $\underline{\mathsf{Shard}}$ oracle for the joint randomness, prover randomness, and query randomness. Accordingly, we pass these joint

```
Adversary $\mathscr{B}^{\mathrm{RO}}[\mathscr{A}]()$:
    $(state_{\mathscr{A}}, \mathsf{V}, (sk_{\hat{j}})_{\hat{j}\in\mathsf{V}}) \leftarrow\!\!\$\, \mathscr{A}^{\mathrm{RO}}()$
    $z \leftarrow\!\!\$\, \mathsf{V}$; $\mathsf{V}' \leftarrow \{z\}$; $T \leftarrow [s] \setminus \mathsf{V}$
    $state_{\mathscr{B}} \leftarrow (state_{\mathscr{A}}, z, T, \mathsf{V}, (sk_{\hat{j}})_{\hat{j}\in\mathsf{V}})$
    ret $(state_{\mathscr{B}}, \mathsf{V}', (sk_z))$

Adversary $\mathscr{B}^{\mathrm{RO},\underline{\mathsf{Shard}},\underline{\mathsf{Setup}},\underline{\mathsf{Prep}},\underline{\mathsf{Agg}}}[\mathscr{A}](state_{\mathscr{B}})$:
    $(state_{\mathscr{A}}, z, T, \mathsf{V}, (sk_{\hat{j}})_{\hat{j}\in\mathsf{V}}) \leftarrow state_{\mathscr{B}}$
    $b' \leftarrow\!\!\$\, \mathscr{A}^{\mathrm{RO},\mathsf{ShardSim},\mathsf{SetupSim},\mathsf{PrepSim},\mathsf{AggSim}}(state_{\mathscr{A}})$
    ret $b'$
```

Figure 5.19. Wrapper adversary $\mathscr{B}$ for the proof of Theorem 19. Algorithms $\mathsf{SetupSim}$, $\mathsf{PrepSim}$, AggSim are evaluated by $\mathscr{B}$ just as the respective oracles in game G0 except that queries pertaining to aggregator z are forwarded to $\mathscr{B}$'s own oracles. Algorithm $\mathsf{ShardSim}$ forwards $\mathscr{A}$'s query to $\underline{\mathsf{Shard}}$ in the natural way, but returns the shares of the aggregators deemed honest by $\mathscr{A}$.

randomness and query randomness to the honest aggregator via its input share ($\mathrm{In}[\hat{k}, z]$; see line 29). This is to simplify bookkeeping in the next step.

Game G2 is identical to game G1 until programming Rand overwrites an already existing value on line 18, 19, 20, or 21.

- Line 18: Adversary $\mathscr{B}$ either has to guess *jseed* or guess the input to RO_6 used to derive it. For the latter it must must guess $\vec{rseed}$ or all of the corresponding inputs to RO_7, which include the blinds generated by oracle $\underline{\mathsf{Shard}}$. Taking union bound over all the queries to $\underline{\mathsf{Shard}}$, the game overwrites Rand at this point with probability at most $q_1 q_{\mathsf{Shard}}/2^{\kappa} + (q_6 + q_7) q_{\mathsf{Shard}}/(2^{s\cdot\kappa})$.

- Line 19: $\mathscr{B}$ must guess the *ps* generated by oracle $\underline{\mathsf{Shard}}$, so the game overwrites the table with probability at most $q_4 q_{\mathsf{Shard}}/2^{\kappa}$.

- Line 20: $\mathscr{B}$ must guess the nonce n generated by the oracle. The game overwrites the table here with probability at most $q_5 q_{\mathsf{Shard}}/|N|$.

- Line 21: $\mathscr{B}$ must guess $\vec{rseed}$ or all of the corresponding inputs to RO_7, so the game overwrites the table with probability at most $(q_6 + q_7) q_{\mathsf{Shard}}/(2^{s\cdot\kappa})$.

We bound the probability of $\mathscr{B}$ distinguishing between these games by the probability that any one of these events occurs, Gathering up the terms yields

$$\begin{aligned} \Pr\big[\,\mathrm{G1}(\mathscr{B})\,\big] &\le \Pr\big[\,\mathrm{G2}(\mathscr{B})\,\big] \qquad (5.10) \\ &\quad + \frac{(q_1 + q_4) q_{\mathsf{Shard}}}{2^{\kappa}} + \frac{(q_6 + q_7) q_{\mathsf{Shard}}}{2^{s\cdot\kappa - 1}} + \frac{q_5 q_{\mathsf{Shard}}}{|N|}\,. \qquad (5.11) \end{aligned}$$

```
Game $\mathrm{G1}(\mathcal{B})$ $\boxed{\mathrm{G2}(\mathcal{B})}$:
  $(state_{\mathcal{B}}, \{z\}, (sk_z,)) \leftarrow\$ \mathcal{B}^{\mathrm{RO}}()$; $T \leftarrow [s] \setminus \{z\}$
  $b \leftarrow\$ \{0,1\}$; $b' \leftarrow\$ \mathcal{B}^{\mathrm{RO},\underline{\mathsf{Shard}},\underline{\mathsf{Setup}},\underline{\mathsf{Prep}},\underline{\mathsf{Agg}}}(state_{\mathcal{B}})$
  ret $b = b'$

$\underline{\mathsf{Shard}}(\hat{k} \in \mathbb{N}, m_0, m_1 \in \mathcal{I})$:
  if $\mathrm{Used}[\hat{k}] \neq \bot$: ret $\bot$
  $n \leftarrow\$ N \setminus N^*$; $N^* \leftarrow N^* \cup \{n\}$
  $inp \leftarrow \mathtt{Encode}(m_b)$
  for $\hat{j} \in [2..s]$:
    $blind_{\hat{j}}, xseed_{\hat{j}}, pseed_{\hat{j}} \leftarrow\$ \{0,1\}^{\kappa}$
    $\vec{x}[\hat{j}] \leftarrow \mathrm{RO}_2(xseed_{\hat{j}}, \hat{j})$
    $\vec{rseed}[\hat{j}] \leftarrow \mathrm{RO}_7(blind_{\hat{j}}, \hat{j} \| n \| \vec{x}[\hat{j}])$
  $\vec{x}[1] \leftarrow inp - \sum_{\hat{j}=2}^{s} \vec{x}[\hat{j}]$
  $blind_1 \leftarrow\$ \{0,1\}^{\kappa}$; $ps \leftarrow\$ \{0,1\}^{\kappa}$
  $\vec{rseed}[1] \leftarrow \mathrm{RO}_7(blind_1, 1 \| n \| \vec{x}[1])$
  $jseed \leftarrow \mathrm{RO}_6(0^{\kappa}, \vec{rseed})$; $jr \leftarrow \mathrm{RO}_1(jseed, \varepsilon)$
  $pr \leftarrow \mathrm{RO}_4(ps, \varepsilon)$
  $\boxed{jseed \leftarrow\$ \{0,1\}^{\kappa}}$
  $\boxed{jr \leftarrow\$ \mathbb{F}^{jl};\ pr \leftarrow\$ \mathbb{F}^{pl};\ qr \leftarrow\$ \mathbb{F}^{ql}}$
  $\boxed{\mathrm{Rand}[1, jseed, \varepsilon] \leftarrow jr}$
  $\boxed{\mathrm{Rand}[4, ps, \varepsilon] \leftarrow pr}$
  $\boxed{\mathrm{Rand}[5, sk_z, n] \leftarrow qr}$
  $\boxed{\mathrm{Rand}[6, 0^{\kappa}, \vec{rseed}] \leftarrow jseed}$
  $\vec{\pi}[1] \leftarrow \mathsf{Prove}(inp, jr; pr)$
  $\vec{\pi}[1] \leftarrow \vec{\pi}[1] - \sum_{\hat{j}=2}^{s} \mathrm{RO}_3(pseed_{\hat{j}}, \hat{j})$
  $\vec{x}[1] \leftarrow (\vec{x}[1], \vec{\pi}[1], blind_1)$
  for $\hat{j} \in [2..s]$:
    $\vec{x}[\hat{j}] \leftarrow (xseed_{\hat{j}}, pseed_{\hat{j}}, blind_{\hat{j}})$
  $\mathrm{Pub}[\hat{k}] \leftarrow \vec{rseed}$
  $\mathrm{In}[\hat{k}, \cdot] \leftarrow \vec{x}$
  $\boxed{\mathrm{In}[\hat{k}, z] \leftarrow (\vec{x}[z], jseed, jr, qr)}$
  $\mathrm{Used}[\hat{k}] \leftarrow (n, m_0, m_1)$
  ret $(n, \mathrm{Pub}[\hat{k}], (\mathrm{In}[\hat{k}, \hat{j}])_{\hat{j} \in T})$

$\underline{\mathsf{Setup}}(\hat{\imath} \in \mathbb{N}, \hat{j} \in \{z\}, st_{\mathrm{Init}} \in \{\varepsilon\})$:
  if $\mathrm{Status}[\hat{\imath}, \hat{j}] \neq \bot$ or $|\mathrm{Setup}[\cdot, \hat{j}]| > 0$: ret $\bot$
  $\mathrm{Setup}[\hat{\imath}, \hat{j}] \leftarrow st_{\mathrm{Init}}$
  $\mathrm{Status}[\hat{\imath}, \hat{j}] \leftarrow \mathsf{running}$

$\underline{\mathsf{Prep}}(\hat{\imath} \in \mathbb{N}, \hat{j} \in \{z\}, \hat{k} \in \mathbb{N}, \vec{M} \in \mathcal{M}^*)$:
  if $\mathrm{Status}[\hat{\imath}, \hat{j}] \neq \mathsf{running}$ or $\mathrm{In}[\hat{k}, \hat{j}] = \bot$:
    ret $\bot$
  if $\mathrm{St}[\hat{\imath}, \hat{j}, \hat{k}] = \bot$:
    $\mathrm{St}[\hat{\imath}, \hat{j}, \hat{k}] \leftarrow \mathrm{Setup}[\hat{\imath}, \hat{j}]$
    $\vec{M} \leftarrow (\mathrm{Pub}[\hat{k}],)$
  $(n, m_0, m_1) \leftarrow \mathrm{Used}[\hat{k}]$
  if $\mathrm{St}[\hat{\imath}, \hat{j}, \hat{k}] = \varepsilon$:  // Process initial message from client
    $(inp, \pi, blind) \leftarrow \mathsf{Unpack}(\hat{j}, \mathrm{In}[\hat{k}, \hat{j}])$
    $\boxed{(x, jseed, jr, qr) \leftarrow \mathrm{In}[\hat{k}, \hat{j}]}$
    $\boxed{(inp, \pi, blind) \leftarrow \mathsf{Unpack}(\hat{j}, x)}$
    $(\vec{rseed},) \leftarrow \vec{M}$
    $\vec{rseed}[\hat{j}] \leftarrow \mathrm{RO}_7(blind, \hat{j} \| n \| inp)$
    $jseed \leftarrow \mathrm{RO}_6(0^{\kappa}, \vec{rseed})$; $jr \leftarrow \mathrm{RO}_1(jseed, \varepsilon)$
    $qr \leftarrow \mathrm{RO}_5(sk_z, n)$
    $M \leftarrow (\mathsf{Query}(inp, \pi, jr; qr), \vec{rseed}[\hat{j}])$
    $\mathrm{St}[\hat{\imath}, \hat{j}, \hat{k}] \leftarrow (jseed, \mathsf{Truncate}(inp))$
    ret $(\mathsf{running}, M)$
  // Process broadcast messages from aggregators
  $(jseed, y) \leftarrow \mathrm{St}[\hat{\imath}, \hat{j}, \hat{k}]$
  $(\vec{vfs}[\hat{j}], \vec{rseed}[\hat{j}])_{\hat{j} \in [s]} \leftarrow \vec{M}$
  $acc \leftarrow \mathsf{Decide}(\sum_{\hat{j}=1}^{s} \vec{vfs}[\hat{j}])$
  $\mathrm{St}[\hat{\imath}, \hat{j}, \hat{k}] \leftarrow \bot$
  if $acc = 0$ or $jseed \neq \mathrm{RO}_6(0^{\kappa}, \vec{rseed})$:
    ret $(\mathtt{failed}, \bot)$
  $\mathrm{Out}[\hat{\imath}, \hat{j}, \hat{k}] \leftarrow y$
  $\mathrm{Batch}_0[\hat{\imath}, \hat{j}, \hat{k}] \leftarrow m_0$
  $\mathrm{Batch}_1[\hat{\imath}, \hat{j}, \hat{k}] \leftarrow m_1$
  ret $(\mathtt{finished}, \bot)$

$\underline{\mathsf{Agg}}(\hat{\imath} \in \mathbb{N}, \hat{j} \in \{z\})$:
  if $\mathrm{Status}[\hat{\imath}, \hat{j}] \neq \mathsf{running}$: ret $\bot$
  if $F(\mathrm{Batch}_0[\hat{\imath}, \hat{j}, \cdot]) \neq F(\mathrm{Batch}_1[\hat{\imath}, \hat{j}, \cdot])$: ret $\bot$
  $\mathrm{Status}[\hat{\imath}, \hat{j}] \leftarrow \mathtt{finished}$
  $\vec{y} \leftarrow \mathrm{Out}[\hat{\imath}, \hat{j}, \cdot]$
  ret $\sum_{i=1}^{|\vec{y}|} \vec{y}[i]$

$\mathrm{RO}_i(seed, cntxt)$:
  $l \leftarrow (jl, n, m, pl, ql)$
  if $\mathrm{Rand}[i, seed, cntxt] = \bot$:
    if $i \leq 5$: $\mathrm{Rand}[i, seed, cntxt] \leftarrow\$ \mathbb{F}^{l[i]}$
    else: $\mathrm{Rand}[i, seed, cntxt] \leftarrow\$ \{0,1\}^{\kappa}$
  ret $\mathrm{Rand}[i, seed, cntxt]$
```

Figure 5.20. Game G2 for the proof of Theorem 19.

In the next game hop (see G3 in the left panel of Figure 5.21) we prepare to ensure that all of the input shares $\vec{x}$, proof shares $\vec{\pi}$, and the public share $\vec{rseed}$ sampled by the Shard oracle are uniform random. We do so by sampling these values prior to processing m_b and programming the random oracle with the sample values (lines 3–12) so long as doing so does overwrite existing values (see procedure PO on lines 37–40). The game sets a flag bad_4 if $\underline{\mathsf{Shard}}$ would have overwritten an existing value. This does not change the adversary's view of the experiment, so

$$\Pr\left[\,\mathrm{G2}(\mathscr{B})\,\right] = \Pr\left[\,\mathrm{G3}(\mathscr{B})\,\right]. \tag{5.12}$$

Next, in game G4 (top-right panel of Figure 5.21) we change oracle $\underline{\mathsf{Shard}}$'s behavior after bad_4 gets set. In particular, if ever PO is called on an input $(i, \mathit{seed}, \mathit{cntxt}, \mathit{out})$ for which $\mathrm{Rand}[i, \mathit{seed}, \mathit{cntxt}]$, the value is overwritten. Game G4 is identical to game G3 until bad_4 gets set. Then we apply the Fundamental Lemma of Game Playing [43] to show that

$$\begin{aligned} \Pr\left[\,\mathrm{G3}(\mathscr{B})\,\right] &\leq \Pr\left[\,\mathrm{G4}(\mathscr{B})\,\right] + \Pr\left[\,\mathrm{G4}(\mathscr{B}) \text{ sets } \mathsf{bad}_4\,\right] && (5.13)\\ &\leq \Pr\left[\,\mathrm{G4}(\mathscr{B})\,\right] + \frac{((s-1)(q_2+q_3)+s(q_7))q_{\mathsf{Shard}}}{2^\kappa}. && (5.14) \end{aligned}$$

The probability that $\mathscr{B}$ sets the bad_4 flag in Game G4 is the probability that $\mathscr{B}$ makes a random oracle query that gets overwritten on line 9, 10, 11, or 12. On each line, the random oracle is programmed with a uniform random string sampled by the oracle prior to being revealed to the adversary. Rolling out the for-loop on line 8 and taking a union bound over all $\underline{\mathsf{Shard}}$ queries yields the claimed bound.

Next, in game G5 (bottom-right panel of Figure 5.21) we simplify the $\underline{\mathsf{Shard}}$ oracle by inlining calls to PO and replacing invocations of RO with corresponding value generated by the oracle. These changes do not change the view of the adversary, so

$$\Pr\left[\,\mathrm{G4}(\mathscr{B})\,\right] = \Pr\left[\,\mathrm{G5}(\mathscr{B})\,\right]. \tag{5.15}$$

