Diss. ETH No. 25512

Composable Anonymous Credentials from Global Random Oracles

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

MANU DRIJVERS

Master of Science, Radboud University
born 07.02.1991
citizen of the Netherlands

accepted on the recommendation of

Prof. Dr. Srdjan Čapkun, examiner
Dr. Jan Camenisch, co-examiner
Prof. Dr. Emiliano De Cristofaro, co-examiner
Prof. Dr. Alessandra Scafuro, co-examiner

2018

Abstract

Authentication is a key aspect of digital security. However, users must authenticate frequently and are often asked to reveal more information than neccessary in the process. Not only does this hurt the privacy of users, it also negatively impacts security, e.g., by increasing the risk of identity theft. Anonymous credentials allow for privacy-friendly authentication, by revealing only minimal information, and by guaranteeing unlinkability of multiple authentications. The most efficient anonymous credential schemes today work in the so-called random-oracle model, in which a hash function is idealized as an oracle implementing a random function. In this thesis, we work towards *composable* anonymous credentials from random oracles, which means that the security remains in tact if we run an instance of this protocol alongside many other protocols. As authentication is typically just one building block of a larger system, composability is a very important property.

First, we investigate the power of *global* random oracles in the generalized universal composability framework. Global random oracles capture the setting in which a single idealized hash function can be used by many different protocols. Consequently, a protocol secure with a global random oracle avoids the unreasonable requirement of an idealized hash function specific to every protocol instance. So far, this seemed to come with a price, and protocols proven secure w.r.t. global random oracles are much less efficient than their counterparts from local random oracles. We show that global random oracles are much more powerful than previously known, by proving that some of the most practical and efficient primitives known can be proven secure with respect to different formulations of global random oracles, without losing efficiency compared to local random oracles.

Second, building on our first set of results, we construct the first composable *delegatable* anonymous credential scheme, which also offers support for attributes. In contrast with basic anonymous credentials, where the user must reveal the identity of the credential issuer to authenticate, delegatable credentials can support a hierarchy of issuers, much like the current public-key infrastructure. A user can authenticate while only revealing the identity of the root issuer, this way preserving the privacy of users in settings with hierarchical issuance of credentials.

Third, we turn to Direct Anonymous Attestation (DAA), which is a variation of basic anonymous credentials where the user has a secure device, such as a Trusted Platform Module (TPM), that holds a part of its secret key. DAA is deployed in practice to attest that a system with a TPM is in a secure state. We show that existing security models have shortcomings and present a new formal security model. Our model protects the privacy of the user even if the TPM is corrupt, removing the need to trust a piece of hardware for privacy. We then present the first composable DAA protocol that is efficient and satisfies our strong security notion, again building on our results on global random oracles.

Overall, the results of this thesis make it easier to design privacy-friendly systems that build on top of anonymous credentials, by defining different notions of anonymous credentials in a composable manner, presenting efficient realizations, and by removing the need for idealized hash functions specific to each protocol instance.

Zusammenfassung

Die Authentifizierung von Nutzern ist ein zentraler Aspekt im Kontext der digitalen Sicherheit und eine häufige Anforderung in einer Vielzahl von Anwendungen. Nutzer müssen sich häufig authentifizieren und geben dabei oft mehr Informationen preis als notwendig. Während dies eine klare Verletzung der Privatsphähre der Nutzer darstellt, bringt es auch andere Sicherheitsrisiken, wie zum Beispiel ein erhöhtes Risiko für Identitätsdiebstahl, mit sich.

Anonyme Berechtigungsnachweise (*anonymous credentials*) geben Nutzern die Möglichkeit sich zu authentifizieren und dabei nur ein Minimum an Information preiszugeben. Daneben garantieren sie, dass mehrere Authentifizierungsvorgänge des selben Nutzers nicht miteinander in Verbindung gebracht werden können. Die effizientesten Protokolle für anonyme Berechtigungsnachweise arbeiten heutzutage in dem sogenannten *Zufallsorakel-Modell*, wo eine Hashfunktion als idealisiertes Orakel, welches eine echte Zufallsfunktion implementiert, betrachtet wird.

In dieser Dissertation legen wir den Fokus auf *zusammensetzbare* anonyme Berechtigungsnachweise im Zufallsorakel-Modell. Zusammensetzbarkeit ist eine wichtige Eigenschaft kryptographischer Protokolle und stellt sicher, dass die Sicherheitsgarantien erhalten bleiben, selbst wenn das Protokoll parallel mit anderen Protokollen läuft. Nachem Authentifizierung typischerweise nur ein Baustein in größeren Systemen ist, ist die Zusammensetzbarkeit eine sehr wichtige Eigenschaft.

Wir beginnen damit die Möglichkeiten zu untersuchen, welche *globale Zufallsorakel* im Rahmen von allgemeiner universeller Zusammensetzbarkeit bieten. Globale Zufallsorakel adressieren das Szenario, in welchem eine einzelne idealisierte Hashfunktion – das globale Zufallsorakel – von vielen verschiedenen Protokollen genutzt werden kann. Folglich umgeht ein Protokoll, welches unter Verwendung eines globa-

len Zufallsorakels beweisbar sicher ist, die unrealistische Anforderung einer spezifischen idealisierten Hashfunktion für jede einzelne Instanz eines Protokolls. Bisher schien das einen hohen Preis zu haben und Protokolle, die sich in Bezug auf globale Zufallsorakel als sicher erwiesen haben, waren weniger effizient als entsprechende Protokolle die auf lokalen Zufallsorakeln basierten. Wir zeigen, dass globale Zufallsorakel mächtiger sind als bisher angenommen, indem wir nachweisen, dass einige der praktischsten und effizientesten bekannten Primitive hinsichtlich verschiedener Formulierungen des globalen Zufallsorakels als sicher bewiesen werden können, ohne dass Effizienz, verglichen mit lokalen Zufallsorakeln, verloren geht.

Aufbauend auf die oben genannten Ergebnisse konstruieren wir das erste zusammensetzbare Protokoll, das *delegierbare* anonyme Berechtigungsnachweise realisiert und zusätzlich Attribute unterstützt. Im Vergleich zu konventionellen anonymen Berechtigungsnachweisen, bei welchen der Nutzer die Identität des Ausstellers offenlegen muss um sich zu authentifizieren, unterstützen delegierbare anonyme Berechtigungsnachweise eine Ausstellungshierarchie, ähnlich zur gegenwärtigen Public-Key Infrastruktur. Ein Benutzer kann sich authentifizieren, indem er nur die Identität des Urausstellers angibt, während die Privatsphäre der Benutzer in der Ausstellungshierarchie gewahrt bleibt.

Daneben betrachten wir direkte anonyme Bescheinigungen (*Direct Anonymous Attestation*, DAA) welche als eine Variation von anonymen Berechtigungsnachweisen gesehen werden können. In der Praxis wird DAA zur Bestätigung eines sichern Systemstatus verwendet. Dabei hat der Benutzer eine vertraute Ausführungsumgebung, wie zum Beispiel ein Trusted Platform Module (TPM), zur Verügung, welche einen Teil seines geheimen Schlüssels enthält. Wir zeigen, dass die bestehenden Sicherheitsmodelle unzulänglich sind und präsentieren ein neues formales Sicherheitsmodell für DAA. Unser Modell bewahrt die Privatsphäre des Nutzers, selbst wenn das TPM korrumpiert ist. Es beseitigt daher die Notwendigkeit einem Chip bezüglich der Wahrung der Privatsphäre seiner Nutzer zu vertrauen. Zusätzlich präsentieren wir das erste zusammensetzbare DAA-Protokoll, welches gleichzeitig effizient ist und unser starkes Sicherheitsmodell erfüllt. Wir bauen dabei auf unsere Ergebnisse im Rahmen globaler Zufallsorakel auf.

Zusammenfassend wird es durch die in dieser Arbeit präsentierten Ergebnisse einfacher privatsphährenschonende Systeme, die auf anonymen Berechtigungsnachweisen basieren, zu entwerfen. Zum einen durch zusammensetzbare Definitionen von verschiedenen Varianten von anonymen Berechtigungshinweisen und deren effiziente Realisierungen. Zum anderen durch das Entfernen der Anforderung eine spezifische idealisierte Hashfunktion je Protokollinstanz verwenden zu müssen.

Acknowledgments

First and foremost, I want to express my gratitude to my supervisor Dr. Jan Camenisch, for teaching me the foundations of cryptography and table soccer. It has been a fantastic experience working under his guidance, Jan's experience and intuition provided a seemingly endless stream of topics to work on, while motivating me with his optimism.

I am truly indebted to my supervisor Prof. Srdjan Čapkun for providing academic advice and guidance where needed while also giving me the space to work at IBM Research. I thank Prof. Alessandra Scafuro and Prof. Emiliano De Cristofaro for co-examining this thesis and for their valuable feedback.

I thank my coauthors Dan Boneh, Liqun Chen, Maria Dubovitskaya, Tommaso Gagliardoni, Jan Hajny, Anja Lehmann, and Gregory Neven, it was a pleasure and honor to work with you. I also thank the not-yet-mentioned group members and former members David Barrera, Cecilia Boschini, Robert Enderlein, Patrick Hough, Anna Kaplan, Mark Korondi, Stephan Krenn, Vadim Lyubashevsky, Michael Osborne, Rafael del Pino, Franz-Stefan Preiss, Alfredo Rial, Kai Samelin, Gregor Seiler, Dieter Sommer, and Patrick Towa, for the lunch conversations, becoming my climbing buddies, for the fun BBQs, the hikes, and for contributing to the great atmosphere at the lab. Special thanks to Cecilia, it's been great fun sharing the office for four years.

I am grateful to Svenja for her love and patience, in particular during the writing of this thesis. Finally, I thank my parents and brother for their support and for helping me stay on track in difficult times.

Manu Drijvers
Zurich, November 2018.

Contents

1 Introduction **1**

2 Preliminaries **9**
- 2.1 Notation 9
- 2.2 Computational Problems 10
- 2.3 Universal Composability Framework 12
- 2.4 Zero-Knowledge Proofs 18

3 Composable Security with Global Random Oracles **21**
- 3.1 Introduction 21
- 3.2 Strict Random Oracle 27
- 3.3 Programmable Global Random Oracle 35
- 3.4 Restricted Programmable Global Random Oracles . . . 42
- 3.5 Unifying the Different Global Random Oracles 53

4 Delegatable Anonymous Credentials **57**
- 4.1 Introduction 57
- 4.2 Definition of Delegatable Credentials 62
- 4.3 Building Blocks 66
- 4.4 A Generic Construction for Delegatable Credentials . . 70
- 4.5 A Concrete Instantiation using Pairings 89

5 Anonymous Attestation **99**
- 5.1 Introduction 99
- 5.2 Issues in Existing Security Definitions 108
- 5.3 A Security Model for DAA with Optimal Privacy 113
- 5.4 Insufficiency of Existing DAA Schemes 122
- 5.5 Building Blocks 123
- 5.6 Construction 135
- 5.7 Concrete Instantiation and Efficiency 176

6 Concluding Remarks **183**

Curriculum Vitae **203**

List of Figures

2.1 Ideal authenticated channel functionality $\mathcal{F}_{\mathsf{auth}}$. 15
2.2 Ideal secure message transmission functionality $\mathcal{F}^{\mathcal{L}}_{\mathsf{smt}}$. . . 15
2.3 The special authenticated communication functionality $\mathcal{F}_{\mathsf{auth}*}$. 16
2.4 Ideal certification authority functionality $\mathcal{F}_{\mathsf{ca}}$. 17
2.5 The ideal common reference string functionality $\mathcal{F}^{D}_{\mathsf{crs}}$. . . 17
2.6 The ideal commitment functionality $\mathcal{F}_{\mathsf{com}}$. 18
2.7 The signature functionality $\mathcal{F}_{\mathsf{sig}}$. 19
2.8 The PKE functionality $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$ with leakage function $\mathcal{L}$. . . 20

3.1 The strict global random oracle functionality $\mathcal{G}_{\mathsf{sRO}}$ that does not give any extra power to anyone. 26
3.2 The UC experiment with a local random oracle (a) and the EUC experiment with a global random oracle (b). . 27
3.3 Reduction $\mathcal{B}$ from a real-world adversary $\mathcal{A}$ and a black-box environment $\mathcal{Z}$, simulating all the ideal functionalities (even the global ones) and playing against an external challenger $\mathcal{C}$. 28
3.4 The programmable global random oracle functionality $\mathcal{G}_{\mathsf{pRO}}$. 36
3.5 Public-key encryption scheme secure against adaptive attacks [CLNS17] based on one-way permutation OWTP and encoding function $(\mathsf{EC}, \mathsf{DC})$. 37
3.6 The EUC simulator $\mathcal{S}$ for protocol π_{PKE}. 39
3.7 The oracle programming routine `Program` 40
3.8 The global random-oracle functionalities $\mathcal{G}_{\mathsf{roRO}}$, $\mathcal{G}_{\mathsf{rpRO}}$, and $\mathcal{G}_{\mathsf{rpoRO}}$ with restricted observability, restricted programming, and combined restricted observability and programming, respectively. 44
3.9 Relations between different notions of global random oracles. 54

4.1 Ideal functionality for delegatable credentials with attributes $\mathcal{F}_{\mathsf{dac}}$ 63
4.2 Our Generic Construction: Delegation 71
4.3 Ideal functionality for GAME 3 in the proof of Theorem 8 77
4.4 Simulator for GAME 3 in the proof of Theorem 8 78

4.5 Ideal functionality for GAME 4 in the proof of Theorem 8 79
4.6 Simulator for GAME 4 in the proof of Theorem 8 80
4.7 Ideal functionality for GAME 5 in the proof of Theorem 8 81
4.8 Simulator for GAME 5 in the proof of Theorem 8 82
4.9 Ideal functionality for GAME 6 in the proof of Theorem 8 83
4.10 Simulator for GAME 6 in the proof of Theorem 8 84
4.11 Ideal functionality for GAME 7 in the proof of Theorem 8 85
4.12 Simulator for GAME 7 in the proof of Theorem 8 86
4.13 Ideal functionality for GAME 8 in the proof of Theorem 8 87
4.14 Simulator for GAME 8 in the proof of Theorem 8 88
4.15 Efficient instantiation of the NIZK used to generate attribute tokens (witness underlined for clarity). 90
4.16 Pseudocode for efficiently computing attribute tokens. . . 92
4.17 Pseudocode for efficiently verifying attribute tokens. . . 94

5.1 The Setup and Join interfaces of our ideal functionality $\mathcal{F}_{\mathsf{pdaa}}$ for DAA with optimal privacy. 114
5.2 The Sign, Verify, and Link interfaces of our ideal functionality $\mathcal{F}_{\mathsf{pdaa}}$ for DAA with optimal privacy. 115
5.3 Modeling of corruption in the real world. 121
5.4 Modeling of corruption in the ideal world. 121
5.5 Unforgeability experiment for signatures on encrypted messages. 126
5.6 Unforgeability-1 (1st signer is corrupt) and unforgeability-2 (2nd signer is corrupt) experiments. 130
5.7 Key-hiding experiment for split signatures. 131
5.8 Key-uniqueness experiment for split signatures. 131
5.9 Signature-uniqueness experiment for split signatures. . . 132
5.10 $\mathcal{F}$ for GAME 3 147
5.11 Simulator for GAME 3 148
5.12 $\mathcal{F}$ for GAME 4 149
5.13 Simulator for GAME 4 150
5.14 $\mathcal{F}$ for GAME 5 151
5.15 Simulator for GAME 5 152
5.16 $\mathcal{F}$ for GAME 6 153
5.17 First part of Simulator for GAME 6 154
5.18 Second part of Simulator for GAME 6 155
5.19 Third part of Simulator for GAME 6 156
5.20 $\mathcal{F}$ for GAME 7 157
5.21 Simulator for GAME 7 158

5.22 $\mathcal{F}$ for GAME 8 159
5.23 Simulator for GAME 8 160
5.24 $\mathcal{F}$ for GAME 9 161
5.25 Simulator for GAME 9 162
5.26 $\mathcal{F}$ for GAME 10 163
5.27 Simulator for GAME 10 164
5.28 $\mathcal{F}$ for GAME 11 165
5.29 Simulator for GAME 11 165
5.30 $\mathcal{F}$ for GAME 12 166
5.31 Simulator for GAME 12 166
5.32 $\mathcal{F}$ for GAME 13 167
5.33 Simulator for GAME 13 167
5.34 $\mathcal{F}$ for GAME 14 168
5.35 Simulator for GAME 14 168
5.36 $\mathcal{F}$ for GAME 15 169
5.37 Simulator for GAME 15 169

List of Tables

4.1 Performace of our delegatable credentials. 95
4.2 Performance measurements of presenting and verifying Level-2 credentials, and our estimated timings following the computation of Table 4.1. No attributes are disclosed. 97

5.1 Efficiency of our concrete DAA scheme ($n\mathbb{G}$ indicates n exponentiations in group $\mathbb{G}$, and nP indicates n pairing operations). 181

Chapter 1

Introduction

Our society is becoming increasingly digital: Many of our purchases are now placed online, email and messenger applications on smartphones replace traditional post, and perhaps cryptocurrencies will replace traditional money in the near future. Strong authentication is a cornerstone of this society. We want to be sure that we buy goods from an authentic manufacturer, we communicate with the intended person, or that we transfer money to the intended receiver. Unfortunately, users are often asked to reveal far too much information. This trend of sharing too much personal information is problematic for multiple reasons. First, requiring users to share personal information harms users' privacy. While every individual transaction may reveal little about a user's habits, malicious service providers may pool their combined data, which may paint a detailed picture of the user's lifestyle. Second, an attacker will benefit from any personal information it can obtain. With sufficient information about a user, one can often act in their name. This can happen through account recovery mechanisms, or even by just stating one's social security number. An attacker can then perform a so-called identity theft, impersonating a victim and performing actions, such as taking loans, in the victim's name. Furthermore, personal information enables an attacker to pretend to be a victim's co-worker, tricking the victim into voluntarily sharing private data and business secrets. This shows that sharing personal information does not only harm the privacy of users but also increases the attack surface of digital systems.

In this thesis, we will work towards enabling privacy-friendly authentication. Anonymous Credentials [Cha85] are a cryptographic tool that allows users to authenticate while disclosing minimal information. An issuer can issue anonymous credentials to users, certifying certain attributes of the user in the credential. The user can then authenticate by proving possession of certain attributes as certified in its credential. The key feature is that the user can choose which attributes from a credential to reveal to a verifier, and the verifier can cryptographically verify the disclosed part. The verifier does not learn anything more than that the disclosed attributes were certified by some issuer. Moreover, multiple authentications are unlinkable, meaning that a verifier cannot distinguish a returning user from a new user, and the user has anonymity in the set of all users possessing the disclosed attributes. Note that certain scenarios require unique identifiability, such as authenticating to see your personal email inbox, and in such settings anonymous credentials do not improve the user's privacy. Many scenarios, however, merely require proving that your attributes satisfy a certain predicate that many users fulfill, such as proving that you are of age, where anonymous credentials provide a large anonymity set. As anonymous credentials allow the user to control which personal information it reveals to whom, they are a primary ingredient for secure and privacy-preserving IT systems.

Hierarchical Issuance for Anonymous Credentials. Despite their strong privacy features, anonymous credentials do reveal the identity of the issuer, which, depending on the use case, still leaks information about the user such as the user's location, organization, or business unit. In practice, credentials are typically issued in a hierarchical manner and thus the chain of issuers will reveal even more information. For instance, consider governmental issued certificates such as drivers licenses, which are typically issued by a local authority whose issuing keys are then certified by a central authority. Thus, when a user presents her drivers license to prove her age, the local authority's public key will reveal her place of residence, which, together with other attributes such as the user's age, might help to identify the user.

Delegatable Anonymous Credentials (DAC), as formally introduced by Belenkiy et al. [BCC$^+$09], can solve this problem. They allow the owner of a credential to *delegate* her credential to another user, who, in turn, can delegate it further as well as present it to a verifier for authentication purposes. Only the identity (or rather the public key) of the initial delegator (root issuer) is revealed for verification. A few DAC constructions have been proposed [CL06, BCC+09, Fuc11, CKLM13a], but none is suitable for practical use due to their computational complexity and their lack of support for attributes.

Direct Anonymous Attestation. Direct Anonymous Attestation (DAA) [BCC04] allows a secure device, such as the Trusted Platform Module (TPM), that is embedded in a host computer to create attestations about the state of the host system. Such attestations, which can be seen as signatures on the current state under the TPM's secret key, convince a remote verifier that the system it is communicating with is running on top of certified hardware and is using the correct software. A crucial feature of DAA is that it performs such attestations in a privacy-friendly manner. That is, the user of the host system can choose to create attestations anonymously ensuring that her transactions are unlinkable and do not leak any information about the particular TPM being used.

DAA can be seen as a variation on anonymous credentials where the credential holder uses a secure device to store (a part of) the secret key. It is one of the most complex cryptographic protocols deployed in practice. The Trusted Computing Group (TCG), the industry standardization group that designed the TPM, standardized the first DAA protocol in the TPM 1.2 specification in 2004 [Tru04] and included support for multiple DAA schemes in the TPM 2.0 specification in 2014 [Tru14]. This sparked a strong interest in the research community in the security and efficiency of DAA schemes [BCC04, BCL08, BCL09, CMS08b, CMS08a, CMS09, Che09, CPS10, BFG+13b]. Unfortunately, existing schemes only offer anonymity properties if the TPM behaves honestly. This is a severe limitation, as verifying that a piece of hardware follows protocol is very difficult. Moreover, in spite of the large scale deployment and the long body of work on the subject, DAA still lacks a sound and comprehensive security definition, meaning it does not live up to the standards of provable security.

Provable Security. Cryptography has already been used for a long time to protect communication, long before computers were invented. In its early days, a cryptographic system was considered secure until somebody managed to break it, which is not very reassuring. As there are infinitely many ways to attack a cryptographic system, it is infeasible to simply "test" its security. In the 1970s, the cryptographic community started following a more scientific approach termed *provable security*, where a mathematical proof of security is required for any new cryptographic scheme. This would typically come in the form of a *reduction*: if there exists an attacker that can break the cryptographic scheme, we can use this attacker to break a computational problem that we assume to be infeasible to solve. A security proof requires a precise definition of security, which can come in two forms. In the first form, security is defined through *games* between an adversary and a challenger, and a protocol is considered secure if no adversary (typically with limited computing power) can win any of the games (with at least a certain probability). This approach considers the cryptographic protocol in isolation, meaning that care has to be taken when composing multiple cryptographic schemes together, e.g., by running multiple instances of the same protocol, or using one protocol as a sub-protocol of higher level protocol. In the second form, security is defined through an *ideal functionality*, which can be seen as a trusted third party that executes the task at hand in an ideal manner. This approach was introduced by Beaver [Bea92] and used by composability frameworks, such as the Universal Composability (UC) framework [Can01] or the Generalized UC framework [CDPW07], that can give stronger security guarantees than game-based security proofs. A proof in the UC framework guarantees that security is maintained under composition, i.e., one can run many instances of the same protocol, or use it as a building block in higher level protocols, without having to worry about affecting the security.

Practical Cryptographic Protocols. Cryptographic protocols are more likely to be used if they are *practical*, meaning efficient in computation and communication, and protocol can easily be deployed. Many interesting cryptographic tasks, such as composable commitments [CF01], are impossible to achieve without assuming some idealized component. One common example of such an idealized component is a common reference string (CRS), which is an honestly generated string sampled from a certain distribution that is available to all parties. To deploy a protocol that relies on a CRS, one typically runs a complex multiparty protocol to first generate a CRS, as was recently done to generate the parameters of the Zcash cryptocurrency [BCG$^+$15]. An alternative approach to bypassing impossibility results is the random-oracle model (ROM) [BR93], which is a truly random function that all parties have oracle access to. The ROM is designed to model an idealized cryptographic hash function, and hence, random oracle based protocols are typically deployed with a hash function that replaces the random oracle. This is a heuristic approach, and the protocol instantiated with a hash function is not formally proven secure. Examples of protocols that are secure with a random oracle but not with any hash function have been constructed [CGH98]. We can gain some confidence in this heuristic approach by the fact that random oracle based protocols deployed with a cryptographic hash function have been used extensively and no attacks have been found. Using random oracles we can obtain more efficient protocols than with a CRS, and we have a way of deploying them that has so far not led to attacks, making the random-oracle model (ROM) a promising approach to obtain practical cryptographic schemes. When composing multiple random oracle based protocols, each protocol typically requires its own random oracle, which means replacing it with a single hash function is unreasonable. Canetti, Jain, and Scafuro [CJS14] put forth the notion of a *global* random oracle, that allows protocols to make use of one globally available random oracle, which proves that the composition of random oracle based schemes instantiated with a hash function will not affect security.