Up to this point, we have constructed the "leader" input and proof shares differently than all of the other shares: we pick all other shares randomly, then set $\vec{x}[1] = \mathit{inp} - \sum_{\hat{j}=2}^{s} \vec{x}[\hat{j}]$ and $\vec{\pi}[1] = \pi - \sum_{\hat{j}=2}^{s} \vec{\pi}[\hat{j}]$. In our next game, we instead sample the "leader" shares randomly and compute the shares of the honest aggregator z in a distinguished manner: $\vec{x}[z] = \mathit{inp} - \sum_{\hat{j}\in T} \vec{x}[\hat{j}]$

```
$\underline{\mathsf{Shard}}(\hat{k} \in \mathbb{N}, m_0, m_1 \in \mathscr{I})$:    Game G2 $\boxed{\text{G3}}$
if $\mathrm{Used}[\hat{k}] \neq \bot$: ret $\bot$
$n \leftarrow\!\$\, N \setminus N^*$; $N^* \leftarrow N^* \cup \{n\}$
$\vec{x} \leftarrow\!\$\, (\mathbb{F}^n)^s$; $\vec{\pi} \leftarrow\!\$\, (\mathbb{F}^m)^s$
$(\mathit{blind}_1, \ldots, \mathit{blind}_s) \leftarrow\!\$\, (\{0,1\}^\kappa)^s$
$(\mathit{xseed}_2, \ldots, \mathit{xseed}_s) \leftarrow\!\$\, (\{0,1\}^\kappa)^{s-1}$
$(\mathit{pseed}_2, \ldots, \mathit{pseed}_s) \leftarrow\!\$\, (\{0,1\}^\kappa)^{s-1}$
$\vec{\mathit{rseed}} \leftarrow\!\$\, (\{0,1\}^\kappa)^s$
for $\hat{j} \in [s]$:
  $\mathsf{PO}_2(\mathit{xseed}_{\hat{j}}, \hat{j}, \vec{x}[\hat{j}])$
  $\mathsf{PO}_3(\mathit{pseed}_{\hat{j}}, \hat{j}, \vec{\pi}[\hat{j}])$
  $\mathsf{PO}_7(\mathit{blind}_{\hat{j}}, \hat{j} \,\|\, n \,\|\, \vec{x}[\hat{j}], \vec{\mathit{rseed}}[\hat{j}])$
$\mathsf{PO}_7(\mathit{blind}_1, 1 \,\|\, n \,\|\, \vec{x}[1], \vec{\mathit{rseed}}[1])$
$\mathit{inp} \leftarrow \mathtt{Encode}(m_b)$
for $\hat{j} \in [2..s]$:
  $\mathit{blind}_{\hat{j}}, \mathit{xseed}_{\hat{j}}, \mathit{pseed}_{\hat{j}} \leftarrow\!\$\, \{0,1\}^\kappa$
  $\vec{x}[\hat{j}] \leftarrow \mathrm{RO}_2(\mathit{xseed}_{\hat{j}}, \hat{j})$
  $\vec{\mathit{rseed}}[\hat{j}] \leftarrow \mathrm{RO}_7(\mathit{blind}_{\hat{j}}, \hat{j} \,\|\, n \,\|\, \vec{x}[\hat{j}])$
$\vec{x}[1] \leftarrow \mathit{inp} - \sum_{\hat{j}=2}^{s} \vec{x}[\hat{j}]$
$\mathit{blind}_1 \leftarrow\!\$\, \{0,1\}^\kappa$; $\mathit{ps} \leftarrow\!\$\, \{0,1\}^\kappa$
$\vec{\mathit{rseed}}[1] \leftarrow \mathrm{RO}_7(\mathit{blind}_1, 1 \,\|\, n \,\|\, \vec{x}[1])$
$\mathit{jseed} \leftarrow\!\$\, \{0,1\}^\kappa$
$\mathit{jr} \leftarrow\!\$\, \mathbb{F}^{jl}$; $\mathit{pr} \leftarrow\!\$\, \mathbb{F}^{pl}$; $\mathit{qr} \leftarrow\!\$\, \mathbb{F}^{ql}$
$\mathrm{Rand}[1, \mathit{jseed}, \varepsilon] \leftarrow \mathit{jr}$
$\mathrm{Rand}[4, \mathit{ps}, \varepsilon] \leftarrow \mathit{pr}$
$\mathrm{Rand}[5, \mathit{sk}_z, n] \leftarrow \mathit{qr}$
$\mathrm{Rand}[6, 0^\kappa, \vec{\mathit{rseed}}] \leftarrow \mathit{jseed}$
$\vec{\pi}[1] \leftarrow \mathsf{Prove}(\mathit{inp}, \mathit{jr}\,; \mathit{pr})$
$\vec{\pi}[1] \leftarrow \vec{\pi}[1] - \sum_{\hat{j}=2}^{s} \mathrm{RO}_3(\mathit{pseed}_{\hat{j}}, \hat{j})$
$\vec{x}[1] \leftarrow (\vec{x}[1], \vec{\pi}[1], \mathit{blind}_1)$
for $\hat{j} \in [2..s]$:
  $\vec{x}[\hat{j}] \leftarrow (\mathit{xseed}_{\hat{j}}, \mathit{pseed}_{\hat{j}}, \mathit{blind}_{\hat{j}})$
$\mathrm{Pub}[\hat{k}] \leftarrow \vec{\mathit{rseed}}$
$\mathrm{In}[\hat{k}, \cdot] \leftarrow \vec{x}$
$\mathrm{In}[\hat{k}, z] \leftarrow (\vec{x}[z], \mathit{jseed}, \mathit{jr}, \mathit{qr})$
$\mathrm{Used}[\hat{k}] \leftarrow (n, m_0, m_1)$
ret $(n, \mathrm{Pub}[\hat{k}], (\mathrm{In}[\hat{k}, \hat{j}])_{\hat{j} \in T})$

Algorithm $\mathsf{PO}_i(\mathit{seed}, \mathit{cntxt}, \mathit{out})$:
if $\mathrm{Rand}[i, \mathit{seed}, \mathit{cntxt}] = \bot$:
  $\mathrm{Rand}[i, \mathit{seed}, \mathit{cntxt}] \leftarrow \mathit{out}$
else:
  $\mathsf{bad}_4 \leftarrow \mathsf{true}$
```

```
Algorithm $\mathsf{PO}_i(\mathit{seed}, \mathit{cntxt}, \mathit{out})$:    Game G3 $\boxed{\text{G4}}$
if $\mathrm{Rand}[i, \mathit{seed}, \mathit{cntxt}] = \bot$:
  $\mathrm{Rand}[i, \mathit{seed}, \mathit{cntxt}] \leftarrow \mathit{out}$
else:
  $\mathsf{bad}_4 \leftarrow \mathsf{true}$; $\boxed{\mathrm{Rand}[i, \mathit{seed}, \mathit{cntxt}] \leftarrow \mathit{out}}$
```

```
$\underline{\mathsf{Shard}}(\hat{k} \in \mathbb{N}, m_0, m_1 \in \mathscr{I})$:    Game G5
if $\mathrm{Used}[\hat{k}] \neq \bot$: ret $\bot$
$n \leftarrow\!\$\, N \setminus N^*$; $N^* \leftarrow N^* \cup \{n\}$
$\vec{x} \leftarrow\!\$\, (\mathbb{F}^n)^s$; $\vec{\pi} \leftarrow\!\$\, (\mathbb{F}^m)^s$
$(\mathit{blind}_1, \ldots, \mathit{blind}_s) \leftarrow\!\$\, (\{0,1\}^\kappa)^s$
$(\mathit{xseed}_2, \ldots, \mathit{xseed}_s) \leftarrow\!\$\, (\{0,1\}^\kappa)^{s-1}$
$(\mathit{pseed}_2, \ldots, \mathit{pseed}_s) \leftarrow\!\$\, (\{0,1\}^\kappa)^{s-1}$
$\vec{\mathit{rseed}} \leftarrow\!\$\, (\{0,1\}^\kappa)^s$
for $\hat{j} \in [s]$:
  $\mathrm{Rand}[2, \mathit{xseed}_{\hat{j}}, \hat{j}] \leftarrow \vec{x}[\hat{j}]$
  $\mathrm{Rand}[3, \mathit{pseed}_{\hat{j}}, \hat{j}] \leftarrow \vec{\pi}[\hat{j}]$
  $\mathrm{Rand}[7, \mathit{blind}_{\hat{j}}, \hat{j} \,\|\, n \,\|\, \vec{x}[\hat{j}]] \leftarrow \vec{\mathit{rseed}}[\hat{j}]$
$\mathrm{Rand}[7, \mathit{blind}_1, 1 \,\|\, n \,\|\, \vec{x}[1]] \leftarrow \vec{\mathit{rseed}}[1]$
$\mathit{inp} \leftarrow \mathtt{Encode}(m_b)$
$\vec{x}[1] \leftarrow \mathit{inp} - \sum_{\hat{j}=2}^{s} \vec{x}[\hat{j}]$
$\mathit{ps} \leftarrow\!\$\, \{0,1\}^\kappa$
$\mathit{jseed} \leftarrow\!\$\, \{0,1\}^\kappa$
$\mathit{jr} \leftarrow\!\$\, \mathbb{F}^{jl}$; $\mathit{pr} \leftarrow\!\$\, \mathbb{F}^{pl}$; $\mathit{qr} \leftarrow\!\$\, \mathbb{F}^{ql}$
$\mathrm{Rand}[1, \mathit{jseed}, \varepsilon] \leftarrow \mathit{jr}$
$\mathrm{Rand}[4, \mathit{ps}, \varepsilon] \leftarrow \mathit{pr}$
$\mathrm{Rand}[5, \mathit{sk}_z, n] \leftarrow \mathit{qr}$
$\mathrm{Rand}[6, 0^\kappa, \vec{\mathit{rseed}}] \leftarrow \mathit{jseed}$
$\vec{\pi}[1] \leftarrow \mathsf{Prove}(\mathit{inp}, \mathit{jr}\,; \mathit{pr})$
$\vec{\pi}[1] \leftarrow \vec{\pi}[1] - \sum_{\hat{j}=2}^{s} \vec{\pi}[\hat{j}]$
$\vec{x}[1] \leftarrow (\vec{x}[1], \vec{\pi}[1], \mathit{blind}_1)$
for $\hat{j} \in [2..s]$:
  $\vec{x}[\hat{j}] \leftarrow (\mathit{xseed}_{\hat{j}}, \mathit{pseed}_{\hat{j}}, \mathit{blind}_{\hat{j}})$
$\mathrm{Pub}[\hat{k}] \leftarrow \vec{\mathit{rseed}}$
$\mathrm{In}[\hat{k}, \cdot] \leftarrow \vec{x}$
$\mathrm{In}[\hat{k}, z] \leftarrow (\vec{x}[z], \mathit{jseed}, \mathit{jr}, \mathit{qr})$
$\mathrm{Used}[\hat{k}] \leftarrow (n, m_0, m_1)$
ret $(n, \mathrm{Pub}[\hat{k}], (\mathrm{In}[\hat{k}, \hat{j}])_{\hat{j} \in T})$
```

Figure 5.21. Games G3 (left), G4 (top-right), and G5 (bottom-right) for the proof of Theorem 19. Only the Shard is shown, as this is the only object that changes in each game hop.

and $\vec{\pi}[z] = \pi - \sum_{\hat{j} \in T} \vec{x}[\hat{j}]$, where $T = [s] \setminus \{z\}$. If $z = 1$, this changes nothing. Otherwise, consider that in G5, we have

$$\vec{x}[1] = inp - \sum_{\hat{j}=2}^{s} \vec{x}[\hat{j}] = inp - \left(\sum_{\hat{j} \in T} \vec{x}[\hat{j}] - \vec{x}[z] + \vec{x}[1] \right) .$$

If we add $\vec{x}[z] - \vec{x}[1]$ to both sides of this equation, we can see that in G4, it was already true that $\vec{x}[z] = inp - \sum_{\hat{j} \in T} \vec{x}[\hat{j}]$. The same holds true for $\vec{\pi}[z]$ by an analogous argument. Therefore the distributions of aggregators' 1 and z's input and proof shares are unchanged between G4 and G5, and we have

$$\Pr[\mathrm{G5}(\mathscr{B})] = \Pr[\mathrm{G6}(\mathscr{B})] . \tag{5.16}$$

In our next game (G7, defined in the right panel of Figure 5.22) we run the query algorithm for aggregator z in the <u>Shard</u> oracle and only send the result to <u>Prep</u>. The adversary cannot detect the timing of when this algorithm is run, so we have

$$\Pr[\mathrm{G6}(\mathscr{B})] = \Pr[\mathrm{G7}(\mathscr{B})] . \tag{5.17}$$

In the next game (G8, defined in the left panel of Figure 5.23) we run $\mathsf{View}_{\mathsf{FLP}}$ (as defined in Section 5.2) on input inp to get jr, qr, and a verifier σ and use these to compute <u>Shard</u>'s output. We have defined $\vec{x}[z] = inp - \sum_{\hat{j} \in T} \vec{x}[\hat{j}]$ and $\vec{x}[z] = \pi - \sum_{\hat{j} \in T} \vec{\pi}[\hat{j}]$. Using the full linearity of FLP, we can the honest aggregator z's verifier share vfs in terms of jr, qr, σ and the corrupt aggregators' shares, since:

$$\mathsf{Query}(inp, \pi, jr; qr) = \mathsf{Query}(\vec{x}[z], \vec{\pi}[z], jr; qr) \tag{5.18}$$

$$+ \sum_{\hat{j} \in T} \mathsf{Query}(\vec{x}[\hat{j}], \vec{\pi}[\hat{j}], jr; qr) \tag{5.19}$$

This revision to the game does not change the outcome of the experiment. However, since we do not have access to the prover randomness generated by $\mathsf{View}_{\mathsf{FLP}}(inp)$, we can no longer consistently program the random oracle (see 19). Fortunately, to trigger this inconsistency, the adversary would have to guess the seed ps used to to program it prior to calling <u>Shard</u>. It follows

```
Shard($\hat{k} \in \mathbb{N}, m_0, m_1 \in \mathcal{I}$):                Game G5 $\boxed{\text{G6}}$
  if Used$[\hat{k}] \neq \bot$: ret $\bot$
  $n \leftarrow\!\!\$\, N \setminus N^*$; $N^* \leftarrow N^* \cup \{n\}$
  $\vec{x} \leftarrow\!\!\$\, (\mathbb{F}^n)^s$; $\vec{\pi} \leftarrow\!\!\$\, (\mathbb{F}^m)^s$
  $(blind_1, \ldots, blind_s) \leftarrow\!\!\$\, (\{0,1\}^\kappa)^s$
  $(xseed_2, \ldots, xseed_s) \leftarrow\!\!\$\, (\{0,1\}^\kappa)^{s-1}$
  $(pseed_2, \ldots, pseed_s) \leftarrow\!\!\$\, (\{0,1\}^\kappa)^{s-1}$
  $\vec{rseed} \leftarrow\!\!\$\, (\{0,1\}^\kappa)^s$
  $jr \leftarrow\!\!\$\, \mathbb{F}^{jl}$; $pr \leftarrow\!\!\$\, \mathbb{F}^{pl}$
  $inp \leftarrow \mathsf{Encode}(m_b)$
  $\pi \leftarrow \mathsf{Prove}(inp, jr\,; pr)$
  $\vec{x}[z] \leftarrow inp - \sum_{\hat{j} \in T} \vec{x}[\hat{j}]$
  $\vec{\pi}[z] \leftarrow \pi - \sum_{\hat{j} \in T} \vec{\pi}[\hat{j}]$
  for $\hat{j} \in [s]$:
    Rand$[2, xseed_{\hat{j}}, \hat{j}] \leftarrow \vec{x}[\hat{j}]$
    Rand$[3, pseed_{\hat{j}}, \hat{j}] \leftarrow \vec{\pi}[\hat{j}]$
    Rand$[7, blind_{\hat{j}}, \hat{j} \,\|\, n \,\|\, \vec{x}[\hat{j}]] \leftarrow \vec{rseed}[\hat{j}]$
  Rand$[7, blind_1, 1 \,\|\, n \,\|\, \vec{x}[1]] \leftarrow \vec{rseed}[1]$
  $inp \leftarrow \mathsf{Encode}(m_b)$
  $\vec{x}[1] \leftarrow inp - \sum_{\hat{j}=2}^{s} \vec{x}[\hat{j}]$
  $ps \leftarrow\!\!\$\, \{0,1\}^\kappa$
  $jseed \leftarrow\!\!\$\, \{0,1\}^\kappa$
  $jr \leftarrow\!\!\$\, \mathbb{F}^{jl}$; $pr \leftarrow\!\!\$\, \mathbb{F}^{pl}$; $qr \leftarrow\!\!\$\, \mathbb{F}^{ql}$
  Rand$[1, jseed, \varepsilon] \leftarrow jr$
  Rand$[4, ps, \varepsilon] \leftarrow pr$
  Rand$[5, sk_z, n] \leftarrow qr$
  Rand$[6, 0^\kappa, \vec{rseed}] \leftarrow jseed$
  $\vec{\pi}[1] \leftarrow \mathsf{Prove}(inp, jr\,; pr)$
  $\vec{\pi}[1] \leftarrow \vec{\pi}[1] - \sum_{\hat{j}=2}^{s} \vec{\pi}[\hat{j}]$
  $\vec{x}[1] \leftarrow (\vec{x}[1], \vec{\pi}[1], blind_1)$
  for $\hat{j} \in [2..s]$:
    $\vec{x}[\hat{j}] \leftarrow (xseed_{\hat{j}}, pseed_{\hat{j}}, blind_{\hat{j}})$
  Pub$[\hat{k}] \leftarrow \vec{rseed}$
  In$[\hat{k}, \cdot] \leftarrow \vec{x}$
  In$[\hat{k}, z] \leftarrow (\vec{x}[z], jseed, jr, qr)$
  Used$[\hat{k}] \leftarrow (n, m_0, m_1)$
  ret $(n, \text{Pub}[\hat{k}], (\text{In}[\hat{k}, \hat{j}])_{\hat{j} \in T})$
```

```
Shard($\hat{k} \in \mathbb{N}, m_0, m_1 \in \mathcal{I}$):                Game G6 $\boxed{\text{G7}}$
  if Used$[\hat{k}] \neq \bot$: ret $\bot$
  $n \leftarrow\!\!\$\, N \setminus N^*$; $N^* \leftarrow N^* \cup \{n\}$
  $\vec{x} \leftarrow\!\!\$\, (\mathbb{F}^n)^s$; $\vec{\pi} \leftarrow\!\!\$\, (\mathbb{F}^m)^s$
  $(blind_1, \ldots, blind_s) \leftarrow\!\!\$\, (\{0,1\}^\kappa)^s$
  $(xseed_2, \ldots, xseed_s) \leftarrow\!\!\$\, (\{0,1\}^\kappa)^{s-1}$
  $(pseed_2, \ldots, pseed_s) \leftarrow\!\!\$\, (\{0,1\}^\kappa)^{s-1}$
  $\vec{rseed} \leftarrow\!\!\$\, (\{0,1\}^\kappa)^s$; $jr \leftarrow\!\!\$\, \mathbb{F}^{jl}$; $pr \leftarrow\!\!\$\, \mathbb{F}^{pl}$
  $inp \leftarrow \mathsf{Encode}(m_b)$
  $\pi \leftarrow \mathsf{Prove}(inp, jr\,; pr)$
  $\vec{x}[z] \leftarrow inp - \sum_{\hat{j} \in T} \vec{x}[\hat{j}]$
  $\vec{\pi}[z] \leftarrow \pi - \sum_{\hat{j} \in T} \vec{\pi}[\hat{j}]$
  for $\hat{j} \in [s]$:
    Rand$[2, xseed_{\hat{j}}, \hat{j}] \leftarrow \vec{x}[\hat{j}]$
    Rand$[3, pseed_{\hat{j}}, \hat{j}] \leftarrow \vec{\pi}[\hat{j}]$
    Rand$[7, blind_{\hat{j}}, \hat{j} \,\|\, n \,\|\, \vec{x}[\hat{j}]] \leftarrow \vec{rseed}[\hat{j}]$
  Rand$[7, blind_1, 1 \,\|\, n \,\|\, \vec{x}[1]] \leftarrow \vec{rseed}[1]$
  $ps \leftarrow\!\!\$\, \{0,1\}^\kappa$; $jseed \leftarrow\!\!\$\, \{0,1\}^\kappa$; $qr \leftarrow\!\!\$\, \mathbb{F}^{ql}$
  Rand$[1, jseed, \varepsilon] \leftarrow jr$; Rand$[4, ps, \varepsilon] \leftarrow pr$
  Rand$[5, sk_z, n] \leftarrow qr$; Rand$[6, 0^\kappa, \vec{rseed}] \leftarrow jseed$
  $\vec{x}[1] \leftarrow (\vec{x}[1], \vec{\pi}[1], blind_1)$
  for $\hat{j} \in [2..s]$: $\vec{x}[\hat{j}] \leftarrow (xseed_{\hat{j}}, pseed_{\hat{j}}, blind_{\hat{j}})$
  Pub$[\hat{k}] \leftarrow \vec{rseed}$; In$[\hat{k}, \cdot] \leftarrow \vec{x}$
  In$[\hat{k}, z] \leftarrow (\vec{x}[z], jseed, jr, qr)$
  $vfs \leftarrow \mathsf{Query}(\vec{x}[z], \vec{\pi}[z], jr; qr)$
  In$[\hat{k}, \hat{j}] \leftarrow (vfs, \vec{rseed}[z], \mathsf{Truncate}(\vec{x}[z]), jseed)$
  Used$[\hat{k}] \leftarrow (n, m_0, m_1)$
  ret $(n, \text{Pub}[\hat{k}], (\text{In}[\hat{k}, \hat{j}])_{\hat{j} \in T})$

Prep($\hat{i} \in \mathbb{N}, \hat{j} \in \{z\}, \hat{k} \in \mathbb{N}, \vec{M} \in \mathcal{M}^*$):
  if Status$[\hat{i}, \hat{j}] \neq$ running or In$[\hat{k}, \hat{j}] = \bot$: ret $\bot$
  if St$[\hat{i}, \hat{j}, \hat{k}] = \bot$:
    St$[\hat{i}, \hat{j}, \hat{k}] \leftarrow$ Setup$[\hat{i}, \hat{j}]$; $\vec{M} \leftarrow (\text{Pub}[\hat{k}],)$
  $(n, m_0, m_1) \leftarrow$ Used$[\hat{k}]$
  if St$[\hat{i}, \hat{j}, \hat{k}] = \varepsilon$:  // Process initial message from client
    $(x, jseed, jr, qr) \leftarrow$ In$[\hat{k}, \hat{j}]$
    $(inp, \pi, blind) \leftarrow \mathsf{Unpack}(\hat{j}, x)$; $(\vec{rseed},) \leftarrow \vec{M}$
    $\vec{rseed}[\hat{j}] \leftarrow \text{RO}_7(blind, \hat{j} \,\|\, n \,\|\, inp)$
    $M \leftarrow (\mathsf{Query}(inp, \pi, jr; qr), \vec{rseed}[\hat{j}])$
    St$[\hat{i}, \hat{j}, \hat{k}] \leftarrow (jseed, \mathsf{Truncate}(inp))$
    $(vfs, rseed, y, jseed) \leftarrow$ In$[\hat{k}, \hat{j}]$
    $M \leftarrow (vfs, rseed)$; St$[\hat{i}, \hat{j}, \hat{k}] \leftarrow (jseed, y)$
    ret (running, $M$)
  // Process broadcast messages from aggregators
  $(jseed, y) \leftarrow$ St$[\hat{i}, \hat{j}, \hat{k}]$; $(\vec{vfs}[\hat{j}], \vec{rseed}[\hat{j}])_{\hat{j} \in [s]} \leftarrow \vec{M}$
  $acc \leftarrow \mathsf{Decide}(\sum_{\hat{j}=1}^{s} \vec{vfs}[\hat{j}])$; St$[\hat{i}, \hat{j}, \hat{k}] \leftarrow \bot$
  if $acc = 0$ or $jseed \neq \text{RO}_6(0^\kappa, \vec{rseed})$: ret (failed, $\bot$)
  Out$[\hat{i}, \hat{j}, \hat{k}] \leftarrow y$
  Batch$_0[\hat{i}, \hat{j}, \hat{k}] \leftarrow m_0$; Batch$_1[\hat{i}, \hat{j}, \hat{k}] \leftarrow m_1$
  ret (finished, $\bot$)
```