Contributions and Outline. In this dissertation, we construct composable anonymous credentials from global random oracles. First, we advance the state-of-the-art in using global random oracles to construct composable cryptographic schemes. We propose multiple notions of global random oracle, and present positive results for each of the notions. Second, we use global random oracles to construct very efficient and composable delegatable anonymous credentials with attributes, which allow for hierarchichal issuance of anonymous credentials, much like the current public-key infrastructure works today. Third, we turn to Direct Anonymous Attestation. We identify flaws in previous security models and present a new notion of security, and a DAA protocol where the user's privacy is guaranteed even with a subverted TPM.

In a bit more detail, our contributions are the following:

Composable Security with Global Random Oracles. In Chapter 3, we discuss global random oracles, which capture the fact that random oracles can be shared by many different instances of one or more protocols. This builds on the work of Canetti, Jain, and Scafuro [CJS14], who put forth a global but non-programmable random oracle in the Generalized

UC (GUC) framework and showed that some basic cryptographic primitives with composable security can be efficiently realized in their model. Because their random-oracle functionality is non-programmable, there are many practical protocols that have no hope of being proved secure using it. We study alternative definitions of a global random oracle and, perhaps surprisingly, show that these allow one to prove GUC-security of existing, very practical realizations of a number of essential cryptographic primitives including public-key encryption, non-committing encryption, commitments, Schnorr signatures, and hash-and-invert signatures. Some of our results hold generically for any suitable scheme proven secure in the traditional random-oracle model, some hold for specific constructions only. Our results include many highly practical protocols, for example, the folklore commitment scheme $\mathcal{H}(m\|r)$ (where m is a message and r is the random opening information) which is far more efficient than the construction of Canetti et al. This chapter is based on the following publication:

- Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 280–312. Springer, Heidelberg, April / May 2018.

Delegatable Anonymous Credentials. In Chapter 4, we present the first hierarchical (or delegatable) anonymous credential system that is practical. To this end, we provide a surprisingly simple ideal functionality for delegatable credentials with attributes and present a generic construction that we prove secure in the GUC model. We then give a concrete instantiation using a recent pairing-based signature scheme by Groth [Gro15] and global random oracles, and describe a number of optimizations and efficiency improvements that can be made when implementing our concrete scheme. The latter might be of independent interest for other pairing-based schemes as well. Finally, we provide concrete performance figures. This chapter is based on the following publication:

- Jan Camenisch, Manu Drijvers, and Maria Dubovitskaya. Practical UC-secure delegatable credentials with attributes and their application to blockchain. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 683–699. ACM Press, October / November 2017.

Anonymous Attestation. In Chapter 5, we first point out many problems in existing security models for direct anonymous attestation. Then, we tackle the challenge of formally defining DAA and present a new ideal functionality $\mathcal{F}_{\mathsf{pdaa}}$ in the UC framework. In addition to the standard security properties such as unforgeability and non-frameability, $\mathcal{F}_{\mathsf{pdaa}}$ also captures *optimal privacy*, ensuring the privacy of honest hosts even when the TPM might try to break anonymity by deviating from the protocol. To this end, we capture subversion attacks in the UC framework.

Next, we discuss why existing protocols do not offer privacy when the TPM is corrupt and propose a new DAA protocol which achieves our strong security definition. Our protocol is constructed from generic building blocks which yields a more modular design. Shifting more responsibility to the host allows us to achieve optimal privacy, while also decreasing the computational burden on the TPM, which is usually the bottleneck in a DAA scheme.

This chapter is based on the following publication:

- Jan Camenisch, Manu Drijvers, and Anja Lehmann. Anonymous attestation with subverted TPMs. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 427–461. Springer, Heidelberg, August 2017.

This chapter also includes partial results form the following publications:

- Jan Camenisch, Manu Drijvers, and Anja Lehmann. Universally composable direct anonymous attestation. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016, Part II*, volume 9615 of *LNCS*, pages 234–264. Springer, Heidelberg, March 2016.

- Jan Camenisch, Manu Drijvers, and Anja Lehmann. Anonymous attestation using the strong diffie hellman assumption revisited. In Michael Franz and Panos Papadimitratos, editors, *Trust and Trustworthy Computing - 9th International Conference, TRUST 2016, Vienna, Austria, August 29-30, 2016, Proceedings*, volume 9824 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2016.

- Jan Camenisch, Liqun Chen, Manu Drijvers, Anja Lehmann, David Novick, and Rainer Urian. One TPM to bind them all: Fixing TPM 2.0 for provably secure anonymous attestation. In *2017 IEEE Symposium on Security and Privacy*, pages 901–920. IEEE Computer Society Press, May 2017.

Chapter 2

Preliminaries

This chapter introduces the required preliminaries, including notation, definitions, and computational problems. Moreover, it introduces the universal composability framework and how many useful primitives such as signatures and encryption can be captured in this model. Finally, it introduces the concept of zero-knowledge proofs.

2.1 Notation

Let $\mathbb{N}$ denote the natural numbers, $\mathbb{R}$ the real numbers, and $\mathbb{R}^+$ the non-negative real numbers. Let $\mathbb{Z}_q$ denote all integers modulo q. Let $\mathbb{Z}_q^*$ denote all integers modulo q that are coprime with q. We will use κ to denote the security parameter. Let 1^κ denote the κ-length string of ones. If S is a set, let $s \xleftarrow{\$} S$ denote sampling s uniformly at random from S.

2.1.1 Negligible functions and Indistinguishability

A function is called *negligible* if it is asymptotically smaller than the inverse of any polynomial.

Definition 1 (Negligible function [KL14]). *A function $f : \mathbb{N} \to \mathbb{R}^+$ is negligible if for every polynomial $p(\cdot)$ there exists an N such that for all integers $n > N$ it holds that $f(n) < \frac{1}{p(n)}$.*

Definition 2 (Overwhelming function)**.** *A function* $f : \mathbb{N} \to \mathbb{R}^+$ *is overwhelming if* $1 - f(x)$ *is negligible.*

Probability ensembles $\mathcal{X}$ and $\mathcal{Y}$ are computationally indistinguishable, denoted $\mathcal{X} \approx \mathcal{Y}$, if no efficient distinguisher has non-negligible success probability.

Definition 3 (Computational indistinguishability [KL14])**.** *Two probability ensembles* $\mathcal{X} = \{X_n\}_{n \in \mathbb{N}}$ *and* $\mathcal{Y} = \{Y_n\}_{n \in \mathbb{N}}$ *are computationally indistinguishable if for every probabilistic polynomial-time distinguisher* D *there exists a negligible function* f *such that*

$$\left| \Pr_{x \leftarrow X_n}[D(1^\kappa, x) = 1] - \Pr_{y \leftarrow Y_n}[D(1^\kappa, y) = 1] \right| \leq f(n).$$

2.1.2 Groups

We write $\mathbb{G} = \langle g \rangle$ to denote a group $\mathbb{G}$ generated by element g. The group operation is always written multiplicatively, and $1_\mathbb{G}$ denotes the identity element of $\mathbb{G}$. Let algorithm $\mathsf{GroupGen}$ on input the security parameter outputs $(\mathbb{G}, g, q)$, such that g generates $\mathbb{G}$ of prime order q, and q is of bitlength κ. Let $\mathbb{G}^*$ denote the elements of $\mathbb{G}$ that generate $\mathbb{G}$.

Bilinear Groups

Let $\mathsf{PairGen}$ be a bilinear group generator that takes as an input a security parameter 1^κ and outputs the descriptions of multiplicative groups $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathrm{t}, g_1, g_2, q, e)$ where $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_\mathrm{t}$ are groups of prime order q, generated by g_1, g_2, and $e(g_1, g_2)$ respectively. Moreover, we require e to be efficiently computable and be bilinear, meaning for any $(\alpha, \beta) \xleftarrow{\$} \mathbb{Z}_q$, $e(g_1^\alpha, g_2^\beta) = e(g_1, g_2)^{\alpha \cdot \beta}$.

2.2 Computational Problems

This section defines the hardness of certain computational problems, which we will assume in later chapters of this thesis.

Definition 4 (Discrete Logarithm (DL) Problem Hardness [KL14])**.** *The discrete logarithm problem is hard relative to* $\mathsf{GroupGen}$ *if for all* *probabilistic polynomial-time algorithms $\mathcal{A}$ there exists a negligible function f such that*

$$\Pr\left[(\mathbb{G}, g, q) \leftarrow \mathsf{GroupGen}(1^\kappa), h \xleftarrow{\$} \mathbb{G}, x \leftarrow \mathcal{A}(\mathbb{G}, g, q, h), g^x = h\right] \leq f(\kappa).$$

Definition 5 (Computational Diffie-Hellman (CDH) Problem Hardness)**.** *The computational Diffie-Hellman problem is hard relative to* GroupGen *if for all probabilistic polynomial-time algorithms $\mathcal{A}$ there exists a negligible function f such that*

$$\Pr\Big[(\mathbb{G}, g, q) \leftarrow \mathsf{GroupGen}(1^\kappa), (\alpha, \beta) \xleftarrow{\$} \mathbb{Z}_q, \\ y \leftarrow \mathcal{A}(\mathbb{G}, g, q, g^\alpha, g^\beta), y = g^{\alpha\cdot\beta}\Big] \leq f(\kappa).$$

Definition 6 (Decisional Diffie-Hellman (DDH) Problem Hardness [KL14])**.** *The decisional Diffie-Hellman problem is hard relative to group generator* GroupGen *if for all probabilistic polynomial-time algorithms $\mathcal{A}$ there exists a negligible function f such that*

$$\Pr\left[\mathcal{A}(\mathbb{G}, g, q, g^x, g^y, g^z) = 1\right] - \left[\mathcal{A}(\mathbb{G}, g, q, g^x, g^y, g^{xy}) = 1\right] \leq f(\kappa),$$

where in each case the probabilities are taken over the experiment in which $\mathsf{GroupGen}(1^\kappa)$ *outputs $(\mathbb{G}, g, q)$, and then uniform $x, y, z \xleftarrow{\$} \mathbb{Z}_q$ are chosen.*

Definition 7 (Computational co-Diffie-Hellman (co-CDH) Problem Hardness [BLS04])**.** *The computational co-Diffie-Hellman problem is hard relative to* PairGen *if for all probabilistic polynomial-time algorithms $\mathcal{A}$ there exists a negligible function f such that*

$$\Pr\Big[(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathrm{t}, g_1, g_2, q, e) \leftarrow \mathsf{PairGen}(1^\kappa), (\alpha, \beta) \xleftarrow{\$} \mathbb{Z}_q, \\ y \leftarrow \mathcal{A}(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathrm{t}, g_1, g_2, q, e, g_1^\alpha, g_1^\beta, g_2^\beta), y = g_1^{\alpha\cdot\beta}\Big] \leq f(\kappa).$$

Definition 8 (External Diffie-Hellman (XDH) Problem Hardness)**.** *The symmetric external Diffie-Hellman problem is hard relative to group generator* PairGen *if for all probabilistic polynomial-time algorithms $\mathcal{A}$ there exists a negligible function f such that*

$$\Pr\left[\mathcal{A}(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathrm{t}, g_1, g_2, q, e, g_1^x, g_1^y, g_1^z) = 1\right] - \\ \left[\mathcal{A}(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathrm{t}, g_1, g_2, q, e, g_1^x, g_1^y, g_1^{xy}) = 1\right] \leq f(\kappa),$$

where in each case the probabilities are taken over the experiment in which $\mathsf{PairGen}(1^\kappa)$ *outputs* $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, g_1, g_2, q, e)$*, and then uniform* $x, y, z \xleftarrow{\$} \mathbb{Z}_q$ *are chosen.*

Definition 9 (Symmetric External Diffie-Hellman (SXDH) Problem Hardness)**.** *The symmetric external Diffie-Hellman problem is hard relative to group generator* $\mathsf{PairGen}$ *if for* $b \in \{1, 2\}$ *for all probabilistic polynomial-time algorithms* $\mathcal{A}$ *there exists a negligible function* f *such that*

$$\Pr\left[\mathcal{A}(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, g_1, g_2, q, e, g_b^x, g_b^y, g_b^z) = 1\right] - \\ \left[\mathcal{A}(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, g_1, g_2, q, e, g_b^x, g_b^y, g_b^{xy}) = 1\right] \leq f(\kappa),$$

where in each case the probabilities are taken over the experiment in which $\mathsf{PairGen}(1^\kappa)$ *outputs* $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, g_1, g_2, q, e)$*, and then uniform* $x, y, z \xleftarrow{\$} \mathbb{Z}_q$ *are chosen.*

2.3 Universal Composability Framework

The universal composability (UC) framework [Can01,Can00] is a framework to define and prove the security of protocols. It follows the simulation-based security paradigm, meaning that security of a protocol is defined as the simulatability of the protocol based on an ideal functionality $\mathcal{F}$. In an imaginary ideal world, parties hand their protocol inputs to a trusted party running $\mathcal{F}$, where $\mathcal{F}$ by construction executes the task at hand in a secure manner. A protocol π is considered a secure realization of $\mathcal{F}$ if the real world, in which parties execute the real protocol, is indistinguishable from the ideal world. Namely, for every real-world adversary $\mathcal{A}$ attacking the protocol, we can design an ideal-world attacker (simulator) $\mathcal{S}$ that performs an equivalent attack in the ideal world. As the ideal world is secure by construction, this means that there are no meaningful attacks on the real-world protocol either.

One of the goals of UC is to simplify the security analysis of protocols, by guaranteeing secure composition of protocols and, consequently, allowing for modular security proofs. One can design a protocol π assuming the availability of an ideal functionality $\mathcal{F}'$, i.e., π is a $\mathcal{F}'$-hybrid protocol. If π securely realizes $\mathcal{F}$, and another protocol π' securely realizes $\mathcal{F}'$, then the composition theorem guarantees that π composed with π' (i.e., replacing π' with $\mathcal{F}'$) is a secure realization of $\mathcal{F}$.

Security is defined through an interactive Turing machine (ITM) $\mathcal{Z}$ that models the environment of the protocol and chooses protocol inputs to all participants. Let $\mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$ denote the output of $\mathcal{Z}$ in the real world, running with protocol π and adversary $\mathcal{A}$, and let $\mathrm{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ denote its output in the ideal world, running with functionality $\mathcal{F}$ and simulator $\mathcal{S}$. Protocol π securely realizes $\mathcal{F}$ if for every polynomial-time adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that for every environment $\mathcal{Z}$, $\mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{Z}} \approx \mathrm{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$.

2.3.1 Generalized Universal Composability.

A UC protocol using random oracles is modeled as a $\mathcal{F}_{\mathsf{RO}}$-hybrid protocol. Since an instance of a UC functionality can only be used by a single protocol instance, this means that every protocol instance uses its own random oracle that is completely independent of other protocol instances' random oracles. As the random-oracle model is supposed to be an idealization of real-world hash functions, this is not a very realistic model: Given that we only have a handful of standardized hash functions, it's hard to argue their independence across many protocol instances.

To address these limitations of the original UC framework, Canetti et al [CDPW07] introduced the Generalized UC (GUC) framework, which allows for shared "global" ideal functionalities (denoted by $\mathcal{G}$) that can be used by all protocol instances. Additionally, GUC gives the environment more powers in the UC experiment. Let $\mathrm{GEXEC}_{\pi,\mathcal{A},\mathcal{Z}}$ be defined as $\mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$, except that the environment $\mathcal{Z}$ is no longer constrained, meaning that it is allowed to start arbitrary protocols in addition to the challenge protocol π. Similarly, $\mathrm{GIDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ is equivalent to $\mathrm{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ but $\mathcal{Z}$ is now unconstrained. If π is a $\mathcal{G}$-hybrid protocol, where $\mathcal{G}$ is some shared functionality, then $\mathcal{Z}$ can start additional $\mathcal{G}$-hybrid protocols, possibly learning information about or influencing the state of $\mathcal{G}$. In GUC, protocol emulation and functionality realization are defined as follows.

Definition 10. *Protocol π GUC-emulates protocol φ if for every adversary $\mathcal{A}$ there exists an adversary $\mathcal{S}$ such that for all unconstrained environments $\mathcal{Z}$,* $\mathrm{GEXEC}_{\pi,\mathcal{A},\mathcal{Z}} \approx \mathrm{GEXEC}_{\varphi,\mathcal{S},\mathcal{Z}}$.

Definition 11. *Protocol π GUC-realizes ideal functionality $\mathcal{F}$ if for every adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that for all unconstrained environments $\mathcal{Z}$,* $\mathrm{GEXEC}_{\pi,\mathcal{A},\mathcal{Z}} \approx \mathrm{GIDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$.

GUC gives very strong security guarantees, as the unconstrained environment can run arbitrary protocols in parallel with the challenge protocol, where the different protocol instances might share access to global functionalities. However, exactly this flexibility makes it hard to reason about the GUC experiment. To address this, Canetti et al. also introduced Externalized UC (EUC). Typically, a protocol π uses many local hybrid functionalities $\mathcal{F}$ but only uses a single shared functionality $\mathcal{G}$. Such protocols are called $\mathcal{G}$-subroutine respecting, and EUC allows for simpler security proofs for such protocols. Rather than considering unconstrained environments, EUC considers $\mathcal{G}$-externally constrained environments. Such environments can invoke only a single instance of the challenge protocol, but can additionally query the shared functionality $\mathcal{G}$ through dummy parties that are not part of the challenge protocol. The EUC experiment is equivalent to the standard UC experiment, except that it considers $\mathcal{G}$-externally constrained environments: A $\mathcal{G}$-subroutine respecting protocol π EUC-emulates a protocol φ if for every polynomial-time adversary $\mathcal{A}$ there is an adversary $\mathcal{S}$ such that for every $\mathcal{G}$-externally constrained environment $\mathrm{EXEC}^{\mathcal{G}}_{\pi,\mathcal{A},\mathcal{Z}} \approx \mathrm{EXEC}^{\mathcal{G}}_{\varphi,\mathcal{S},\mathcal{Z}}$. Figure 3.2(b) depicts EUC-emulation and shows that this setting is much simpler to reason about than GUC-emulation: We can reason about this static setup, rather than having to imagine arbitrary protocols running alongside the challenge protocol. Canetti et al. [CDPW07] prove that showing EUC-emulation is useful to obtain GUC-emulation.

Theorem 1. *Let π be a $\mathcal{G}$-subroutine respecting protocol, then protocol π GUC-emulates protocol φ if and only if π $\mathcal{G}$-EUC-emulates φ.*

2.3.2 Ideal Functionalities

We now present some standard ideal functionalities that model common tasks. When specifying ideal functionalities, we will use some conventions for ease of notation. For a non-shared functionality with session id sid, we write "On input x from party $\mathcal{P}$", where it is understood the input comes from machine $(\mathcal{P}, \mathsf{sid})$. For shared functionalities, machines from any session may provide input, so we always specify both the party identity and the session identity of machines. If a functionality makes a *delayed output*, it means the adversary first receives the output, and only when the adversary indicates it allows the output, the output is sent to the party. In a public delayed output, the adversary receives the full output, whereas in a private delayed output, the

Functionality $\mathcal{F}_{\mathsf{auth}}$

1. On input $(\mathsf{SEND}, \mathsf{sid}, m)$ from a party $\mathcal{P}$, abort if $\mathsf{sid} \neq (\mathcal{S}, \mathcal{R}, \mathsf{sid}')$, Generate a public delayed output $(\mathsf{SENT}, \mathsf{sid}, m)$ to $\mathcal{R}$ and halt.

Figure 2.1: Ideal authenticated channel functionality $\mathcal{F}_{\mathsf{auth}}$.

Functionality $\mathcal{F}_{\mathsf{smt}}^{\mathcal{L}}$

1. On input $(\mathsf{SEND}, \mathsf{sid}, m)$ from a party $\mathcal{P}$, abort if $\mathsf{sid} \neq (\mathcal{S}, \mathcal{R}, \mathsf{sid}')$, send $(\mathsf{SEND}, \mathsf{sid}, \mathcal{L}(m))$ to the adversary. When the adversary allows, output $(\mathsf{SENT}, \mathsf{sid}, m)$ to $\mathcal{R}$ and halt.

Figure 2.2: Ideal secure message transmission functionality $\mathcal{F}_{\mathsf{smt}}^{\mathcal{L}}$.

adversary does not obtain the contents of the output. In some cases an ideal functionality requires immediate input from the adversary. In such cases we write “wait for input x from the adversary”, which is formally defined by Camenisch et al. [CEK^{+}16].

Authenticated Channels

Figure 2.1 depicts functionality $\mathcal{F}_{\mathsf{auth}}$, as defined by Canetti [Can00], sends a single message from a sender to a receiver in an authenticated manner.

Secure Message Transfer

Figure 2.2 depicts functionality $\mathcal{F}_{\mathsf{smt}}^{\mathcal{L}}$, as defined by Canetti [Can00], sends a single message to a receiver in a authenticated and confidential manner. The functionality is parameterized by a leakage function $\mathcal{L} : \{0,1\}^* \rightarrow \{0,1\}^*$ that leaks information about the transmitted message, e.g., the message length.

Special Authenticated Communication

We introduce a special authenticated channel functionality $\mathcal{F}_{\mathsf{auth}*}$ that sends an authenticated message from one party to another via a third party. This will model the authentication a TPM performs to an issuer, where messages are forwareded by an unauthenticated host. $\mathcal{F}_{\mathsf{auth}*}$ is depicted in Figure 2.3.

Functionality $\mathcal{F}_{\mathsf{auth}*}$

1. On input $(\mathsf{SEND}, \mathsf{sid}, ssid, m_1, m_2, F)$ from S. Check that $\mathsf{sid} = (S, R, \mathsf{sid}')$ for some R an output $(\mathsf{REPLACE1}, \mathsf{sid}, ssid, m_1, m_2, F)$ to the adversary.
2. On input $(\mathsf{REPLACE1}, \mathsf{sid}, ssid, m_2')$ from the adversary, output $(\mathsf{APPEND}, \mathsf{sid}, ssid, m_1, m_2')$ to F.
3. On input $(\mathsf{APPEND}, \mathsf{sid}, ssid, m_2'')$ from F, output $(\mathsf{REPLACE2}, \mathsf{sid}, ssid, m_1, m_2'')$ to the adversary.
4. On input $(\mathsf{REPLACE2}, \mathsf{sid}, ssid, m_2''')$ from the adversary, output $(\mathsf{SENT}, \mathsf{sid}, ssid, m_1, m_2''')$ to R.

Figure 2.3: The special authenticated communication functionality $\mathcal{F}_{\mathsf{auth}*}$.

Certification Authority

Ideal certification authority functionality $\mathcal{F}_{\mathsf{ca}}$, as defined by Canetti [Can04], allows parties to register data (such as a public key) in an authenticated manner, such that other users can look up this data knowing only the identity of the party. We extend $\mathcal{F}_{\mathsf{ca}}$ to allow one party to register multiple keys, i.e., we check $\mathsf{sid} = (P, \mathsf{sid}')$ for some sid' instead of checking $\mathsf{sid} = P$. $\mathcal{F}_{\mathsf{ca}}$ is depicted in Figure 2.4.

Common Reference String

Figure 2.5 depicts $\mathcal{F}_{\mathsf{crs}}^{D}$, the ideal common reference string functionality as defined by Canetti and Fischlin [CF01]. This functionality is parametrized by a distribution D, from which the string is sampled.

Commitments.

Figure 2.6 depics the ideal commitment functionality $\mathcal{F}_{\mathsf{com}}$, as defined by Canetti [Can00]. Party $\mathcal{C}$ first commits to a value, upon which receiver $\mathcal{R}$ learns that $\mathcal{C}$ chose a value. At a later point, $\mathcal{C}$ can reveal to $\mathcal{R}$ the value it committed to earlier.

Functionality $\mathcal{F}_{\mathsf{ca}}$

1. Upon receiving the first message $(\mathsf{Register}, \mathsf{sid}, v)$ from P, send $(\mathsf{Registered}, \mathsf{sid}, v)$ to the adversary; upon receiving $(\mathsf{sid}, \mathsf{ok})$ from the adversary, and if $\mathsf{sid} = (P, \mathsf{sid}')$, and this is the first request from P, then record the pair (P, v).
2. Upon receiving a message $(\mathsf{Retrieve}, \mathsf{sid})$ from party P', send $(\mathsf{Retrieve}, \mathsf{sid}, P')$ to the adversary, and wait for an ok from the adversary. Then, if there is a recorded pair (sid, v) output $(\mathsf{Retrieve}, \mathsf{sid}, \mathsf{v})$ to P'. Else output $(\mathsf{Retrieve}, \mathsf{sid}, \perp)$ to P'.

Figure 2.4: Ideal certification authority functionality $\mathcal{F}_{\mathsf{ca}}$.

Functionality $\mathcal{F}_{\mathsf{crs}}^{D}$

1. When receiving input $(\mathsf{crs}, \mathsf{sid})$ from party $\mathcal{P}$, look up a recorded value r. If there is no value r recorded then choose and record $r \xleftarrow{\$} \mathcal{D}$. Finally, send a public delayed output $(\mathsf{crs}, \mathsf{sid}, r)$ to $\mathcal{P}$.