Figure 5.22. Game G6 (left) and game G7 (right) for the proof of Theorem 19.

```
$\underline{\mathsf{Shard}}(\hat{k} \in \mathbb{N}, m_0, m_1 \in \mathcal{I})$:    Game G7 $\boxed{\text{G8}}$
  if $\text{Used}[\hat{k}] \neq \bot$: ret $\bot$
  $n \leftarrow\!\!\$\, N \setminus N^*$; $N^* \leftarrow N^* \cup \{n\}$
  $\vec{x} \leftarrow\!\!\$\, (\mathbb{F}^n)^s$; $\vec{\pi} \leftarrow\!\!\$\, (\mathbb{F}^m)^s$
  $(blind_1, \dots, blind_s) \leftarrow\!\!\$\, (\{0,1\}^\kappa)^s$
  $(xseed_2, \dots, xseed_s) \leftarrow\!\!\$\, (\{0,1\}^\kappa)^{s-1}$
  $(pseed_2, \dots, pseed_s) \leftarrow\!\!\$\, (\{0,1\}^\kappa)^{s-1}$
  $\vec{rseed} \leftarrow\!\!\$\, (\{0,1\}^\kappa)^s$; $jr \leftarrow\!\!\$\, \mathbb{F}^{jl}$; $pr \leftarrow\!\!\$\, \mathbb{F}^{pl}$
  $inp \leftarrow \mathsf{Encode}(m_b)$
  $\pi \leftarrow \mathsf{Prove}(inp, jr; pr)$
  $\boxed{jr \,\|\, \|\, qr \,\|\, \sigma \leftarrow\!\!\$\, \mathsf{View}_{\mathsf{FLP}}(inp)}$
  $\vec{x}[z] \leftarrow inp - \sum_{\hat{j} \in T} \vec{x}[\hat{j}]$
  $\vec{\pi}[z] \leftarrow \pi - \sum_{\hat{j} \in T} \vec{\pi}[\hat{j}]$
  for $\hat{j} \in [s]$:
    $\text{Rand}[2, xseed_{\hat{j}}, \hat{j}] \leftarrow \vec{x}[\hat{j}]$
    $\text{Rand}[3, pseed_{\hat{j}}, \hat{j}] \leftarrow \vec{\pi}[\hat{j}]$
    $\text{Rand}[7, blind_{\hat{j}}, \hat{j} \,\|\, n \,\|\, \vec{x}[\hat{j}]] \leftarrow \vec{rseed}[\hat{j}]$
  $\text{Rand}[7, blind_1, 1 \,\|\, n \,\|\, \vec{x}[1]] \leftarrow \vec{rseed}[1]$
  $ps \leftarrow\!\!\$\, \{0,1\}^\kappa$; $jseed \leftarrow\!\!\$\, \{0,1\}^\kappa$; $qr \leftarrow\!\!\$\, \mathbb{F}^{ql}$
  $\text{Rand}[1, jseed, \varepsilon] \leftarrow jr$; $\text{Rand}[4, ps, \varepsilon] \leftarrow pr$
  $\text{Rand}[5, sk_z, n] \leftarrow qr$; $\text{Rand}[6, 0^\kappa, \vec{rseed}] \leftarrow jseed$
  $\vec{x}[1] \leftarrow (\vec{x}[1], \vec{\pi}[1], blind_1)$
  for $\hat{j} \in [2..s]$: $\vec{x}[\hat{j}] \leftarrow (xseed_{\hat{j}}, pseed_{\hat{j}}, blind_{\hat{j}})$
  $\text{Pub}[\hat{k}] \leftarrow \vec{rseed}$; $\text{In}[\hat{k}, \cdot] \leftarrow \vec{x}$
  $vfs \leftarrow \mathsf{Query}(\vec{x}[z], \vec{\pi}[z], jr; qr)$
  $\boxed{vfs \leftarrow \sigma - \sum_{\hat{j} \in T} \mathsf{Query}(\vec{x}[\hat{j}], \vec{\pi}[\hat{j}], jr; qr)}$
  $\text{In}[\hat{k}, \hat{j}] \leftarrow (vfs, \vec{rseed}[z], \mathsf{Truncate}(\vec{x}[z]), jseed)$
  $\text{Used}[\hat{k}] \leftarrow (n, m_0, m_1)$
  ret $(n, \text{Pub}[\hat{k}], (\text{In}[\hat{k}, \hat{j}])_{\hat{j} \in T})$
```

```
$\underline{\mathsf{Shard}}(\hat{k} \in \mathbb{N}, m_0, m_1 \in \mathcal{I})$:    Game G8 $\boxed{\text{G9}^i}$
  if $\text{Used}[\hat{k}] \neq \bot$: ret $\bot$
  $n \leftarrow\!\!\$\, N \setminus N^*$; $N^* \leftarrow N^* \cup \{n\}$
  $\vec{x} \leftarrow\!\!\$\, (\mathbb{F}^n)^s$; $\vec{\pi} \leftarrow\!\!\$\, (\mathbb{F}^m)^s$
  $(blind_1, \dots, blind_s) \leftarrow\!\!\$\, (\{0,1\}^\kappa)^s$
  $(xseed_2, \dots, xseed_s) \leftarrow\!\!\$\, (\{0,1\}^\kappa)^{s-1}$
  $(pseed_2, \dots, pseed_s) \leftarrow\!\!\$\, (\{0,1\}^\kappa)^{s-1}$
  $\vec{rseed} \leftarrow\!\!\$\, (\{0,1\}^\kappa)^s$
  $inp \leftarrow \mathsf{Encode}(m_b)$
  $jr \,\|\, \|\, qr \,\|\, \sigma \leftarrow\!\!\$\, \mathsf{View}_{\mathsf{FLP}}(inp)$
  $\boxed{ctr \leftarrow ctr + 1}$
  $\boxed{\text{if } ctr < i\text{: } jr \,\|\, \|\, qr \,\|\, \sigma \leftarrow\!\!\$\, \mathsf{Sim}()}$
  $\boxed{\text{else: } jr \,\|\, \|\, qr \,\|\, \sigma \leftarrow\!\!\$\, \mathsf{View}_{\mathsf{FLP}}(inp)}$
  $\vec{x}[z] \leftarrow inp - \sum_{\hat{j} \in T} \vec{x}[\hat{j}]$
  for $\hat{j} \in [s]$:
    $\text{Rand}[2, xseed_{\hat{j}}, \hat{j}] \leftarrow \vec{x}[\hat{j}]$
    $\text{Rand}[3, pseed_{\hat{j}}, \hat{j}] \leftarrow \vec{\pi}[\hat{j}]$
    $\text{Rand}[7, blind_{\hat{j}}, \hat{j} \,\|\, n \,\|\, \vec{x}[\hat{j}]] \leftarrow \vec{rseed}[\hat{j}]$
  $\text{Rand}[7, blind_1, 1 \,\|\, n \,\|\, \vec{x}[1]] \leftarrow \vec{rseed}[1]$
  $jseed \leftarrow\!\!\$\, \{0,1\}^\kappa$
  $\text{Rand}[1, jseed, \varepsilon] \leftarrow jr$
  $\text{Rand}[5, sk_z, n] \leftarrow qr$; $\text{Rand}[6, 0^\kappa, \vec{rseed}] \leftarrow jseed$
  $\vec{x}[1] \leftarrow (\vec{x}[1], \vec{\pi}[1], blind_1)$
  for $\hat{j} \in [2..s]$: $\vec{x}[\hat{j}] \leftarrow (xseed_{\hat{j}}, pseed_{\hat{j}}, blind_{\hat{j}})$
  $\text{Pub}[\hat{k}] \leftarrow \vec{rseed}$; $\text{In}[\hat{k}, \cdot] \leftarrow \vec{x}$
  $vfs \leftarrow \sigma -$
        $\sum_{\hat{j} \in T} \mathsf{Query}(\vec{x}[\hat{j}], \vec{\pi}[\hat{j}], jr; qr)$
  $\text{In}[\hat{k}, \hat{j}] \leftarrow (vfs, \vec{rseed}[z], \mathsf{Truncate}(\vec{x}[z]), jseed)$
  $\text{Used}[\hat{k}] \leftarrow (n, m_0, m_1)$
  ret $(n, \text{Pub}[\hat{k}], (\text{In}[\hat{k}, \hat{j}])_{\hat{j} \in T})$
```

Figure 5.23. Game G8 (left) and game G9 for the proof of Theorem 19.

that

$$\Pr[\text{G7}(\mathscr{B})] \leq \Pr[\text{G8}(\mathscr{B})] + \frac{q_4 q_{\mathsf{Shard}}}{2^\kappa}\,. \tag{5.20}$$

Let Sim be the simulator hypothesized by δ-privacy of FLP. In the right panel of Figure 5.23 we define a series of hybrid games that replace $\mathsf{View}_{\mathsf{FLP}}$ with a simulator S for the privacy of FLP. Recall from Section 5.2 that Sim outputs a string $jr \,\|\, qr \,\|\, \sigma$. In $\text{G9}^i(\mathscr{B})$, the first $i-1$ queries to $\underline{\mathsf{Shard}}$ generate jr, qr, σ by calling $\mathsf{Sim}()$; the remaining queries call $\mathsf{View}_{\mathsf{FLP}}$

instead. This means that $\mathrm{G9}^1$ is identical to G8, so

$$\Pr\big[\,\mathrm{G8}(\mathscr{B})\,\big] = \Pr\big[\,\mathrm{G9}^1(\mathscr{B})\,\big]\,. \tag{5.21}$$

For every $v \in \mathbb{F}^{jl \times ql \times v}$, we let $p_{i,v}$ denote the probability that $\mathscr{B}$ wins hybrid $\mathrm{G9}^i$, conditioned on the event that the i^{th} query to $\underline{\mathsf{Shard}}$ sets $v = jr \,\|\, qr \,\|\, \sigma$. A union bound over all v shows that

$$\Pr[\mathrm{G9}^i(\mathscr{B})] = \sum_{v \in \mathbb{F}^{jl \times ql \times v}} p_{i,v}\,. \tag{5.22}$$

We are now ready to bound the quantity $\Pr[\mathrm{G9}^{i+1}(\mathscr{B})] - \Pr[\mathrm{G9}^i(\mathscr{B})]$. The two games $\mathrm{G9}^{i+1}$ and $\mathrm{G9}^i$ differ only in the tuple v chosen by of the $(i+1)^{\text{th}}$ query to $\underline{\mathsf{Shard}}$: the former calls $\mathsf{View}_{\mathsf{FLP}}$ and the latter calls Sim. We therefore decompose both probabilities over the possible choices of v, and substitute in the statement

$$\sum_{v \in \mathbb{F}^{jl \times ql \times v}} \Big|\Pr\big[\,\mathsf{View}_{\mathsf{FLP}}(\mathit{inp}) = v\,\big] - \Pr\big[\,\mathsf{Sim}(\,) = v\,\big]\Big| \leq \delta$$

that follows from the δ-privacy of FLP for all inp. Since $p_{i,v} \leq 1$ for all i and v and $\Pr[\mathsf{View}_{\mathsf{FLP}}(\mathit{inp}) = v] - \Pr[\mathsf{Sim}(\,) = v] \leq |\Pr[\mathsf{View}_{\mathsf{FLP}}(\mathit{inp}) = v] - \Pr[\mathsf{Sim}(\,) = v]|$:

$$\begin{aligned}
\Pr[\mathrm{G9}^{i+1}(\mathscr{B})] - \Pr[\mathrm{G9}^i(\mathscr{B})] = {} & \\
& \sum_v p_{i,v} \cdot \Pr[\mathsf{View}_{\mathsf{FLP}}(\mathit{inp}) = v] - p_{i,v} \cdot \Pr[\mathsf{Sim}(\,) = v] \\
& = \sum_v p_{i,v} \cdot (\Pr[\mathsf{View}_{\mathsf{FLP}}(\mathit{inp}) = v] - \Pr[\mathsf{Sim}(\,) = v]) \\
& \leq \sum_v |\Pr[\mathsf{View}_{\mathsf{FLP}}(\mathit{inp}) = v] - \Pr[\mathsf{Sim}(\,) = v]| \\
& \leq \delta\,.
\end{aligned}$$

A union bound over all $i \in [q_{\underline{\mathsf{Shard}}}]$ produces the final inequality:

$$\Pr[\mathrm{G9}^{q_{\mathsf{Shard}}}(\mathscr{B})] - \Pr[\mathrm{G9}^1(\mathscr{B})] \leq \delta \cdot q_{\underline{\mathsf{Shard}}}\,. \tag{5.23}$$

Finally, we observe that game $\mathrm{G9}^{q_{\mathsf{Shard}}}$ can now be rewritten so that the outcome is independent of the challenge bit b. Hence

$$\Pr[\mathrm{G9}^{q_{\mathsf{Shard}}}(\mathscr{B})] = \frac{1}{2}\,. \tag{5.24}$$

Collecting bounds across all games and simplifying yields the theorem.

5.9.3 Doplar Robustness (Theorem 20)

The proof is by a game-playing argument. We begin with the game G0 defined in Figure 5.24 played by the given adversary $\mathscr{A}$. This game was constructed from $\mathsf{Exp}^{\mathrm{robust}}_{\Pi}(\mathscr{A})$ by applying the following revisions. First, we have replaced Prep with its implementation, rolled out the loops in the $\underline{\mathsf{Prep}}$ oracle, and simplified some of the control flow. Second, we have removed the call to $\mathsf{refineFromShares}$ and set the purported refined measurement with the sum of the refined shares output by the calls to $\mathsf{VIDPF.VEval}$. (This is equivalent by refinement consistency of Π.) Third, we use the fact that the allowed-state $\mathsf{validSt}$ algorithm for Π only permits $\underline{\mathsf{Prep}}$ queries with unique (n,ℓ) pairs to make the contents of table Used more explicit. Finally, we lazy-evaluate each random oracle, denoted RO_i, with a table Rand. We use RO' to denote the random oracle for VIDPF. By construction we have that

$$\mathbf{Adv}\mathrm{robust}_{\Pi}(\mathscr{A}) = \mathsf{Pr}\left[\,\mathrm{G0}(\mathscr{A})\,\right]. \tag{5.25}$$

In the remainder, we let q_i denote the number queries $\mathscr{A}$ makes to RO_i and q_i denote the number of queries $\mathscr{A}$ makes to RO'; note that $q_{\mathsf{RG}} = q_1 + \cdots + q_6 + q'$.

Similar to the proof of Theorem 18, note that we have dropped the winning condition on line 16 of the robustness game (Figure 5.3). The refined measurement computed from the input shares is equal to $\Pi.\mathsf{Unshard}(1,(\Pi.\mathsf{Agg}(\vec{y}_1),\Pi.\mathsf{Agg}(\vec{y}_2))) = \vec{y}_1 + \vec{y}_2$, so this condition is never met by definition.

Next, in game G1 (left panel of Figure 5.24) we revise the definition of the RO oracle so that for each $i \in \{5,6\}$, the values of $\mathrm{Rand}[i, \mathit{seed}, \mathit{cntxt}]$ are sampled without replacement. The new game is identical to G0 up to a collision in the output for either $\mathrm{Rand}[5,\cdot,\cdot]$ or $\mathrm{Rand}[6,\cdot,\cdot]$.

```
Game G0($\mathscr{A}$) $\boxed{\text{G1}(\mathscr{A})}$ :
$sk \leftarrow\!\!\$\ \mathscr{SK}$; win $\leftarrow$ false; $\mathscr{A}^{\mathrm{RO},\underline{\mathsf{Prep}}}()$; ret $w$

$\underline{\mathsf{Prep}}(n, \vec{x}, msg_{\mathrm{Init}}, st_{\mathrm{Init}})$:
  $(\ell, \vec{pfx}) \leftarrow state$; $u \leftarrow |\vec{pfx}|$
  if $\mathrm{Used}[n,\ell] \neq \bot$: ret $\bot$
  $\mathrm{Used}[n,\ell] \leftarrow \top$
  $(pub, \vec{rseed}) \leftarrow msg_{\mathrm{Init}}$
  $(key_1, seed_1, \pi_1) \leftarrow \mathsf{Unpack}(1, \vec{x}[1], n, \ell)$
  $(key_2, seed_2, \pi_2) \leftarrow \mathsf{Unpack}(2, \vec{x}[2], n, \ell)$
  $\Delta_1 \leftarrow \mathrm{RO}_2(seed_1, n \,\|\, \ell \,\|\, 1)$
  $\Delta_2 \leftarrow \mathrm{RO}_2(seed_2, n \,\|\, \ell \,\|\, 2)$
  $\rho_1 \leftarrow \mathrm{RO}_5(seed_1, n \,\|\, 1 \,\|\, pub \,\|\, key_1)$
  $\rho_2 \leftarrow \mathrm{RO}_5(seed_2, n \,\|\, 2 \,\|\, pub \,\|\, key_2)$
  $jseed_1 \leftarrow \mathrm{RO}_6(0^\kappa, \ell \,\|\, \rho_1 \,\|\, \vec{rseed}[2])$
  $jseed_2 \leftarrow \mathrm{RO}_6(0^\kappa, \ell \,\|\, \vec{rseed}[1] \,\|\, \rho_2)$
  $jr_1 \leftarrow \mathrm{RO}_1(jseed_1, n \,\|\, \ell)$
  $jr_2 \leftarrow \mathrm{RO}_1(jseed_2, n \,\|\, \ell)$
  $qr \leftarrow \mathrm{RO}_4(sk, n \,\|\, \ell \,\|\,)$
  $(h_1, \vec{y}_1) \leftarrow \mathsf{VIDPF.VEval}^{\mathrm{RO}'}(1, pub, key_1, \vec{pfx})$
  $(h_2, \vec{y}_2) \leftarrow \mathsf{VIDPF.VEval}^{\mathrm{RO}'}(2, pub, key_2, \vec{pfx})$
  $\vec{y} \leftarrow \vec{y}_1 + \vec{y}_2$
  $inp_1 \leftarrow \sum_{i \in [u]} \vec{y}_1[i]$
  $inp_2 \leftarrow \sum_{i \in [u]} \vec{y}_2[i]$
  $\sigma_1 \leftarrow \mathsf{DFLP.Query}(inp_1, \Delta_1, \pi_1, jr_1;\ qr)$
  $\sigma_2 \leftarrow \mathsf{DFLP.Query}(inp_2, \Delta_2, \pi_2, jr_2;\ qr)$
  $\overline{jseed} \leftarrow \mathrm{RO}_6(0^\kappa, \ell \,\|\, \rho_1 \,\|\, \rho_2)$
  $b_1 \leftarrow jseed_1 = \overline{jseed}$
  $b_2 \leftarrow jseed_2 = \overline{jseed}$
  $v \leftarrow \mathsf{VIDPF.Verify}^{\mathrm{RO}'}(h_1, h_2)$
  $d \leftarrow \mathsf{DFLP.Decide}(\sigma_1 + \sigma_2)$
  if $\vec{y} \notin \mathscr{V}_{st_{\mathrm{Init}}}$
      and $(b_1 \wedge v \wedge d)$ or $(b_2 \wedge v \wedge d)$: win $\leftarrow$ true
  ret (win, $(msg_{\mathrm{Init}}, ((\sigma_1, \rho_1, h_1), (\sigma_2, \rho_2, h_2))))$

$\mathrm{RO}_i(seed, cntxt)$:
  $l \leftarrow (jl, el, m, ql)$
  if $\mathrm{Rand}[i, seed, cntxt] = \bot$:
    if $i \leq 4$: $\mathrm{Rand}[i, seed, cntxt] \leftarrow\!\!\$\ \mathbb{F}^{l[i]}$
    else: $\mathrm{Rand}[i, seed, cntxt] \leftarrow\!\!\$\ \{0,1\}^\kappa$
      $\boxed{out \leftarrow\!\!\$\ \{0,1\}^\kappa \setminus \mathsf{Q}_i;\ \mathsf{Q}_i \leftarrow \mathsf{Q}_i \cup \{out\}}$
      $\boxed{\mathrm{Rand}[i, seed, cntxt] \leftarrow out}$
  ret $\mathrm{Rand}[i, seed, cntxt]$

$\mathrm{RO}'(inp)$:
  if $\mathrm{Rand}'[inp] = \bot$: $\mathrm{Rand}'[inp] \leftarrow\!\!\$\ Y$
  ret $\mathrm{Rand}'[inp]$
```

```
$\underline{\mathsf{Prep}}(n, \vec{x}, msg_{\mathrm{Init}}, st_{\mathrm{Init}})$:          G1 $\boxed{\text{G2}}$
  $(\ell, \vec{pfx}) \leftarrow state$; $u \leftarrow |\vec{pfx}|$
  if $\mathrm{Used}[n,\ell] \neq \bot$: ret $\bot$
  $\mathrm{Used}[n,\ell] \leftarrow \top$
  $(pub, \vec{rseed}) \leftarrow msg_{\mathrm{Init}}$
  $(key_1, seed_1, \pi_1) \leftarrow \mathsf{Unpack}(1, \vec{x}[1], n, \ell)$
  $(key_2, seed_2, \pi_2) \leftarrow \mathsf{Unpack}(2, \vec{x}[2], n, \ell)$
  $\Delta_1 \leftarrow \mathrm{RO}_2(seed_1, n \,\|\, \ell \,\|\, 1)$
  $\Delta_2 \leftarrow \mathrm{RO}_2(seed_2, n \,\|\, \ell \,\|\, 2)$
  $\rho_1 \leftarrow \mathrm{RO}_5(seed_1, n \,\|\, 1 \,\|\, pub \,\|\, key_1)$
  $\rho_2 \leftarrow \mathrm{RO}_5(seed_2, n \,\|\, 2 \,\|\, pub \,\|\, key_2)$
  $jseed_1 \leftarrow \mathrm{RO}_6(0^\kappa, \ell \,\|\, \rho_1 \,\|\, \vec{rseed}[2])$
  $jseed_2 \leftarrow \mathrm{RO}_6(0^\kappa, \ell \,\|\, \vec{rseed}[1] \,\|\, \rho_2)$
  $jr_1 \leftarrow \mathrm{RO}_1(jseed_1, n \,\|\, \ell)$
  $jr_2 \leftarrow \mathrm{RO}_1(jseed_2, n \,\|\, \ell)$
  $\boxed{jseed \leftarrow \mathrm{RO}_6(0^\kappa, \ell \,\|\, \rho_1 \,\|\, \rho_2)}$
  $\boxed{jr \leftarrow \mathrm{RO}_1(jseed, n \,\|\, \ell)}$
  $qr \leftarrow \mathrm{RO}_4(sk, n \,\|\, \ell \,\|\,)$
  $(h_1, \vec{y}_1) \leftarrow \mathsf{VIDPF.VEval}^{\mathrm{RO}'}(1, pub, key_1, \vec{pfx})$
  $(h_2, \vec{y}_2) \leftarrow \mathsf{VIDPF.VEval}^{\mathrm{RO}'}(2, pub, key_2, \vec{pfx})$
  $\vec{y} \leftarrow \vec{y}_1 + \vec{y}_2$
  $inp_1 \leftarrow \sum_{i \in [u]} \vec{y}_1[i]$
  $inp_2 \leftarrow \sum_{i \in [u]} \vec{y}_2[i]$
  $\sigma_1 \leftarrow \mathsf{DFLP.Query}(inp_1, \Delta_1, \pi_1,\ jr_1 \boxed{jr};\ qr)$
  $\sigma_2 \leftarrow \mathsf{DFLP.Query}(inp_2, \Delta_2, \pi_2,\ jr_2 \boxed{jr};\ qr)$
  $\overline{jseed} \leftarrow \mathrm{RO}_6(0^\kappa, \ell \,\|\, \rho_1 \,\|\, \rho_2)$
  $b_1 \leftarrow jseed_1 = \overline{jseed}$
  $b_2 \leftarrow jseed_2 = \overline{jseed}$
  $\boxed{b_1 \leftarrow \rho_1 \neq \vec{rseed}[1]}$
  $\boxed{b_2 \leftarrow \rho_2 \neq \vec{rseed}[2]}$
  $v \leftarrow \mathsf{VIDPF.Verify}^{\mathrm{RO}'}(h_1, h_2)$
  $d \leftarrow \mathsf{DFLP.Decide}(\sigma_1 + \sigma_2)$
  if $\vec{y} \notin \mathscr{V}_{st_{\mathrm{Init}}}$
      and $(b_1 \wedge v \wedge d)$ or $(b_2 \wedge v \wedge d)$: win $\leftarrow$ true
  ret (win, $(msg_{\mathrm{Init}}, ((\sigma_1, \rho_1, h_1), (\sigma_2, \rho_2, h_2))))$
```

Figure 5.24. Games G0, G1, and G2 for the proof of Theorem 20. Let Y denote the co-domain of the random oracle used by VIDPF.