Figure 2.5: The ideal common reference string functionality $\mathcal{F}_{\mathsf{crs}}^{D}$.

Signatures.

Figure 2.7 depicts the ideal signature functionality $\mathcal{F}_{\mathsf{sig}}$, as defined by Canetti [Can04]. The functionality lets the adversary choose signature values and verify signatures, but it uses internal lists to guarantee security properties. $\mathcal{F}_{\mathsf{sig}}$ keeps records of the honestly signed messages, and its verification interface rejects any purported signature on a message that was not honestly signed to guarantee unforgeability. It logs handled verification queries to guarantee consistency, and while generating a signature, it makes sure that this signature will be accepted by its verification interface, to enforce completeness.

Public-key Encryption.

Figure 2.8 depicts the ideal public-key encryption functionality $\mathcal{F}_{\mathsf{pke}}^{\mathcal{L}}$, as defined by Camenisch et al. [CLNS17]. Similar to $\mathcal{F}_{\mathsf{sig}}$, it lets the adversary create ciphertexts and provide decryptions, while guaranteeing

Functionality $\mathcal{F}_{\mathsf{com}}$

1. `Commit`: on input $(\mathsf{Commit}, \mathsf{sid}, x)$ from a party $\mathcal{C}$ proceed as follows.
    - Check that $\mathsf{sid} = (\mathcal{C}, \mathcal{R}, \mathsf{sid}')$.
    - Store x and generate public delayed output $(\mathsf{Receipt}, \mathsf{sid})$ to $\mathcal{R}$. Ignore subsequent Commit inputs.
2. `Open`: on input $(\mathsf{Open}, \mathsf{sid})$ from $\mathcal{C}$ proceed as follows.
    - Check that a committed value x is stored.
    - Generate public delayed output $(\mathsf{Open}, \mathsf{sid}, x)$ to $\mathcal{R}$.

Figure 2.6: The ideal commitment functionality $\mathcal{F}_{\mathsf{com}}$.

the desired security properties.

2.4 Zero-Knowledge Proofs

Feige, Fiat, and Shamir [FFS88] were the first to formalize the proof of knowledge, while the concept of zero-knowledge was introduced by Goldwasser et al. [GMR85]. When referring to the interactive proofs, one usually uses the notation introduced by Camenisch and Stadler [CS97] and formally defined by Camenisch, Kiayias, and Yung [CKY09]. For instance, $\mathsf{PK}\{(a, b, c) : Y = g_1^a H^b \wedge \tilde{Y} = \tilde{g_1}^a \tilde{H}^c\}$ denotes a "*zero-knowledge Proof of Knowledge of integers a, b, c such that* $Y = g_1^a H^b$ *and* $\tilde{Y} = \tilde{g_1}^a \tilde{H}^c$ *holds*," where $y, g, h, \tilde{Y}, \tilde{g_1}$, and $\tilde{H}$ are elements of some groups $G = \langle g_1 \rangle = \langle H \rangle$ and $\tilde{G} = \langle \tilde{g_1} \rangle = \langle \tilde{H} \rangle$. The convention is that the letters in the parenthesis (a, b, c) denote quantities of which knowledge is being proven, while all other values are known to the verifier. $\mathsf{SPK}\{\ldots\}(m)$ denotes a signature proof of knowledge on m, which is a non-interactive transformation of such proofs using the Fiat-Shamir heuristic [FS87].

We can create similar proofs proving knowledge of group elements instead of exponents, e.g., $\mathsf{SPK}\{a \in \mathbb{G}_1 : y = e(a, b)\}$ by using $e(\cdot, b)$ instead of $b^{(\cdot)}$: Take $r \xleftarrow{\$} \mathbb{G}_1$, $t \leftarrow e(r, b)$, $c \leftarrow \mathsf{H}(\ldots) \in \mathbb{Z}_q$, and $s \leftarrow r \cdot a^c$. Verification computes $\hat{t} = e(s, b) \cdot y^{-c}$ and checks that the Fiat-Shamir hash [FS87] equals c. With the same mechanism we can prove knowledge of elements in $\mathbb{G}_2$.

$\mathcal{F}_{\mathsf{sig}}$ – functionality for public-key signatures.

Variables: initially empty records keyrec and sigrec.

1. `Key Generation.` On input $(\mathsf{KeyGen}, \mathsf{sid})$ from a party $\mathcal{P}$.
    - If $\mathsf{sid} \neq (\mathcal{P}, \mathsf{sid}')$ or a record $(\mathsf{keyrec}, \mathsf{sid}, pk)$ exists, then abort.
    - Send $(\mathsf{KeyGen}, \mathsf{sid})$ to $\mathcal{A}$ and wait for $(\mathsf{KeyConf}, \mathsf{sid}, pk)$ from $\mathcal{A}$. If a record $(\mathsf{sigrec}, \mathsf{sid}, *, *, pk, *)$ exists, abort (*Consistency*).
    - Create record $(\mathsf{keyrec}, \mathsf{sid}, pk)$.
    - Output $(\mathsf{KeyConf}, \mathsf{sid}, pk)$ to $\mathcal{P}$.
2. `Signature Generation.` On input $(\mathsf{Sign}, \mathsf{sid}, m)$ from $\mathcal{P}$.
    - If $\mathsf{sid} \neq (\mathcal{P}, \mathsf{sid}')$ or no record $(\mathsf{keyrec}, \mathsf{sid}, pk)$ exists, then abort.
    - Send $(\mathsf{Sign}, \mathsf{sid}, m)$ to $\mathcal{A}$, and wait for $(\mathsf{Signature}, \mathsf{sid}, \sigma)$ from $\mathcal{A}$.
    - If a record $(\mathsf{sigrec}, \mathsf{sid}, m, \sigma, pk, \mathsf{false})$ exists, then abort.
    - Create record $(\mathsf{sigrec}, \mathsf{sid}, m, \sigma, pk, \mathsf{true})$ (*Completeness*).
    - Output $(\mathsf{Signature}, \mathsf{sid}, \sigma)$ to $\mathcal{P}$.
3. `Signature Verification.` On input $(\mathsf{Verify}, \mathsf{sid}, m, \sigma, pk')$ from some party $\mathcal{V}$.
    - If a record $(\mathsf{sigrec}, \mathsf{sid}, m, \sigma, pk', b)$ exists, set $f \leftarrow b$ (*Consistency*).
    - Else, if a record $(\mathsf{keyrec}, \mathsf{sid}, pk)$ exists, $\mathcal{P}$ is honest, and no record $(\mathsf{sigrec}, \mathsf{sid}, m, *, pk, \mathsf{true})$ exists, set $f \leftarrow 0$ (*Unforgeability*).
    - Else, send $(\mathsf{Verify}, \mathsf{sid}, m, \sigma, pk')$ to $\mathcal{A}$ and wait for $(\mathsf{Verified}, \mathsf{sid}, b)$, and set $f \leftarrow b$.
    - Create a record $(\mathsf{sigrec}, \mathsf{sid}, m, \sigma, pk', f)$ and output $(\mathsf{Verified}, \mathsf{sid}, f)$ to $\mathcal{V}$.

Figure 2.7: The signature functionality $\mathcal{F}_{\mathsf{sig}}$.

We use $\mathsf{NIZK}\{w : s(w)\}$ to denote a generic non-interactive zero-knowledge proof proving knowledge of witness w such that statement $s(w)$ is true. Sometimes we need a witness to be online extractable by a simulator, i.e., extratable without making use of rewinding. We denote online-extractability by drawing a box around the witness: $\mathsf{NIZK}\{\boxed{w} : s(w)\}$.

$\mathcal{F}_{\mathsf{pke}}^{\mathcal{L}}$ – public-key encryption functionality with leakage function $\mathcal{L}$.

Parameters: message space $\mathcal{M}$
Variables: initially empty records keyrec, encrec, decrec.

1. `KeyGen`. On input $(\mathsf{KeyGen}, \mathsf{sid})$ from party $\mathcal{P}$:
   - If $\mathsf{sid} \neq (\mathcal{P}, \mathsf{sid}')$ or a record $(\mathsf{keyrec}, \mathsf{sid}, pk)$ exists, then abort.
   - Send $(\mathsf{KeyGen}, \mathsf{sid})$ to $\mathcal{A}$ and wait for $(\mathsf{KeyConf}, \mathsf{sid}, pk)$ from $\mathcal{A}$.
   - Create record $(\mathsf{keyrec}, \mathsf{sid}, pk)$.
   - Output $(\mathsf{KeyConf}, \mathsf{sid}, pk)$ to $\mathcal{P}$.
2. `Encrypt`. On input $(\mathsf{Encrypt}, \mathsf{sid}, pk', m)$ from party $\mathcal{Q}$ with $m \in \mathcal{M}$:
   - Retrieve record $(\mathsf{keyrec}, \mathsf{sid}, pk)$ for sid.
   - If $pk' = pk$ and $\mathcal{P}$ is honest, then:
     - Send $(\mathsf{Enc\text{-}L}, \mathsf{sid}, pk, \mathcal{L}(m))$ to $\mathcal{A}$, and wait for $(\mathsf{Ciphertext}, \mathsf{sid}, c)$ from $\mathcal{A}$.
     - If a record $(\mathsf{encrec}, \mathsf{sid}, \cdot, c)$ exists, then abort.
     - Create record $(\mathsf{encrec}, \mathsf{sid}, m, c)$.
   - Else (i.e., $pk' \neq pk$ or $\mathcal{P}$ is corrupt) then:
     - Send $(\mathsf{Enc\text{-}M}, \mathsf{sid}, pk', m)$ to $\mathcal{A}$, and wait for $(\mathsf{Ciphertext}, \mathsf{sid}, c)$ from $\mathcal{A}$.
   - Output $(\mathsf{Ciphertext}, \mathsf{sid}, c)$ to $\mathcal{Q}$.
3. `Decrypt`. On input $(\mathsf{Decrypt}, \mathsf{sid}, c)$ from party $\mathcal{P}$:
   - If $\mathsf{sid} \neq (\mathcal{P}, \mathsf{sid}')$ or no record $(\mathsf{keyrec}, \mathsf{sid}, pk)$ exists, then abort.
   - If a record $(\mathsf{encrec}, \mathsf{sid}, m, c)$ for c exists:
     - Output $(\mathsf{Plaintext}, \mathsf{sid}, m)$ to $\mathcal{P}$.
   - Else (i.e., if no such record exists):
     - Send $(\mathsf{Decrypt}, \mathsf{sid}, c)$ to $\mathcal{A}$ and wait for $(\mathsf{Plaintext}, \mathsf{sid}, m)$ from $\mathcal{A}$.
     - Create record $(\mathsf{encrec}, \mathsf{sid}, m, c)$.
     - Output $(\mathsf{Plaintext}, \mathsf{sid}, m)$ to $\mathcal{P}$.

Figure 2.8: The PKE functionality $\mathcal{F}_{\mathsf{pke}}^{\mathcal{L}}$ with leakage function $\mathcal{L}$.

Chapter 3

Composable Security with Global Random Oracles

This chapter focuses on cryptographic protocols using an idealized hash function, the so-called random oracle. Typically, protocols assume random oracles specific to the protocol instance. We present new key insights on proving protocols secure with respect to *global* random oracles, that allow the random oracle to be shared by different protocols, correctly modeling the fact that the same hash functions are used by most protocols. These insights will be used in the later chapters to construct practical and composable protocols.

3.1 Introduction

The random-oracle model (ROM) [BR93] is an overwhelmingly popular tool in cryptographic protocol design and analysis. Part of its success is due to its intuitive idealization of cryptographic hash functions, which it models through calls to an external oracle that implements a random function. Another important factor is its capability to provide security proofs for highly practical constructions of important cryptographic building blocks such as digital signatures, public-key encryption, and key exchange. In spite of its known inability to provide provable guarantees when instantiated with a real-world hash function [CGH98], the ROM is still widely seen as convincing evidence that a protocol will resist attacks in practice.

Most proofs in the ROM, however, are for property-based security notions, where the adversary is challenged in a game where he faces a single, isolated instance of the protocol. Security can therefore no longer be guaranteed when a protocol is composed. Addressing this requires composable security notions such as Canetti's Universal Composability (UC) framework [Can01], which have the advantage of guaranteeing security even if protocols are arbitrarily composed.

UC modeling. In the UC framework, a random oracle is usually modeled as an ideal functionality that a protocol uses as a subroutine in a so-called *hybrid model*, similarly to other setup constructs such as a common reference string (CRS). For example, the random-oracle functionality $\mathcal{F}_{\mathsf{RO}}$ [Nie02] simply assigns a random output value h to each input m and returns h. In the security proof, the simulator executes the code of the subfunctionality, which enables it to observe the queries of all involved parties and to program any random-looking values as outputs. Setup assumptions play an important role for protocols in the UC model, as many important cryptographic primitives such as commitments simply cannot be achieved [CF01]; other tasks can, but have more efficient instantiations with a trusted setup.

An important caveat is that this way of modeling assumes that each instance of each protocol uses its own separate and independent instance of the subfunctionality. For a CRS this is somewhat awkward, because it raises the question of how the parties should agree on a common CRS, but it is even more problematic for random oracles if all, supposedly independent, instances of $\mathcal{F}_{\mathsf{RO}}$ are replaced in practice with the *same* hash function. This can be addressed using the Generalized UC (GUC) framework [CDPW07] that allows one to model different protocol instances sharing access to global functionalities. Thus one can make the setup functionality globally available to all parties, meaning, including those outside of the protocol execution as well as the external environment.

Global UC random oracle. Canetti, Jain, and Scafuro [CJS14] indeed applied the GUC framework to model globally accessible random oracles. In doing so, they discard the globally accessible variant of $\mathcal{F}_{\mathsf{RO}}$

described above as of little help for proving security of protocols because it is too “strict”, allowing the simulator neither to observe the environment’s random-oracle queries, nor to program its answers. They argue that any shared functionality that provides only public information is useless as it does not give the simulator any advantage over the real adversary. Instead, they formulate a global random-oracle functionality that grants the ideal-world simulator access to the list of queries that the environment makes outside of the session. They then show that this shared functionality can be used to design a reasonably efficient GUC-secure commitment scheme, as well as zero-knowledge proofs and two-party computation. However, their global random-oracle functionality rules out security proofs for a number of practical protocols, especially those that require one to program the random oracle.

Our Contributions.

In this chapter, which is based on [CDG$^+$18], we investigate different alternative formulations of globally accessible random-oracle functionalities and protocols that can be proven secure with respect to these functionalities. For instance, we show that the simple variant discarded by Canetti et al. surprisingly suffices to prove the GUC-security of a number of truly practical constructions for useful cryptographic primitives such as digital signatures and public-key encryption. We achieve these results by carefully analyzing the minimal capabilities that the *simulator* needs in order to simulate the real-world (hybrid) protocol, while fully exploiting the additional capabilities that one has in proving the indistinguishability between the real and the ideal worlds. In the following, we briefly describe the different random-oracle functionalities we consider and which we prove GUC-secure using them.

Strict global random oracle. First, we revisit the strict global random-oracle functionality $\mathcal{G}_{\mathsf{sRO}}$ described above and show that, in spite of the arguments of Canetti et al. [CJS14], it actually suffices to prove the GUC-security of many practical constructions. In particular, we show that any digital signature scheme that is existentially unforgeable under chosen-message attack in the traditional ROM also GUC-realizes the signature functionality with $\mathcal{G}_{\mathsf{sRO}}$, and that any public-key encryption (PKE) scheme that is indistinguishable under adaptive chosen-ciphertext attack in the traditional ROM GUC-realizes the PKE functionality under $\mathcal{G}_{\mathsf{sRO}}$ with static corruptions.

This result may be somewhat surprising as it includes many schemes that, in their property-based security proofs, rely on invasive proof techniques such as rewinding, observing, and programming the random oracle, all of which are tools that the GUC simulator is not allowed to use. We demonstrate, however, that none of these techniques are needed during the simulation of the protocol, but rather only show up when proving indistinguishability of the real and the ideal worlds, where they are allowed. A similar technique was used It also does not contradict the impossibility proof of commitments based on global setup functionalities that simply provide public information [CDPW07,CF01] because, in the GUC framework, signatures and PKE do not imply commitments.

Programmable global random oracles. Next, we present a global random-oracle functionality $\mathcal{G}_{\mathsf{pRO}}$ that allows the simulator as well as the real-world adversary to program arbitrary points in the random oracle, as long as they are not yet defined. We show that it suffices to prove the GUC-security of Camenisch et al.'s non-committing encryption scheme [CLNS17], i.e., PKE scheme secure against adaptive corruptions. Here, the GUC simulator needs to produce dummy ciphertexts that can later be made to decrypt to a particular message when the sender or the receiver of the ciphertext is corrupted. The crucial observation is that, to embed a message in a dummy ciphertext, the simulator only needs to program the random oracle at *random* inputs, which have negligible chance of being already queried or programmed. Again, this result is somewhat surprising as $\mathcal{G}_{\mathsf{pRO}}$ does not give the simulator any advantage over the real adversary either.

We also define a restricted variant $\mathcal{G}_{\mathsf{rpRO}}$ that, analogously to the observable random oracle of Canetti et al. [CJS14], offers programming subject to some restrictions, namely that protocol parties can check whether the random oracle was programmed on a particular point. If the adversary tries to cheat by programming the random oracle, then honest parties have a means of detecting this misbehavior. However, we will see that the simulator can hide its programming from the adversary, giving it a clear advantage over the real-world adversary. We use it to GUC-realize the commitment functionality through a new construction that, with only two exponentiations per party and two rounds of communication, is considerably more efficient than the one of Canetti et al. [CJS14], which required five exponentiations and five rounds of communication.

Programmable and observable global random oracle. Finally, we describe a global random-oracle functionality $\mathcal{G}_{\mathsf{rpoRO}}$ that combines the restricted forms of programmability and observability. We then show that this functionality allows us to prove that commitments can be GUC-realized by the most natural and efficient random-oracle based scheme where a commitment $c = \mathcal{H}(m\|r)$ is the hash of the random opening information r and the message m.

Transformations between different oracles. While our different types of oracles allow us to securely realize different protocols, the variety in oracles partially defies the original goal of modeling the situation where all protocols use the *same* hash function. We therefore explore some relations among the different types by presenting efficient protocol transformations that turn any protocol that securely realizes a functionality with one type of random oracle into a protocol that securely realizes the same functionality with a different type.

Other related work.

Dodis et al. [DSW08] already realized that rewinding can be used in the indistinguishability proof in the GUC model, as long as it's not used in the simulation itself. In a broader sense, our work complements existing studies on the impact of programmability and observability of random oracles in security reductions. Fischlin et al. [FLR$^+$10] and Bhattacharyya and Mukherjee [BM15] have proposed formalizations of non-programmable and weakly-programmable random oracles, e.g., only allowing non-adaptive programmability. Both works give a number of possibility and impossibility results, in particular that full-domain hash (FDH) signatures can only be proven secure (via black-box reductions) if the random oracle is fully programmable [FLR$^+$10]. Non-observable random oracles and their power are studied by Ananth and Bhaskarin [AB13], showing that Schnorr and probabilistic RSA-FDH signatures can be proven secure. All these works focus on the use of random oracles in individual reductions, whereas our work proposes globally re-usable random-oracle functionalities within the UC framework. The strict random oracle functionality $\mathcal{G}_{\mathsf{sRO}}$ that we analyze is comparable to a non-programmable and non-observable random oracle, so our result that any unforgeable signature scheme is also GUC-secure

$\mathcal{G}_{\mathsf{sRO}}$ – functionality for the strict global random oracle.

Parameters: output size $\ell(\kappa)$
Variables: initially empty list $\mathsf{List}_{\mathcal{H}}$

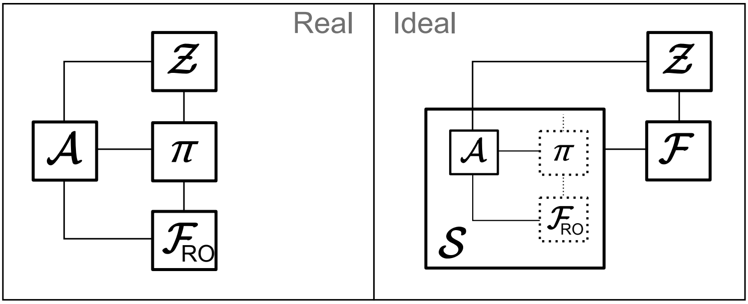
1. `Query`: on input $(\mathsf{HashQuery}, m)$ from a machine $(\mathcal{P}, \mathsf{sid})$, proceed as follows.
    - Find h such that $(m, h) \in \mathsf{List}_{\mathcal{H}}$. If no such h exists, let $h \xleftarrow{\$} \{0,1\}^{\ell(\kappa)}$ and store (m, h) in $\mathsf{List}_{\mathcal{H}}$.
    - Output $(\mathsf{HashConfirm}, h)$ to $(\mathcal{P}, \mathsf{sid})$.

Figure 3.1: The strict global random oracle functionality $\mathcal{G}_{\mathsf{sRO}}$ that does not give any extra power to anyone (mentioned but not defined by Canetti et al. [CJS14]).

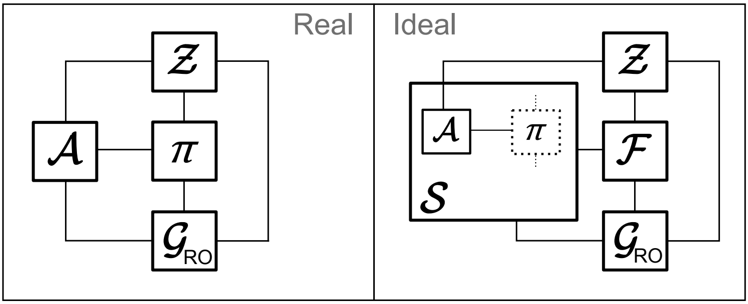
w.r.t. $\mathcal{G}_{\mathsf{sRO}}$ may seem to contradict the above results. However, the $\mathcal{G}_{\mathsf{sRO}}$ functionality imposes these restrictions only for the GUC simulator, whereas the reduction can fully program the random oracle.

Summary.

Our results clearly paint a much more positive picture for global random oracles than was given in the literature so far. We present several formulations of globally accessible random-oracle functionalities that allow to prove the composable security of some of the most efficient signature, PKE, and commitment schemes that are currently known. We even show that the most natural formulation, the strict global random oracle $\mathcal{G}_{\mathsf{sRO}}$ that was previously considered useless, suffices to prove GUC-secure a large class of efficient signature and encryption schemes. By doing so, our work brings the (composable) ROM back closer to its original intention: to provide an *intuitive* idealization of hash functions that enables to prove the security of *highly efficient* protocols. We expect that our results will give rise to many more practical cryptographic protocols that can be proven GUC-secure, among them known protocols that have been proven secure in the traditional ROM model.



(a) Local random oracle: the simulator simulates the RO and has full control.



(b) Global random oracle: the random oracle is external to the simulator.

Figure 3.2: The UC experiment with a local random oracle (a) and the EUC experiment with a global random oracle (b).

3.2 Strict Random Oracle

This section focuses on the so-called *strict* global random oracle $\mathcal{G}_{\mathsf{sRO}}$ depicted in Figure 3.1, which is the most natural definition of a global random oracle: on a fresh input m, a random value h is chosen, while on repeating inputs, a consistent answer is given back. This natural definition was discussed by Canetti et al. [CJS14] but discarded as it does not suffice to realize $\mathcal{F}_{\mathsf{com}}$. While this is true, we will argue that $\mathcal{G}_{\mathsf{sRO}}$ is still useful to realize other functionalities.

The code of $\mathcal{G}_{\mathsf{sRO}}$ is identical to that of a *local* random oracle $\mathcal{F}_{\mathsf{RO}}$ in UC. In standard UC, this is a very strong definition, as it gives the simulator a lot of power: In the ideal world, it can simulate the random

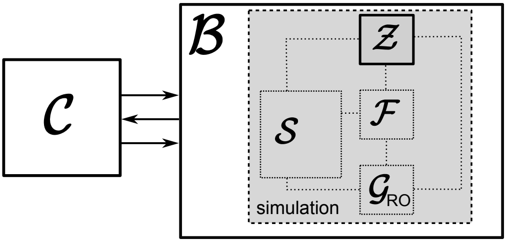


Figure 3.3: Reduction $\mathcal{B}$ from a real-world adversary $\mathcal{A}$ and a black-box environment $\mathcal{Z}$, simulating all the ideal functionalities (even the global ones) and playing against an external challenger $\mathcal{C}$.

oracle $\mathcal{F}_{\mathsf{RO}}$, which gives it the ability to observe all queries and program the random oracle on the fly (cf. Figure 3.2(a)). In GUC, the global random oracle $\mathcal{G}_{\mathsf{sRO}}$ is present in both worlds and the environment can access it (cf. Figure 3.2(b)). In particular, the simulator is not given control of $\mathcal{G}_{\mathsf{sRO}}$ and hence cannot simulate it. Therefore, the simulator has no more power over the random oracle than explicitly offered through the interfaces of the global functionality. In the case of $\mathcal{G}_{\mathsf{sRO}}$, the simulator can neither program the random oracle, nor observe the queries made.