Applying a birthday bound over all queries by $\mathscr{A}$ or by the Prep oracle yields

$$\Pr\big[\,\mathrm{G0}(\mathscr{A})\,\big] \le \Pr\big[\,\mathrm{G1}(\mathscr{A})\,\big] + \frac{(q_5 + 2q_{\mathsf{Prep}})^2}{2^{\kappa+1}} + \frac{(q_6 + 3q_{\mathsf{Prep}})^2}{2^{\kappa+1}}\,. \tag{5.26}$$

Next, in game G2 (right panel of Figure 5.25) we simplify the Prep oracle by substituting aggregator $\hat{j}$'s local computation of the joint randomness seed $\mathit{jseed}_{\hat{j}}$ with a direct computation of the seed jseed from the parts ρ_1, ρ_2 computed on lines 9–10. Accordingly, We simplify the joint local randomness checks (lines 26–27) to just check if the purported hint $\vec{\mathit{rseed}}[\hat{j}]$ matches the computed part $\rho_{\hat{j}}$ (28–29). This change is only detectable to the adversary if it can find a joint randomness seed and hints such that the check succeeds, but the aggregators compute distinct $\mathit{jseed}_1 \neq \mathit{jseed}_2$. This is impossible by construction (transition from G1 to G2), so

$$\Pr\big[\,\mathrm{G1}(\mathscr{A})\,\big] = \Pr\big[\,\mathrm{G2}(\mathscr{A})\,\big]\,. \tag{5.27}$$

Next, in game G3 (Figure 5.25), we make the following changes. First, we modify oracle RO_4 so that, for any query that coincides with the secret verification key sk sampled at the beginning of the game, the oracle immediately returns $\perp$ without programming the RO table. Second, we modify Prep by replacing the call to

$$\mathit{qr} \leftarrow \mathrm{RO}_4(\mathit{sk}, n \,\|\, \ell \,\|\,)$$

with

$$\mathit{qr} \leftarrow \mathrm{Rand}[4, \mathit{sk}, n \,\|\, \ell] \leftarrow\!\!\$\; \mathbb{F}^{ql}\,.$$

That way each call to Prep samples fresh query randomness. The second change does not overwrite any value in Rand due to the first change. Thus the new game is identical to G2 until the adversary makes a query to RO_4 with the seed equal to sk. Taking a union bound over all of $\mathscr{A}$'s queries, we have that

$$\Pr\big[\,\mathrm{G2}(\mathscr{A})\,\big] \le \Pr\big[\,\mathrm{G3}(\mathscr{A})\,\big] + \frac{q_4 q_{\mathsf{Prep}}}{2^{\kappa}}\,. \tag{5.28}$$

```
$\underline{\mathsf{Prep}}(n, \vec{x}, msg_{\mathrm{Init}}, st_{\mathrm{Init}})$:                    G2 $\boxed{\mathrm{G3}}$
1   $(\ell, \vec{pfx}) \leftarrow state$; $u \leftarrow |\vec{pfx}|$
2   if $\mathrm{Used}[n, \ell] \neq \bot$: ret $\bot$
3   $\mathrm{Used}[n, \ell] \leftarrow \top$
4   $(pub, \vec{rseed}) \leftarrow msg_{\mathrm{Init}}$
5   $(key_1, seed_1, \pi_1) \leftarrow \mathsf{Unpack}(1, \vec{x}[1], n, \ell)$
6   $(key_2, seed_2, \pi_2) \leftarrow \mathsf{Unpack}(2, \vec{x}[2], n, \ell)$
7   $\Delta_1 \leftarrow \mathrm{RO}_2(seed_1, n \,\|\, \ell \,\|\, 1)$
8   $\Delta_2 \leftarrow \mathrm{RO}_2(seed_2, n \,\|\, \ell \,\|\, 2)$
9   $\rho_1 \leftarrow \mathrm{RO}_5(seed_1, n \,\|\, 1 \,\|\, pub \,\|\, key_1)$
10  $\rho_2 \leftarrow \mathrm{RO}_5(seed_2, n \,\|\, 2 \,\|\, pub \,\|\, key_2)$
11  $jseed \leftarrow \mathrm{RO}_6(0^\kappa, \ell \,\|\, \rho_1 \,\|\, \rho_2)$
12  $jr \leftarrow \mathrm{RO}_1(jseed, n \,\|\, \ell)$
13  $qr \leftarrow \mathrm{RO}_4(sk, n \,\|\, \ell \,\|\, )$
14  $\boxed{qr \leftarrow \mathrm{Rand}[4, sk, n \,\|\, \ell] \leftarrow\!\!\$\, \mathbb{F}^{ql}}$
15  $(h_1, \vec{y}_1) \leftarrow \mathsf{VIDPF.VEval}^{\mathrm{RO}'}(1, pub, key_1, \vec{pfx})$
16  $(h_2, \vec{y}_2) \leftarrow \mathsf{VIDPF.VEval}^{\mathrm{RO}'}(2, pub, key_2, \vec{pfx})$
17  $\vec{y} \leftarrow \vec{y}_1 + \vec{y}_2$
18  $inp_1 \leftarrow \sum_{i \in [u]} \vec{y}_1[i]$
19  $inp_2 \leftarrow \sum_{i \in [u]} \vec{y}_2[i]$
20  $\sigma_1 \leftarrow \mathsf{DFLP.Query}(inp_1, \Delta_1, \pi_1, jr;\, qr)$
21  $\sigma_2 \leftarrow \mathsf{DFLP.Query}(inp_2, \Delta_2, \pi_2, jr;\, qr)$
22  $b_1 \leftarrow \rho_1 \neq \vec{rseed}[1]$
23  $b_2 \leftarrow \rho_2 \neq \vec{rseed}[2]$
24  $v \leftarrow \mathsf{VIDPF.Verify}^{\mathrm{RO}'}(h_1, h_2)$
25  $d \leftarrow \mathsf{DFLP.Decide}(\sigma_1 + \sigma_2)$
26  if $\vec{y} \notin \mathscr{V}_{st_{\mathrm{Init}}}$
27      and $(b_1 \wedge v \wedge d)$ or $(b_2 \wedge v \wedge d)$: $\mathsf{win} \leftarrow \mathsf{true}$
28  ret $(\mathsf{win}, (msg_{\mathrm{Init}}, ((\sigma_1, \rho_1, h_1), (\sigma_2, \rho_2, h_2))))$

$\mathrm{RO}_i(seed, cntxt)$:
29  $\boxed{\text{if } i = 4 \wedge seed = sk\text{: ret } \bot}$
30  $l \leftarrow (jl, el, m, ql)$
31  if $\mathrm{Rand}[i, seed, cntxt] = \bot$:
32    if $i \leq 4$: $\mathrm{Rand}[i, seed, cntxt] \leftarrow\!\!\$\, \mathbb{F}^{l[i]}$
33    else:
34      $out \leftarrow\!\!\$\, \{0,1\}^\kappa \setminus \mathrm{Q}_i$; $\mathrm{Q}_i \leftarrow \mathrm{Q}_i \cup \{out\}$
35      $\mathrm{Rand}[i, seed, cntxt] \leftarrow out$
36  ret $\mathrm{Rand}[i, seed, cntxt]$
```

```
$\underline{\mathsf{Prep}}(n, \vec{x}, msg_{\mathrm{Init}}, st_{\mathrm{Init}})$:                    G3 $\boxed{\mathrm{G4}}$
1   $(\ell, \vec{pfx}) \leftarrow state$; $u \leftarrow |\vec{pfx}|$
2   if $\mathrm{Used}[n, \ell] \neq \bot$: ret $\bot$
3   $\mathrm{Used}[n, \ell] \leftarrow \top$
4   $(pub, \vec{rseed}) \leftarrow msg_{\mathrm{Init}}$
5   $(key_1, seed_1, \pi_1) \leftarrow \mathsf{Unpack}(1, \vec{x}[1], n, \ell)$
6   $(key_2, seed_2, \pi_2) \leftarrow \mathsf{Unpack}(2, \vec{x}[2], n, \ell)$
7   $\boxed{\text{if } \mathrm{T}[n] = \bot\text{: } \mathrm{T}[n] \leftarrow\!\!\$\, \mathscr{E}(key_1, key_2, pub, \mathrm{Rand}')}$
8   $\Delta_1 \leftarrow \mathrm{RO}_2(seed_1, n \,\|\, \ell \,\|\, 1)$
9   $\Delta_2 \leftarrow \mathrm{RO}_2(seed_2, n \,\|\, \ell \,\|\, 2)$
10  $\rho_1 \leftarrow \mathrm{RO}_5(seed_1, n \,\|\, 1 \,\|\, pub \,\|\, key_1)$
11  $\rho_2 \leftarrow \mathrm{RO}_5(seed_2, n \,\|\, 2 \,\|\, pub \,\|\, key_2)$
12  $jseed \leftarrow \mathrm{RO}_6(0^\kappa, \ell \,\|\, \rho_1 \,\|\, \rho_2)$
13  $jr \leftarrow \mathrm{RO}_1(jseed, n \,\|\, \ell)$
14  $qr \leftarrow \mathrm{Rand}[4, sk, n \,\|\, \ell] \leftarrow\!\!\$\, \mathbb{F}^{ql}$
15  $(h_1, \vec{y}_1) \leftarrow \mathsf{VIDPF.VEval}^{\mathrm{RO}'}(1, pub, key_1, \vec{pfx})$
16  $(h_2, \vec{y}_2) \leftarrow \mathsf{VIDPF.VEval}^{\mathrm{RO}'}(2, pub, key_2, \vec{pfx})$
17  $\vec{y} \leftarrow \vec{y}_1 + \vec{y}_2$
18  $inp_1 \leftarrow \sum_{i \in [u]} \vec{y}_1[i]$
19  $inp_2 \leftarrow \sum_{i \in [u]} \vec{y}_2[i]$
20  $\sigma_1 \leftarrow \mathsf{DFLP.Query}(inp_1, \Delta_1, \pi_1, jr;\, qr)$
21  $\sigma_2 \leftarrow \mathsf{DFLP.Query}(inp_2, \Delta_2, \pi_2, jr;\, qr)$
22  $b_1 \leftarrow \rho_1 \neq \vec{rseed}[1]$
23  $b_2 \leftarrow \rho_2 \neq \vec{rseed}[2]$
24  $v \leftarrow \mathsf{VIDPF.Verify}^{\mathrm{RO}'}(h_1, h_2)$
25  $\boxed{\text{if } v = 1\text{: } (\alpha, \vec{\beta}) \leftarrow\!\!\$\, \mathrm{T}[n];\ \vec{y} \leftarrow f_{\alpha,\vec{\beta}}(\vec{pfx})}$
26  $\boxed{\text{else } \vec{y} \leftarrow \vec{y}_1 + \vec{y}_2}$
27  $d \leftarrow \mathsf{DFLP.Decide}(\sigma_1 + \sigma_2)$
28  if $\vec{y} \notin \mathscr{V}_{st_{\mathrm{Init}}}$ $\boxed{(\sum_{i \in [u]} \vec{y}[i]) \notin X}$
29      and $(b_1 \wedge v \wedge d)$ or $(b_2 \wedge v \wedge d)$: $\mathsf{win} \leftarrow \mathsf{true}$
30  ret $(\mathsf{win}, (msg_{\mathrm{Init}}, ((\sigma_1, \rho_1, h_1), (\sigma_2, \rho_2, h_2))))$
```

Figure 5.25. Games G3 and G4 for the proof of Theorem 20. Let $X = \{0,1\}$ denote the delayed-input set for DFLP.

In the last game, G4 (right-hand panel of Figure 5.25), we use the extractability of VIDPF to simplify the winning condition. First, we change how the IDPF output vector $\vec{y}$ is computed by $\underline{\mathsf{Prep}}$: If the one-hot check succeeds, i.e., v is set to 1 on line 24, then we use the extractor $\mathscr{E}$ to extract $(\alpha, \vec{\beta})$ from the transcript of the random oracle (7) and set $\vec{y}$ to $f_{\alpha,\vec{\beta}}(\vec{pfx})$. Second, we revise the winning condition (28) by requiring only that the sum of the elements of $\vec{y}$ is not in the delayed-input set $X = \{0,1\}$ for DFLP. In particular, we no longer require $\vec{y}$ to be one-hot for the adversary to win. (Recall that $\mathscr{V}_{st_{\mathrm{Init}}}$ is the set of one-hot vectors where the non-zero element is in X.) These conditions are equivalent in the revised game, since (1) $\mathscr{A}$ cannot set win if $v = 0$, and if $v = 1$, vector $\vec{y}$ is one-hot by definition.

We claim that there exists an $O(t_{\mathscr{A}} + q_{\mathsf{Prep}} t_{\mathscr{E}})$-time adversary $\mathscr{B}$ for which

$$\Pr\big[\,\mathrm{G3}(\mathscr{A})\,\big] \le \Pr\big[\,\mathrm{G4}(\mathscr{A})\,\big] + q_{\mathsf{Prep}} \cdot \mathbf{Adv}\mathrm{extract}_{\mathsf{VIDPF},\mathscr{E}}(\mathscr{B})\,. \tag{5.29}$$

The proof is by a hybrid argument. For each $i \in [q_{\mathsf{Prep}}]$ let $\mathrm{G}i'$ be the game G3 except that only the first i queries to $\underline{\mathsf{Prep}}$ are answered in the usual way; the remaining queries are answered as they are in game G4. Adversary $\mathscr{B}$ first samples $i \leftarrow\!\!\$\, [q_{\mathsf{Prep}}]$ then runs $\mathrm{G}i'(\mathscr{A})$ as usual, except that it simulates $\underline{\mathsf{Prep}}$ queries for one of the reports using its own game. Specifically, after unpacking IDPF public share pub and key shares key_1, key_2 on lines 4–6, it pauses the simulation, outputs (pub, key_1, key_2), and waits to be invoked again. On its second invocation, it resumes the simulation of the $\underline{\mathsf{Prep}}$ query until it reaches the computation of $\vec{y}$ on lines 23–26: At this point it queries its own $\underline{\mathsf{Eval}}$ oracle on the candidate prefixes $\vec{pfx}$ and sets $\vec{y}$ to the return value. Thereafter, it simulates the remainder of the game faithfully. if $\mathscr{A}$ sets $w \leftarrow \mathsf{true}$ in its game, then $\mathscr{B}$ guesses 1; otherwise it guesses 0.

Let δ_1^i (resp. δ_0^i) denote the probability that $\mathscr{B}$ samples i and guesses 1 in the VIDPF extractability experiment, conditioned on the outcome of the coin toss being 1 (resp. 0). Then for all i,

$$\mathbf{Adv}\mathrm{extract}_{\mathsf{VIDPF},\mathscr{E}}(\mathscr{A}) \ge \frac{1}{q_{\mathsf{Prep}}}\left(\delta_1^i - \delta_0^i\right). \tag{5.30}$$

Moreover, by construction we have that

$$\delta_1^i - \delta_0^i = \Pr\big[\,\mathrm{G}i'(\mathscr{A})\,\big] - \Pr\big[\,\mathrm{G}i+1'(\mathscr{A})\,\big]\,. \tag{5.31}$$

for all i. The claim follows from the observation that $\Pr\big[\,\mathrm{G3}(\mathscr{A})\,\big] = \Pr\big[\,\mathrm{G0}'(\mathscr{A})\,\big]$ and $\Pr\big[\,\mathrm{G4}(\mathscr{A})\,\big] = \Pr\big[\,\mathrm{G}q_{\mathsf{Prep}}{}'(\mathscr{A})\,\big]$.

```
Adversary $\mathscr{P}^*[\mathscr{A}]()$:
$i^* \leftarrow\!\!\$\, [q_1 + q_{\mathsf{Prep}}]$; $n^*, \ell^* \leftarrow \bot$; $ctr \leftarrow 0$
$sk \leftarrow\!\!\$\, \mathscr{SK}$; win $\leftarrow$ false; $\mathscr{A}^{\textsc{ROExt},\mathsf{PrepSim}}()$

$\textsc{ROExt}_i(seed, cntxt)$:
if $i = 4 \wedge seed = sk$: ret $\bot$
$l \leftarrow (jl, el, m, ql)$
if $\mathrm{Rand}[i, seed, cntxt] = \bot$:
  if $i = 1$:
    $ctr \leftarrow ctr + 1$
    if $i = i^* \wedge$
      if $(\exists n, \ell, pub, key_1, key_2, \rho_1, \rho_2, seed_1, seed_2)$
          $\wedge\ \rho_1 = \mathrm{Rand}[5, seed_1, n \,\|\, 1 \,\|\, pub \,\|\, key_1]$
          $\wedge\ \rho_2 = \mathrm{Rand}[5, seed_2, n \,\|\, 2 \,\|\, pub \,\|\, key_2]$
          $\wedge\ seed = \mathrm{Rand}[6, 0^\kappa, \ell \,\|\, \rho_1, \,\|\, \rho_2]$:
        $(n^*, \ell^*) \leftarrow (n, \ell)$
        // We don't know $\vec{pfx}$, so guess what the sum will
be!
        $inp^* \leftarrow\!\!\$\, \{0,1\}$
        $\Delta_1 \leftarrow \textsc{ROExt}_2(seed_1, n \,\|\, \ell \,\|\, 1)$
        $\Delta_2 \leftarrow \textsc{ROExt}_2(seed_2, n \,\|\, \ell \,\|\, 2)$
        $\Delta \leftarrow \Delta_1 + \Delta_2$
        $e \leftarrow \mathsf{DFLP}.\mathtt{Encode}(\Delta, inp^*)$
        output $(e, \Delta)$ and wait for $jr$.
        $\mathrm{Rand}[1, seed, cntxt] \leftarrow jr$
    else: $\mathrm{Rand}[1, seed, cntxt] \leftarrow\!\!\$\, \mathbb{F}^{jl}$
  else if $i \in \{2,3,4\}$: $\mathrm{Rand}[i, seed, cntxt] \leftarrow\!\!\$\, \mathbb{F}^{l[i]}$
  else:
    $out \leftarrow\!\!\$\, \{0,1\}^\kappa \setminus \mathsf{Q}_i$; $\mathsf{Q}_i \leftarrow \mathsf{Q}_i \cup \{out\}$
    $\mathrm{Rand}[i, seed, cntxt] \leftarrow out$
ret $\mathrm{Rand}[i, seed, cntxt]$

$\textsc{ROExt}'(inp)$:
if $\mathrm{Rand}'[inp] = \bot$: $\mathrm{Rand}'[inp] \leftarrow\!\!\$\, Y$
ret $\mathrm{Rand}'[inp]$
```

```
$\mathsf{PrepSim}(n, \vec{x}, msg_{\mathrm{Init}}, st_{\mathrm{Init}})$:
$(\ell, \vec{pfx}) \leftarrow state$; $u \leftarrow |\vec{pfx}|$
if $\mathrm{Used}[n, \ell] \neq \bot$: ret $\bot$
$\mathrm{Used}[n, \ell] \leftarrow \top$
$(pub, \vec{rseed}) \leftarrow msg_{\mathrm{Init}}$
$(key_1, seed_1, \pi_1) \leftarrow \mathsf{Unpack}(1, \vec{x}[1], n, \ell)$
$(key_2, seed_2, \pi_2) \leftarrow \mathsf{Unpack}(2, \vec{x}[2], n, \ell)$
if $(n^*, \ell^*) = (n, \ell)$: output $\pi_1 + \pi_2$ and halt.
if $\mathrm{T}[n] = \bot$: $\mathrm{T}[n] \leftarrow\!\!\$\, \mathscr{E}(key_1, key_2, pub, \mathrm{Rand}')$
$\Delta_1 \leftarrow \mathrm{RO}_2(seed_1, n \,\|\, \ell \,\|\, 1)$
$\Delta_2 \leftarrow \mathrm{RO}_2(seed_2, n \,\|\, \ell \,\|\, 2)$
$\rho_1 \leftarrow \mathrm{RO}_5(seed_1, n \,\|\, 1 \,\|\, pub \,\|\, key_1)$
$\rho_2 \leftarrow \mathrm{RO}_5(seed_2, n \,\|\, 2 \,\|\, pub \,\|\, key_2)$
$jseed \leftarrow \mathrm{RO}_6(0^\kappa, \ell \,\|\, \rho_1, \,\|\, \rho_2)$
$jr \leftarrow \mathrm{RO}_1(jseed, n \,\|\, \ell)$
$qr \leftarrow \mathrm{Rand}[4, sk, n \,\|\, \ell] \leftarrow\!\!\$\, \mathbb{F}^{ql}$
$(h_1, \vec{y}_1) \leftarrow \mathsf{VIDPF.VEval}^{\mathrm{RO}'}(1, pub, key_1, \vec{pfx})$
$(h_2, \vec{y}_2) \leftarrow \mathsf{VIDPF.VEval}^{\mathrm{RO}'}(2, pub, key_2, \vec{pfx})$
$inp_1 \leftarrow \sum_{i \in [u]} \vec{y}_1[i]$
$inp_2 \leftarrow \sum_{i \in [u]} \vec{y}_2[i]$
$\sigma_1 \leftarrow \mathsf{DFLP.Query}(inp_1, \Delta_1, \pi_1, jr;\ qr)$
$\sigma_2 \leftarrow \mathsf{DFLP.Query}(inp_2, \Delta_2, \pi_2, jr;\ qr)$
$b_1 \leftarrow \rho_1 \neq \vec{rseed}[1]$
$b_2 \leftarrow \rho_2 \neq \vec{rseed}[2]$
$v \leftarrow \mathsf{VIDPF.Verify}^{\mathrm{RO}'}(h_1, h_2)$
if $v = 1$: $(\alpha, \vec{\beta}) \leftarrow\!\!\$\, \mathrm{T}[n]$; $\vec{y} \leftarrow f_{\alpha, \vec{\beta}}(\vec{pfx})$
else $\vec{y} \leftarrow \vec{y}_1 + \vec{y}_2$
$d \leftarrow \mathsf{DFLP.Decide}(\sigma_1 + \sigma_2)$
if $\left(\sum_{i \in [u]} \vec{y}[i]\right) \notin X$
    and $(b_1 \wedge v \wedge d)$ or $(b_2 \wedge v \wedge d)$: win $\leftarrow$ true
ret $(\mathsf{win}, (msg_{\mathrm{Init}}, ((\sigma_1, \rho_1, h_1), (\sigma_2, \rho_2, h_2))))$
```

Figure 5.26. Malicious prover P^* against the soundness of DFLP for the proof of Theorem 20.

Consider what $\mathscr{A}$ must do to set win $\leftarrow$ true in game G4. For some Prep query, the delayed-input proof check must succeed when in fact the sum $\sum_{i \in [u]} \vec{y}[i]$ is not a valid encoded input. We bound $\mathscr{A}$'s advantage in game G4 by a reduction to the soundness of DFLP. Recall from the definition of soundness in Section 5.5.2 that the malicious prover P^* first commits to an encoded input (e, Δ), then gets a fresh joint randomness jr, then picks a proof forgery π. It wins if $\mathsf{DFLP.Decode}(e) \notin \mathscr{L}$ but the verifier deems the input valid (i.e., $\mathsf{DFLP.Decide}(\mathsf{DFLP.Query}(e, \Delta, \pi, jr;\ qr)) = 1$, where qr is a fresh query randomness sampled by the game).

Consider the malicious prover P^* in Figure 5.26. The basic idea is that P^* simulates $\mathrm{G4}(\mathscr{A})$ and extracts its commitment from queries to the random oracle. Specifically, the prover samples $i^* \leftarrow\!\!\$\, [q_1 + q_{\mathsf{Prep}}]$ at the beginning of the game, and for the i^*-th query to RO_1, it attempts to compute (e, Δ) as follows (see lines 15–19).

The prover maintains a reverse look-up table for random oracle queries for computing the query randomness (i.e., RO_4), the joint randomness seed parts (RO_5), and the joint randomness seed (RO_5). On the i^*-th query, it looks for values n, ℓ, pub, key_1, key_2, $seed_1$, and $seed_2$ that would be used by a query to $\underline{\mathsf{Prep}}$. If successful, it uses these to construct its encoded input $(\mathsf{DFLP}.\mathtt{Encode}(\Delta, inp^*), \Delta)$ to output in its game (20). It computes Δ as the sum of the $\Delta_{\hat{j}}$'s corresponding to that query (16–17). So how does it compute inp^*? Well, in G4, the $\underline{\mathsf{Prep}}$ query corresponding to i^* evaluates IDPF keys shares at a set of candidate prefixes $\vec{pfx}$ chosen by the adversary. But because $\vec{pfx}$ is not known at this point, the best it can do is guess. It therefore chooses inp^* by sampling uniform randomly from the set $X = \{0,1\}$ of delayed-input values.

If extraction of the commitment is successful, then the prover outputs it, awaits the response from its game, and programs the table with the response jr (21). Thereafter, prover P^* runs $\mathrm{G5}(\mathscr{A})$ as usual until a $\underline{\mathsf{Prep}}$ query is made for the session (n^*, ℓ^*) that coincides with the distinguished RO_1 query i^*. At this point, the prover cannot compute the decision bit d and the verifier shares σ_1, σ_2 consistently, as it does not have access to the query randomness sampled by its game. Instead, it simply halts and outputs $\pi_1 + \pi_2$ as its proof forgery (35–37).

Observe that P^*'s simulation of $\mathrm{G5}(\mathscr{A})$ is perfect up until the point it it halts and outputs its forgery. This is due to the full linearity of DFLP, which allows us to substitute the computation of the query-generation algorithm secret-shared data in G5 with the computation of the query-generation algorithm on plaintext inputs in the prover's soundness game. It follows that P^* wins precisely when $\mathscr{A}$ sets $\mathsf{win} \leftarrow \mathsf{true}$ in the call to $\underline{\mathsf{Prep}}$ that coincides with the distinguished session. Conditioning on the probability that P^* guesses the correct call to RO_1, and that we guessed the value of inp^* correctly, we conclude that

$$\Pr\big[\,\mathrm{G4}(\mathscr{A})\,\big] \leq 2(q_1 + q_{\mathsf{Prep}}) \cdot \varepsilon\,. \tag{5.32}$$

```
Game $\mathrm{G0}(\mathscr{A})$ $\boxed{\mathrm{G1}(\mathscr{A})}$:
  $(state_{\mathscr{A}}, \{z\}, (sk,)) \leftarrow\!\!\$\, \mathscr{A}^{\mathrm{RO}}()$; $\bar{z} \leftarrow 3 - z$
  $b \leftarrow\!\!\$\, \{0,1\}$; $b' \leftarrow\!\!\$\, \mathscr{A}^{\mathrm{RO},\underline{\mathrm{Shard}},\underline{\mathrm{Setup}},\underline{\mathrm{Prep}},\underline{\mathrm{Agg}}}(state_{\mathscr{A}})$
  ret $b = b'$

$\underline{\mathrm{Shard}}(\hat{k} \in \mathbb{N}, \alpha_0, \alpha_1 \in \mathscr{I})$:
  if $\mathrm{Used}[\hat{k}] \neq \bot$: ret $\bot$
  $n \leftarrow\!\!\$\, N$ $\boxed{n \leftarrow\!\!\$\, N \setminus N^*; N^* \leftarrow N^* \cup \{n\}}$
  // Construct the VIDPF key shares.
  $seed_1, seed_2 \leftarrow\!\!\$\, \{0,1\}^{\kappa}$
  for $\ell \in [\eta]$:
    $\mathrm{D}[\hat{k}, \ell] \leftarrow \mathrm{RO}_2(seed_1, n \,\|\, \ell \,\|\, 1)$
          $+ \mathrm{RO}_2(seed_2, n \,\|\, \ell \,\|\, 2)$
    $\vec{\beta}[\ell] \leftarrow \mathtt{Encode}(\mathrm{D}[\hat{k}, \ell], 1)$
  $(key_1, key_2, pub) \leftarrow\!\!\$\, \mathsf{VIDPF.Gen}(\alpha_b, \vec{\beta})$
  // Prepare the joint randomness.
  $\vec{rseed}[1] \leftarrow \mathrm{RO}_5(seed_1, n \,\|\, 1 \,\|\, pub \,\|\, key_1)$
  $\vec{rseed}[2] \leftarrow \mathrm{RO}_5(seed_2, n \,\|\, 2 \,\|\, pub \,\|\, key_2)$
  // Generate the level proofs.
  for $\ell \in [\eta]$:
    $jseed \leftarrow \mathrm{RO}_6(0^{\kappa}, \ell \,\|\, \vec{rseed})$
    $jr \leftarrow \mathrm{RO}_1(jseed, n \,\|\, \ell)$
    $\pi \leftarrow\!\!\$\, \mathsf{DFLP.Prove}(\{0,1\}, \mathrm{D}[\hat{k}, \ell], jr)$
    $\vec{pf}[\ell] \leftarrow \pi - \mathrm{RO}_3(seed_2, n \,\|\, \ell)$
  // Prepare the initial message and input shares.
  $x_1 \leftarrow (key_1, seed_1, \vec{pf})$
  $x_2 \leftarrow (key_2, seed_2)$
  $\mathrm{In}[\hat{k}] \leftarrow x_z$
  $\mathrm{Pub}[\hat{k}] \leftarrow (pub, \vec{rseed})$
  $\mathrm{Used}[\hat{k}] \leftarrow (n, \alpha_0, \alpha_1)$
  ret $(n, \mathrm{Pub}[\hat{k}], (x_{\bar{z}},))$

$\underline{\mathrm{Setup}}(\hat{\imath} \in \mathbb{N}, st_{\mathrm{Init}} \in \mathscr{Q}_{\mathrm{Init}})$:
  $(\ell, \vec{pfx}) \leftarrow st_{\mathrm{Init}}$
  if $\mathrm{Status}[\hat{\imath}] \neq \bot$ or $\ell \in \mathscr{U}$ or $\vec{pfx}$ not distinct: ret $\bot$
  $\mathscr{U} \leftarrow \mathscr{U} \cup \{\ell\}$
  $\mathrm{Setup}[\hat{\imath}] \leftarrow st_{\mathrm{Init}}$; $\mathrm{Status}[\hat{\imath}] \leftarrow \mathsf{running}$
```

```
$\underline{\mathrm{Prep}}(\hat{\imath} \in \mathbb{N}, \hat{k} \in \mathbb{N}, \vec{M} \in \mathscr{M}^*)$:
  if $\mathrm{Status}[\hat{\imath}] \neq \mathsf{running}$ or $\mathrm{In}[\hat{k}] = \bot$: ret $\bot$
  if $\mathrm{St}[\hat{\imath}, \hat{k}] = \bot$: $\mathrm{St}[\hat{\imath}, \hat{k}] \leftarrow \mathrm{Setup}[\hat{\imath}]$
  $(n, \alpha_0, \alpha_1) \leftarrow \mathrm{Used}[\hat{k}]$
  if $\mathrm{St}[\hat{\imath}, \hat{k}] \in \mathscr{Q}_{\mathrm{Init}}$: // Process initial message from client
    $(\ell, \vec{pfx}) \leftarrow \mathrm{St}[\hat{\imath}, \hat{k}]$; $u \leftarrow |\vec{pfx}|$
    $(pub, \vec{rseed}) \leftarrow \mathrm{Pub}[\hat{k}]$
    $(key, seed, \pi) \leftarrow \mathsf{Unpack}(z, \mathrm{In}[\hat{k}], n, \ell)$
    $\Delta \leftarrow \mathrm{RO}_2(seed, n \,\|\, \ell \,\|\, z)$
    $\vec{rseed}[z] \leftarrow \mathrm{RO}_5(seed, n \,\|\, z \,\|\, pub \,\|\, key)$
    $jseed \leftarrow \mathrm{RO}_6(0^{\kappa}, \ell \,\|\, \vec{rseed})$
    $jr \leftarrow \mathrm{RO}_1(jseed, n \,\|\, \ell)$; $qr \leftarrow \mathrm{RO}_4(sk, n \,\|\, \ell)$
    $(h, \vec{y}) \leftarrow \mathsf{VIDPF.VEval}(z, pub, key, \vec{pfx})$
    $inp \leftarrow \sum_{i \in [u]} \vec{y}[i]$
    $\sigma \leftarrow \mathsf{DFLP.Query}(inp, \Delta, \pi, jr; qr)$
    $M \leftarrow (\sigma, \vec{rseed}[z], h)$
    $\mathrm{St}[\hat{\imath}, \hat{k}] \leftarrow (jseed, (\mathsf{DFLP.Decode}(\vec{y}[i]))_{i \in [u]})$
    ret $(\mathsf{running}, M)$
  // Process broadcast messages from aggregators
  $(jseed, \vec{y}) \leftarrow \mathrm{St}[\hat{\imath}, \hat{k}]$; $\mathrm{St}[\hat{\imath}, \hat{k}] \leftarrow \bot$
  $\Big((\sigma_1, rseed_1, h_1), (\sigma_2, rseed_2, h_2)\Big) \leftarrow \vec{M}$
  $acc_{\mathsf{DFLP}} \leftarrow \mathsf{DFLP.Decide}(\sigma_1 + \sigma_2)$
  $acc_{\mathsf{VIDPF}} \leftarrow \mathsf{VIDPF.Verify}(h_1, h_2)$
  $acc_0 \leftarrow jseed = \mathrm{RO}_6(0^{\kappa}, \ell \,\|\, rseed_1 \,\|\, rseed_2)$
  if $acc_{\mathsf{DFLP}}$ and $acc_{\mathsf{VIDPF}}$ and $acc_0$:
    $\mathrm{Out}[\hat{\imath}, \hat{k}] \leftarrow \vec{y}$; $\mathrm{Batch}_0[\hat{\imath}, \hat{k}] \leftarrow \alpha_0$; $\mathrm{Batch}_1[\hat{\imath}, \hat{k}] \leftarrow \alpha_1$
    ret finished
  ret failed

$\underline{\mathrm{Agg}}(\hat{\imath} \in \mathbb{N})$:
  if $\mathrm{Status}[\hat{\imath}] \neq \mathsf{running}$: ret $\bot$
  $st_{\mathrm{Init}} \leftarrow \mathrm{Setup}[\hat{\imath}]$
  if $F(st_{\mathrm{Init}}, \mathrm{Batch}_0[\hat{\imath}, \cdot]) \neq F(st_{\mathrm{Init}}, \mathrm{Batch}_1[\hat{\imath}, \cdot])$: ret $\bot$
  $\mathrm{Status}[\hat{\imath}] \leftarrow$ finished
  ret $\sum_{\vec{y} \in \mathrm{Out}[\hat{\imath}, \cdot]} \vec{y}$
```

Figure 5.27. Games G0 and G1 for the proof of Theorem 21.

The bound follows from gathering up each of the equations in simplifying.

5.9.4 Doplar Privacy (Theorem 21)

We begin with a game G0 (Figure 5.27) in which we instantiate $\mathsf{Exp}^{\mathrm{PRIV}}_{\Pi}(\mathscr{A})$ in the random oracle model, in-line the sub-routines of Π, and simplify the code. Calls to RG have been replaced with a random oracle RO; as usual, RO is implemented by lazy-evaluating a table Rand. In the remainder, we let q_i denote the number of queries $\mathscr{A}$ makes to RO_i. Another simplifying change we have made is to hard-code the index of the corrupt aggregator, which we denote by $\tilde{z}$. (We denote the honest aggregator by z.) Accordingly, we have removed the share index $\hat{j}$ from the oracle parameters and tables, as there is only one valid choice for these. (This is without loss of generality.) None of these changes impact the outcome of the experiment, so

$$\Pr\big[\,\mathsf{Exp}^{\mathrm{PRIV}}_{\Pi}(\mathscr{A})\,\big] = \Pr\big[\,\mathrm{G0}(\mathscr{A})\,\big]\,. \tag{5.33}$$

In game G1 (Figure 5.27) we revise the $\underline{\mathsf{Shard}}$ oracle by sampling the nonce without replacement (line 5). This ensures each report has a unique nonce, which will be useful in subsequent steps. By a birthday bound, we have that

$$\Pr\big[\,\mathrm{G0}(\mathscr{A})\,\big] \le \Pr\big[\,\mathrm{G1}(\mathscr{A})\,\big] + \frac{{q_{\mathsf{Shard}}}^2}{|N|}\,. \tag{5.34}$$

In our next step, G2 (Figure 5.28), we modify the $\underline{\mathsf{Shard}}$ oracle such that, instead of querying the random oracle RO, it *programs* the random oracle using a new sub-routine, PO (31–34). This ensures that the output of $\underline{\mathsf{Shard}}$ is not correlated with the game's current state, allowing us to treat the sampled values as fresh. This has a cost, however, since if any of the values programmed by the oracle overwrite existing values in table Rand, then the adversary will end up with an inconsistent view. We can bound this by considering the probability of any one of the following events occurring:

- Seed seed_1 or seed_2 sampled on line 4 coincides with a query to RO_2 made by $\mathscr{A}$ (see lines 6–7). We write this as Rand_2 for short in the remainder.

- Seed seed_1 or seed_2 coincides with an element of Rand_5 (11–12).

- Vector $\vec{\mathit{rseed}}$ sampled on lines 11–12 coincides with an element of Rand_6 (13).

- Seed jseed sampled on line 15 coincides with an element of Rand_1 (16).

- Seed seed_2 coincides with an element of Rand_3 (18).

```
$\underline{\mathsf{Shard}}(\hat{k} \in \mathbb{N}, \alpha_0, \alpha_1 \in \mathscr{I})$:
if $\mathrm{Used}[\hat{k}] \neq \bot$: ret $\bot$
$n \leftarrow\!\!\$\, N \setminus N^*; N^* \leftarrow N^* \cup \{n\}$
// Construct the VIDPF key shares.
$seed_1, seed_2 \leftarrow\!\!\$\, \{0,1\}^\kappa$
for $\ell \in [\eta]$:
  $\mathsf{D}[\hat{k},\ell] \leftarrow \mathrm{RO}_2\boxed{\mathsf{PO}_2}(seed_1, n \,\|\, \ell \,\|\, 1)$
        $+\mathrm{RO}_2\boxed{\mathsf{PO}_2}(seed_2, n \,\|\, \ell \,\|\, 2)$
  $\vec{\beta}[\ell] \leftarrow \mathtt{Encode}(\mathsf{D}[\hat{k},\ell], 1)$
$(key_1, key_2, pub) \leftarrow\!\!\$\, \mathsf{VIDPF.Gen}(\alpha_b, \vec{\beta})$
// Prepare the joint randomness.
$\vec{rseed}[1] \leftarrow \mathrm{RO}_5\boxed{\mathsf{PO}_5}(seed_1, n \,\|\, 1 \,\|\, pub \,\|\, key_1)$
$\vec{rseed}[2] \leftarrow \mathrm{RO}_5\boxed{\mathsf{PO}_5}(seed_2, n \,\|\, 2 \,\|\, pub \,\|\, key_2)$
// Generate the level proofs.
for $\ell \in [\eta]$:
  $jseed \leftarrow \mathrm{RO}_6\boxed{\mathsf{PO}_6}(0^\kappa, \ell \,\|\, \vec{rseed})$
  $jr \leftarrow \mathrm{RO}_1\boxed{\mathsf{PO}_1}(jseed, n \,\|\, \ell)$
  $\pi \leftarrow\!\!\$\, \mathsf{DFLP.Prove}(\{0,1\}, \mathsf{D}[\hat{k},\ell], jr)$
  $\vec{pf}[\ell] \leftarrow \pi - \mathrm{RO}_3\boxed{\mathsf{PO}_3}(seed_2, n \,\|\, \ell)$
// Prepare the initial message and input shares.
$x_1 \leftarrow (key_1, seed_1, \vec{pf})$
$x_2 \leftarrow (key_2, seed_2)$
$\mathrm{In}[\hat{k}] \leftarrow x_z$
$\mathrm{Pub}[\hat{k}] \leftarrow (pub, \vec{rseed})$
$\mathrm{Used}[\hat{k}] \leftarrow (n, \alpha_0, \alpha_1)$
ret $(n, \mathrm{Pub}[\hat{k}], (x_{\bar{z}},))$
```

```
$\mathrm{RO}_i(seed, cntxt)$:                                  G1 $\boxed{\mathrm{G2}}$
$l \leftarrow (jl, el, m, ql)$
if $\mathrm{Rand}[i, seed, cntxt] = \bot$:
  if $i \le 4$: $\mathrm{Rand}[i, seed, cntxt] \leftarrow\!\!\$\, \mathbb{F}^{l[i]}$
  else: $\mathrm{Rand}[i, seed, cntxt] \leftarrow\!\!\$\, \{0,1\}^\kappa$
ret $\mathrm{Rand}[i, seed, cntxt]$

$\mathsf{PO}_i(seed, cntxt)$:
$l \leftarrow (jl, el, m, ql)$
if $i \le 4$: $\mathrm{Rand}[i, seed, cntxt] \leftarrow\!\!\$\, \mathbb{F}^{l[i]}$
else: $\mathrm{Rand}[i, seed, cntxt] \leftarrow\!\!\$\, \{0,1\}^\kappa$
ret $\mathrm{Rand}[i, seed, cntxt]$
```

Figure 5.28. Game G2 for the proof of Theorem 21.