As the simulator obtains no relevant advantage over the real-world adversary when interacting with $\mathcal{G}_{\mathsf{sRO}}$, one might wonder how it could help in security proofs. The main observation is that the situation is different when one proves that the real and ideal world are indistinguishable. Here one needs to show that no environment can distinguish between the real and ideal world and thus, when doing so, one has full control over the global functionality. This is for instance the case when using the (distinguishing) environment in a cryptographic reduction: as depicted in Figure 3.3, the reduction algorithm $\mathcal{B}$ simulates the complete view of the environment $\mathcal{Z}$, including the global $\mathcal{G}_{\mathsf{sRO}}$, allowing $\mathcal{B}$ to freely observe and program $\mathcal{G}_{\mathsf{sRO}}$. As a matter of facts, $\mathcal{B}$ can also rewind the environment here – another power that the simulator $\mathcal{S}$ does not have but is useful in the security analysis of many schemes. It turns out that for some primitives, the EUC simulator does not need to program or observe the random oracle, but only needs to do so when proving that no environment can distinguish between the real and the ideal world.

This allows us to prove a surprisingly wide range of practical protocols secure with respect to $\mathcal{G}_{\mathsf{sRO}}$. First, we prove that any signature scheme proven to be EUF-CMA in the local random-oracle model yields UC secure signatures with respect the global $\mathcal{G}_{\mathsf{sRO}}$. Second, we show that any public-key encryption scheme proven to be IND-CCA2 secure with local random oracles yields UC secure public-key encryption (with respect to static corruptions), again with the global $\mathcal{G}_{\mathsf{sRO}}$. These results show that highly practical schemes such as Schnorr signatures [Sch91], RSA full-domain hash signatures [BR93, Cor00], RSA-PSS signatures [BR96], RSA-OAEP encryption [BR95], and the Fujisaki-Okamoto transform [FO13] all remain secure when all schemes share a single hash function that is modeled as a strict global random oracle. This is remarkable, as their security proofs in the local random-oracle model involve techniques that are not available to an EUC simulator: signature schemes typically require programming of random-oracle outputs to simulate signatures, PKE schemes typically require observing the adversary's queries to simulate decryption queries, and Schnorr signatures need to rewind the adversary in a forking argument [PS00] to extract a witness. However, it turns out, this rewinding is only necessary in the reduction $\mathcal{B}$ showing that no distinguishing environment $\mathcal{Z}$ can exist and we can show that all these schemes can safely be used in composition with arbitrary protocols and with a natural, globally accessible random-oracle functionality $\mathcal{G}_{\mathsf{sRO}}$.

3.2.1 Composable Signatures using $\mathcal{G}_{\mathsf{sRO}}$

Let $\mathsf{SIG} = (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Verify})$ be an EUF-CMA secure signature scheme in the ROM. We show that this directly yields a secure realization of UC signatures $\mathcal{F}_{\mathsf{sig}}$ with respect to a strict global random oracle $\mathcal{G}_{\mathsf{sRO}}$. We assume that SIG uses a single random oracle that maps to $\{0,1\}^{\ell(\kappa)}$. Protocols requiring multiple random oracles or mapping into different ranges can be constructed using standard domain separation and length extension techniques.

We define π_{SIG} to be SIG phrased as a GUC protocol. Whenever an algorithm of SIG makes a call to the random oracle, π_{SIG} makes a call to $\mathcal{G}_{\mathsf{sRO}}$.

1. On input $(\mathsf{KeyGen}, \mathsf{sid})$, signer $\mathcal{P}$ proceeds as follows.
    - Check that $\mathsf{sid} = (\mathcal{P}, \mathsf{sid}')$ for some sid', and no record (sid, sk) exists.
    - Run $(pk, sk) \leftarrow \mathsf{SIG.KGen}(\kappa)$ and store (sid, sk).

   - Output $(\mathsf{KeyConf}, \mathsf{sid}, pk)$.

2. On input $(\mathsf{Sign}, \mathsf{sid}, m)$, signer $\mathcal{P}$ proceeds as follows.
   - Retrieve record (sid, sk), abort if no record exists.
   - Output $(\mathsf{Signature}, \mathsf{sid}, \sigma)$ with $\sigma \leftarrow \mathsf{SIG.Sign}(sk, m)$.
3. On input $(\mathsf{Verify}, \mathsf{sid}, m, \sigma, pk')$ a verifier $\mathcal{V}$ proceeds as follows.
   - Output $(\mathsf{Verified}, \mathsf{sid}, f)$ with $f \leftarrow \mathsf{SIG.Verify}(pk', \sigma, m)$.

We will prove that π_{SIG} will realize UC signatures. There are two main approaches to defining a signature functionality: using adversarially provided algorithms to generate and verify signature objects (e.g., the 2005 version of [Can00]), or by asking the adversary to create and verify signature objects (e.g., [Can04]). For a version using algorithms, the functionality will locally create and verify signature objects using the algorithm, without activating the adversary. This means that the algorithms cannot interact with external parties, and in particular communication with external functionalities such as a global random oracle is not permitted. We could modify an algorithm-based $\mathcal{F}_{\mathsf{sig}}$ to allow the sign and verify algorithms to communicate *only with a global random oracle*, but we choose to use an $\mathcal{F}_{\mathsf{sig}}$ that interacts with the adversary as this does not require special modifications for signatures with global random oracles.

Theorem 2. *If* SIG *is EUF-CMA in the random-oracle model, then* π_{SIG} *GUC-realizes* $\mathcal{F}_{\mathsf{sig}}$ *(as defined in Figure 2.7) in the* $\mathcal{G}_{\mathsf{sRO}}$*-hybrid model.*

Proof. By the fact that π_{SIG} is $\mathcal{G}_{\mathsf{sRO}}$-subroutine respecting and by Theorem 1, it is sufficient to show that π_{SIG} $\mathcal{G}_{\mathsf{sRO}}$-EUC-realizes $\mathcal{F}_{\mathsf{sig}}$. We define the UC simulator $\mathcal{S}$ as follows.

1. `Key Generation.` On input $(\mathsf{KeyGen}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{sig}}$, where $\mathsf{sid} = (\mathcal{P}, \mathsf{sid}')$ and $\mathcal{P}$ is honest.
   - Simulate honest signer "$\mathcal{P}$", and give it input $(\mathsf{KeyGen}, \mathsf{sid})$.
   - When "$\mathcal{P}$" outputs $(\mathsf{KeyConf}, \mathsf{sid}, pk)$ (where pk is generated according to π_{SIG}), send $(\mathsf{KeyConf}, \mathsf{sid}, pk)$ to $\mathcal{F}_{\mathsf{sig}}$.
2. `Signature Generation.` On input $(\mathsf{Sign}, \mathsf{sid}, m)$ from $\mathcal{F}_{\mathsf{sig}}$, where $\mathsf{sid} = (\mathcal{P}, \mathsf{sid}')$ and $\mathcal{P}$ is honest.
   - Run simulated honest signer "$\mathcal{P}$" with input $(\mathsf{Sign}, \mathsf{sid}, m)$.
   - When "$\mathcal{P}$" outputs $(\mathsf{Signature}, \mathsf{sid}, \sigma)$ (where σ is generated according to π_{SIG}), send $(\mathsf{Signature}, \mathsf{sid}, \sigma)$ to $\mathcal{F}_{\mathsf{sig}}$.

3. `Signature Verification.` On input $(\mathsf{Verify}, \mathsf{sid}, m, \sigma, pk')$ from $\mathcal{F}_{\mathsf{sig}}$, where $\mathsf{sid} = (\mathcal{P}, \mathsf{sid}')$.
   - Run $f \leftarrow \mathsf{SIG.Verify}(pk', \sigma, m)$, and send $(\mathsf{Verified}, \mathsf{sid}, f)$ to $\mathcal{F}_{\mathsf{sig}}$.

We must show that π_{SIG} realizes $\mathcal{F}_{\mathsf{sig}}$ in the standard UC sense, but with respect to $\mathcal{G}_{\mathsf{sRO}}$-externally constrained environments, i.e., the environment is now allowed to access $\mathcal{G}_{\mathsf{sRO}}$ via dummy parties in sessions unequal to the challenge session. Without loss of generality, we prove this with respect to the dummy adversary.

During key generation, $\mathcal{S}$ invokes the simulated honest signer $\mathcal{P}$, so the resulting keys are exactly like in the real world. The only difference is that in the ideal world $\mathcal{F}_{\mathsf{sig}}$ can abort key generation in case the provided public key pk already appears in a previous sigrec record. But if this happens it means that $\mathcal{A}$ has successfully found a collision in the public key space, which must be exponentially large as the signature scheme is EUF-CMA by assumption. This means that such event can only happen with negligible probability.

For a corrupt signer, the rest of the simulation is trivially correct: the adversary generates keys and signatures locally, and if an honest party verifies a signature, the simulator simply executes the verification algorithm as a real world party would do, and $\mathcal{F}_{\mathsf{sig}}$ does not make further checks (the unforgeability check is only made when the signer is honest). When an honest signer signs, the simulator creates a signature using the real world signing algorithm, and when $\mathcal{F}_{\mathsf{sig}}$ asks the simulator to verify a signature, $\mathcal{S}$ runs the real world verification algorithm, and $\mathcal{F}_{\mathsf{sig}}$ keeps records of the past verification queries to ensure consistency. As the real world verification algorithm is deterministic, storing verification queries does not cause a difference. Finally, when $\mathcal{S}$ provides $\mathcal{F}_{\mathsf{sig}}$ with a signature, $\mathcal{F}_{\mathsf{sig}}$ checks that there is no stored verification query exists that states the provided signature is invalid. By completeness of the signature scheme, this check will never trigger.

The only remaining difference is that $\mathcal{F}_{\mathsf{sig}}$ prevents forgeries: if a verifier uses the correct public key, the signer is honest, and we verify a signature on a message that was never signed, $\mathcal{F}_{\mathsf{sig}}$ rejects. This would change the verification outcome of a signature that would be accepted by the real-world verification algorithm. As this event is the only difference between the real and ideal world, what remains to show is that this check changes the verification outcome only with negligible probability. We prove that if there is an environment that causes this event with non-negligible probability, then we can use it to construct a forger $\mathcal{B}$ that breaks the EUF-CMA unforgeability of SIG.

Our forger $\mathcal{B}$ plays the role of $\mathcal{F}_{\mathsf{sig}}$, $\mathcal{S}$, and even the random oracle $\mathcal{G}_{\mathsf{sRO}}$, and has black-box access to the environment $\mathcal{Z}$. Then $\mathcal{B}$ receives a challenge public key pk and is given access to a signing oracle $\mathcal{O}^{\mathsf{Sign}(sk,\cdot)}$ and to a random oracle RO. It responds $\mathcal{Z}$'s $\mathcal{G}_{\mathsf{sRO}}$ queries by relaying queries and responses to and from RO. It runs the code of $\mathcal{F}_{\mathsf{sig}}$ and $\mathcal{S}$, but $\mathcal{S}$ now uses pk as the public key of "$\mathcal{P}$", and uses $\mathcal{O}^{\mathsf{Sign}(sk,\mathtt{m})}$ whenever $\mathcal{F}_{\mathsf{sig}}$ requests $\mathcal{S}$ to generate a signature. If the unforgeability check of $\mathcal{F}_{\mathsf{sig}}$ triggers for a cryptographically valid signature σ on message m, then we know that $\mathcal{B}$ made no query $\mathcal{O}^{\mathsf{Sign}(sk,\mathtt{m})}$, meaning that $\mathcal{B}$ can submit (σ, m) to win the EUF-CMA game. $\square$

3.2.2 Composable Public-Key Encryption using $\mathcal{G}_{\mathsf{sRO}}$

Let $\mathsf{PKE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ be a CCA2 secure public-key encryption scheme in the ROM. We show that this directly yields a secure realization of GUC public-key encryption $\mathcal{F}_{\mathsf{pke}}^{\mathcal{L}}$, as recently defined by Camenisch et al. [CLNS17] and depicted in Figure 2.8), with respect to a strict global random oracle $\mathcal{G}_{\mathsf{sRO}}$ and static corruptions. As with our result for signature schemes, we require that PKE uses a single random oracle that maps to $\{0,1\}^{\ell(\kappa)}$.

We define π_{PKE} to be PKE phrased as a GUC protocol.

1. On input $(\mathsf{KeyGen}, \mathsf{sid}, \kappa)$, party $\mathcal{P}$ proceeds as follows.
    - Check that $\mathsf{sid} = (\mathcal{P}, \mathsf{sid}')$ for some sid', and no record (sid, sk) exists.
    - Run $(pk, sk) \leftarrow \mathsf{PKE.KGen}(\kappa)$ and store (sid, sk).
    - Output $(\mathsf{KeyConf}, \mathsf{sid}, pk)$.
2. On input $(\mathsf{Encrypt}, \mathsf{sid}, pk', m)$, party $\mathcal{Q}$ proceeds as follows.
    - Set $c \leftarrow \mathsf{PKE.Enc}(pk', m)$ and output $(\mathsf{Ciphertext}, \mathsf{sid}, c)$.
3. On input $(\mathsf{Decrypt}, \mathsf{sid}, c)$, party $\mathcal{P}$ proceeds as follows.
    - Retrieve (sid, sk), abort if no such record exist.
    - Set $m \leftarrow \mathsf{PKE.Dec}(sk, c)$ and output $(\mathsf{Plaintext}, \mathsf{sid}, m)$.

Theorem 3. *Protocol π_{PKE} GUC-realizes $\mathcal{F}_{\mathsf{pke}}^{\mathcal{L}}$ with static corruptions with leakage function $\mathcal{L}$ in the $\mathcal{G}_{\mathsf{sRO}}$-hybrid model if PKE is CCA2 secure with leakage $\mathcal{L}$ in the ROM.*

Proof. By the fact that π_{PKE} is $\mathcal{G}_{\mathsf{sRO}}$-subroutine respecting and by Theorem 1, it is sufficient to show that π_{PKE} $\mathcal{G}_{\mathsf{sRO}}$-EUC-realizes $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$.

We define simulator $\mathcal{S}$ as follows.

1. On input $(\mathsf{KEYGEN}, \mathsf{sid})$.
    - Parse sid as $(\mathcal{P}, \mathsf{sid}')$. Note that $\mathcal{P}$ is honest, as $\mathcal{S}$ does not make KeyGen queries on behalf of corrupt parties.
    - Invoke the simulated receiver "$\mathcal{P}$" on input $(\mathsf{KeyGen}, \mathsf{sid})$ and wait for output $(\mathsf{KeyConf}, \mathsf{sid}, pk)$ from "$\mathcal{P}$".
    - Send $(\mathsf{KeyConf}, \mathsf{sid}, pk)$ to $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$.
2. On input $(\mathsf{Enc\text{-}M}, \mathsf{sid}, pk', m)$ with $m \in \mathcal{M}$.
    - $\mathcal{S}$ picks some honest party "$\mathcal{Q}$" and gives it input $(\mathsf{Encrypt}, \mathsf{sid}, pk', m)$. Wait for output $(\mathsf{Ciphertext}, \mathsf{sid}, c)$ from "$\mathcal{Q}$".
    - Send $(\mathsf{Ciphertext}, \mathsf{sid}, c)$ to $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$.
3. On input $(\mathsf{Enc\text{-}L}, \mathsf{sid}, pk, l)$.
    - $\mathcal{S}$ does not know which message is being encrypted, so it chooses a dummy plaintext $m' \in \mathcal{M}$ with $\mathcal{L}(m') = l$.
    - Pick some honest party "$\mathcal{Q}$" and give it input $(\mathsf{Encrypt}, \mathsf{sid}, pk, m')$, Wait for output $(\mathsf{Ciphertext}, \mathsf{sid}, c)$ from "$\mathcal{Q}$".
    - Send $(\mathsf{Ciphertext}, \mathsf{sid}, c)$ to $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$.
4. On input $(\mathsf{Decrypt}, \mathsf{sid}, c)$.
    - Note that $\mathcal{S}$ only receives such input when $\mathcal{P}$ is honest, and therefore $\mathcal{S}$ simulates "$\mathcal{P}$" and knows its secret key sk.
    - Give "$\mathcal{P}$" input $(\mathsf{Decrypt}, \mathsf{sid}, c)$ and wait for output $(\mathsf{Plaintext}, \mathsf{sid}, m)$ from "$\mathcal{P}$".
    - Send $(\mathsf{Plaintext}, \mathsf{sid}, m)$ to $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$.

What remains to show is that $\mathcal{S}$ is a satisfying simulator, i.e., no $\mathcal{G}_{\mathsf{sRO}}$-externally constrained environment can distinguish the real protocol π_{PKE} from $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$ with $\mathcal{S}$. If the receiver $\mathcal{P}$ (i.e., such that $\mathsf{sid} = (\mathcal{P}, \mathsf{sid}')$) is corrupt, the simulation is trivially correct: $\mathcal{S}$ only creates ciphertexts when it knows the plaintext, so it can simply follow the real protocol. If $\mathcal{P}$ is honest, $\mathcal{S}$ does not know the message for which it is computing ciphertexts, so a dummy plaintext is encrypted. When the environment submits that ciphertext for decryption by $\mathcal{P}$, the functionality $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$ will still return the correct message. Using a sequence of games, we show that if an environment exists that can notice this difference, it can break the CCA2 security of PKE.

Let $\mathsf{Game}\ 0$ be the game where $\mathcal{S}$ and $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$ act as in the ideal world, except that $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$ passes the full message m in $\mathsf{Enc\text{-}L}$ inputs to $\mathcal{S}$, and $\mathcal{S}$ returns a real encryption of m as the ciphertext. It is clear that $\mathsf{Game}\ 0$ is identical to the real world $\mathrm{EXEC}^{\mathcal{G}}_{\pi,\mathcal{A},\mathcal{Z}}$. Let $\mathsf{Game}\ i$ for $i = 1, \ldots, q_{\mathrm{E}}$, where q_{E} is the number of $\mathsf{Encrypt}$ queries made by $\mathcal{Z}$, be defined as the game where for $\mathcal{Z}$'s first i $\mathsf{Encrypt}$ queries, $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$ passes only $\mathcal{L}(m)$ to $\mathcal{S}$ and $\mathcal{S}$ returns the encryption of a dummy message m' so that $\mathcal{L}(m') = \mathcal{L}(m)$, while for the $i+1$-st to q_{E}-th queries, $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$ passes m to $\mathcal{S}$ and $\mathcal{S}$ returns an encryption of m. It is clear that $\mathsf{Game}\ q_{\mathrm{E}}$ is identical to the ideal world $\mathrm{IDEAL}^{\mathcal{G}}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$.

By a hybrid argument, for $\mathcal{Z}$ to have non-negligible probability to distinguish between $\mathrm{EXEC}^{\mathcal{G}}_{\pi,\mathcal{A},\mathcal{Z}}$ and $\mathrm{IDEAL}^{\mathcal{G}}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$, there must exist an i such that $\mathcal{Z}$ distinguishes with non-negligible probability between $\mathsf{Game}\ (i-1)$ and $\mathsf{Game}\ i$. Such an environment gives rise to the following CCA2 attacker $\mathcal{B}$ against PKE.

Algorithm $\mathcal{B}$ receives a challenge public key pk as input and is given access to decryption oracle $\mathcal{O}^{\mathsf{Dec}(sk,\cdot)}$ and random oracle RO. It answers $\mathcal{Z}$'s queries $\mathcal{G}_{\mathsf{sRO}}(m)$ by relaying responses from its own oracle $\mathsf{RO}(m)$ and lets $\mathcal{S}$ use pk as the public key of $\mathcal{P}$. It largely runs the code of $\mathsf{Game}\ (i-1)$ for $\mathcal{S}$ and $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$, but lets $\mathcal{S}$ respond to inputs $(\mathsf{Dec}, \mathsf{sid}, c)$ from $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$ by calling its decryption oracle $m = \mathcal{O}^{\mathsf{Decrypt}(sk,\mathsf{c})}$. Note that $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$ only hands such inputs to $\mathcal{S}$ for ciphertexts c that were *not* produced via the $\mathsf{Encrypt}$ interface of $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$, as all other ciphertexts are handled by $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$ itself.

Let m_0 denote the message that Functionality $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$ hands to $\mathcal{S}$ as part of the i-th $\mathsf{Enc\text{-}L}$ input. Algorithm $\mathcal{B}$ now sets m_1 to be a dummy message m' such that $\mathcal{L}(m') = \mathcal{L}(m_0)$ and hands (m_0, m_1) to the challenger to obtain the challenge ciphertext c^* that is an encryption of m_b. It is clear that if $b = 0$, then the view of $\mathcal{Z}$ is identical to that in $\mathsf{Game}\ (i-1)$, while if $b = 1$, it is identical to that in $\mathsf{Game}\ i$. Moreover, $\mathcal{B}$ will never have to query its decryption oracle on the challenge ciphertext c^*, because any decryption queries for c^* are handled by $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$ directly. By outputting 0 if $\mathcal{Z}$ decides it runs in $\mathsf{Game}\ (i-1)$ and outputting 1 if $\mathcal{Z}$ decides it runs in $\mathsf{Game}\ i$, $\mathcal{B}$ wins the CCA2 game with non-negligible probability. $\square$

3.3 Programmable Global Random Oracle

We now turn our attention to a new functionality that we call the *programmable global random oracle*, denoted by $\mathcal{G}_{\mathsf{pRO}}$. The functionality simply extends the strict random oracle $\mathcal{G}_{\mathsf{sRO}}$ by giving the adversary (real-world adversary $\mathcal{A}$ and ideal-world adversary $\mathcal{S}$) the power to program input-output pairs. Because we are in GUC or EUC, that also means that the environment gets this power. Thus, as in the case of $\mathcal{G}_{\mathsf{sRO}}$, the simulator is thus not given any extra power compared to the environment (through the adversary), and one might well think that this model would not lead to the realization of any useful cryptographic primitives either. To the contrary, one would expect that the environment being able to program outputs would interfere with security proofs, as it destroys many properties of the random oracle such as collision or preimage resistance.

As it turns out, we can actually realize public-key encryption secure against adaptive corruptions (also known as non-committing encryption) in this model: we prove that the PKE scheme of Camenisch et al. [CLNS17] GUC-realizes $\mathcal{F}_{\mathsf{pke}}^{\mathcal{L}}$ against adaptive corruptions in the $\mathcal{G}_{\mathsf{pRO}}$-hybrid model. The security proof works out because the simulator equivocates dummy ciphertexts by programming the random oracle on *random* points, which are unlikely to have been queried by the environment before.

3.3.1 The Programmable Global Random Oracle

The programmable global random oracle functionality $\mathcal{G}_{\mathsf{pRO}}$ (cf. Figure 3.4) is simply obtained from $\mathcal{G}_{\mathsf{sRO}}$ by adding an interface for the adversary to program the oracle on a single point at a time. To this end, the functionality $\mathcal{G}_{\mathsf{pRO}}$ keeps an internal list of preimage-value assignments and, if programming fails (because it would overwrite a previously taken value), the functionality aborts, i.e., it replies with an error message $\perp$.

Notice that our $\mathcal{G}_{\mathsf{pRO}}$ functionality does not guarantee common random-oracle properties such as collision resistance: an adversary can simply program collisions into $\mathcal{G}_{\mathsf{pRO}}$. However, this choice is by design, because we are interested in achieving security with the *weakest* form of a *programmable* global random oracle to see what can be achieved against the strongest adversary possible.

$\mathcal{G}_{\mathsf{pRO}}$ – functionality for the programmable global random oracle.

Parameters: output size $\ell(\kappa)$
Variables: initially empty list $\mathsf{List}_{\mathcal{H}}$

1. `Query`: on input $(\mathsf{HashQuery}, m)$ from a machine $(\mathcal{P}, \mathsf{sid})$, proceed as follows.
   - Find h such that $(m, h) \in \mathsf{List}_{\mathcal{H}}$. If no such h exists, let $h \xleftarrow{\$} \{0,1\}^{\ell(\kappa)}$ and store (m, h) in $\mathsf{List}_{\mathcal{H}}$.
   - Output $(\mathsf{HashConfirm}, h)$ to $(\mathcal{P}, \mathsf{sid})$.
2. `Program`: on input $(\mathsf{ProgramRO}, m, h)$ from adversary $\mathcal{A}$
   - If $\exists\ h' \in \{0,1\}^{\ell(\kappa)}$ such that $(m, h') \in \mathsf{List}_{\mathcal{H}}$ and $h \neq h'$, then abort
   - Else, add (m, h) to $\mathsf{List}_{\mathcal{H}}$ and output $(\mathsf{ProgramConfirm})$ to $\mathcal{A}$

Figure 3.4: The programmable global random oracle functionality $\mathcal{G}_{\mathsf{pRO}}$.