Because the nonces sampled by $\underline{\mathsf{Shard}}$ are unique, and because each of this oracle queries encodes the nonce, we can be certain that points programmed into the table by each $\underline{\mathsf{Shard}}$ query do not collide with one another. Indeed, it is only possible for these values to coincide with random oracle queries made by $\mathscr{A}$. Apply a union bound over all q_{Shard} queries, we conclude that

$$\Pr\big[\,\mathrm{G1}(\mathscr{A})\,\big] \le \Pr\big[\,\mathrm{G2}(\mathscr{A})\,\big] + \frac{q_2 q_{\mathsf{Shard}}}{2^{\kappa-1}} + \frac{q_5 q_{\mathsf{Shard}}}{2^{\kappa-1}} + \frac{q_6 q_{\mathsf{Shard}}}{2^{2\kappa}} + \frac{q_1 q_{\mathsf{Shard}}}{2^{\kappa}}\,. \tag{5.35}$$

In the next step, G3 (Figure 5.29), we substitute calls to $\mathsf{VIDPF.Gen}$ and $\mathsf{VIDPF.VEval}$ with calls to the simulator $\mathscr{S} = (\mathscr{S}^1_{\mathsf{VIDPF}}, \mathscr{S}^2_{\mathsf{VIDPF}})$. The first part, $\mathscr{S}^1_{\mathsf{VIDPF}}$, is used to simulate the public share corrupt aggregator's key share (10); the second part, $\mathscr{S}^2_{\mathsf{VIDPF}}$, is used to simulate the honest aggregators one-hot check, based on the output of the first (41). After this second point, we no longer compute the honest aggregator's refined share $\vec{y}$ consistently. Instead, we

```
$\underline{\mathsf{Shard}}(\hat{k} \in \mathbb{N}, \alpha_0, \alpha_1 \in \mathscr{I})$:
if $\mathrm{Used}[\hat{k}] \neq \bot$: ret $\bot$
$n \leftarrow\!\!\$\, N \setminus N^*; N^* \leftarrow N^* \cup \{n\}$
// Construct the VIDPF key shares.
$seed_1, seed_2 \leftarrow\!\!\$\, \{0,1\}^\kappa$
for $\ell \in [\eta]$:
  $\mathrm{D}[\hat{k},\ell] \leftarrow \mathsf{PO}_2(seed_1, n \,\|\, \ell \,\|\, 1)$
        $+\, \mathsf{PO}_2(seed_2, n \,\|\, \ell \,\|\, 2)$
  $\vec{\beta}[\ell] \leftarrow \mathtt{Encode}(\mathrm{D}[\hat{k},\ell], 1)$
$(key_1, key_2, pub) \leftarrow\!\!\$\, \mathsf{VIDPF.Gen}(\alpha_b, \vec{\beta})$
$\boxed{(\mathrm{T}[\hat{k}], pub) \leftarrow\!\!\$\, \mathscr{S}^1_{\mathsf{VIDPF}}(\tilde{z});\ key_{\tilde{z}} \leftarrow \mathrm{T}[\hat{k}];\ key_z \leftarrow \bot}$
// Prepare the joint randomness.
$\vec{rseed}[1] \leftarrow \mathsf{PO}_5(seed_1, n \,\|\, 1 \,\|\, pub \,\|\, key_1)$
$\vec{rseed}[2] \leftarrow \mathsf{PO}_5(seed_2, n \,\|\, 2 \,\|\, pub \,\|\, key_2)$
// Generate the level proofs.
for $\ell \in [\eta]$:
  $jseed \leftarrow \mathsf{PO}_6(0^\kappa, \ell \,\|\, \vec{rseed})$
  $jr \leftarrow \mathsf{PO}_1(jseed, n \,\|\, \ell)$
  $\pi \leftarrow\!\!\$\, \mathsf{DFLP.Prove}(\{0,1\}, \mathrm{D}[\hat{k},\ell], jr)$
  $\vec{pf}[\ell] \leftarrow \pi - \mathsf{PO}_3(seed_2, n \,\|\, \ell)$
// Prepare the initial message and input shares.
$x_1 \leftarrow (key_1, seed_1, \vec{pf})$
$x_2 \leftarrow (key_2, seed_2)$
$\mathrm{In}[\hat{k}] \leftarrow x_z$
$\mathrm{Pub}[\hat{k}] \leftarrow (pub, \vec{rseed})$
$\mathrm{Used}[\hat{k}] \leftarrow (n, \alpha_0, \alpha_1)$
ret $(n, \mathrm{Pub}[\hat{k}], (x_{\tilde{z}},))$
```

```
$\underline{\mathsf{Prep}}(\hat{i} \in \mathbb{N}, \hat{k} \in \mathbb{N}, \vec{M} \in \mathscr{M}^*)$:        G2 $\boxed{\text{G3}}$
if $\mathrm{Status}[\hat{i}] \neq \mathsf{running}$ or $\mathrm{In}[\hat{k}] = \bot$: ret $\bot$
if $\mathrm{St}[\hat{i},\hat{k}] = \bot$: $\mathrm{St}[\hat{i},\hat{k}] \leftarrow \mathrm{Setup}[\hat{i}]$
$(n, \alpha_0, \alpha_1) \leftarrow \mathrm{Used}[\hat{k}]$
if $\mathrm{St}[\hat{i},\hat{k}] \in \mathscr{Q}_{\mathrm{Init}}$: // Process initial message from client
  $(\ell, \vec{pfx}) \leftarrow \mathrm{St}[\hat{i},\hat{k}];\ u \leftarrow |\vec{pfx}|$
  $(pub, \vec{rseed}) \leftarrow \mathrm{Pub}[\hat{k}]$
  $(key_{\boxed{\phantom{z}}}, seed, \pi) \leftarrow \mathsf{Unpack}(z, \mathrm{In}[\hat{k}], n, \ell)$
  $\Delta \leftarrow \mathrm{RO}_2(seed, n \,\|\, \ell \,\|\, z)$
  $\vec{rseed}[z] \leftarrow \mathrm{RO}_5(seed, n \,\|\, z \,\|\, pub \,\|\, key)$
  $jseed \leftarrow \mathrm{RO}_6(0^\kappa, \ell \,\|\, \vec{rseed})$
  $jr \leftarrow \mathrm{RO}_1(jseed, n \,\|\, \ell);\ qr \leftarrow \mathrm{RO}_4(sk, n \,\|\, \ell)$
  $(h, \vec{y}) \leftarrow \mathsf{VIDPF.VEval}(z, pub, key, \vec{pfx})$
  $inp \leftarrow \sum_{i \in [u]} \vec{y}[i]$
  $\boxed{key_{\tilde{z}} \leftarrow \mathrm{T}[\hat{k}]}$
  $\boxed{h \leftarrow\!\!\$\, \mathscr{S}^2_{\mathsf{VIDPF}}(\tilde{z}, pub, key_{\tilde{z}}, \vec{pfx})}$
  $\boxed{(\_, \vec{y}) \leftarrow \mathsf{VIDPF.VEval}(\tilde{z}, pub, key_{\tilde{z}}, \vec{pfx})}$
  $\boxed{x_b \leftarrow |\{\vec{pfx}[i] : \vec{pfx}[i] \text{ prefixes } \alpha_b\}_{i \in [u]}|}$
  $\boxed{inp_b \leftarrow \mathsf{DFLP.Encode}(\Delta[\hat{k},\ell], x_b)}$
  $\boxed{inp \leftarrow inp_b - \sum_{i \in [u]} \vec{y}[i]}$
  $\sigma \leftarrow \mathsf{DFLP.Query}(inp, \Delta, \pi, jr;\ qr)$
  $M \leftarrow (\sigma, \vec{rseed}[z], h)$
  $\mathrm{St}[\hat{i},\hat{k}] \leftarrow (jseed, (\mathsf{DFLP.Decode}(\vec{y}[i]))_{i \in [u]})$
  ret $(\mathsf{running}, M)$
// Process broadcast messages from aggregators
$(jseed, \vec{y}) \leftarrow \mathrm{St}[\hat{i},\hat{k}];\ \mathrm{St}[\hat{i},\hat{k}] \leftarrow \bot$
$\big((\sigma_1, rseed_1, h_1), (\sigma_2, rseed_2, h_2)\big) \leftarrow \vec{M}$
$acc_{\mathsf{DFLP}} \leftarrow \mathsf{DFLP.Decide}(\sigma_1 + \sigma_2)$
$acc_{\mathsf{VIDPF}} \leftarrow \mathsf{VIDPF.Verify}(h_1, h_2)$
$acc_0 \leftarrow jseed = \mathrm{RO}_6(0^\kappa, \ell \,\|\, rseed_1 \,\|\, rseed_2)$
if $acc_{\mathsf{DFLP}}$ and $acc_{\mathsf{VIDPF}}$ and $acc_0$:
  $\mathrm{Out}[\hat{i},\hat{k}] \leftarrow \vec{y};\ \mathrm{Batch}_0[\hat{i},\hat{k}] \leftarrow \alpha_0;\ \mathrm{Batch}_1[\hat{i},\hat{k}] \leftarrow \alpha_1$
  ret $\mathtt{finished}$
ret $\mathtt{failed}$
```

Figure 5.29. Game G3 for the proof of Theorem 21.

compute the *corrupt aggregator's refined share* $\vec{y}$ and compute the the challenge input *inp* by subtracting the sum from the true sum for the input α_b (43–44).

There exists an adversary $\mathscr{B}$ for which

$$\Pr\left[\,\mathrm{G2}(\mathscr{A})\,\right] \le \Pr\left[\,\mathrm{G3}(\mathscr{A})\,\right] + q_{\mathsf{Shard}} \cdot \mathbf{Adv}\textsc{priv}_{\mathsf{VIDPF},\mathscr{S}}(\mathscr{B})\,. \tag{5.36}$$

The proof is by a standard argument. In each hybrid game, we answer one more Shard query (and the corresponding Prep query) using $\mathscr{S}$. Adversary $\mathscr{B}$ simply runs $\mathscr{A}$ in one of these hybrid games, chosen at random, and outputs whatever $\mathscr{A}$ outputs.

In game G4 (Figure 5.30), we prepare for the Shard oracle for the reduction to DFLP privacy. The primary change is that we have Shard sample the query randomness qr that will be used to query the proof at each level (see line 18 in the left panel). This ensures that the query randomness is "committed" even before the query is made. We use the unpredictability of the nonce to bound the probability that this change leads to an inconsistent view of the experiment. In particular,

$$\Pr\left[\,\mathrm{G3}(\mathscr{A})\,\right] \le \Pr\left[\,\mathrm{G4}(\mathscr{A})\,\right] + \frac{\eta q_4 q_{\mathsf{Shard}}}{|N|}\,. \tag{5.37}$$

In this step, we also make a couple of non-breaking changes. First, we in-line programming of the random oracle with the joint randomness and encoding randomness (16–17,19). Second, we store each proof and encoding randomness in tables $\mathbf{P}$ and $\mathbf{D}$ respectively. These changes are made to clarify the next step.

In game G5 (Figure 5.30) we prepare the Prep oracle by re-arranging the proof query. In particular, we run the query-generation algorithm on the plaintext encoded input and proof, and generate the verifier share that is output by subtracting from the verifier (denoted $\mathbf{V}[\hat{k},\ell]$; see line 19 of the right panel) the verifier share generated from the corrupt aggregator's share. The adversary's view is consistent with the previous game by the full linearity of DFLP.

Lastly, in game G6 (not pictured) we modify the Prep oracle by replacing computation of the verifier from α_b with the DFLP-privacy simulator $\mathscr{T}$. There exists an adversary $\mathscr{C}$ for which

$$\Pr\left[\,\mathrm{G5}(\mathscr{A})\,\right] \le \Pr\left[\,\mathrm{G6}(\mathscr{A})\,\right] + \eta q_{\mathsf{Shard}} \cdot \mathbf{Adv}\textsc{priv}_{\mathsf{DFLP},\mathscr{T}}(\mathscr{C})\,. \tag{5.38}$$

```
$\underline{\mathsf{Shard}}(\hat{k} \in \mathbb{N}, \alpha_0, \alpha_1 \in \mathscr{I})$:    G3 $\boxed{\text{G4}}$
if $\mathrm{Used}[\hat{k}] \neq \bot$: ret $\bot$
$n \leftarrow\!\!\$\, N \setminus N^*; N^* \leftarrow N^* \cup \{n\}$
// Construct the VIDPF key shares.
$seed_1, seed_2 \leftarrow\!\!\$\, \{0,1\}^\kappa$
for $\ell \in [\eta]$:
  $\mathrm{D}[\hat{k}, \ell] \leftarrow \mathsf{PO}_2(seed_1, n \,\|\, \ell \,\|\, 1)$
        $+ \mathsf{PO}_2(seed_2, n \,\|\, \ell \,\|\, 2)$
$(\mathrm{T}[\hat{k}], pub) \leftarrow\!\!\$\, \mathscr{S}^1_{\mathsf{VIDPF}}(\tilde{z})$; $key_{\tilde{z}} \leftarrow \mathrm{T}[\hat{k}]$; $key_z \leftarrow \bot$
// Prepare the joint randomness.
$\vec{rseed}[1] \leftarrow \mathsf{PO}_5(seed_1, n \,\|\, 1 \,\|\, pub \,\|\, key_1)$
$\vec{rseed}[2] \leftarrow \mathsf{PO}_5(seed_2, n \,\|\, 2 \,\|\, pub \,\|\, key_2)$
// Generate the level proofs.
for $\ell \in [\eta]$:
  $jseed \leftarrow \mathsf{PO}_6(0^\kappa, \ell \,\|\, \vec{rseed})$
  $jr \leftarrow\!\!\$\, \mathbb{F}^{jl}$; $qr \leftarrow\!\!\$\, \mathbb{F}^{ql}$; $\mathrm{D}[\hat{k}, \ell], \tilde{\Delta} \leftarrow\!\!\$\, \mathbb{F}^{el}$
  $\mathrm{Rand}[2, seed_z, n \,\|\, \ell \,\|\, z] \leftarrow \mathrm{D}[\hat{k}, \ell] - \tilde{\Delta}$
  $\mathrm{Rand}[2, seed_{\tilde{z}}, n \,\|\, \ell \,\|\, \tilde{z}] \leftarrow \tilde{\Delta}$
  $\mathrm{Rand}[4, sk, n \,\|\, \ell] \leftarrow qr$
  $\mathrm{Rand}[1, jseed, n \,\|\, \ell] \leftarrow jr$
  $\mathrm{P}[\hat{k}, \ell] \leftarrow\!\!\$\, \mathsf{DFLP.Prove}(\{0,1\}, \Delta, jr)$
  $\vec{pf}[\ell] \leftarrow \mathrm{P}[\hat{k}, \ell] - \mathsf{PO}_3(seed_2, n \,\|\, \ell)$
  $jr \leftarrow \mathsf{PO}_1(jseed, n \,\|\, \ell)$
  $\pi \leftarrow\!\!\$\, \mathsf{DFLP.Prove}(\{0,1\}, \mathrm{D}[\hat{k}, \ell], jr)$
  $\vec{pf}[\ell] \leftarrow \pi - \mathsf{PO}_3(seed_2, n \,\|\, \ell)$
// Prepare the initial message and input shares.
$x_1 \leftarrow (key_1, seed_1, \vec{pf})$; $x_2 \leftarrow (key_2, seed_2)$
$\mathrm{In}[\hat{k}] \leftarrow x_z$; $\mathrm{Pub}[\hat{k}] \leftarrow (pub, \vec{rseed})$
$\mathrm{Used}[\hat{k}] \leftarrow (n, \alpha_0, \alpha_1)$
ret $(n, \mathrm{Pub}[\hat{k}], (x_{\tilde{z}},))$
```

```
$\underline{\mathsf{Prep}}(\hat{i} \in \mathbb{N}, \hat{k} \in \mathbb{N}, \vec{M} \in \mathscr{M}^*)$:    G4 $\boxed{\text{G5}}$
if $\mathrm{Status}[\hat{i}] \neq \mathsf{running}$ or $\mathrm{In}[\hat{k}] = \bot$: ret $\bot$
if $\mathrm{St}[\hat{i}, \hat{k}] = \bot$: $\mathrm{St}[\hat{i}, \hat{k}] \leftarrow \mathrm{Setup}[\hat{i}]$
$(n, \alpha_0, \alpha_1) \leftarrow \mathrm{Used}[\hat{k}]$
if $\mathrm{St}[\hat{i}, \hat{k}] \in \mathscr{Q}_{\mathrm{Init}}$:  // Process initial message from client
  $(\ell, \vec{pfx}) \leftarrow \mathrm{St}[\hat{i}, \hat{k}]$; $u \leftarrow |\vec{pfx}|$
  $(pub, \vec{rseed}) \leftarrow \mathrm{Pub}[\hat{k}]$
  $(\_, seed, \pi) \leftarrow \mathsf{Unpack}(z, \mathrm{In}[\hat{k}], n, \ell)$
  $\Delta \leftarrow \mathrm{RO}_2(seed, n \,\|\, \ell \,\|\, z)$
  $\vec{rseed}[z] \leftarrow \mathrm{RO}_5(seed, n \,\|\, z \,\|\, pub \,\|\, key)$
  $jseed \leftarrow \mathrm{RO}_6(0^\kappa, \ell \,\|\, \vec{rseed})$
  $jr \leftarrow \mathrm{RO}_1(jseed, n \,\|\, \ell)$; $qr \leftarrow \mathrm{RO}_4(sk, n \,\|\, \ell)$
  $key_{\tilde{z}} \leftarrow \mathrm{T}[\hat{k}]$
  $h \leftarrow\!\!\$\, \mathscr{S}^2_{\mathsf{VIDPF}}(\tilde{z}, pub, key_{\tilde{z}}, \vec{pfx})$
  $(\_, \vec{\vec{y}}) \leftarrow \mathsf{VIDPF.VEval}(\tilde{z}, pub, key_{\tilde{z}}, \vec{pfx})$
  $x_b \leftarrow |\{\vec{pfx}[i] : \vec{pfx}[i] \text{ prefixes } \alpha_b\}_{i \in [u]}|$
  $inp_b \leftarrow \mathsf{DFLP.Encode}(\Delta[\hat{k}, \ell], x_b)$
  $inp \leftarrow inp_b - \sum_{i \in [u]} \vec{\vec{y}}[i]$
  $\sigma \leftarrow \mathsf{DFLP.Query}(inp, \Delta, \pi, jr; qr)$
  $\mathrm{V}[\hat{k}, \ell] \leftarrow \mathsf{DFLP.Query}(inp_b, \mathrm{D}[\hat{k}, \ell], \mathrm{P}[\hat{k}, \ell], jr; qr)$
  $\sigma \leftarrow \mathrm{V}[\hat{k}, \ell] - \mathsf{DFLP.Query}(\sum_{i \in [u]} \vec{\vec{y}}[i], \Delta, \pi, jr; qr)$
  $M \leftarrow (\sigma, \vec{rseed}[z], h)$
  $\mathrm{St}[\hat{i}, \hat{k}] \leftarrow (jseed, (\mathsf{DFLP.Decode}(\vec{y}[i]))_{i \in [u]})$
  ret $(\mathsf{running}, M)$
// Process broadcast messages from aggregators
$(jseed, \vec{y}) \leftarrow \mathrm{St}[\hat{i}, \hat{k}]$; $\mathrm{St}[\hat{i}, \hat{k}] \leftarrow \bot$
$\big((\sigma_1, rseed_1, h_1), (\sigma_2, rseed_2, h_2)\big) \leftarrow \vec{M}$
$acc_{\mathsf{DFLP}} \leftarrow \mathsf{DFLP.Decide}(\sigma_1 + \sigma_2)$
$acc_{\mathsf{VIDPF}} \leftarrow \mathsf{VIDPF.Verify}(h_1, h_2)$
$acc_0 \leftarrow jseed = \mathrm{RO}_6(0^\kappa, \ell \,\|\, rseed_1 \,\|\, rseed_2)$
if $acc_{\mathsf{DFLP}}$ and $acc_{\mathsf{VIDPF}}$ and $acc_0$:
  $\mathrm{Out}[\hat{i}, \hat{k}] \leftarrow \vec{y}$; $\mathrm{Batch}_0[\hat{i}, \hat{k}] \leftarrow \alpha_0$; $\mathrm{Batch}_1[\hat{i}, \hat{k}] \leftarrow \alpha_1$
  ret finished
ret failed
```

Figure 5.30. Games G4 and G5 for the proof of Theorem 21.

The proof is by a hybrid argument, where each hybrid game $\mathrm{G}u,v'$ is defined as follows. For the first u reports and for the first v levels of the VIDPF tree, the verifier $\mathrm{V}[u,v]$ is generated as specified in game G5 (line 19 in the right panel of Figure 5.30); all other verifiers are generated by $\mathscr{T}$ as specified in game G6. By construction,

$$\Pr\big[\,\mathrm{G5}(\mathscr{A})\,\big] - \Pr\big[\,\mathrm{G6}(\mathscr{A})\,\big] = \Pr\big[\,\mathrm{G0},0'(\mathscr{A})\,\big] - \Pr\big[\,\mathrm{G}q_{\mathsf{Shard}},\eta'(\mathscr{A})\,\big]\,. \tag{5.39}$$

Define DFLP-privacy attacker $\mathscr{C}$ as follows. (Refer to Figure 5.6.) On its first invocation, it simply outputs $X = \{0,1\}$ as the input set, as this is what is required by the game. On its next invocation, it is given joint randomness jr^* and query randomness qr^*. It proceeds by simulating $\mathscr{A}$ in a random hybrid game. It first samples $u^* \leftarrow\!\!\$\, [q_{\mathsf{Shard}}]$ and $v^* \leftarrow\!\!\$\, [\eta]$. It then runs $\mathrm{G}u^*,v^{*\prime}(\mathscr{A})$ except:

- On the u^*-th query to $\underline{\mathsf{Shard}}$, for the v^*-th level, it uses jr^* and qr^* to program the random oracles for the joint and query randomness respectively.
- When $\mathscr{A}$ makes a $\underline{\mathsf{Prep}}$ query corresponding to report u^* and level v^*, it halts and outputs x_b and awaits a response from its game. Upon being invoked once more on input σ, it sets $\mathrm{V}[u^*,v^*] \leftarrow \sigma$ and continues the simulation.

Finally, when $\mathscr{A}$ halts, $\mathscr{C}$ halts and returns whatever $\mathscr{A}$ output. Then $\mathscr{C}$ perfectly simulates $\mathrm{G}u^*,v^{*\prime}(\mathscr{A})$ when the value of its challenge bit is 1, and it perfectly simulates $\mathrm{G}u^*,v^*+1'(\mathscr{A})$ when its challenge bit is equal to 0. The claimed bound follows from a standard conditioning argument.

To complete the proof, we note that

$$\Pr\big[\,\mathrm{G6}(\mathscr{A})\,\big] = \frac{1}{2}\,. \tag{5.40}$$

Gathering up all of the terms and simplifying yields the desired bound.

Bibliography

[1] Privacy preserving measurement, 2022. (Cited on page 5, 251.)

[2] Private Advertising Tecnology Community Group, 2022. (Cited on page 250.)

[3] M. Abdalla, J. H. An, M. Bellare, and C. Namprempre. From identification to signatures via the Fiat-Shamir transform: Minimizing assumptions for security and forward-security. In L. R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 418–433. Springer, Heidelberg, Apr. / May 2002. (Cited on page 214, 215, 217, 218, 228, 229, 230, 237.)

[4] M. Abdalla, M. Bellare, and P. Rogaway. The oracle Diffie-Hellman assumptions and an analysis of DHIES. In D. Naccache, editor, *CT-RSA 2001*, volume 2020 of *LNCS*, pages 143–158. Springer, Heidelberg, Apr. 2001. (Cited on page 51, 52, 53.)

[5] M. Abdalla, F. Benhamouda, and P. MacKenzie. Security of the J-PAKE password-authenticated key exchange protocol. In *2015 IEEE Symposium on Security and Privacy*, pages 571–587. IEEE Computer Society Press, May 2015. (Cited on page 57.)

[6] M. Abdalla, P.-A. Fouque, and D. Pointcheval. Password-based authenticated key exchange in the three-party setting. In S. Vaudenay, editor, *PKC 2005*, volume 3386 of *LNCS*, pages 65–84. Springer, Heidelberg, Jan. 2005. (Cited on page 57.)

[7] M. Abdalla, B. Haase, and J. Hesse. Security analysis of cpace. Cryptology ePrint Archive, Paper 2021/114, 2021. (Cited on page 254.)

[8] M. Abdalla, B. Haase, and J. Hesse. CPace, a balanced composable PAKE. Internet-Draft draft-irtf-cfrg-cpace-06, Internet Engineering Task Force, July 2022. Work in Progress. (Cited on page 254.)

[9] S. Addanki, K. Garbe, E. Jaffe, R. Ostrovsky, and A. Polychroniadou. Prio+: Privacy preserving aggregate statistics via boolean shares. Cryptology ePrint Archive, Report 2021/576, 2021. https://ia.cr/2021/576. (Cited on page 251, 257.)

[10] M. Albrecht, C. Cid, K. G. Paterson, C. J. Tjhai, and M. Tomlinson. NTS-KEM. NIST PQC Round 2 Submission, 2019. (Cited on page 9, 22.)

[11] E. Alkim, R. Avanzi, J. Bos, L. Ducas, A. de la Piedra, T. Pöppelmann, P. Schwabe, and D. Stebila. NewHope: Algorithm specifications and supporting documentation. NIST PQC Round 2 Submission, 2019. (Cited on page 3, 8, 20, 41.)

[12] E. Anderson, M. Chase, F. B. Durak, E. Ghosh, K. Laine, and C. Weng. Aggregate measurement via oblivious shuffling. Cryptology ePrint Archive, Report 2021/1490, 2021. https://ia.cr/2021/1490. (Cited on page 253, 257.)

[13] Apple and Google. Exposure Notification Privacy-preserving Analytics (ENPA). White paper, 2021. https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf. (Cited on page 5, 251, 252.)

[14] N. Aragon, P. S. L. M. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Gueron, T. Güneysu, C. Aguilar Melchor, R. Misoczki, E. Persichetti, N. Sendrier, J.-P. Tillich, V. Vasseur, and G. Zémor. BIKE: Bit flipping key encapsulation. NIST PQC Round 2 Submission, 2019. (Cited on page 9, 20.)

[15] N. Aragon, O. Blazy, J.-C. Deneuville, P. Gaborit, A. Hauteville, O. Ruatta, J.-P. Tillich, and G. Zémor. LOCKER: Low rank parity check codes encryption. NIST PQC Round 1 Submission, 2017. (Cited on page 9, 20.)

[16] G. Arfaoui, X. Bultel, P.-A. Fouque, A. Nedelcu, and C. Onete. The privacy of the TLS 1.3 protocol. *PoPETs*, 2019(4):190–210, Oct. 2019. (Cited on page 155.)

[17] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3):501–555, may 1998. (Cited on page 260.)

[18] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS-Kyber: Algorithm specifications and supporting documentation. NIST PQC Round 2 Submission, 2019. (Cited on page 9, 22.)

[19] H. Baan, S. Bhattacharya, S. Fluhrer, O. Garcia-Morchon, T. Laarohoven, R. Player, R. Rietman, M.-J. O. Saarinen, L. Tolhuizen, J. L. Torre-Arce, and Z. Zhang. Round5: KEM and PKE based on (ring) learning with rounding. NIST PQC Round 2 Submission, 2019. (Cited on page 9, 20.)

[20] M. Backendal, M. Bellare, J. Sorrell, and J. Sun. The fiat-shamir zoo: Relating the security of different signature variants. In N. Gruschka, editor, *Secure IT Systems - 23rd Nordic Conference, NordSec 2018*, volume 11252 of *Lecture Notes in Computer Science*, pages 154–170. Springer, 2018. (Cited on page 229.)

[21] C. Bader, D. Hofheinz, T. Jager, E. Kiltz, and Y. Li. Tightly-secure authenticated key exchange. In Y. Dodis and J. B. Nielsen, editors, *TCC 2015, Part I*, volume 9014 of *LNCS*, pages 629–658. Springer, Heidelberg, Mar. 2015. (Cited on page 52, 55, 64, 117.)

[22] C. Bader, T. Jager, Y. Li, and S. Schäge. On the impossibility of tight cryptographic reductions. In M. Fischlin and J.-S. Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 273–304. Springer, Heidelberg, May 2016. (Cited on page 54, 65, 117.)

[23] G. Banegas, P. S. L. M. Barreto, B. O. Boidje, P.-L. Cayrel, G. N. Dione, K. Gaj, C. T. Gueye, R. Haeussler, J. B. Klamti, O. N'diaye, D. T. Nguyen, E. Persichetti, and J. E. Ricardini. DAGS: Key encapsulation from dyadic GS codes. NIST PQC Round 1 Submission, 2017. (Cited on page 8, 17.)

[24] M. Bardet, É. Barelli, O. Blazy, R. Canto-Torres, A. Couvreur, P. Gaborit, A. Otmani, N. Sendrier, and J.-P. Tillich. BIG QUAKE: Binary goppa quasi-cyclic key encapsulation. NIST PQC Round 1 Submission, 2017. (Cited on page 8, 17.)

[25] R. Barnes, C. Patton, and P. Schoppmann. Verifiable Distributed Aggregation Functions. Internet-Draft draft-irtf-cfrg-vdaf-03, Internet Engineering Task Force, Aug. 2022. Work in Progress. (Cited on page 5, 252, 255, 259, 261, 262, 265, 270, 272, 273, 275, 283, 284, 296.)

[26] J. Bell, A. Gascón, B. Ghazi, R. Kumar, P. Manurangsi, M. Raykova, and P. Schoppmann. Distributed, private, sparse histograms in the two-server model. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 307–321, 2022. (Cited on page 251, 253, 257, 287.)

[27] J. Bell, A. Gascón, T. Lepoint, B. Li, S. Meiklejohn, M. Raykova, and C. Yun. Acorn: Input validation for secure aggregation. *Cryptology ePrint Archive*, 2022. (Cited on page 257.)

[28] J. H. Bell, K. A. Bonawitz, A. Gascón, T. Lepoint, and M. Raykova. Secure single-server aggregation with (poly) logarithmic overhead. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1253–1269, 2020. (Cited on page 257.)

[29] M. Bellare, D. J. Bernstein, and S. Tessaro. Hash-function based PRFs: AMAC and its multi-user security. In M. Fischlin and J.-S. Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 566–595. Springer, Heidelberg, May 2016. (Cited on page 13, 54, 64, 67, 76, 216, 217, 233, 240.)

[30] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In N. Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 1–15. Springer, Heidelberg, Aug. 1996. (Cited on page 212.)

[31] M. Bellare, R. Canetti, and H. Krawczyk. Pseudorandom functions revisited: The cascade construction and its concrete security. In *37th FOCS*, pages 514–523. IEEE Computer Society Press, Oct. 1996. (Cited on page 63.)

[32] M. Bellare and W. Dai. The multi-base discrete logarithm problem: Non-rewinding proofs and improved reduction tightness for identification and signatures. In *INDOCRYPT 2020*, 2020. https://eprint.iacr.org/2020/416. (Cited on page 4, 67, 215, 218, 230, 237.)

[33] M. Bellare, H. Davis, and Z. Di. Hardening signature schemes via derive-then-derandomize: Stronger security proofs for eddsa. In A. Boldyreva and V. Kolesnikov, editors, *Public-Key Cryptography - PKC 2023 - 26th IACR International Conference on Practice and Theory of Public-Key Cryptography, Atlanta, GA, USA, May 7-10, 2023, Proceedings, Part I*, volume 13940 of *Lecture Notes in Computer Science*, pages 223–250. Springer, 2023. (Cited on page .)

[34] M. Bellare, O. Goldreich, and A. Mityagin. The power of verification queries in message authentication and authenticated encryption. Cryptology ePrint Archive, Report 2004/309, 2004. http://eprint.iacr.org/2004/309. (Cited on page 67.)

[35] M. Bellare and A. Palacio. GQ and Schnorr identification schemes: Proofs of security against impersonation under active and concurrent attacks. In M. Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 162–177. Springer, Heidelberg, Aug. 2002. (Cited on page 215.)

[36] M. Bellare, B. Poettering, and D. Stebila. From identification to signatures, tightly: A framework and generic transforms. In J. H. Cheon and T. Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 435–464. Springer, Heidelberg, Dec. 2016. (Cited on page 213, 218.)

[37] M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In B. Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 139–155. Springer, Heidelberg, May 2000. (Cited on page 50.)

[38] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, pages 62–73, New York, NY, USA, 1993. ACM. (Cited on page 2, 259.)

[39] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In D. E. Denning, R. Pyle, R. Ganesan, R. S. Sandhu, and V. Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, Nov. 1993. (Cited on page 6, 13, 43, 52, 53, 55, 212, 220, 223.)

[40] M. Bellare and P. Rogaway. Entity authentication and key distribution. In D. R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 232–249. Springer, Heidelberg, Aug. 1994. (Cited on page 50, 53, 56.)

[41] M. Bellare and P. Rogaway. Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331, 2004. http://eprint.iacr.org/2004/331. (Cited on page 163.)

[42] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In S. Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, Heidelberg, May / June 2006. (Cited on page 24, 47, 53, 59, 76, 77, 219, 225.)

[43] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In S. Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006*, pages 409–426, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. (Cited on page 253, 306, 311.)

[44] M. Bellare and B. Tackmann. Nonce-based cryptography: Retaining security when randomness fails. In M. Fischlin and J.-S. Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 729–757. Springer, Heidelberg, May 2016. (Cited on page 213, 218.)

[45] D. J. Bernstein. Multi-user Schnorr security, revisited. Cryptology ePrint Archive, Report 2015/996, 2015. http://eprint.iacr.org/2015/996. (Cited on page 229.)

[46] D. J. Bernstein, T. Chou, T. Lange, I. von Maurich, R. N. Rafael Misoczki, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier, J. Szefer, and W. Wang. Classic McEliece: conservative code-based cryptography. NIST PQC Round 2 Submission, 2019. (Cited on page 9, 22.)

[47] D. J. Bernstein, C. Chuengsatiansup, T. Lange, and C. van Vredendaal. NTRU Prime. NIST PQC Round 2 Submission, 2019. (Cited on page 9, 22.)

[48] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. In B. Preneel and T. Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 124–142. Springer, Heidelberg, Sept. / Oct. 2011. (Cited on page 54.)

[49] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, Sept. 2012. (Cited on page 4.)

[50] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of cryptographic engineering*, 2(2):77–89, 2012. (Cited on page 211, 213, 215, 230.)

[51] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. On the indifferentiability of the sponge construction. In N. P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 181–197. Springer, Heidelberg, Apr. 2008. (Cited on page 2.)

[52] K. Bhargavan, C. Brzuska, C. Fournet, M. Green, M. Kohlweiss, and S. Zanella-Béguelin. Downgrade resilience in key-exchange protocols. In *2016 IEEE Symposium on Security and Privacy*, pages 506–525. IEEE Computer Society Press, May 2016. (Cited on page 120.)

[53] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Zanella Béguelin. Proving the TLS handshake secure (as it is). In J. A. Garay and R. Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 235–255. Springer, Heidelberg, Aug. 2014. (Cited on page 119.)

[54] N. Bitansky, A. Chiesa, Y. Ishai, O. Paneth, and R. Ostrovsky. Succinct non-interactive arguments via linear interactive proofs. In A. Sahai, editor, *Theory of Cryptography*, pages 315–333, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. (Cited on page 260.)

[55] D. Bleichenbacher. A forgery attack on RSA signatures based on implementation errors in the verification. Rump Session Presentation, Crypto 2006, August 2006. (Cited on page 211.)

[56] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. Practical secure aggregation for privacy-preserving machine learning. In *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1175–1191, 2017. (Cited on page 257.)

[57] D. Boneh. The decision Diffie-Hellman problem. In *Third Algorithmic Number Theory Symposium (ANTS)*, volume 1423 of *LNCS*. Springer, Heidelberg, 1998. Invited paper. (Cited on page 50, 53, 62.)

[58] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In A. Boldyreva and D. Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 67–97, Cham, 2019. Springer International Publishing. (Cited on page xii, 5, 251, 255, 259, 261, 265, 270, 272, 283, 284, 296, 297, 298.)

[59] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. Lightweight techniques for private heavy hitters. In *IEEE Symposium on Security and Privacy*, pages 762–776. IEEE, 2021. (Cited on page 5, 251, 255, 256, 262, 265, 270, 275, 277, 282, 285.)

[60] C. Boyd, C. Cremers, M. Feltz, K. G. Paterson, B. Poettering, and D. Stebila. ASICS: Authenticated key exchange security incorporating certification systems. In J. Crampton, S. Jajodia, and K. Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 381–399. Springer, Heidelberg, Sept. 2013. (Cited on page 127.)

[61] C. Boyd, G. T. Davies, B. de Kock, K. Gellert, T. Jager, and L. Millerjord. Symmetric key exchange with full forward security and robust synchronization. In *ASIACRYPT 2021*, 2021. To appear. Available as Cryptology ePrint Archive, Report 2021/702. https://ia.cr/2021/702. (Cited on page 115.)

[62] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1292–1303, 2016. (Cited on page 276.)

[63] J. Brendel, M. Fischlin, F. Günther, and C. Janson. PRF-ODH: Relations, instantiations, and impossibility results. In J. Katz and H. Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 651–681. Springer, Heidelberg, Aug. 2017. (Cited on page 52, 53, 55, 62, 113.)

[64] J. Brickell and V. Shmatikov. Efficient anonymity-preserving data collection. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 76–85, 2006. (Cited on page 257.)

[65] C. Brzuska, M. Fischlin, B. Warinschi, and S. C. Williams. Composability of Bellare-Rogaway key exchange protocols. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *ACM CCS 2011*, pages 51–62. ACM Press, Oct. 2011. (Cited on page 157.)

[66] R. Canetti. Universally composable security. *J. ACM*, 67(5), sep 2020. (Cited on page 254.)

[67] R. Canetti and H. Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In B. Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 453–474. Springer, Heidelberg, May 2001. (Cited on page 50, 55, 61, 111, 112.)

[68] R. Canetti and H. Krawczyk. Security analysis of IKE's signature-based key-exchange protocol. In M. Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 143–161. Springer, Heidelberg, Aug. 2002. http://eprint.iacr.org/2002/120/. (Cited on page xiv, 50, 51, 52, 53, 54, 55, 57, 108, 109, 110, 111, 112.)

[69] K. Chalkias, F. Garillot, and V. Nikolaenko. Taming the many eddsas. In T. van der Merwe, C. Mitchell, and M. Mehrnezhad, editors, *Security Standardisation Research*, pages 67–90, Cham, 2020. Springer International Publishing. (Cited on page 217.)

[70] M.-S. Chen, A. Hülsing, J. Rijneveld, S. Samardjiska, and P. Schwabe. MQDSS specifications. NIST PQC Round 2 Submission, 2019. (Cited on page 23.)

[71] J. H. Cheon, S. Park, J. Lee, D. Kim, Y. Song, S. Hong, D. Kim, J. Kim, S.-M. Hong, A. Yun, J. Kim, H. Park, E. Choi, K. Kim, J.-S. Kim, and J. Lee. Lizard public key encryption. NIST PQC Round 1 Submission, 2017. (Cited on page 9, 20, 21.)

[72] K. Cohn-Gordon, C. Cremers, and L. Garratt. On post-compromise security. In *2016 Computer Security Foundations Symposium*, pages 164–178. IEEE, 2016. (Cited on page 61.)

[73] K. Cohn-Gordon, C. Cremers, K. Gjøsteen, H. Jacobsen, and T. Jager. Highly efficient key exchange protocols with optimal tightness. In A. Boldyreva and D. Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 767–797. Springer, Heidelberg, Aug. 2019. (Cited on page 3, 52, 53, 54, 55, 57, 61, 63, 76, 85, 117, 118, 120, 170.)

[74] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-Damgrard revisited: How to construct a hash function. In V. Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 430–448. Springer, Heidelberg, Aug. 2005. (Cited on page 2, 4, 10, 25, 212, 213, 216, 238, 240, 241.)

[75] H. Corrigan-Gibbs and D. Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 259–282, Boston, MA, Mar. 2017. USENIX Association. (Cited on page 5, 251, 253, 254, 255, 265, 266, 267, 270.)

[76] R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2003. (Cited on page 7, 44.)

[77] I. Damgrard. A design principle for hash functions. In G. Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 416–427. Springer, Heidelberg, Aug. 1990. (Cited on page 212, 215, 220.)

[78] G. Danezis, C. Fournet, M. Kohlweiss, and S. Zanella-Béguelin. Smart meter aggregation via secret-sharing. In *Proceedings of the first ACM workshop on Smart energy grid security*, pages 75–80, 2013. (Cited on page 257.)

[79] J.-P. D'Anvers, A. Karmakar, S. S. Roy, and F. Vercauteren. SABER: Mod-LWR based KEM. NIST PQC Round 2 Submission, 2019. (Cited on page 9, 20.)

[80] A. Davidson, P. Snyder, E. Quirk, J. Genereux, B. Livshits, and H. Haddadi. Star: Secret sharing for private threshold aggregation reporting. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 697–710, 2022. (Cited on page 257.)

[81] H. Davis, D. Diemert, F. Günther, and T. Jager. On the concrete security of TLS 1.3 PSK mode. In O. Dunkelman and S. Dziembowski, editors, *Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 - June 3, 2022, Proceedings, Part II*, volume 13276 of *Lecture Notes in Computer Science*, pages 876–906. Springer, 2022. (Cited on page .)

[82] H. Davis and F. Günther. Tighter proofs for the SIGMA and TLS 1.3 key exchange protocols. In *19th International Conference on Applied Cryptography and Network Security (ACNS 2021)*, 2021. (Cited on page 117, 118, 119, 120, 126, 137, 190, 193, 195.)

[83] H. Davis, C. Patton, M. Rosulek, and P. Schoppmann. Verifiable distributed aggreagtion functions. *Proc. Priv. Enhancing Technol.*, 2023(4), 2023. (Cited on page .)

[84] H. Davis, C. Patton, M. Rosulek, and P. Schoppmann. Verifiable distributed aggregation functions. Cryptology ePrint Archive, Paper 2023/130, 2023. (Cited on page 258.)

[85] L. de Castro and A. Polychroniadou. Lightweight, maliciously secure verifiable function secret sharing. In O. Dunkelman and S. Dziembowski, editors, *Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 - June 3, 2022, Proceedings, Part I*, volume 13275 of *Lecture Notes in Computer Science*, pages 150–179. Springer, 2022. (Cited on page 256, 276, 277, 278, 288.)

[86] G. Demay, P. Gaži, M. Hirt, and U. Maurer. Resource-restricted indifferentiability. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 664–683. Springer, Heidelberg, May 2013. (Cited on page 10, 25.)

[87] A. W. Dent. A designer's guide to KEMs. In K. G. Paterson, editor, *9th IMA International Conference on Cryptography and Coding*, volume 2898 of *LNCS*, pages 133–151. Springer, Heidelberg, Dec. 2003. (Cited on page 7, 14, 15.)

[88] D. Diemert and T. Jager. On the tight security of TLS 1.3: Theoretically-sound cryptographic parameters for real-world deployments. *Journal of Cryptology*, 2020. To appear. Available as Cryptology ePrint Archive, Report 2020/726. https://eprint.iacr.org/2020/726. (Cited on page 3, 55, 56, 57, 61, 63, 93, 111.)

[89] Y. Dodis, T. Ristenpart, and T. Shrimpton. Salvaging Merkle-Damgrard for practical applications. In A. Joux, editor, *EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 371–388. Springer, Heidelberg, Apr. 2009. (Cited on page 212, 213, 214, 234.)

[90] Y. Dodis, T. Ristenpart, J. Steinberger, and S. Tessaro. To hash or not to hash again? (In)differentiability results for H^2 and HMAC. Cryptology ePrint Archive, Report 2013/382, 2013. http://eprint.iacr.org/2013/382. (Cited on page 145.)

[91] Y. Dodis, T. Ristenpart, J. P. Steinberger, and S. Tessaro. To hash or not to hash again? (In)differentiability results for H^2 and HMAC. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 348–366. Springer, Heidelberg, Aug. 2012. (Cited on page 140, 144, 145.)

[92] B. Dowling, M. Fischlin, F. Günther, and D. Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol. *Journal of Cryptology*, 2021. To appear. Available as Cryptology ePrint Archive, Report 2020/1044. https://eprint.iacr.org/2020/1044. (Cited on page xiv, xv, 50, 51, 52, 53, 54, 55, 63, 91, 108, 109, 110, 112, 113, 116, 119, 120, 126, 131, 193, 196, 197.)

[93] B. Dowling, M. Fischlin, F. Günther, and D. Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In I. Ray, N. Li, and C. Kruegel, editors, *ACM CCS 2015*, pages 1197–1210. ACM Press, Oct. 2015. (Cited on page 3, 50, 55, 63, 91, 116, 119, 120, 157.)

[94] B. Dowling, M. Fischlin, F. Günther, and D. Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. Cryptology ePrint Archive, Report 2015/914, 2015. http://eprint.iacr.org/2015/914. (Cited on page 157.)

[95] B. Dowling, M. Fischlin, F. Günther, and D. Stebila. A cryptographic analysis of the TLS 1.3 draft-10 full and pre-shared key handshake protocol. Cryptology ePrint Archive, Report 2016/081, 2016. http://eprint.iacr.org/2016/081. (Cited on page 50, 55, 63, 91, 116, 120.)

[96] B. Dowling and D. Stebila. Modelling ciphersuite and version negotiation in the TLS protocol. In E. Foo and D. Stebila, editors, *ACISP 15*, volume 9144 of *LNCS*, pages 270–288. Springer, Heidelberg, June / July 2015. (Cited on page 120.)

[97] Y. Duan, J. Canny, and J. Zhan. {P4P}: Practical {Large-Scale}{Privacy-Preserving} distributed computation robust against malicious users. In *19th USENIX Security Symposium (USENIX Security 10)*, 2010. (Cited on page 257.)