3.3.2 Public-Key Encryption with Adaptive Corruptions from $\mathcal{G}_{\mathsf{pRO}}$

We show that GUC-secure non-interactive PKE with adaptive corruptions (often referred to as non-committing encryption) is achievable in the hybrid $\mathcal{G}_{\mathsf{pRO}}$ model by proving the PKE scheme by Camenisch et al. [CLNS17] secure in this model. We recall the scheme in Figure 3.5 based on the following building blocks:

- a family of one-way trapdoor permutations $\mathsf{OWTP} = (\mathsf{OWTP.Gen}, \mathsf{OWTP.Sample}, \mathsf{OWTP.Eval}, \mathsf{OWTP.Invert})$, where domains Σ generated by $\mathsf{OWTP.Gen}(1^\kappa)$ have cardinality at least 2^κ;
- a block encoding scheme $(\mathsf{EC}, \mathsf{DC})$ such that $\mathsf{EC} : \{0,1\}^* \to (\{0,1\}^{\ell(\kappa)})^*$ is an encoding function such that the number of blocks that it outputs for a given message m depends only on the leakage $\mathcal{L}(m)$, and DC its deterministic inverse (possibly rejecting with $\perp$ if no preimage exists).

Theorem 4. *Protocol π_{PKE} in Figure 3.5 GUC-realizes $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$ with adaptive corruptions and leakage function $\mathcal{L}$ in the $\mathcal{G}_{\mathsf{pRO}}$-hybrid model.*

Proof. We need to show that π_{PKE} GUC-realizes $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$, i.e., that, given any environment $\mathcal{Z}$ and any real-world adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that the output distribution of $\mathcal{Z}$ interacting with $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$,

π_{PKE} – public-key encryption secure against adaptive corruptions.

Parameters: block size $\ell(\kappa)$

1. `KeyGen`. On input $(\mathsf{KeyGen}, \mathsf{sid})$ from party $\mathcal{P}$:
    - Check that $\mathsf{sid} = (\mathcal{P}, \mathsf{sid}')$ and no record $(\mathsf{keyrec}, \mathsf{sid}, sk)$ exist.
    - Sample $(\varphi, \varphi^{-1}, \Sigma) \leftarrow \mathsf{OWTP.Gen}(1^\kappa)$.
    - Set $pk \leftarrow (\varphi, \Sigma)$, $sk \leftarrow (\varphi, \varphi^{-1}, \Sigma)$.
    - Create record $(\mathsf{keyrec}, \mathsf{sid}, pk, sk)$.
    - Output $(\mathsf{KeyConf}, \mathsf{sid}, pk)$ to $\mathcal{P}$.
2. `Encrypt`. On input $(\mathsf{Encrypt}, \mathsf{sid}, pk', m)$ from party $\mathcal{Q}$:
    - Parse pk' as (φ, Σ), get $(m_1, \ldots, m_k) \leftarrow \mathsf{EC}(m)$ and $x \leftarrow \mathsf{OWTP.Sample}(\Sigma)$.
    - Let $c_1 \leftarrow \mathsf{OWTP.Eval}(\Sigma,\varphi,x), c_{2,i} \leftarrow m_i \oplus h_i, \forall\, i = 1, \ldots, k$, and $c_3 \leftarrow h$ where h and all h_i are obtained as $(\mathsf{HashConfirm}, h_i) \leftarrow \mathcal{G}_{\mathsf{pRO}}(\mathsf{HashQuery}, (x \| i))$ and $(\mathsf{HashConfirm}, h) \leftarrow \mathcal{G}_{\mathsf{pRO}}(\mathsf{HashQuery}, (x \| k \| m))$, respectively.
    - Set $c \leftarrow (c_1, c_{2,1}, \ldots, c_{2,k}, c_3)$.
    - Output $(\mathsf{Ciphertext}, \mathsf{sid}, c)$ to $\mathcal{Q}$.
3. `Decrypt`. On input $(\mathsf{Decrypt}, \mathsf{sid}, c)$ from party $\mathcal{P}$:
    - Check that $\mathsf{sid} = (\mathcal{P}, \mathsf{sid}')$ and $(\mathsf{keyrec}, \mathsf{sid}, sk)$ exist, if not, then abort.
    - Parse sk as $(\varphi, \varphi^{-1}, \Sigma)$, and c as $(c_1, c_{2,1}, \ldots, c_{2,k}, c_3)$.
    - Set $x' \leftarrow \mathsf{OWTP.Invert}(\Sigma, \varphi, \varphi^{-1}, c_1)$, $m'_i \leftarrow c_{2,i} \oplus h'_i$ for $i = 1, \ldots, k$, and $m' \leftarrow \mathsf{DC}(m'_1, \ldots, m'_k)$, where all h'_i are obtained as $(\mathsf{HashConfirm}, h'_i) \leftarrow \mathcal{G}_{\mathsf{pRO}}(\mathsf{HashQuery}, (x' \| i))$.
    - If $m' = \perp_m$ or $h' \neq c_3$, then output $(\mathsf{Plaintext}, \mathsf{sid}, \perp_m)$ to $\mathcal{P}$, where h' is obtained from $(\mathsf{HashConfirm}, h') \leftarrow \mathcal{G}_{\mathsf{pRO}}(\mathsf{HashQuery}, (x' \| k \| m'))$.
    - Else, output $(\mathsf{Plaintext}, \mathsf{sid}, m)$ to $\mathcal{P}$.

Figure 3.5: Public-key encryption scheme secure against adaptive attacks [CLNS17] based on one-way permutation OWTP and encoding function $(\mathsf{EC}, \mathsf{DC})$.

$\mathcal{G}_{\mathsf{pRO}}$, and $\mathcal{S}$ is indistinguishable from its output distribution when interacting with π_{PKE}, $\mathcal{G}_{\mathsf{pRO}}$, and $\mathcal{A}$. Because π_{PKE} is $\mathcal{G}_{\mathsf{sRO}}$-subroutine respecting, by Theorem 1 it suffices to show that π_{PKE} $\mathcal{G}_{\mathsf{pRO}}$-EUC-realizes $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$.

The simulator $\mathcal{S}$ is depicted in Figure 3.6. Basically, it generates an honest key pair for the receiver and responds to Enc-M and Decrypt inputs by using the honest encryption and decryption algorithms, respectively. On Enc-L inputs, however, it creates a dummy ciphertext c composed of $c_1 = \varphi(x)$ for a freshly sampled x (but rejecting values of x that were used before) and randomly chosen $c_{2,1}, \ldots, c_{2,k}$ and c_3 for the correct number of blocks k. Only when either the secret key or the randomness used for this ciphertext must be revealed to the adversary, i.e., only when either the receiver or the party $\mathcal{Q}$ who created the ciphertext is corrupted, does the simulator program the random oracle so that the dummy ciphertext decrypts to the correct message m. If the receiver is corrupted, the simulator obtains m by having it decrypted by $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$; if the encrypting party $\mathcal{Q}$ is corrupted, then m is included in the history of inputs and outputs that is handed to $\mathcal{S}$ upon corruption. The programming is done through the `Program` subroutine, but the simulation aborts in case programming fails, i.e., when a point needs to be programmed that is already assigned. We will prove in the reduction that any environment causing this to happen can be used to break the one-wayness of the trapdoor permutation.

We now have to show that $\mathcal{S}$ successfully simulates a real execution of the protocol π_{PKE} to a real-world adversary $\mathcal{A}$ and environment $\mathcal{Z}$. To see this, consider the following sequence of games played with $\mathcal{A}$ and $\mathcal{Z}$ that gradually evolve from a real execution of π_{PKE} to the simulation by $\mathcal{S}$.

Let Game 0 be a game that is generated by letting an ideal functionality $\mathcal{F}_0$ and a simulator $\mathcal{S}_0$ collaborate, where $\mathcal{F}_0$ is identical to $\mathcal{F}^{\mathcal{L}}_{\mathsf{pke}}$, except that it passes the full message m along with Enc-L inputs to $\mathcal{S}_0$. The simulator $\mathcal{S}_0$ simply performs all key generation, encryption, and decryption using the real algorithms, without any programming of the random oracle. The only difference between Game 0 and the real world is that the ideal functionality $\mathcal{F}_0$ aborts when the same ciphertext c is generated twice during an encryption query for the honest public key. Because $\mathcal{S}_0$ generates honest ciphertexts, the probability that the same ciphertext is generated twice can be bounded by the probability that two honest ciphertexts share the same first component c_1. Given that c_1 is computed as $\varphi(x)$ for a freshly sampled x from Σ, and given

1. On input $(\mathsf{KeyGen}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{pke}}^{\mathcal{L}}$:
   - Sample $r \xleftarrow{\$} \{0,1\}^{\kappa}$ and honestly generate keys with randomness r by generating $(\Sigma, \varphi, \varphi^{-1}) \leftarrow \mathsf{OWTP.Gen}(\kappa; r)$ and setting $pk \leftarrow (\Sigma, \varphi)$, $sk \leftarrow \varphi^{-1}$. Record (pk, sk, r) and send $(\mathsf{KeyConf}, \mathsf{sid}, pk)$ to $\mathcal{F}_{\mathsf{pke}}^{\mathcal{L}}$.
2. On input $(\mathsf{Enc\text{-}L}, \mathsf{sid}, pk, \lambda)$ from $\mathcal{F}_{\mathsf{pke}}^{\mathcal{L}}$:
   - Parse pk as (Σ, φ), sample $r \xleftarrow{\$} \{0,1\}^{\kappa}$, and generate $x \leftarrow \mathsf{OWTP.Sample}(\Sigma; r)$ until x does not appear in EncL.
   - Choose a dummy plaintext m such that $\mathcal{L}(m) = \lambda$ and let k be such that $(m_1, \ldots, m_k) \leftarrow \mathsf{EC}(m)$.
   - Generate a dummy ciphertext c with $c_1 \leftarrow \mathsf{OWTP.Eval}(\Sigma, \varphi, x)$ and with random $c_{2,1}, \ldots, c_{2,k}, c_3 \xleftarrow{\$} \{0,1\}^{\ell(\kappa)}$.
   - Record $(c, \perp_m, r, x, pk)$ in EncL and send $(\mathsf{Ciphertext}, \mathsf{sid}, c)$ to $\mathcal{F}_{\mathsf{pke}}^{\mathcal{L}}$.
3. On input $(\mathsf{Enc\text{-}M}, \mathsf{sid}, pk', m)$ from $\mathcal{F}_{\mathsf{pke}}^{\mathcal{L}}$:
   - Sample $r \xleftarrow{\$} \{0,1\}^{\kappa}$ and produce ciphertext c honestly from m using key pk' and randomness r.
   - Send $(\mathsf{Ciphertext}, \mathsf{sid}, c)$ to $\mathcal{F}_{\mathsf{pke}}^{\mathcal{L}}$.
4. On input $(\mathsf{Decrypt}, \mathsf{sid}, c)$ from $\mathcal{F}_{\mathsf{pke}}^{\mathcal{L}}$:
   - Decrypt c honestly using the recorded secret key sk to yield plaintext m.
   - Send $(\mathsf{Plaintext}, \mathsf{sid}, m)$ to $\mathcal{F}_{\mathsf{pke}}^{\mathcal{L}}$.
5. On corruption of party $\mathcal{Q}$, receive as input from $\mathcal{F}_{\mathsf{pke}}^{\mathcal{L}}$ the history of $\mathcal{Q}$'s inputs and outputs, then compose $\mathcal{Q}$'s state as follows and hand it to $\mathcal{F}_{\mathsf{pke}}^{\mathcal{L}}$:
   - For every input $(\mathsf{Encrypt}, \mathsf{sid}, pk', m)$ and corresponding response $(\mathsf{Ciphertext}, \mathsf{sid}, c)$ in $\mathcal{Q}$'s history:
     - If $pk' \neq pk$, then include the randomness r that $\mathcal{S}$ used in the corresponding $\mathsf{Enc\text{-}M}$ query into $\mathcal{Q}$'s state.
     - If $pk' = pk$, then find $(c, \perp_m, r, x, pk)$ in EncL, update it to (c, m, r, x, pk), and include r into $\mathcal{Q}$'s state. Execute $\mathtt{Program}(m, c, r)$.
   - If $\mathcal{Q}$ is the receiver, i.e., $\mathsf{sid} = (\mathcal{Q}, \mathsf{sid}')$, then include the randomness r used at key generation into $\mathcal{Q}$'s state, and for all remaining $(c, \perp_m, r, x, pk)$ in EncL do:
   - Send $(\mathsf{Decrypt}, \mathsf{sid}, c)$ to $\mathcal{F}_{\mathsf{pke}}^{\mathcal{L}}$ in name of $\mathcal{Q}$ and wait for response $(\mathsf{Plaintext}, \mathsf{sid}, m)$.
   - If $m \neq \perp_m$, then execute $\mathtt{Program}(m, c, r)$.
   - Update record $(c, \perp_m, r, x, pk)$ in EncL to (c, m, r, x, pk)

Figure 3.6: The EUC simulator $\mathcal{S}$ for protocol π_{PKE}.

```
On input $(m, c, r)$ do the following:
  • Parse $(m_1, \ldots, m_k) := \mathsf{EC}(m)$, and $c := (c_1, c_{2,1}, \ldots, c_{2,k'}, c_3)$;
    let $x := \mathsf{OWTP.Sample}(\Sigma; r)$.
  • For $i = 1, \ldots, k$:
      – Execute $\mathcal{G}_{\mathsf{pRO}}.\mathtt{Program}(x \| i, m_i \oplus c_{2,i})$ ; abort if unsuccessful.
  • Execute $\mathcal{G}_{\mathsf{pRO}}.\mathtt{Program}(x \| k \| m, c_3)$ ; abort if unsuccessful.
```

Figure 3.7: The oracle programming routine `Program` .

that x is uniformly distributed over Σ which has size at least 2^κ, the probability of a collision occurring over q_{E} encryption queries is at most $q_{\mathrm{E}}^2/2^\kappa$.

Let Game 1 to Game q_{E} be games for a hybrid argument where gradually all ciphertexts by honest users are replaced with dummy ciphertexts. Let Game i be the game with a functionality $\mathcal{F}_{\mathsf{i}}$ and simulator $\mathcal{S}_i$ where the first $i-1$ $\mathsf{Enc\text{-}L}$ inputs of $\mathcal{F}_{\mathsf{i}}$ to $\mathcal{S}_i$ include only the leakage $\mathcal{L}(m)$, and the remaining such inputs include the full message. For the first $i-1$ encryptions, $\mathcal{S}_i$ creates a dummy ciphertext and programs the random oracle upon corruption of the party or the receiver as done by $\mathcal{S}$ in Figure 3.6, aborting in case programming fails. For the remaining $\mathsf{Enc\text{-}L}$ inputs, $\mathcal{S}_i$ generates honest encryptions of the real message.

One can see that Game q_{E} is identical to the ideal world with $\mathcal{F}_{\mathsf{pke}}^{\mathcal{L}}$ and $\mathcal{S}$. To have a non-negligible advantage distinghuishing the real from the ideal world, there must exist an $i \in \{1, \ldots, q_{\mathrm{E}}\}$ such that $\mathcal{Z}$ and $\mathcal{A}$ can distinguish between Game $(i-1)$ and Game i. These games are actually identical, *except* in the case that abort happens during the programming of the random oracle $\mathcal{G}_{\mathsf{pRO}}$ for the i-th ciphertext, which is a real ciphertext in Game $(i-1)$ and a dummy ciphertext in Game i. We call this the $\mathsf{ROABORT}$ event. We show that if there exists an environment $\mathcal{Z}$ and real-world adversary $\mathcal{A}$ that make $\mathsf{ROABORT}$ happen with non-negligible probability ν, then we can construct an efficient algorithm $\mathcal{B}$ (the "reduction") with black-box access to $\mathcal{Z}$ and $\mathcal{A}$ that is able to invert OWTP.

Our reduction $\mathcal{B}$ must only simulate honest parties, and in particular must provide to $\mathcal{A}$ a consistent view of their secrets (randomness used for encryption, secret keys, and decrypted plaintexts, just like $\mathcal{S}$ does) when they become corrupted. Moreover, since we are not in the idealized scenario, there is no external global random oracle functionality $\mathcal{G}_{\mathsf{pRO}}$: instead, $\mathcal{B}$ simulates $\mathcal{G}_{\mathsf{pRO}}$ for all the parties involved, and answers all their oracle calls.

Upon input the OWTP challenge (Σ, φ, y), $\mathcal{B}$ runs the code of Game $(i-1)$, but sets the public key of the receiver to $pk = (\Sigma, \varphi)$. Algorithm $\mathcal{B}$ answers the first $i-1$ encryption requests with dummy ciphertexts and the $(i+1)$-st to q_{E}-th queries with honestly generated ciphertexts. For the i-th encryption request, however, it returns a special dummy ciphertext with $c_1 = y$.

To simulate $\mathcal{G}_{\mathsf{pRO}}$, $\mathcal{B}$ maintains an initially empty list $\mathsf{List}_{\mathcal{H}}$ to which pairs (m, h) are either added by lazy sampling for $\mathsf{HashQuery}$ queries, or by programming for $\mathsf{ProgramRO}$ queries. (Remember that the environment $\mathcal{Z}$ can program entries in $\mathcal{G}_{\mathsf{pRO}}$ as well.) For requests from $\mathcal{Z}$, $\mathcal{B}$ actually performs some additional steps that we describe further below.

It answers $\mathsf{Decrypt}$ requests for a ciphertext $c = (c_1, c_{2,1}, \ldots, c_{2,k}, c_3)$ by searching for a pair of the form $(x\|k\|m, c_3) \in \mathsf{List}_{\mathcal{H}}$ such that $\varphi(x) = c_1$ and $m = \mathsf{DC}(c_{2,1} \oplus h_1, \ldots, c_{2,k} \oplus h_k)$, where $h_j = \mathcal{H}(x\|j)$, meaning that h_j is assigned the value of a simulated request $(\mathsf{HashQuery}, x\|j)$ to $\mathcal{G}_{\mathsf{pRO}}$. Note that at most one such pair exists for a given ciphertext c, because if a second $(x'\|k\|m', c_3) \in \mathsf{List}_{\mathcal{H}}$ would exist, then it must hold that $\varphi(x') = c_1$. Because φ is a permutation, this means that $x = x'$. Since for each $j = 1, \ldots, k$, only one pair $(x\|j, h_j) \in \mathsf{List}_{\mathcal{H}}$ can be registered, this means that $m' = \mathsf{DC}(c_{2,1} \oplus h_1, \ldots, c_{2,k} \oplus h_k) = m$ because DC is deterministic. If such a pair $(x\|k\|m, c_3)$ exists, it returns m, otherwise it rejects by returning $\perp_m$.

One problem with the decryption simulation above is that it does not necessarily create the same entries into $\mathsf{List}_{\mathcal{H}}$ as an honest decryption would have, and $\mathcal{Z}$ could detect this by checking whether programming for these entries succeeds. In particular, $\mathcal{Z}$ could first ask to decrypt a ciphertext $c = (\varphi(x), c_{2,1}, \ldots, c_{2,k}, c_3)$ for random $x, c_{2,1}, \ldots, c_{2,k}, c_3$ and then try to program the random oracle on any of the points $x\|j$ for $j = 1, \ldots, k$ or on $x\|k\|m$. In Game $(i-1)$ and Game i, such programming would fail because the entries were created during the decryption of c. In the simulation by $\mathcal{B}$, however, programming would succeed, because no valid pair $(x\|k\|m, c_3) \in \mathsf{List}_{\mathcal{H}}$ was found to perform decryption.

To preempt the above problem, $\mathcal{B}$ checks all incoming requests $\mathsf{HashQuery}$ and $\mathsf{ProgramRO}$ by $\mathcal{Z}$ for points of the form $x\|j$ or $x\|k\|m$ against all previous decryption queries $c = (c_1, c_{2,1}, \ldots, c_{2,k}, c_3)$. If

$\varphi(x) = c_1$, then $\mathcal{B}$ immediately triggers the creation of all random-oracle entries (by making appropriate HashQuery calls) that would have been generated by a decryption of c by computing $m' = \mathsf{DC}(c_{2,1} \oplus \mathcal{H}(x\|1), \ldots, c_{2,k} \oplus \mathcal{H}(x\|k))$ and $c_3' = \mathcal{H}(x\|k\|m')$. Only then does $\mathcal{B}$ handle $\mathcal{Z}$'s original HashQuery or ProgramRO request.

The only remaining problem is if during this procedure $c_3' = c_3$, meaning that c was previously rejected during by $\mathcal{B}$, but it becomes a valid ciphertext by the new assignment of $\mathcal{H}(x\|k\|m) = c_3' = c_3$. This happens with negligible probability, though: a random value c_3' will only hit a fixed c_3 with probability $1/|\Sigma| \leq 1/2^\kappa$. Since up to q_D ciphertexts may have been submitted with the same first component $c_1 = \varphi(x)$ and with different values for c_3, the probability that it hits any of them is at most $q_\mathrm{D}/2^\kappa$. The probability that this happens for at least one of $\mathcal{Z}$'s q_H HashQuery queries or one of its q_P ProgramRO queries during the entire execution is at most $(q_\mathrm{H} + q_\mathrm{P})q_\mathrm{D}/2^\kappa$.

When $\mathcal{A}$ corrupts a party, $\mathcal{B}$ provides the encryption randomness that it used for all ciphertexts that such party generated. If $\mathcal{A}$ corrupts the receiver or the party that generated the i-th ciphertext, then $\mathcal{B}$ cannot provide that randomness. Remember, however, that $\mathcal{B}$ is running $\mathcal{Z}$ and $\mathcal{A}$ in the hope for the ROABORT event to occur, meaning that the programming of values for the i-th ciphertext fails because the relevant points in $\mathcal{G}_{\mathsf{pRO}}$ have been assigned already. Event ROABORT can only occur at the corruption of either the receiver or of the party that generated the i-th ciphertext, whichever comes first. Algorithm $\mathcal{B}$ therefore checks $\mathsf{List}_\mathcal{H}$ for points of the form $x\|j$ or $x\|k\|m$ such that $\varphi(x) = y$. If ROABORT occurred, then $\mathcal{B}$ will find such a point and output x as its preimage for y. If it did not occur, then $\mathcal{B}$ gives up. Overall, $\mathcal{B}$ will succeed whenever ROABORT occurs. Given that Game $(i-1)$ and Game i are different only when ROABORT occurs, and given that $\mathcal{Z}$ and $\mathcal{A}$ have non-negligible probability of distinguishing between Game $(i-1)$ and Game i, we conclude that $\mathcal{B}$ succeeds with non-negligible probability. □

3.4 Restricted Programmable Global Random Oracles

The strict and the programmable global random oracles, $\mathcal{G}_{\mathsf{sRO}}$ and $\mathcal{G}_{\mathsf{pRO}}$, respectively, do not give the simulator any extra power compared to the real world adversary/environment. Canetti and Fischlin [CF01] proved that it is impossible to realize UC commitments without a setup assumption that gives the simulator an advantage over the environment. This means that, while $\mathcal{G}_{\mathsf{sRO}}$ and $\mathcal{G}_{\mathsf{pRO}}$ allowed for security proofs of many practical schemes, we cannot hope to realize even the seemingly simple task of UC commitments with this setup. In this section, we turn our attention to programmable global random oracles that do grant an advantage to the simulator.

3.4.1 Restricting Programmability to the Simulator

Canetti et al. [CJS14] defined a global random oracle that restricts observability only adversarial queries, (hence, we call it the *restricted observable global random oracle* $\mathcal{G}_{\mathsf{roRO}}$), and show that this is sufficient to construct UC commitments. More precisely, if sid is the identifier of the challenge session, a list of so-called *illegitimate* queries for sid can be obtained by the adversary, which are queries made on inputs of the form $(\mathsf{sid}, \ldots)$ by machines that are not part of session sid. If honest parties only make legitimate queries, then clearly this restricted observability will not give the adversary any new information, as it contains only queries made by the adversary. In the ideal world, however, the simulator $\mathcal{S}$ can observe all queries made through corrupt machines within the challenge session sid as it is the ideal-world attacker, which means it will see all legitimate queries in sid. With the observability of illegitimate queries, that means $\mathcal{S}$ can observe *all* hash queries of the form $(\mathsf{sid}, \ldots)$, regardless of whether they are made by honest or corrupt parties, whereas the real-world attacker does not learn anything form the observe interface.

We recall the restricted observable global random oracle $\mathcal{G}_{\mathsf{roRO}}$ due to Canetti et al. [CJS14] in a slightly modified form in Fig. 3.8. In their definition, it allows *ideal functionalities* to obtain the illegitimate queries corresponding to their own session. These functionalities then allow the adversary to obtain the illegitimate queries by forwarding the request to the global random oracle. Since the adversary can spawn any new machine, and in particular an ideal functionality, the adversary can create such an ideal functionality and use it to obtain the illegitimate queries. We chose to explicitly model this adversarial power by allowing the adversary to query for the illegitimate queries directly.

Also in Fig. 3.8, we define a *restricted* programmable global random oracle $\mathcal{G}_{\mathsf{rpRO}}$ by using a similar approach to restrict programming ac-

$\mathcal{G}_{\mathsf{roRO}}$, $\mathcal{G}_{\mathsf{rpRO}}$, and $\mathcal{G}_{\mathsf{rpoRO}}$ – functionalities of the global random oracle with restricted programming and/or restricted observability.

Parameters: output size function ℓ.
Variables: initially empty lists $\mathsf{List}_{\mathcal{H}}$, prog.

1. `Query`. On input $(\mathsf{HashQuery}, m)$ from a machine $(\mathcal{P}, \mathsf{sid})$ or from the adversary:
    - Look up h such that $(m, h) \in \mathsf{List}_{\mathcal{H}}$. If no such h exists:
        - draw $h \xleftarrow{\$} \{0,1\}^{\ell(\kappa)}$
        - set $\mathsf{List}_{\mathcal{H}} := \mathsf{List}_{\mathcal{H}} \cup \{(m, h)\}$
    - Parse m as (s, m').
    - If this query is made by the adversary, or if $s \neq \mathsf{sid}$, then add (s, m', h) to the (initially empty) list of illegitimate queries $\mathcal{Q}_s$.
    - Output $(\mathsf{HashConfirm}, h)$ to the caller.
2. `Observe`. ($\mathcal{G}_{\mathsf{roRO}}$ and $\mathcal{G}_{\mathsf{rpoRO}}$ only) On input $(\mathsf{Observe}, \mathsf{sid})$ from the adversary:
    - If $\mathcal{Q}|_{\mathsf{sid}}$ does not exist yet, then set $\mathcal{Q}_{\mathsf{sid}} = \varnothing$.
    - Output $(\mathsf{ListObserve}, \mathcal{Q}_{\mathsf{sid}})$ to the adversary.
3. `Program`. ($\mathcal{G}_{\mathsf{rpRO}}$ and $\mathcal{G}_{\mathsf{rpoRO}}$ only) On input $(\mathsf{ProgramRO}, m, h)$ with $h \in \{0,1\}^{\ell(\kappa)}$ from the adversary:
    - If $\exists\, h' \in \{0,1\}^{\ell(\kappa)}$ such that $(m, h') \in \mathsf{List}_{\mathcal{H}}$ and $h \neq h'$, ignore this input.
    - Set $\mathsf{List}_{\mathcal{H}} := \mathsf{List}_{\mathcal{H}} \cup \{(m, h)\}$ and $\mathsf{prog} := \mathsf{prog} \cup \{m\}$.
    - Output $(\mathsf{ProgramConfirm})$ to the adversary.
4. `IsProgrammed`: ($\mathcal{G}_{\mathsf{rpRO}}$ and $\mathcal{G}_{\mathsf{rpoRO}}$ only) On input $(\mathsf{IsProgrammed}, m)$ from a machine $(\mathcal{P}, \mathsf{sid})$ or from the adversary:
    - If the input was given by $(\mathcal{P}, \mathsf{sid})$, parse m as (s, m'). If $s \neq \mathsf{sid}$, ignore this input.
    - Set $b \leftarrow m \in \mathsf{prog}$ and output $(\mathsf{IsProgrammed}, b)$ to the caller.

Figure 3.8: The global random-oracle functionalities $\mathcal{G}_{\mathsf{roRO}}$, $\mathcal{G}_{\mathsf{rpRO}}$, and $\mathcal{G}_{\mathsf{rpoRO}}$ with restricted observability, restricted programming, and combined restricted observability and programming, respectively. Functionality $\mathcal{G}_{\mathsf{roRO}}$ contains only the `Query` and `Observe` interfaces, $\mathcal{G}_{\mathsf{rpRO}}$ contains only the `Query`, `Program`, and `IsProgrammed` interfaces, and $\mathcal{G}_{\mathsf{rpoRO}}$ contains all interfaces.

cess from the real-world adversary. The adversary can program points, but parties in session sid can *check* whether the random oracle was programmed on a particular point (sid, ...). In the real world, the adversary is allowed to program, but honest parties can check whether points were programmed and can, for example, reject signatures based on a programmed hash. In the ideal world, the simulator controls the corrupt parties in sid and is therefore the only entity that can check whether points are programmed. Note that while it typically internally simulates the real-world adversary that may want to check whether points of the form (sid, ...) are programmed, the simulator can simply "lie" and pretend that no points are programmed. Therefore, the extra power that the simulator has over the real-world adversary is programming points without being detected.

It may seem strange to offer a new interface allowing all parties to check whether certain points are programmed, even though a real-world hash function does not have such an interface. However, we argue that if one accepts a programmable random oracle as a proper idealization of a clearly non-programmable real-world hash function, then it should be a small step to accept the instantiation of the `IsProgrammed` interface that always returns "false" to the question whether any particular entry was programmed into the hash function.

3.4.2 UC-Commitments from $\mathcal{G}_{\mathsf{rpRO}}$

We now show that we can create a UC-secure commitment protocol from $\mathcal{G}_{\mathsf{rpRO}}$. A UC-secure commitment scheme must allow the simulator to extract the message from adversarially created commitments, and to equivocate dummy commitments created for honest committers, i.e., first create a commitment that it can open to any message after committing. Intuitively, achieving the equivocability with a programmable random oracle is simple: we can define a commitment that uses the random-oracle output, and the adversary can later change the committed message by programming the random oracle. Achieving extractability, however, seems difficult, as we cannot extract by observing the random-oracle queries. We overcome this issue with the following approach. The receiver of a commitment chooses a nonce on which we query random oracle, interpreting the random oracle output as a public key pk. Next, the committer encrypts the message to pk and sends the ciphertext to the receiver, which forms the commitment. To open, the committer reveals the message and the randomness used to encrypt it.

This solution is extractable as the simulator that plays the role of receiver can program the random oracle such that it knows the secret key corresponding to pk, and simply decrypt the commitment to find the message. However, we must take care to still achieve equivocability. If we use standard encryption, the simulator cannot open a ciphertext to any value it learns later. The solution is to use *non-committing encryption*, which, as shown in Section 3.3, can be achieved using a programmable random oracle. We use a slightly different encryption scheme, as the security requirements here are slightly less stringent than full non-committing encryption, and care must be taken that we can interpret the result of the random oracle as a public key, which is difficult for constructions based on trapdoor one-way permutations such as RSA. This approach results in a very efficient commitment scheme: with two exponentiations per party (as opposed to five) and two rounds of communication (as opposed to five), it is considerably more efficient than the one of [CJS14].

Let $\mathsf{COM}_{\mathcal{G}_{\mathsf{rpRO}}}$ be the following commitment protocol, parametrized by a group $\mathbb{G} = \langle g \rangle$ of prime order q. We require an algorithm Embed that maps elements of $\{0,1\}^{\ell(\kappa)}$ into $\mathbb{G}$, such that for $h \xleftarrow{\$} \{0,1\}^{\ell(\kappa)}$, $\mathsf{Embed}(h)$ is computationally indistinguishable from uniform in $\mathbb{G}$. Furthermore, we require an efficiently computable probabilistic algorithm Embed^{-1}, such that for all $x \in \mathbb{G}$, $\mathsf{Embed}(\mathsf{Embed}^{-1}(x)) = x$ and for $x \xleftarrow{\$} \mathbb{G}$, $\mathsf{Embed}^{-1}(x)$ is computationally indistinguishable from uniform in $\{0,1\}^{\ell(\kappa)}$. $\mathsf{COM}_{\mathcal{G}_{\mathsf{rpRO}}}$ assumes authenticated channels $\mathcal{F}_{\mathsf{auth}}$ as defined by Canetti [Can00].

1. On input $(\mathsf{Commit}, \mathsf{sid}, x)$, party $\mathcal{C}$ proceeds as follows.
   - Check that $\mathsf{sid} = (\mathcal{C}, \mathcal{R}, \mathsf{sid}')$ for some $\mathcal{C}$, sid'. Send Commit to $\mathcal{R}$ over $\mathcal{F}_{\mathsf{auth}}$ by giving $\mathcal{F}_{\mathsf{auth}}$ input $(\mathsf{Send}, (\mathcal{C}, \mathcal{R}, \mathsf{sid}, 0),$ "Commit").
   - $\mathcal{R}$, upon receiving $(\mathsf{Sent}, (\mathcal{C}, \mathcal{R}, \mathsf{sid}, 0),$ "Commit") from $\mathcal{F}_{\mathsf{auth}}$, takes a nonce $n \xleftarrow{\$} \{0,1\}^{\kappa}$ and sends the nonce back to $\mathcal{C}$ by giving $\mathcal{F}_{\mathsf{auth}}$ input $(\mathsf{Send}, (\mathcal{R}, \mathcal{C}, \mathsf{sid}, 0), n)$.
   - $\mathcal{C}$, upon receiving $(\mathsf{Sent}, (\mathcal{R}, \mathcal{C}, \mathsf{sid}, 0), n)$, queries $\mathcal{G}_{\mathsf{rpRO}}$ on (sid, n) to obtain h_n. It checks whether this point was programmed by giving $\mathcal{G}_{\mathsf{roRO}}$ input $(\mathsf{IsProgrammed}, (\mathsf{sid}, n))$ and aborts if $\mathcal{G}_{\mathsf{roRO}}$ returns $(\mathsf{IsProgrammed}, 1)$.
   - Set $pk \leftarrow \mathsf{Embed}(h_n)$.
   - Pick a random $r \xleftarrow{\$} \mathbb{G}$ and $\rho \in \mathbb{Z}_q$. Set $c_1 \leftarrow g^r$, query $\mathcal{G}_{\mathsf{rpRO}}$

on (sid, pk^r) to obtain h_r and let $c_2 \leftarrow h_r \oplus x$.

- Store (r, x) and send the commitment to $\mathcal{R}$ by giving $\mathcal{F}_{\mathsf{auth}}$ input $(\mathsf{Send}, (\mathcal{C}, \mathcal{R}, \mathsf{sid}, 1), (c_1, c_2))$.
- $\mathcal{R}$, upon receiving $(\mathsf{Sent}, (\mathcal{C}, \mathcal{R}, \mathsf{sid}, 1), (c_1, c_2))$ from $\mathcal{F}_{\mathsf{auth}}$ outputs $(\mathsf{Receipt}, \mathsf{sid})$.

2. On input $(\mathsf{Open}, \mathsf{sid})$, $\mathcal{C}$ proceeds as follows.
   - It sends (r, x) to $\mathcal{R}$ by giving $\mathcal{F}_{\mathsf{auth}}$ input $(\mathsf{Send}, (\mathcal{C}, \mathcal{R}, \mathsf{sid}, 2), (r, x))$.
   - $\mathcal{R}$, upon receiving $(\mathsf{Sent}, (\mathcal{C}, \mathcal{R}, \mathsf{sid}, 1), (r, x))$:
     - Query $\mathcal{G}_{\mathsf{rpRO}}$ on (sid, n) to obtain h_n and compute $pk \leftarrow \mathsf{Embed}(h_n)$.
     - Check that $c_1 = g^r$.
     - Query $\mathcal{G}_{\mathsf{rpRO}}$ on (sid, pk^r) to obtain h_r and check that $c_2 = h_r \oplus x$.
     - Check that none of the points was programmed by giving $\mathcal{G}_{\mathsf{roRO}}$ inputs $(\mathsf{IsProgrammed}, (\mathsf{sid}, n))$ and $(\mathsf{IsProgrammed}, pk^r)$ and asserting that it returns $(\mathsf{IsProgrammed}, 0)$ for both queries.
     - Output $(\mathsf{Open}, \mathsf{sid}, x)$.

$\mathsf{COM}_{\mathcal{G}_{\mathsf{rpRO}}}$ is a secure commitment scheme under the computational Diffie-Hellman assumption, which given a group $\mathbb{G}$ generated by g of prime order q, challenges the adversary to compute $g^{\alpha\beta}$ on input (g^α, g^β), with $(\alpha, \beta) \xleftarrow{\$} \mathbb{Z}_q^2$.

Theorem 5. *$\mathsf{COM}_{\mathcal{G}_{\mathsf{rpRO}}}$ GUC-realizes $\mathcal{F}_{\mathsf{com}}$ (as defined in Figure 2.6) in the $\mathcal{G}_{\mathsf{rpRO}}$ and $\mathcal{F}_{\mathsf{auth}}$ hybrid model under the CDH assumption.*

Proof. By the fact that $\mathsf{COM}_{\mathcal{G}_{\mathsf{rpRO}}}$ is $\mathcal{G}_{\mathsf{rpRO}}$-subroutine respecting and by Theorem 1, it is sufficient to show that $\mathsf{COM}_{\mathcal{G}_{\mathsf{rpRO}}}$ $\mathcal{G}_{\mathsf{rpRO}}$-EUC-realizes $\mathcal{F}_{\mathsf{com}}$.

We describe a simulator $\mathcal{S}$ by defining its behavior in the different corruption scenarios. In all scenarios, whenever the simulated real-world adversary makes an $\mathsf{IsProgrammed}$ query or instructs a corrupt party to make such a query on a point that $\mathcal{S}$ has programmed, the simulator intercepts this query and simply replies $(\mathsf{IsProgrammed}, 0)$, lying that the point was not programmed.

When both the sender and the receiver are honest, $\mathcal{S}$ works as follows.

1. When $\mathcal{F}_{\mathsf{com}}$ asks $\mathcal{S}$ for permission to output $(\mathsf{Receipt}, \mathsf{sid})$:

  - Parse sid as $(\mathcal{C}, \mathcal{R}, \mathsf{sid}')$ and let "$\mathcal{C}$" create a dummy commitment by choosing $r \xleftarrow{\$} \mathbb{Z}_q$, letting $c_1 = g^r$, and choosing $c_2 \xleftarrow{\$} \{0,1\}^{\ell(\kappa)}$.
  - When "$\mathcal{R}$" outputs $(\mathsf{Receipt}, \mathsf{sid})$, allow $\mathcal{F}_{\mathsf{com}}$ to proceed.

2. When $\mathcal{F}_{\mathsf{com}}$ asks $\mathcal{S}$ for permission to output $(\mathsf{Open}, \mathsf{sid}, x)$:
   - Program $\mathcal{G}_{\mathsf{rpRO}}$ by giving $\mathcal{G}_{\mathsf{roRO}}$ input $(\mathsf{ProgramRO}, (\mathsf{sid}, pk^r), c_2 \oplus x)$, such that the commitment (c_1, c_2) commits to x.
   - Give "$\mathcal{C}$" input $(\mathsf{Open}, \mathsf{sid})$ instructing it to open its commitment to x.
   - When "$\mathcal{R}$" outputs $(\mathsf{Open}, \mathsf{sid}, x)$, allow $\mathcal{F}_{\mathsf{com}}$ to proceed.

If the committer is corrupt but the receiver is honest, $\mathcal{S}$ works as follows.

1. When the simulated receiver "$\mathcal{R}$" notices the commitment protocol starting (i.e., receives $(\mathsf{Sent}, (\mathcal{C}, \mathcal{R}, \mathsf{sid}, 0), \text{“}\mathsf{Commit}\text{”})$ from "$\mathcal{F}_{\mathsf{auth}}$"):
   - Choose nonce n as in the protocol.
   - Before sending n, choose $sk \xleftarrow{\$} \mathbb{Z}_q$ and set $pk \leftarrow g^{sk}$.
   - Program $\mathcal{G}_{\mathsf{rpRO}}$ by giving $\mathcal{G}_{\mathsf{rpRO}}$ input $(\mathsf{ProgramRO}, (\mathsf{sid}, n), \mathsf{Embed}^{-1}(pk))$. Note that this simulation will succeed with overwhelming probability as n is freshly chosen, and note that as pk is uniform in $\mathbb{G}$, by definition of Embed^{-1} the programmed value $\mathsf{Embed}^{-1}(pk)$ is uniform in $\{0,1\}^{\ell(\kappa)}$.
   - $\mathcal{S}$ now lets "$\mathcal{R}$" execute the remainder the protocol honestly.
   - When "$\mathcal{R}$" outputs $(\mathsf{Receipt}, \mathsf{sid})$, $\mathcal{S}$ extracts the committed value from (c_1, c_2). Query $\mathcal{G}_{\mathsf{rpRO}}$ on (sid, c_1^{sk}) to obtain h_r and set $x \leftarrow c_2 \oplus h_r$.
   - Make a query with $\mathcal{F}_{\mathsf{com}}$ on $\mathcal{C}$'s behalf by sending $(\mathsf{Commit}, \mathsf{sid}, x)$ on $\mathcal{C}$'s behalf to $\mathcal{F}_{\mathsf{com}}$.
   - When $\mathcal{F}_{\mathsf{com}}$ asks permission to output $(\mathsf{Receipt}, \mathsf{sid})$, allow.
2. When "$\mathcal{R}$" outputs $(\mathsf{Open}, \mathsf{sid}, x)$:
   - Send $(\mathsf{Open}, \mathsf{sid})$ on $\mathcal{C}$'s behalf to $\mathcal{F}_{\mathsf{com}}$.
   - When $\mathcal{F}_{\mathsf{com}}$ asks permission to output $(\mathsf{Open}, \mathsf{sid}, x)$, allow.

If the receiver is corrupt but the committer is honest, $\mathcal{S}$ works as follows.

1. When $\mathcal{F}_{\mathsf{com}}$ asks permission to output $(\mathsf{Receipt}, \mathsf{sid})$:
   - Parse sid as $(\mathcal{C}, \mathcal{R}, \mathsf{sid}')$.

- Allow $\mathcal{F}_{\mathsf{com}}$ to proceed.
- When $\mathcal{F}_{\mathsf{com}}$ receives $(\mathsf{Receipt}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{com}}$ as $\mathcal{R}$ is corrupt, it simulates "$\mathcal{C}$" by choosing $r \xleftarrow{\$} \mathbb{Z}_q$, computing $c_1 = g^r$, and choosing $c_2 \xleftarrow{\$} \{0,1\}^{\ell(\kappa)}$.

2. When $\mathcal{F}_{\mathsf{com}}$ asks permission to output $(\mathsf{Open}, \mathsf{sid}, x)$:
    - Allow $\mathcal{F}_{\mathsf{com}}$ to proceed.
    - When $\mathcal{S}$ receives $(\mathsf{Open}, \mathsf{sid}, x)$ from $\mathcal{F}_{\mathsf{com}}$ as $\mathcal{R}$ is corrupt, $\mathcal{S}$ programs $\mathcal{G}_{\mathsf{rpRO}}$ by giving $\mathcal{G}_{\mathsf{rpRO}}$ input $(\mathsf{ProgramRO}, (\mathsf{sid}, pk^r), c_2 \oplus x)$, such that the commitment (c_1, c_2) commits to x.
    - $\mathcal{S}$ inputs $(\mathsf{Open}, \mathsf{sid})$ to "$\mathcal{C}$", instructing it to open its commitment to x.

What remains to show is that $\mathcal{S}$ is a satisfying simulator, i.e., no $\mathcal{G}_{\mathsf{rpRO}}$-externally constrained environment can distinguish $\mathcal{F}_{\mathsf{com}}$ and $\mathcal{S}$ from $\mathsf{COM}_{\mathcal{G}_{\mathsf{rpRO}}}$ and $\mathcal{A}$. When simulating an honest receiver, $\mathcal{S}$ extracts the committed message correctly: Given pk and $c_1 = g^r$ for some r, there is a unique value pk^r, and the message x is uniquely determined by c_2 and pk^r. Simulator $\mathcal{S}$ also simulates an honest committer correctly. When committing, it does not know the message, but can still produce a commitment that is identically distributed as long as the environment does not query the random oracle on (sid, pk^r). When $\mathcal{S}$ later learns the message x, it must equivocate the commitment to open to x, by programming $\mathcal{G}_{\mathsf{rpRO}}$ on (sid, pk^r), which again succeeds unless the environment makes a random oracle query on (sid, pk^r). If there is an environment that triggers such a $\mathcal{G}_{\mathsf{rpRO}}$ with non-negligible probability, we can construct an attacker $\mathcal{B}$ that breaks the CDH problem in $\mathbb{G}$.

Our CDH attacker $\mathcal{B}$ plays the role of $\mathcal{F}_{\mathsf{com}}$, $\mathcal{S}$, and $\mathcal{G}_{\mathsf{rpRO}}$, and has black-box access to the environment. $\mathcal{B}$ receives CDH problem g^α, g^β and is challenged to compute $g^{\alpha\beta}$. It simulates $\mathcal{G}_{\mathsf{rpRO}}$ to return $h_n \leftarrow \mathsf{Embed}^{-1}(g^\alpha)$ on random query (sid, n). When simulating an honest committer committing with respect to this pk, set $c_1 \leftarrow g^\beta$ and $c_2 \xleftarrow{\$} \{0,1\}^{\ell(\kappa)}$. Note that $\mathcal{S}$ cannot successfully open this commitment, but remember that we consider an environment that with non-negligible probability makes a $\mathcal{G}_{\mathsf{rpRO}}$ query on $pk^r (= g^{\alpha\beta})$ before the commitment is being opened. Next, $\mathcal{B}$ will choose a random $\mathcal{G}_{\mathsf{rpRO}}$ query on (sid, m). With nonnegligible probability, we have $m = g^{\alpha\beta}$, and $\mathcal{B}$ found the solution to the CDH challenge. $\square$

3.4.3 Adding Observability for Efficient Commitments

While the commitment scheme $\mathsf{COM}_{\mathcal{G}_{\mathsf{rpRO}}}$ from the restricted programmable global random oracle is efficient for a composable commitment scheme, there is still a large efficiency gap between composable commitments from global random oracles and standalone commitments or commitments from local random oracles. Indeed, $\mathsf{COM}_{\mathcal{G}_{\mathsf{rpRO}}}$ still requires multiple exponentiations and rounds of interaction, whereas the folklore commitment scheme $c = \mathcal{H}(m\|r)$ for message m and random opening information r consists of computing a single hash function.

We extend $\mathcal{G}_{\mathsf{rpRO}}$ to, on top of programmability, offer the restricted observability interface of the global random oracle due to Canetti et al. [CJS14]. With this restricted programmable *and observable* global random oracle $\mathcal{G}_{\mathsf{rpoRO}}$ (as shown in Figure 3.8), we can close this efficiency gap and prove that the folklore commitment scheme above is a secure composable commitment scheme with a global random oracle.

Let $\mathsf{COM}_{\mathcal{G}_{\mathsf{rpoRO}}}$ be the commitment scheme that simply hashes the message and opening, phrased as a GUC protocol using $\mathcal{G}_{\mathsf{rpoRO}}$ and using authenticated channels, which is formally defined as follows.

1. On input $(\mathsf{Commit}, \mathsf{sid}, x)$, party C proceeds as follows.
   - Check that $\mathsf{sid} = (C, R, \mathsf{sid}')$ for some C, sid'.
   - Pick $r \xleftarrow{\$} \{0,1\}^{\kappa}$ and query $\mathcal{G}_{\mathsf{rpoRO}}$ on (sid, r, x) to obtain c.
   - Send c to R by giving $\mathcal{F}_{\mathsf{auth}}$ input $(\mathsf{Send}, (C, R, \mathsf{sid}, 0), c)$.
   - R, upon receiving $(\mathsf{Sent}, (C, R, \mathsf{sid}, 0), c)$ from $\mathcal{F}_{\mathsf{auth}}$, outputs $(\mathsf{Receipt}, \mathsf{sid})$.
2. On input $(\mathsf{Open}, \mathsf{sid})$, C proceeds as follows.
   - It sends (r, x) to R by giving $\mathcal{F}_{\mathsf{auth}}$ input $(\mathsf{Send}, (C, R, \mathsf{sid}, 1), (r, x))$.
   - R, upon receiving subroutine output $(\mathsf{Sent}, (C, R, \mathsf{sid}, 1), (r, x))$ from $\mathcal{F}_{\mathsf{auth}}$, queries $\mathcal{G}_{\mathsf{rpoRO}}$ on (sid, r, x) and checks that the result is equal to c, and checks that (sid, r, x) is not programmed by giving $\mathcal{G}_{\mathsf{rpoRO}}$ input $(\mathsf{IsProgrammed}, (\mathsf{sid}, r, x))$ and aborting if the result is not $(\mathsf{IsProgrammed}, 0)$. Output $(\mathsf{Open}, \mathsf{sid}, x)$.

Theorem 6. $\mathsf{COM}_{\mathcal{G}_{\mathsf{rpoRO}}}$ *GUC-realizes* $\mathcal{F}_{\mathsf{com}}$ *(as defined in Figure 2.6), in the* $\mathcal{G}_{\mathsf{rpoRO}}$ *and* $\mathcal{F}_{\mathsf{auth}}$ *hybrid model.*

Proof. By the fact that $\mathsf{COM}_{\mathcal{G}_{\mathsf{rpoRO}}}$ is $\mathcal{G}_{\mathsf{rpoRO}}$-subroutine respecting and by Theorem 1, it is sufficient to show that $\mathsf{COM}_{\mathcal{G}_{\mathsf{rpoRO}}}$ $\mathcal{G}_{\mathsf{rpoRO}}$-EUC-realizes $\mathcal{F}_{\mathsf{com}}$.