[98] C. Dwork. Differential privacy. In M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, editors, *Automata, Languages and Programming*, pages 1–12, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. (Cited on page 287.)

[99] T. Elahi, G. Danezis, and I. Goldberg. Privex: Private collection of traffic statistics for anonymous communication networks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1068–1079, 2014. (Cited on page 257.)

[100] M. Fischlin and F. Günther. Multi-stage key exchange and the case of Google's QUIC protocol. In G.-J. Ahn, M. Yung, and N. Li, editors, *ACM CCS 2014*, pages 1193–1204. ACM Press, Nov. 2014. (Cited on page 56, 57, 61, 63, 112, 119, 120.)

[101] M. Fischlin and F. 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*, pages 60–75. IEEE, Apr. 2017. (Cited on page 50, 55, 63, 91, 116, 119, 120.)

[102] M. Fischlin, F. Günther, B. Schmidt, and B. Warinschi. Key confirmation in key exchange: A formal treatment and implications for TLS 1.3. In *2016 IEEE Symposium on Security and Privacy*, pages 452–469. IEEE Computer Society Press, May 2016. (Cited on page 131.)

[103] O. Garcia-Morchon and Z. Zhang. Round2: KEM and PKE based on GLWR. NIST PQC Round 1 Submission, 2017. (Cited on page 8, 17.)

[104] T. Geoghegan, C. Patton, E. Rescorla, and C. A. Wood. Distributed Aggregation Protocol for Privacy Preserving Measurement. Internet-Draft draft-ietf-ppm-dap-02, Internet Engineering Task Force, Sept. 2022. Work in Progress. (Cited on page 251, 253, 255, 266, 268, 287.)

[105] F. Giesen, F. Kohlar, and D. Stebila. On the security of TLS renegotiation. In A.-R. Sadeghi, V. D. Gligor, and M. Yung, editors, *ACM CCS 2013*, pages 387–398. ACM Press, Nov. 2013. (Cited on page 119.)

[106] N. Gilboa and Y. Ishai. Distributed point functions and their applications. In P. Q. Nguyen and E. Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, pages 640–658, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. (Cited on page 256.)

[107] D. K. Gillmor. Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS). RFC 7919, Aug. 2016. (Cited on page 204.)

[108] K. Gjøsteen and T. Jager. Practical and tightly-secure digital signatures and authenticated key exchange. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 95–125. Springer, Heidelberg, Aug. 2018. (Cited on page 52, 117, 120.)

[109] O. Goldreich. Two remarks concerning the Goldwasser-Micali-Rivest signature scheme. In A. M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 104–110. Springer, Heidelberg, Aug. 1987. (Cited on page 213, 218.)

[110] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, Apr. 1988. (Cited on page 64, 214, 223.)

[111] M. Green, W. Ladd, and I. Miers. A protocol for privately reporting ad impressions at scale. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, page 1591–1601, New York, NY, USA, 2016. Association for Computing Machinery. (Cited on page 251, 257.)

[112] C. G. Günther. An identity-based key-exchange protocol. In J.-J. Quisquater and J. Vandewalle, editors, *EUROCRYPT'89*, volume 434 of *LNCS*, pages 29–37. Springer, Heidelberg, Apr. 1990. (Cited on page 115.)

[113] F. Günther. *Modeling Advanced Security Aspects of Key Exchange and Secure Channel Protocols*. PhD thesis, Technische Universität Darmstadt, Darmstadt, Germany, 2018. http://tuprints.ulb.tu-darmstadt.de/7162/. (Cited on page 120, 157.)

[114] C. Guo, J. Katz, X. Wang, and Y. Yu. Efficient and secure multiparty computation from fixed-key block ciphers. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 825–841. IEEE, 2020. (Cited on page 259.)

[115] X. Guo, K. Yang, X. Wang, W. Zhang, X. Xie, J. Zhang, and Z. Liu. Half-tree: Halving the cost of tree expansion in cot and dpf. Cryptology ePrint Archive, Paper 2022/1431, 2022. (Cited on page 259.)

[116] M. Hamburg. Post-quantum cryptography proposal: ThreeBears. NIST PQC Round 2 Submission, 2019. (Cited on page 9, 22.)

[117] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). IETF RFC 2409 (Proposed Standard), 1998. (Cited on page 49, 72.)

[118] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In T. Kohno, editor, *USENIX Security 2012*, pages 205–220. USENIX Association, Aug. 2012. (Cited on page 211.)

[119] D. Hofheinz, K. Hövelmanns, and E. Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Y. Kalai and L. Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 341–371. Springer, Heidelberg, Nov. 2017. (Cited on page 7, 14, 15.)

[120] S. Hohenberger, S. Myers, R. Pass, et al. Anonize: A large-scale anonymous survey system. In *2014 IEEE Symposium on Security and Privacy*, pages 375–389. IEEE, 2014. (Cited on page 257.)

[121] A. Hülsing, J. Rijneveld, J. M. Schanck, and P. Schwabe. NTRU-HRSS-KEM: Algorithm specifications and supporting documentations. NIST PQC Round 1 Submission, 2017. (Cited on page 9, 22.)

[122] T. Humphries, R. Akhavan Mahdavi, S. Veitch, and F. Kerschbaum. Selective mpc: Distributed computation of differentially private key-value statistics. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1459–1472, 2022. (Cited on page 287.)

[123] IANIX. Things that use Ed25519. https://ianix.com/pub/ed25519-deployment.html. (Cited on page 211, 213.)

[124] J. J. P. III, D. Charles, D. Gilton, Y. H. Jung, M. Parsana, and E. Anderson. Masked lark: Masked learning, aggregation and reporting workflow, 2021. (Cited on page 253, 257.)

[125] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short pcps. In *Twenty-Second Annual IEEE Conference on Computational Complexity (CCC'07)*, pages 278–291, 2007. (Cited on page 260.)

[126] T. Jager, E. Kiltz, D. Riepel, and S. Schäge. Tightly-secure authenticated key exchange, revisited. In *EUROCRYPT 2021*, 2021. To appear. Available as Cryptology ePrint Archive, Report 2020/1279. https://eprint.iacr.org/2020/1279. (Cited on page 55, 120.)

[127] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk. On the security of TLS-DHE in the standard model. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 273–293. Springer, Heidelberg, Aug. 2012. (Cited on page 52, 53, 55, 62, 63, 119, 120.)

[128] T. Jager, J. Schwenk, and J. Somorovsky. On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption. In I. Ray, N. Li, and C. Kruegel, editors, *ACM CCS 2015*, pages 1185–1196. ACM Press, Oct. 2015. (Cited on page 120.)

[129] P. Jangir, N. Koti, V. B. Kukkala, A. Patra, B. R. Gopal, and S. Sangal. Vogue: Faster computation of private heavy hitters. Cryptology ePrint Archive, Paper 2022/1561, 2022. (Cited on page 257.)

[130] D. Jao, R. Azarderakhsh, M. Campagna, C. Costello, L. D. Feo, B. Hess, A. Jalali, B. Koziel, B. LaMacchia, P. Longa, M. Naehring, J. Renes, V. Soukharev, and D. Urbanik. Supersingular isogeny key encapsulation. NIST PQC Round 2 Submission, 2019. (Cited on page 9, 22.)

[131] M. Jawurek and F. Kerschbaum. Fault-tolerant privacy-preserving statistics. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 221–238. Springer, 2012. (Cited on page 257.)

[132] H. Jiang, Z. Zhang, L. Chen, H. Wang, and Z. Ma. IND-CCA-secure key encapsulation mechanism in the quantum random oracle model, revisited. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 96–125. Springer, Heidelberg, Aug. 2018. (Cited on page 7, 14, 15.)

[133] S. Josefsson and I. Liusvaara. Edwards-curve digital signature algorithm (EdDSA). RFC 8032, Jan. 2017. https://datatracker.ietf.org/doc/html/rfc8032. (Cited on page 211, 213.)

[134] J. Katz and R. Ostrovsky. Round-optimal secure two-party computation. In *Annual International Cryptology Conference*, pages 335–354. Springer, 2004. (Cited on page 279.)

[135] C. Kaufman (Ed.). Internet Key Exchange (IKEv2) Protocol. RFC 4306 (Proposed Standard), Dec. 2005. Obsoleted by RFC 5996, updated by RFC 5282. (Cited on page 72.)

[136] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401 (Proposed Standard), Nov. 1998. Obsoleted by RFC 4301, updated by RFC 3168. (Cited on page 49, 72.)

[137] E. Kiltz, D. Masny, and J. Pan. Optimal security proofs for signatures from identification schemes. In M. Robshaw and J. Katz, editors, *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 33–61. Springer, Heidelberg, Aug. 2016. (Cited on page 217.)

[138] H. Krawczyk. SIGMA: The "SIGn-and-MAc" approach to authenticated Diffie-Hellman and its use in the IKE protocols. In D. Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 400–425. Springer, Heidelberg, Aug. 2003. (Cited on page 3, 49, 53, 57, 72, 93.)

[139] H. Krawczyk. SIGMA: the 'SIGn-and-MAc' approach to authenticated Diffie-Hellman and its use in the IKE protocols, 2003. Full version. https://webee.technion.ac.il/~hugo/sigma-pdf.pdf. (Cited on page 72, 93.)

[140] H. Krawczyk. HMQV: A high-performance secure Diffie-Hellman protocol. Cryptology ePrint Archive, Report 2005/176, 2005. http://eprint.iacr.org/2005/176. (Cited on page 130.)

[141] H. Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In T. Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 631–648. Springer, Heidelberg, Aug. 2010. (Cited on page 56, 91, 124.)

[142] H. Krawczyk. A unilateral-to-mutual authentication compiler for key exchange (with applications to client authentication in TLS 1.3). In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016*, pages 1438–1450. ACM Press, Oct. 2016. (Cited on page 121.)

[143] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), Feb. 1997. Updated by RFC 6151. (Cited on page 124.)

[144] H. Krawczyk and P. Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869 (Informational), May 2010. (Cited on page 124.)

[145] H. Krawczyk, K. G. Paterson, and H. Wee. On the security of the TLS protocol: A systematic analysis. In R. Canetti and J. A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 429–448. Springer, Heidelberg, Aug. 2013. (Cited on page 119, 120.)

[146] H. Krawczyk, K. G. Paterson, and H. Wee. On the security of the TLS protocol: A systematic analysis. Cryptology ePrint Archive, Report 2013/339, 2013. http://eprint.iacr.org/2013/339. (Cited on page 119.)

[147] H. Krawczyk and H. Wee. The OPTLS protocol and TLS 1.3. In *2016 IEEE European Symposium on Security and Privacy*, pages 81–96. IEEE, Mar. 2016. (Cited on page 50, 63, 116.)

[148] K. Kursawe, G. Danezis, and M. Kohlweiss. Privacy-friendly aggregation for the smart-grid. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 175–191. Springer, 2011. (Cited on page 257.)

[149] B. A. LaMacchia, K. Lauter, and A. Mityagin. Stronger security of authenticated key exchange. In W. Susilo, J. K. Liu, and Y. Mu, editors, *ProvSec 2007*, volume 4784 of *LNCS*, pages 1–16. Springer, Heidelberg, Nov. 2007. (Cited on page 55, 61.)

[150] A. Langley, M. Hamburg, and S. Turner. Elliptic Curves for Security. RFC 7748 (Informational), Jan. 2016. (Cited on page 51, 204.)

[151] D. Lapidot and A. Shamir. Publicly verifiable non-interactive zero-knowledge proofs. In A. Menezes and S. A. Vanstone, editors, *Advances in Cryptology - CRYPTO '90, 10th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1990, Proceedings*, volume 537 of *Lecture Notes in Computer Science*, pages 353–365. Springer, 1990. (Cited on page 276, 279.)

[152] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter. Ron was wrong, whit is right. Cryptology ePrint Archive, Report 2012/064, 2012. http://eprint.iacr.org/2012/064. (Cited on page 211.)

[153] Y. Li, S. Schäge, Z. Yang, F. Kohlar, and J. Schwenk. On the security of the pre-shared key ciphersuites of TLS. In H. Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 669–684. Springer, Heidelberg, Mar. 2014. (Cited on page 119.)

[154] X. Lu, Y. Liu, D. Jia, H. Xue, J. He, and Z. Zhang. LAC: Lattice-based cryptosystems. NIST PQC Round 2 Submission, 2019. (Cited on page 9, 20.)

[155] U. M. Maurer. Abstract models of computation in cryptography (invited paper). In N. P. Smart, editor, *10th IMA International Conference on Cryptography and Coding*, volume 3796 of *LNCS*, pages 1–12. Springer, Heidelberg, Dec. 2005. (Cited on page 53, 63.)

[156] U. M. Maurer, R. Renner, and C. Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In M. Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 21–39. Springer, Heidelberg, Feb. 2004. (Cited on page 2, 7, 10, 25, 27, 43, 118, 134, 135, 136, 138, 139, 141, 213, 216, 233, 235, 237.)

[157] C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, A. Hauteville, O. Ruatta, J.-P. Tillich, and G. Zémor. ROLLO: Rank-ouroboros, LAKE, & LOCKER. NIST PQC Round 2 Submission, 2018. (Cited on page 9, 20, 21.)

[158] C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, and G. Zémor. Rank quasi-cyclic (RQC). NIST PQC Round 2 Submission, 2019. (Cited on page 9, 22.)

[159] C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. D. P. Gaborit, and E. P. G. Zémor. Hamming quasi-cyclic (HQC). NIST PQC Round 2 Submission, 2019. (Cited on page 9, 22.)

[160] L. Melis, G. Danezis, and E. De Cristofaro. Efficient private statistics with succinct sketches. *arXiv preprint arXiv:1508.06110*, 2015. (Cited on page 257.)

[161] A. Menezes and N. Smart. Security of signature schemes in a multi-user setting. *Designs, Codes and Cryptography*, 33(3):261–274, Nov. 2004. (Cited on page 64.)

[162] R. C. Merkle. A certified digital signature. In G. Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 218–238. Springer, Heidelberg, Aug. 1990. (Cited on page 212, 215, 220.)

[163] I. Mironov, O. Pandey, O. Reingold, and S. Vadhan. Computational differential privacy. In *Annual International Cryptology Conference*, pages 126–142. Springer, 2009. (Cited on page 287.)

[164] A. Mittelbach. Salvaging indifferentiability in a multi-stage setting. In P. Q. Nguyen and E. Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 603–621. Springer, Heidelberg, May 2014. (Cited on page 10, 25.)

[165] A. Mittelbach and M. Fischlin. *The Theory of Hash Functions and Random Oracles*. Springer Nature, Switzerland, 2021. (Cited on page 216, 238, 240, 241.)

[166] D. Molteni. Improving the WAF with machine learning. Cloudflare blog, 2022. (Cited on page 251.)

[167] D. Mouris, P. Sarkar, and N. G. Tsoutsos. PLASMA: Private, lightweight aggregated statistics against malicious adversaries with full security. Cryptology ePrint Archive, Paper 2023/080, 2023. https://eprint.iacr.org/2023/080. (Cited on page 258.)

[168] Mozilla. Origin Telemetry, 2022. (Cited on page 250, 251, 252.)

[169] D. M'Raïhi, D. Naccache, D. Pointcheval, and S. Vaudenay. Computational alternatives to random number generators. In S. E. Tavares and H. Meijer, editors, *SAC 1998*, volume 1556 of *LNCS*, pages 72–80. Springer, Heidelberg, Aug. 1999. (Cited on page 213, 218.)

[170] M. Naehrig, E. Alkim, J. W. Bos, L. Ducas, K. Easterbrook, B. LaMacchia, P. Longa, I. Mironov, V. Nikolaenko, C. Peikert, A. Raghunathan, and D. Stebila. FrodoKEM: Learning with errors key encapsulation. NIST PQC Round 2 Submission, 2019. (Cited on page 9, 22.)

[171] National Institute of Standards and Technology. FIPS PUB 180-4: Secure Hash Standard (SHS), 2012. (Cited on page 123.)

[172] National Institute of Standards and Technology. FIPS PUB 186-4: Digital Signature Standard (DSS), 2013. (Cited on page 51, 54.)

[173] National Institute of Standards and Technology. FIPS PUB 186-4: Digital Signature Standard (DSS), 2013. (Cited on page 204.)

[174] National Institute of Standards and Technology. Digital Signature Standard (DSS). FIPS PUB 186-5, Oct. 2019. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5-draft.pdf. (Cited on page 211, 213, 214.)

[175] G. Neven, N. P. Smart, and B. Warinschi. Hash function requirements for schnorr signatures. *Journal of Mathematical Cryptology*, 3(1):69–87, 2009. (Cited on page 217.)

[176] NIST. Post-Quantum Cryptography Standardization Process. https://csrc.nist.gov/projects/post-quantum-cryptography. (Cited on page 6, 7.)

[177] NIST. Federal Information Processing Standard 202, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, Aug 2015. (Cited on page 8.)

[178] NIST. PQC Standardization Process: Second Round Candidate Announcement. https://csrc.nist.gov/news/2019/pqc-standardization-process-2nd-round-candidates, Jan. 2019. (Cited on page 7.)

[179] K. Ohta and T. Okamoto. On concrete security treatment of signatures derived from identification. In H. Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 354–369. Springer, Heidelberg, Aug. 1998. (Cited on page 217.)

[180] T. Okamoto and D. Pointcheval. The gap-problems: A new class of problems for the security of cryptographic schemes. In K. Kim, editor, *PKC 2001*, volume 1992 of *LNCS*, pages 104–118. Springer, Heidelberg, Feb. 2001. (Cited on page 62.)

[181] C. Patton and T. Shrimpton. Quantifying the security cost of migrating protocols to practice. In D. Micciancio and T. Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020*, pages 94–124, Cham, 2020. Springer International Publishing. (Cited on page 287.)

[182] T. Plantard. Odd manhattan's algorithm specifications and supporting documentation. NIST PQC Round 1 Submission, 2017. (Cited on page 9, 20.)

[183] D. Pointcheval and J. Stern. Security proofs for signature schemes. In U. M. Maurer, editor, *EUROCRYPT'96*, volume 1070 of *LNCS*, pages 387–398. Springer, Heidelberg, May 1996. (Cited on page 2.)

[184] D. Pointcheval and J. Stern. Security arguments for digital signatures and blind signatures. *Journal of Cryptology*, 13(3):361–396, June 2000. (Cited on page 4, 214, 215, 217, 218, 230, 237.)

[185] R. A. Popa, A. J. Blumberg, H. Balakrishnan, and F. H. Li. Privacy and accountability for location-based aggregate statistics. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 653–666, 2011. (Cited on page 257.)

[186] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard), Aug. 2018. (Cited on page 3, 49, 53, 54, 57, 72, 90, 114, 116, 200, 201, 202, 203, 206, 207, 208.)

[187] T. Ristenpart, H. Shacham, and T. Shrimpton. Careful with composition: Limitations of the indifferentiability framework. In K. G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 487–506. Springer, Heidelberg, May 2011. (Cited on page 10, 13, 25, 27, 32, 43, 134, 135, 213.)

[188] E. Roth, D. Noble, B. H. Falk, and A. Haeberlen. Honeycrisp: Large-scale differentially private aggregation without a trusted core. In *Proceedings of the 27th ACM Symposium*

on Operating Systems Principles, SOSP '19, page 196–210, New York, NY, USA, 2019. Association for Computing Machinery. (Cited on page 287.)

[189] T. Saito, K. Xagawa, and T. Yamakawa. Tightly-secure key-encapsulation mechanism in the quantum random oracle model. In J. B. Nielsen and V. Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 520–551. Springer, Heidelberg, Apr. / May 2018. (Cited on page 7, 14, 15.)

[190] C.-P. Schnorr. Efficient identification and signatures for smart cards. In G. Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 239–252. Springer, Heidelberg, Aug. 1990. (Cited on page 4.)

[191] C.-P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, Jan. 1991. (Cited on page 213, 215, 229.)

[192] P. Schoppmann, L. Vogelsang, A. Gascón, and B. Balle. Secure and scalable document similarity on distributed databases: Differential privacy to the rescue. *Proceedings on Privacy Enhancing Technologies*, 2:209–229, 2020. (Cited on page 287.)

[193] M. Seo, J. H. Park, D. H. Lee, S. Kim, and S.-J. Lee. Proposal for NIST post-quantum cryptography standard: EMBLEM and R.EMBLEM. NIST PQC Round 1 Submission, 2017. (Cited on page 9, 22.)

[194] N. Shah. The challenges of inspecting encrypted network traffic. https://www.fortinet.com/blog/industry-trends/keeping-up-with-performance-demands-of-encrypted-web-traffic, 2020. (Cited on page 1.)

[195] V. Shoup. Lower bounds for discrete logarithms and related problems. In W. Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 256–266. Springer, Heidelberg, May 1997. (Cited on page 53, 63, 72, 217.)

[196] N. P. Smart, M. R. Albrecht, Y. Lindell, E. Orsini, V. Osheter, K. G. Paterson, and G. Peer. LIMA: A PQC encryption scheme. NIST PQC Round 1 Submission, 2017. (Cited on page 9, 22.)

[197] R. Steinfeld, A. Sakzad, and R. K. Zhao. Titanium: Proposal for a NIST post-quantum public-key encryption and KEM standard. NIST PQC Round 1 Submission, 2017. (Cited on page 9, 20.)

[198] E. Taubeneck, M. Thomson, B. Savage, B. Case, D. Masny, and R. Jain. Ipa end to end protocol. Proposal submitted to the PATCG working group of the W3, 2022. (Cited on page 253.)

[199] K. Yoneyama, S. Miyagawa, and K. Ohta. Leaky random oracle. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 92(8):1795–1807, 2009. (Cited on page 213, 214.)

[200] Y. Zhao, Z. Jin, B. Gong, and G. Sui. A modular and systematic approach to key establishment and public-key encryption based on LWE and its variants. NIST PQC Round 1 Submission, 2017. (Cited on page 9, 20.)