We define a simulator $\mathcal{S}$ by describing its behavior in the different corruption scenarios. For all scenarios, $\mathcal{S}$ will internally simulate $\mathcal{A}$ and forward any messages between $\mathcal{A}$ and the environment, the corrupt parties, and $\mathcal{G}_{\mathsf{rpoRO}}$. It stores all $\mathcal{G}_{\mathsf{rpoRO}}$ queries that it makes for $\mathcal{A}$ and for corrupt parties. Only when $\mathcal{A}$ directly or through a corrupt party makes an `IsProgrammed` query on a point that $\mathcal{S}$ programmed, $\mathcal{S}$ will not forward this query to $\mathcal{G}_{\mathsf{rpoRO}}$ but instead return $(\mathsf{IsProgrammed}, 0)$. When we say that $\mathcal{S}$ queries $\mathcal{G}_{\mathsf{rpoRO}}$ on a point (s, m) where s is the challenge sid, for example when simulating an honest party, it does so through a corrupt dummy party that it spawns, such that the query is not marked as illegitimate.

When both the sender and the receiver are honest, $\mathcal{S}$ works as follows.

1. When $\mathcal{F}_{\mathsf{com}}$ asks $\mathcal{S}$ for permission to output $(\mathsf{Receipt}, \mathsf{sid})$:
   - Parse sid as (C, R, sid') and let "C" commit to a dummy value by giving it input $(\mathsf{Commit}, \mathsf{sid}, \perp)$, except that it takes $c \xleftarrow{\$} \{0,1\}^{\ell(\kappa)}$ instead of following the protocol.
   - When "R" outputs $(\mathsf{Receipt}, \mathsf{sid})$, allow $\mathcal{F}_{\mathsf{com}}$ to proceed.
2. When $\mathcal{F}_{\mathsf{com}}$ asks $\mathcal{S}$ for permission to output $(\mathsf{Open}, \mathsf{sid}, x)$:
   - Choose a random $r \xleftarrow{\$} \{0,1\}^{\kappa}$ and program $\mathcal{G}_{\mathsf{rpoRO}}$ by giving it input $(\mathsf{ProgramRO}, (\mathsf{sid}, r, x), c)$, such that the commitment c commits to x. Note that since r is freshly chosen at random, the probability that $\mathcal{G}_{\mathsf{rpoRO}}$ is already defined on (sid, r, x) is negligible, so the programming will succeed with overwhelming probability.
   - Give "C" input $(\mathsf{Open}, \mathsf{sid})$ instructing it to open its commitment to x.
   - When "R" outputs $(\mathsf{Open}, \mathsf{sid}, x)$, allow $\mathcal{F}_{\mathsf{com}}$ to proceed.

If the committer is corrupt but the receiver is honest, $\mathcal{S}$ works as follows.

1. When simulated receiver "R" outputs $(\mathsf{Receipt}, \mathsf{sid})$:
   - Obtain the list $\mathcal{Q}_{\mathsf{sid}}$ of all random oracle queries of form $(\mathsf{sid}, \ldots)$, by combining the queries that $\mathcal{S}$ made on behalf of the corrupt parties and the simulated honest parties, and

by obtaining the illegitimate queries made outside of $\mathcal{S}$ by giving $\mathcal{G}_{\mathsf{rpoRO}}$ input $(\mathsf{Observe}, \mathsf{sid})$.
- Find a non-programmed record $((\mathsf{sid}, r, x), c) \in \mathcal{Q}_{\mathsf{sid}}$. If no such record is found, set x to a dummy value.
- Make a query with $\mathcal{F}_{\mathsf{com}}$ on C's behalf by sending $(\mathsf{Commit}, \mathsf{sid}, x)$ on C's behalf to $\mathcal{F}_{\mathsf{com}}$.
- When $\mathcal{F}_{\mathsf{com}}$ asks permission to output $(\mathsf{Receipt}, \mathsf{sid})$, allow.

2. When "R" outputs $(\mathsf{Open}, \mathsf{sid}, x)$:
   - Send $(\mathsf{Open}, \mathsf{sid})$ on C's behalf to $\mathcal{F}_{\mathsf{com}}$.
   - When $\mathcal{F}_{\mathsf{com}}$ asks permission to output $(\mathsf{Open}, \mathsf{sid}, x)$, allow.

If the receiver is corrupt but the committer is honest, $\mathcal{S}$ works as follows.

1. When $\mathcal{F}_{\mathsf{com}}$ asks permission to output $(\mathsf{Receipt}, \mathsf{sid})$:
   - Parse sid as (C, R, sid').
   - Allow $\mathcal{F}_{\mathsf{com}}$ to proceed.
   - When $\mathcal{S}$ receives $(\mathsf{Receipt}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{com}}$ as R is corrupt, it simulates "C" by choosing $c \xleftarrow{\$} \{0,1\}^{\ell(\kappa)}$ instead of following the protocol.
2. When $\mathcal{F}_{\mathsf{com}}$ asks permission to output $(\mathsf{Open}, \mathsf{sid}, x)$:
   - Allow $\mathcal{F}_{\mathsf{com}}$ to proceed.
   - When $\mathcal{S}$ receives $(\mathsf{Open}, \mathsf{sid}, x)$ from $\mathcal{F}_{\mathsf{com}}$ as R is corrupt, choose $r \xleftarrow{\$} \{0,1\}^{\kappa}$ and program $\mathcal{G}_{\mathsf{rpoRO}}$ by giving it input $(\mathsf{ProgramRO}, (\mathsf{sid}, r, x), c)$, such that the commitment c commits to x. Note that since r is freshly chosen at random, the probability that $\mathcal{G}_{\mathsf{rpoRO}}$ is already defined on (sid, r, x) is negligible, so the programming will succeed with overwhelming probability.
   - $\mathcal{S}$ inputs $(\mathsf{Open}, \mathsf{sid})$ to "C", instructing it to open its commitment to x.

We must show that $\mathcal{S}$ extracts the correct value from a corrupt commitment. It obtains a list of all $\mathcal{G}_{\mathsf{rpoRO}}$ queries of the form $(\mathsf{sid}, \ldots)$ and looks for a non-programmed entry (sid, r, x) that resulted in output c. If this does not exist, then the environment can only open its commitment successfully by later finding a preimage of c, as the honest receiver will check that the point was not programmed. Finding such a preimage happens with negligible probability, so committing to a dummy value is sufficient. The probability that there are multiple satisfying entries is also negligible, as this means the environment found collisions on the random oracle.

Next, we argue that the simulated commitments are indistinguishable from honest commitments. Observe that the commitment c is distributed equally to real commitments, namely uniform in $\{0,1\}^{\ell(\kappa)}$. The simulator can open this value to the desired x if programming the random oracle succeeds. As it first takes a fresh nonce $r \xleftarrow{\$} \{0,1\}^{\kappa}$ and programs (sid, r, x), the probability that $\mathcal{G}_{\mathsf{rpoRO}}$ is already defined on this input is negligible. $\square$

3.5 Unifying the Different Global Random Oracles

At this point, we have considered several notions of global random oracles that differ in whether they offer programmability or observability, and in whether this power is restricted to machines within the local session, or also available to other machines. Having several coexisting variants of global random oracles, each with their own set of schemes that they can prove secure, is somewhat unsatisfying. Indeed, if different schemes require different random oracles that in practice end up being replaced with the same hash function, then we're back to the problem that motivated the concept of global random oracles.

We were able to distill a number of relations and transformations among the different notions, allowing a protocol that realizes a functionality with access to one type of global random oracle to be efficiently transformed into a protocol that realizes the same functionality with respect to a different type of global random oracle. A graphical representation of our transformation is given in Figure 3.9.

The transformations are very simple and hardly affect efficiency of the protocol. The $\mathsf{s2ro}$ transformation takes as input a $\mathcal{G}_{\mathsf{sRO}}$-subroutine-respecting protocol π and transforms it into a $\mathcal{G}_{\mathsf{roRO}}$-subroutine respecting protocol $\pi' = \mathsf{s2ro}(\pi)$ by replacing each query $(\mathsf{HashQuery}, m)$ to $\mathcal{G}_{\mathsf{sRO}}$ with a query $(\mathsf{HashQuery}, (\mathsf{sid}, m))$ to $\mathcal{G}_{\mathsf{roRO}}$, where sid is the session identifier of the calling machine. Likewise, the $\mathsf{p2rp}$ transformation takes as input a $\mathcal{G}_{\mathsf{pRO}}$-subroutine-respecting protocol π and transforms it into a $\mathcal{G}_{\mathsf{rpRO}}$-subroutine respecting protocol $\pi' = \mathsf{p2rp}(\pi)$ by replacing each query $(\mathsf{HashQuery}, m)$ to $\mathcal{G}_{\mathsf{pRO}}$ with a query $(\mathsf{HashQuery}, (\mathsf{sid}, m))$ to $\mathcal{G}_{\mathsf{rpRO}}$ and replacing each query $(\mathsf{ProgramRO}, m, h)$ to $\mathcal{G}_{\mathsf{pRO}}$ with a

$$\begin{tikzcd}
 & \mathcal{G}_{\mathsf{sRO}} \arrow[r, "\mathsf{s2ro}"] & \mathcal{G}_{\mathsf{roRO}} \\
\mathcal{G}_{\mathsf{pRO}} \arrow[r, "\mathsf{p2rp}"] & \mathcal{G}_{\mathsf{rpRO}} \arrow[r, "\mathsf{rp2rpo}"] & \mathcal{G}_{\mathsf{rpoRO}}
\end{tikzcd}$$

Figure 3.9: Relations between different notions of global random oracles. An arrow from $\mathcal{G}$ to $\mathcal{G}'$ indicates the existence of simple transformation such that any protocol that $\mathcal{G}$-EUC-realizes a functionality $\mathcal{F}$, the transformed protocol $\mathcal{G}'$-EUC-realizes the transformed functionality $\mathcal{F}$ (cf. Theorem 7).

query $(\mathsf{ProgramRO}, (\mathsf{sid}, m), h)$ to $\mathcal{G}_{\mathsf{rpRO}}$, where sid is the session identifier of the calling machine. The other transformation $\mathsf{rp2rpo}$ simply replaces $\mathsf{HashQuery}$, $\mathsf{ProgramRO}$, and $\mathsf{IsProgrammed}$ queries to $\mathcal{G}_{\mathsf{rpRO}}$ with identical queries to $\mathcal{G}_{\mathsf{rpoRO}}$.

Theorem 7. *Let π be a $\mathcal{G}_{\mathsf{x}\mathrm{RO}}$-subroutine-respecting protocol and let $\mathcal{G}_{\mathsf{y}\mathrm{RO}}$ be such that there is an edge from $\mathcal{G}_{\mathsf{x}\mathrm{RO}}$ to $\mathcal{G}_{\mathsf{y}\mathrm{RO}}$ in Figure 3.9, where $\mathsf{x}, \mathsf{y} \in \{\mathsf{s}, \mathsf{ro}, \mathsf{p}, \mathsf{rp}, \mathsf{rpo}\}$. Then if π $\mathcal{G}_{\mathsf{x}\mathrm{RO}}$-EUC-realizes a functionality $\mathcal{F}$, where $\mathcal{F}$ is an ideal functionality that does not communicate with $\mathcal{G}_{\mathsf{x}\mathrm{RO}}$, then $\pi' = \mathsf{x2y}(\pi)$ is a $\mathcal{G}_{\mathsf{y}\mathrm{RO}}$-subroutine-respecting protocol that $\mathcal{G}_{\mathsf{y}\mathrm{RO}}$-EUC-realizes $\mathcal{F}$.*

Proof sketch. We first provide some detail for the $\mathsf{s2ro}$ transformation. The other transformations can be proved in a similar fashion, so we only provide an intuition here.

As protocol π $\mathcal{G}_{\mathsf{sRO}}$-EUC-realizes $\mathcal{F}$, there exists a simulator $\mathcal{S}_{\mathsf{s}}$ that correctly simulates the protocol with respect to the dummy adversary. Observe that $\mathcal{G}_{\mathsf{roRO}}$ offers the same $\mathsf{HashQuery}$ interface to the adversary as $\mathcal{G}_{\mathsf{sRO}}$, and that the $\mathcal{G}_{\mathsf{roRO}}$ only gives the simulator extra powers. Therefore, given the dummy-adversary simulator $\mathcal{S}_{\mathsf{s}}$ for π, one can build a dummy-adversary simulator $\mathcal{S}_{\mathsf{ro}}$ for $\mathsf{s2ro}(\pi)$ as follows. If the environment makes a query $(\mathsf{HashQuery}, x)$, either directly through the dummy adversary, or indirectly by instructing a corrupt party to make that query, $\mathcal{S}_{\mathsf{ro}}$ checks whether x can be parsed as (sid, x') where sid is the challenge session. If so, then it passes a direct or indirect query $(\mathsf{HashQuery}, x')$ to $\mathcal{S}_{\mathsf{s}}$, depending whether the environment's original query was direct or indirect. If x cannot be parsed as (sid, x'), then it simply relays the query to $\mathcal{G}_{\mathsf{roRO}}$. Simulator $\mathcal{S}_{\mathsf{ro}}$ relays $\mathcal{S}_{\mathsf{s}}$'s inputs to and outputs from $\mathcal{F}$. When $\mathcal{S}_{\mathsf{s}}$ makes a $(\mathsf{HashQuery}, x')$ query to

$\mathcal{G}_{\mathsf{sRO}}$, $\mathcal{S}_{\mathsf{ro}}$ makes a query $(\mathsf{HashQuery}, (\mathsf{sid}, x'))$ to $\mathcal{G}_{\mathsf{roRO}}$ and relays the response back to $\mathcal{S}_{\mathsf{s}}$. Finally, $\mathcal{S}_{\mathsf{ro}}$ simply relays any $\mathsf{Observe}$ queries by the environment to $\mathcal{G}_{\mathsf{roRO}}$. Note, however, that these queries do not help the environment in observing the honest parties, as they only make legitimate queries.

To see that $\mathcal{S}_{\mathsf{ro}}$ is a good simulator for $\mathsf{s2ro}(\pi)$, we show that if there exists a distinguishing dummy-adversary environment $\mathcal{Z}_{\mathsf{ro}}$ for $\mathsf{s2ro}(\pi)$ and $\mathcal{S}_{\mathsf{ro}}$, then there also exists a distinguishing environment $\mathcal{Z}_{\mathsf{s}}$ for π and $\mathcal{S}_{\mathsf{s}}$, which would contradict the security of π. The environment $\mathcal{Z}_{\mathsf{s}}$ runs $\mathcal{Z}_{\mathsf{ro}}$ by internally executing the code of $\mathcal{G}_{\mathsf{roRO}}$ to respond to $\mathcal{Z}_{\mathsf{ro}}$'s $\mathcal{G}_{\mathsf{roRO}}$ queries, except for queries $(\mathsf{HashQuery}, x)$ where x can be parsed as (sid, x'), for which $\mathcal{Z}_{\mathsf{s}}$ reaches out to its own $\mathcal{G}_{\mathsf{sRO}}$ functionality with a query $(\mathsf{HashQuery}, x')$.

The $\mathsf{p2rp}$ transformation is very similar to $\mathsf{s2ro}$ and prepends sid to random oracle queries. Moving to the *restricted* programmable RO only reduces the power of the adversary by making programming detectable to honest users through the $\mathsf{IsProgrammed}$ interface. The simulator, however, maintains its power to program without being detected, because it can intercept the environment's $\mathsf{IsProgrammed}$ queries for the challenge sid and pretend that they were not programmed. The environment cannot circumvent the simulator and query $\mathcal{G}_{\mathsf{rpRO}}$ directly, because $\mathsf{IsProgrammed}$ queries for sid must be performed from a machine within sid.

Finally, the $\mathsf{rp2rpo}$ transformation increases the power of both the simulator and the adversary by adding a $\mathsf{Observe}$ interface. Similarly to the $\mathsf{s2ro}$ simulator, however, the interface cannot be used by the adversary to observe queries made by honest parties, as these queries are all legitimate. □

Unfortunately, we were unable to come up with security-preserving transformations from non-programmable to programmable random oracles that apply to any protocol. One would expect that the capability to program random-oracle entries destroys the security of many protocols that are secure for non-programmable random oracles. Often this effect can be mitigated by letting the protocol, after performing a random-oracle query, additionally check whether the entry was programmed through the $\mathsf{IsProgrammed}$ interface, and rejecting or aborting if it was. While this seems to work for signature or commitment schemes where rejection is a valid output, it may not always work for arbitrary protocols with interfaces that may not be able to indicate rejection. We leave the study of more generic relations and transformations between programmable and non-programmable random oracles as interesting future work.

Chapter 4

Delegatable Anonymous Credentials

Anonymous credentials allow users to prove that they were issued certain attributes by an issuer in an anonymous manner, meaning that they are indistinguishable from other users who prove the same statement. However, since the identity of the issuer must be known, anonymous credentials cannot offer anonymity when different intermediate authorities may issue credentials. This chapter introduces the first practical delegatable anonymous credential scheme, which supports exactly this scenario. Moreover, it is the first delegatable credential scheme that supports attributes, and we prove the scheme to be secure in a composable manner. This chapter builds on the results of the previous chapter and is secure with respect to a global random oracle.

4.1 Introduction

Privacy-preserving attribute-based credentials (PABCs) [CDE$^+$14], originally introduced as anonymous credentials [Cha85, CL04], allow users to authenticate to service providers in a privacy-protecting way, only revealing the information absolutely necessary to complete a transaction. The growing legal demands for better protection of personal data and more generally the increasingly stronger security requirements make PABCs a primary ingredient for building secure and privacy-preserving IT systems.

An *(attribute-based) anonymous credential* is a set of attributes certified to a user by an issuer. Every time a user presents her credential, she creates a fresh *token* which is a zero-knowledge proof of possession of a credential. When creating a token, the user can select which attributes she wants to disclose from the credential or choose to include only predicates on the attributes. Verification of a token requires knowledge of the issuer public key only. Despite their strong privacy features, anonymous credentials do reveal the identity of the issuer, which, depending on the use case, still leaks information about the user such as the user's location, organization, or business unit. In practice, credentials are typically issued in a hierarchical manner and thus the chain of issuers will reveal even more information. For instance, consider governmental issued certificates such as drivers licenses, which are typically issued by a local authority whose issuing keys are then certified by a regional authority, etc. So there is a hierarchy of at least two levels if not more. Thus, when a user presents her drivers license to prove her age, the local issuer's public key will reveal her place of residence, which, together with other attributes such as the user's age, might help to identify the user. As another example consider a (permissioned) blockchain. Such a system is run by multiple organizations that issue certificates (possibly containing attributes) to parties that are allowed to submit transactions. By the nature of blockchain, transactions are public or at least viewable by many blockchain members. Recorded transactions are often very sensitive, in particular when they pertain to financial or medical data and thus require protection, including the identity of the transaction originator. Again, issuing credential in a permissioned blockchain is a hierarchical process, typically consisting of two levels, a (possibly distributed) root authority, the first level consisting of CAs by the different organizations running the blockchain, and the second level being users who are allowed to submit transactions.

Delegatable anonymous credentials (DAC), formally introduced by Belenkiy et al. [BCC$^+$09], can solve this problem. They allow the owner of a credential to *delegate* her credential to another user, who, in turn, can delegate it further as well as present it to a verifier for authentication purposes. Thereby, only the identity (or rather the public key) of the initial delegator (root issuer) is revealed for verification. A few DAC constructions have been proposed [CL06, BCC$^+$09, Fuc11, CKLM13a], but none is suitable for practical use for the following reasons:

- While being efficient in a complexity theoretic sense, they are not practical because they use generic zero-knowledge proofs or Groth-

Sahai proofs with many expensive pairing operations and a large credential size.

- The provided constructions are described mostly in a black-box fashion (to hide the complexity of their concrete instantiations), often leaving out the details that would be necessary for their implementation. Therefore, a substantial additional effort would be required to translate these schemes to a software specification or perform a concrete efficiency analysis.

- The existing DAC security models do not consider attributes, which, however, are necessary in many practical applications. Also, extending the proposed schemes to include attributes on different delegation levels is not straightforward and will definitely not improve their efficiency.

- Finally, the existing schemes either do not provide an ideal functionality for DAC ([CL06]) or are proven secure in standalone models ([BCC$^+$09,Fuc11,CKLM13a]) that guarantee security only if a protocol is run in isolation, which is not the case for a real environment. In other words, no security guarantees are provided if they are used to build a system, i.e., the security of the overall system would have to be proved from scratch. This usually results in complex monolithic security proofs that are prone to mistakes and hard to verify.

The main reason why the existing schemes are sufficiently efficient, is that they hide the identities of the delegator and delegatee during credential delegation. Thus privacy is ensured for both delegation and presentation of credentials. While this is a superior privacy guarantee, we think that privacy is not necessary for delegation. Indeed, in real-world scenarios a delegator and a delegatee would typically know each other when an attribute-based credential is delegated, especially in the most common case of a hierarchal issuance. Therefore, we think that ensuring privacy only for presentation is a natural way to model delegatable credentials. Furthermore, revealing the full credential chain including the public keys and attribute values to the delegatee would allow us to avoid using expensive cryptographic building blocks such as generic zero-knowledge proofs, re-randomizable proofs, and malleable signatures.

4.1.1 Our Contribution

Let us look at delegatable credentials with a different privacy assumptions for delegation in mind and see how such system would work. The root delegator (we call it *issuer*) generates a signing and a corresponding verification key and publishes the latter. User A, to whom a credential gets issued on the first level (we call it a *Level-1 credential*), generates a fresh credential secret and a public key and sends the public key to the issuer. The issuer signs this public key together with the set of attributes and sends the generated signature to user A. User A can then delegate her credential further to another user, say B, by signing B's freshly generated credential public key and (possibly another) set of attributes with the credential secret key of user A. A sends her signature together with her original credential and A's attributes to user B. User B's credential, therefore, consists of two signatures with the corresponding attribute sets, credential public keys of user A and user B, and B's credential secret key. User B, using his credential secret key, can delegate his credential further as described above or use it to sign a message by generating a presentation token. The token is essentially a non-interactive zero-knowledge (NIZK) proof of possession of the signatures and the corresponding public keys from the delegation chain that does not reveal their values. The signed attributes can also be only selectively revealed using NIZK. Verification of the token requires only the public key of the issuer and, thus, hides the identities of both users A and B and (selectively) their attributes. Since all attributes, signatures, and public keys are revealed to the delegatee during delegation, we can use the most efficient zero-knowledge proofs (Schnorr proofs) that would make a protocol practical.

Contribution Summary In this chapter, which is based on [CDD17], we propose the first *practical* delegatable anonymous credential system with attributes that is well-suited for real-world applications.

More concretely, we first provide a (surprisingly simple) ideal functionality $\mathcal{F}_{\mathsf{dac}}$ for delegatable credentials with attributes. Attributes can be different on any level of delegation. Each attribute at any level can be selectively revealed when generating presentation token. Tokens can be used to sign arbitrary messages. Privacy is guaranteed only during presentation, during delegation the delegatee knows the full credential chain delegated to her.

Second, we propose a generic DAC construction from signature schemes and zero-knowledge proofs and prove it secure in the universal composability (UC) framework introduced by Canetti [Can01]. Our construction can be used as a secure building block to build a higher-level system as a hybrid protocol, enabling a modular design and simpler security analysis.

Third, we describe a very efficient instantiation of our DAC scheme based on a recent pairing-based signature scheme by Groth [Gro15] and on Schnorr zero-knowledge proofs [Sch90]. We further provide a thorough efficiency analysis of this instantiation and detailed pseudocode that can be easily translated into a computer program. We also discus a few optimization techniques for the type of zero-knowledge proofs we use (i.e., proofs of knowledge of group elements under pairings). These techniques are of independent interest.

Finally, we report on an implementation of our scheme and give concrete performance figures, demonstrating the practicality of our construction. For instance, generating an attribute token with four undisclosed attributes from a delegated credential takes only 50 miliseconds, and verification requires only 40 miliseconds, on a 3.1GHz Intel I7-5557U laptop CPU.

4.1.2 Related Work

There is only a handful of constructions of delegatable anonymous credentials [CL06,BCC$^+$09,Fuc11,CKLM13a]. All of them provide privacy for both delegator and delegatee during credential delegation and presentation. The first one is by Chase and Lysyanskaya [CL06] which uses generic zero-knowledge proofs. The size of a credential in their scheme is exponential in the number of delegations, which, as authors admit themselves, makes it impractical and allows only for a constant number of delegations. Our ideal functionality for DAC is also quite different from the signature of knowledge functionality that they use to build a DAC system. For example, we distinguish between the delegation and presentation interfaces and ping the adversary for the delegation. We also do not require the extractability for the verification interface, which makes our scheme much more efficient.

The construction by Belenkiy et al. [BCC$^+$09] employs Groth-Sahai NIZK proofs and in particular their randomization property. It allows for a polynomial number of delegations and requires a common reference string (CRS). Fuchsbauer [Fuc11] proposed a delegatable credential system that is inspired by the construction of Belenkiy et al.

and supports non-interactive issuing and delegation of credentials. It is based on the commuting signatures and Groth-Sahai proofs and is at least twice as efficient as the scheme by Belenkiy et al. [BCC+09]. Our construction also requires a CRS, but still outperforms both schemes. For example, without attributes, the token size increases with every level by 4 group elements ($\mathbb{G}_1^2 \times \mathbb{G}_2^2$) for our scheme versus $\mathbb{G}_1^{50} \times \mathbb{G}_2^{40}$ for Belenkiy et al. [BCC+09] and $\mathbb{G}_1^{20} \times \mathbb{G}_2^{18}$ for Fuchsbauer [Fuc11]. Due to our optimization techniques, the number of expensive operations (exponentiations and pairings) is also minimized.

Finally, Chase et al. [CKLM13a, CKLM14] propose a DAC instantiation that is also non-interactive and scales linearly with the number of delegations. Their unforgeability definition is a bit different from the one by Belenkiy et al. [BCC+09] and implements the simulation extractability notion. However, none of the schemes accommodate attributes in their security definitions. As we mentioned above, it is hard to derive the exact efficiency figures from the "black-box"-type construction of [CKLM13a], which is built from malleable signatures, which, in turn, are built from the malleable proofs. The efficiency of their scheme depends on the concrete instantiation of malleable proofs: either Groth-Sahai proofs [CKLM12], which would be in the same spirit as [Fuc11], or non-interactive arguments of knowledge (SNARKs) and homomorphic encryption [CKLM13b], which, as the authors claim themselves, is less efficient.

Hierarchical group signatures, as introduced by Trolin and Wikström [TW05] and improved by Fuchsbauer and Pointcheval [FP09], are an extension of group signatures that allow for a tree of group managers. Users that received a credential from any of the managers can anonymously sign on behalf of the group, as is the case with delegatable credentials. However, in contrast to delegatable credentials, parties can serve either as manager or as user, but not both simultaneously. Additionally, hierarchical group signatures differ from delegatable credentials in the fact that signatures can be deanonymized by group managers.

4.2 Definition of Delegatable Credentials

We now define delegatable credentials in the form of an ideal functionality $\mathcal{F}_{\mathsf{dac}}$. For simplicity we consider the functionality with a single root delegator (issuer), but using multiple instances of $\mathcal{F}_{\mathsf{dac}}$ allows for

1. `Setup`. On input $(\mathsf{SETUP}, \mathsf{sid}, \langle n_i \rangle_i)$ from $\mathcal{I}$.
    - Verify that $\mathsf{sid} = (\mathcal{I}, \mathsf{sid}')$.
    - Output $(\mathsf{SETUP}, \mathsf{sid}, \langle n_i \rangle_i)$ to $\mathcal{A}$ and wait for response $(\mathsf{SETUP}, \mathsf{sid}, \mathsf{Present}, \mathsf{Ver}, \langle \mathbb{A}_i \rangle_i)$ from $\mathcal{A}$, where $\mathsf{Present}$ is a probablistic ITM Ver is a deterministic ITM, both interacting only with random oracle $\mathcal{G}_{\mathsf{sRO}}$.
    - Store algorithms $\mathsf{Present}$ and Ver and credential parameters $\langle \mathbb{A}_i \rangle_i, \langle n_i \rangle_i$, initialize $\mathcal{L}_{\mathsf{de}} \leftarrow \varnothing$; $\mathcal{L}_{\mathsf{at}} \leftarrow \varnothing$.
    - Output $(\mathsf{SETUPDONE}, \mathsf{sid})$ to $\mathcal{I}$.
2. `Delegate`. On input $(\mathsf{DELEGATE}, \mathsf{sid}, ssid, \vec{a_1}, \ldots, \vec{a_L}, \mathcal{P}_j)$ from some party $\mathcal{P}_i$, with $\vec{a_L} \in \mathbb{A}_L^{n_L}$.
    - If $L = 1$, check $\mathsf{sid} = (\mathcal{P}_i, \mathsf{sid}')$ and add an entry $\langle \mathcal{P}_j, \vec{a_1} \rangle$ to $\mathcal{L}_{\mathsf{de}}$.
    - If $L > 1$, check that an entry $\langle \mathcal{P}_i, \vec{a_1}, \ldots, a_{\overrightarrow{L-1}} \rangle$ exists in $\mathcal{L}_{\mathsf{de}}$.
    - Output $(\mathsf{ALLOWDEL}, \mathsf{sid}, ssid, \mathcal{P}_i, \mathcal{P}_j, L)$ to $\mathcal{A}$ and wait for input $(\mathsf{ALLOWDEL}, \mathsf{sid}, ssid)$ from $\mathcal{A}$.
    - Add an entry $\langle \mathcal{P}_j, \vec{a_1}, \ldots, \vec{a_L} \rangle$ to $\mathcal{L}_{\mathsf{de}}$.
    - Output $(\mathsf{DELEGATE}, \mathsf{sid}, ssid, \vec{a_1}, \ldots, \vec{a_L}, \mathcal{P}_i)$ to $\mathcal{P}_j$.
3. `Present`. On input $(\mathsf{PRESENT}, \mathsf{sid}, m, \vec{a}_1, \ldots, \vec{a}_L)$ from some party $\mathcal{P}_i$, with $\vec{a}_i \in (\mathbb{A}_i \cup \bot)^{n_i}$ for $i = 1, \ldots, L$.
    - Check that an entry $\langle \mathcal{P}_i, \vec{a}'_1, \ldots, \vec{a}'_L \rangle$ exists in $\mathcal{L}_{\mathsf{de}}$ such that $\vec{a_i} \preceq \vec{a_i}'$ for $i = 1, \ldots, L$.
    - Set $at \leftarrow \mathsf{Present}(m, \vec{a}_1, \ldots, \vec{a}_L)$ and abort if $\mathsf{Ver}(at, m, \vec{a}_1, \ldots, \vec{a}_L) = 0$.
    - Store $\langle m, \vec{a}_1, \ldots, \vec{a}_L \rangle$ in $\mathcal{L}_{\mathsf{at}}$.
    - Output $(\mathsf{TOKEN}, \mathsf{sid}, at)$ to $\mathcal{P}_i$.
4. `Verify`. On input $(\mathsf{VERIFY}, \mathsf{sid}, at, m, \vec{a}_1, \ldots, \vec{a}_L)$ from some party $\mathcal{P}_i$.
    - If there is no record $\langle m, \vec{a}_1, \ldots, \vec{a}_L \rangle$ in $\mathcal{L}_{\mathsf{at}}$, $\mathcal{I}$ is honest, and for $i = 1, \ldots, L$, there is no corrupt $\mathcal{P}_j$ such that $\langle \mathcal{P}_j, \vec{a}'_1, \ldots, \vec{a}'_i \rangle \in \mathcal{L}_{\mathsf{de}}$ with $\vec{a}_j \preceq \vec{a}'_j$ for $j = 1, \ldots, i$, set $f \leftarrow 0$.
    - Else, set $f \leftarrow \mathsf{Ver}(at, m, \vec{a}_1, \ldots, \vec{a}_L)$.
    - Output $(\mathsf{VERIFIED}, \mathsf{sid}, f)$ to $\mathcal{P}_i$.

Figure 4.1: Ideal functionality for delegatable credentials with attributes $\mathcal{F}_{\mathsf{dac}}$

many issuers. $\mathcal{F}_{\mathsf{dac}}$ allows for multiple levels delegation. A Level-1 credential is issued directly by the issuer. Any further delegations are done between users: the owner of a Level-$(L-1)$ credential can delegate it further, giving the receiver a Level L credential. $\mathcal{F}_{\mathsf{dac}}$ supports attributes on every level; attributes can be selectively disclosed during credential presentation. A presentation of a delegated credential creates a so-called *attribute token*, which can be verified with respect to the identity of the issuer, hiding the identity of the delegators.

$\mathcal{F}_{\mathsf{dac}}$ interacts with the issuer $\mathcal{I}$ and parties $\mathcal{P}_i$ who can delegate, present, and verify the credentials through the following four interfaces: $\mathsf{SETUP}, \mathsf{DELEGATE}, \mathsf{PRESENT}, \mathsf{VERIFY}$, that we describe here. The formal definition is presented in Fig. 4.1, where we use two conventions that ease the notation. First, the SETUP interface can only be called once, and all other interfaces ignore all input until a SETUP message has been completed. Second, whenever $\mathcal{F}_{\mathsf{dac}}$ performs a check, it means that if the check fails, it aborts by outputting $\perp$ to the caller.

Setup. The SETUP message is sent by the issuer $\mathcal{I}$, whose identity is fixed in the session identifier sid: $\mathcal{F}_{\mathsf{dac}}$ first checks that $\mathsf{sid} = (\mathcal{I}, \mathsf{sid}')$, which guarantees that each issuer can initialize its own instance of the functionality. The issuer defines the number of attributes for every delegation level i by specifying $\langle n_i \rangle_i$. This can be done efficiently by describing a function $f(i)$. We fix the number of attributes on the same delegation level since different number of attributes used by different delegators on the same level may leak information about the delegators. $\mathcal{I}$ does not need to specify the maximum number of the delegation levels.

$\mathcal{F}_{\mathsf{dac}}$ then asks the adversary for algorithms and credential parameters. The adversary provides algorithms $\mathsf{Present}, \mathsf{Ver}$ for presenting and verifying attribute tokens, respectively, and specifies the attribute spaces $\langle \mathbb{A}_i \rangle_i$ for different credential levels. To support random-oracle based realizations of $\mathcal{F}_{\mathsf{dac}}$, $\mathsf{Present}$ and Ver are allowed to interact with a global random oracle $\mathcal{G}_{\mathsf{sRO}}$. Observe that here we make sure of the fact that the random oracle is global, as two local functionalities cannot interact in the standard UC framework. $\mathcal{F}_{\mathsf{dac}}$ stores $\mathsf{Present}, \mathsf{Ver}, \langle \mathbb{A}_i \rangle_i$, $\langle n_i \rangle_i$ and initializes two empty sets: $\mathcal{L}_{\mathsf{de}}$ for delegation and $\mathcal{L}_{\mathsf{at}}$ for presentation bookkeeping.

Delegate. The $\mathsf{DELEGATE}$ message is sent by a user $\mathcal{P}_i$ with a Level-$(L-1)$ credential to delegate it to a user $\mathcal{P}_j$, giving $\mathcal{P}_j$ a Level-L credential. $\mathcal{P}_i$ specifies a list of attribute vectors for all the previous levels in the delegation chain $\vec{a}_1, \ldots, \vec{a}_{L-1}$ and the vector of attributes $\vec{a}_L$ to certify in a freshly delegated Level-L credential. All attribute vectors should satisfy the corresponding attribute space and length requirements. We use subsession identifiers in this interface since multiple delegation sessions might be interleaved due to the communication with the adversary. If this delegation gives $\mathcal{P}_j$ a Level-1 credential, then $\mathcal{F}_{\mathsf{dac}}$ verifies that party $\mathcal{P}_i$ is the issuer by checking the sid and adds an entry $\langle \mathcal{P}_j, \vec{a}_1 \rangle$ to $\mathcal{L}_{\mathsf{de}}$. If this is not the first level delegation $(L > 1)$, $\mathcal{F}_{\mathsf{dac}}$ checks if $\mathcal{P}_i$ indeed has a Level-$(L-1)$ credential with the specified attributes $\vec{a}_1, \ldots, \vec{a}_{L-1}$ by looking it up in $\mathcal{L}_{\mathsf{de}}$. $\mathcal{F}_{\mathsf{dac}}$ then asks the adversary if the delegation should proceed and, after receiving a response from $\mathcal{A}$, adds the corresponding delegation record to $\mathcal{L}_{\mathsf{de}}$ and sends the output that includes the full attribute chain to $\mathcal{P}_j$, notifying it of the successful delegation.

Note that in contrast to previous work on delegatable credentials, we model no privacy in delegation. That is, $\mathcal{P}_i$ and $\mathcal{P}_j$ will learn the identity of eachother during delegation. While this is a weaker privacy definition than previous definitions, we think privacy for delegation is not neccesary. in real-world scenarios, the delegator and delegatee will typically know eachother when a credential with attributes is delegated.

Present. The $\mathsf{PRESENT}$ message is sent by a user $\mathcal{P}_i$ to create an attribute token. A token selectively reveals attributes from the delegated credential and also signs a message m, which can be an arbitrary string. $\mathcal{P}_i$ inputs attribute vectors by specifying only the values of the disclosed attributes and using special symbol $\perp$ to indicate the hidden attributes. $\mathcal{F}_{\mathsf{dac}}$ checks if a delegation entry exists in $\mathcal{L}_{\mathsf{de}}$ such that the corresponding disclosed attributes were indeed delegated to $\mathcal{P}_i$. For this, it uses the following relation for attribute vectors: We say that for two vectors $\vec{a} = (a_1, \ldots, a_n)$; $\vec{b} = (b_1, \ldots, b_n)$: $\vec{a} \preceq \vec{b}$ if $a_i = b_i$ or $a_i = \perp$ for $i = 1, \ldots, n$.

If this is the case it runs the $\mathsf{Present}$ algorithm to generate the attribute token. The $\mathsf{Present}$ algorithm does not take the identity of the user and the non-disclosed attributes as input - the attribute token is computed independently of these values. This ensures the user's privacy and hiding the non-disclosed attributes on all levels of the delegated credential chain. Next, it checks that the computed attribute token is valid using the Ver algorithm, which ensures completeness. It outputs the token value to user $\mathcal{P}_i$.

Verify. The VERIFY message is sent by a user $\mathcal{P}_i$ to verify an attribute token. Message m and the disclosed attribute values are also provided as input for verification. $\mathcal{F}_{\mathsf{dac}}$ performs the unforgeability check: if the message together with the corresponding disclosed attribute values were not signed by calling the PRESENT interface (there is no corresponding bookkeeping record), the issuer is honest, and on any delegation level there is no corrupted party with the matching attributes, then $\mathcal{F}_{\mathsf{dac}}$ outputs a negative verification result; otherwise, $\mathcal{F}_{\mathsf{dac}}$ runs the verification algorithm and outputs the result to $\mathcal{P}_i$.

Our ideal functionality $\mathcal{F}_{\mathsf{dac}}$ can be easily extended to also accept as input and output commitments to attribute values, following the recent work by Camenisch et al. [CDR16], which would allow extending our delegatable credential scheme with existing revocation schemes for anonymous credentials in a hybrid protocol.

4.3 Building Blocks

This section introduces the building blocks used in our delegatable credential scheme. We recall Groth's structure preserving signature scheme and define a new primitive we call a *sibling signature scheme*, that allows for two different signing algorithms sharing a single key pair.

4.3.1 Signature Schemes

A digital signature scheme SIG is a set of PPT algorithms $\mathsf{SIG} = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$:

$\mathsf{SIG.Setup}(1^\kappa) \xrightarrow{\$} sp$: The setup algorithm takes as input a security parameter and outputs public system parameters that also specify a message space $\mathcal{M}$.

$\mathsf{SIG.Gen}(sp) \xrightarrow{\$} (sk, pk)$: The key generation algorithm takes as input system parameters and outputs a public key pk and a corresponding secret key sk.

$\mathsf{SIG.Sign}(sk, m) \xrightarrow{\$} \sigma$: The signing algorithm takes as input a private key sk and a message $m \in \mathcal{M}$ and outputs a signature σ.

$\mathsf{SIG.Verify}(pk, m, \sigma) \rightarrow 1/0$: The verification algorithm takes as input a public key pk, a message m and a signature σ and outputs 1 for acceptance or 0 for rejection according to the input.

Structure-Preserving Signature scheme by Groth

We recall the structure-preserving signature scheme by Groth [Gro15], which we refer to as Groth. Note that the original scheme supports signing blocks of messages in a form of "matrix", whereas we provide a simplified description for "vectors" of messages only, since we use this version later in the chapter. Let a message be a vector of group elements of length n: $\vec{m} = (m_1, \ldots, m_n)$. Groth can sign messages in either $\mathbb{G}_1$ or $\mathbb{G}_2$, by choosing a public key in $\mathbb{G}_2$ or $\mathbb{G}_1$, respectively. Let $\mathsf{Groth}_{\mathbb{G}_1}$ be the Groth signature scheme signing messages in $\mathbb{G}_1$ with a public key in $\mathbb{G}_2$, and $\mathsf{Groth}_{\mathbb{G}_2}$ signs messages in $\mathbb{G}_2$ with a public key in $\mathbb{G}_1$. We describe the $\mathsf{Groth}_{\mathbb{G}_2}$ scheme below. $\mathsf{Groth}_{\mathbb{G}_1}$ follows immediately.

$\mathsf{Groth}_{\mathbb{G}_2}$.Setup: Let $\Lambda^* = (q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, e)$ and $y_i \xleftarrow{\$} \mathbb{G}_2$ for $i = 1, \ldots, n$. Output parameters $sp = (\Lambda^*, \{y_i\}_{i=1,\ldots,n})$.

$\mathsf{Groth}_{\mathbb{G}_2}$.Gen($sp$): Choose random $v \xleftarrow{\$} \mathbb{Z}_q$ and set $V \xleftarrow{\$} g_1^v$. Output public key $pk = V$ and secret key $sk = v$.

$\mathsf{Groth}_{\mathbb{G}_2}$.Sign($sk; \vec{m}$): To sign message $\vec{m} \in \mathbb{G}_2^n$ choose a random $r \xleftarrow{\$} \mathbb{Z}_q^*$ and set

$$R \leftarrow g_1^r \ , \quad S \leftarrow (y_1 \cdot g_2^v)^{\frac{1}{r}} \ , \quad \text{and} \quad T_i \leftarrow (y_i^v \cdot m_i)^{\frac{1}{r}} \ .$$

Output signature $\sigma = (R, S, T_1, \ldots, T_n)$.

$\mathsf{Groth}_{\mathbb{G}_2}$.Verify($pk, \sigma, \vec{m}$) On input public key $pk = V \in \mathbb{G}_1$, message $\vec{m} \in \mathbb{G}_2^n$ and signature $\sigma = (R, S, T_1, \ldots, T_n) \in \mathbb{G}_1 \times \mathbb{G}_2^{n+1}$, output 1 iff

$$e(R, S) = e(g_1, y_1)e(V, g_2) \ \wedge \ \bigwedge_{i=1}^{n} e(R, T_i) = e(V, y_i)e(g_1, m_i) \ .$$

$\mathsf{Groth}_{\mathbb{G}_2}.\overset{\$}{\leftarrow}(\sigma)$ To randomize signature $\sigma = (R, S, T_1, \ldots, T_n)$, pick $r' \overset{\$}{\leftarrow} \mathbb{Z}_q$ and set

$$R' \leftarrow R^{r'} \quad , \quad S' \leftarrow S^{\frac{1}{r'}} \quad , \quad \text{and} \quad T_i' \leftarrow T_i^{\frac{1}{r'}} \quad .$$

Output randomized signature $\sigma' = (R', S', T_1', \ldots, T_n')$.

4.3.2 Sibling Signatures

We introduce a new type of signatures that we call *sibling signatures.* It allows a signer with one key pair to use two different signing algorithms, each with a dedicated verification algorithm. In our generic construction, this will allow a user to hold a single key pair that it can use for both presentation and delegation of a credential.

A sibling signature scheme consists of algorithms Setup, Gen, Sign_1, Sign_2, Verify_1, Verify_2.

$\mathsf{Sib.Setup}(1^\kappa) \overset{\$}{\rightarrow} sp$: The setup algorithm takes as input a security parameter and outputs public system parameters that also specify two message spaces $\mathcal{M}_1$ and $\mathcal{M}_2$.

$\mathsf{Sib.Gen}(sp) \overset{\$}{\rightarrow} (sk, pk)$: The key generation algorithm takes as input system parameters and outputs a public key pk and a corresponding secret key sk.

$\mathsf{Sib.Sign}_1(sk, m) \overset{\$}{\rightarrow} \sigma$: The signing algorithm takes as input a private key sk and a message $m \in \mathcal{M}_1$ and outputs a signature σ.

$\mathsf{Sib.Sign}_2(sk, m) \overset{\$}{\rightarrow} \sigma$: The signing algorithm takes as input a private key sk and a message $m \in \mathcal{M}_2$ and outputs a signature σ.

$\mathsf{Sib.Verify}_1(pk, m, \sigma) \rightarrow 1/0$: The verification algorithm takes as input a public key pk, a message m and a signature σ and outputs 1 for acceptance or 0 for rejection according to the input.

$\mathsf{Sib.Verify}_2(pk, m, \sigma) \rightarrow 1/0$: The verification algorithm takes as input a public key pk, a message m and a signature σ and outputs 1 for acceptance or 0 for rejection according to the input.

We require sibling signatures to be *complete* and *unforgeable.*

Definition 12 (Completeness). *A sibling signature scheme is complete if for $b \in \{0,1\}$ and for all $m \in \mathcal{M}_b$ we have*

$$\Pr\big[\mathsf{Sib.Verify_b}(pk, m, \sigma) = 1 | sp \xleftarrow{\$} \mathsf{Sib.Setup}(1^\kappa),\\ (sk, pk) \xleftarrow{\$} \mathsf{Sib.Gen}(sp), \sigma \xleftarrow{\$} \mathsf{Sib.Sign_b}(sk, m)\big] = 1 \ .$$

Definition 13 (Unforgeability). *No adversary with oracle access to* $\mathsf{Sign_1}$ *and* $\mathsf{Sign_2}$ *can create a signature that correctly verifies with* $\mathsf{Verify_b}$*, if no* $\mathsf{Sign_b}$ *query was made for message m. For every such $b \in \{1,2\}$ we call it unforgeability-b. More precisely, a sibling signature scheme is unforgeable-b if the probability*

$$\Pr\big[\mathsf{Sib.Verify_b}(pk, m, \sigma) = 1 \wedge m \notin Q_{\mathsf{Sign_b}} |\\ sp \xleftarrow{\$} \mathsf{Sib.Setup}(1^\kappa), (sk, pk) \xleftarrow{\$} \mathsf{Sib.Gen}(sp),\\ (\sigma, m) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{Sib.Sign_1}(sk,\cdot)}, \mathcal{O}^{\mathsf{Sib.Sign_2}(sk,\cdot)}}(sp, pk)\big]$$

is negligible in κ for every PPT adversary $\mathcal{A}$ and $b \in \{1,2\}$, where oracle $\mathcal{O}^{\mathsf{Sib.Sign_b}(sk,\cdot)}$ on input m stores m in $Q_{\mathsf{Sign_b}}$ and returns $\mathsf{Sib.Sign_b}(sk, m)$. A sibling signature scheme is unforgeable if it is both unforgeable-1 and unforgeable-2.

Constructing Sibling Signatures

Note that one can trivially construct a sibling signature scheme from two standard signature schemes by setting the public key pk as (pk_1, pk_2) and the signing key as $sk = (sk_1, sk_2)$, and simply using one signature scheme as $\mathsf{Sign_1}$ and $\mathsf{Verify_1}$ and the other as $\mathsf{Sign_2}$ and $\mathsf{Verify_2}$. However, this generalization also allows for instantiations that securely share key material between the two algorithms.

We now show that one can combine $\mathsf{Groth}_{\mathbb{G}_1}$ signatures with Schnorr-signatures to form a sibling signature scheme we call $\mathsf{SibGS1}$. $\mathsf{SibGS1}$ uses only a single key pair. It uses the Setup and Gen algorithms of $\mathsf{Groth}_{\mathbb{G}_1}$. Algorithm $\mathsf{Sign_1}$ is instantiated with $\mathsf{Groth}_{\mathbb{G}_1}.\mathsf{Sign}$, and $\mathsf{Sign_2}$ creates a Schnorr signature. Let $\mathsf{SibGS2}$ denote the analogously defined Groth-Schnorr sibling signature where we use $\mathsf{Groth}_{\mathbb{G}_2}$ instead of $\mathsf{Groth}_{\mathbb{G}_1}$.

Lemma 1. SibGSb *is a secure sibling signature scheme in the random oracle and generic group model.*

Proof. Completeness of SibGSb directly follows from the completeness of Grothb and Schnorr signatures. We can reduce the unforgeability-1 and unforgeability-2 of SibGSb to the unforgeability of Grothb, which is proven to be unforgeable in the generic group model. The reduction algorithm $\mathcal{B}$ receives the Grothb public key pk from the challenger and has access to signing oracle $\mathcal{O}^{\mathsf{Grothb.Sign}(sk,\cdot)}$ that creates signatures valid under pk. $\mathcal{B}$ simulates the random oracle honestly and must answer $\mathcal{A}$'s signing queries by simulating oracles $\mathcal{O}^{\mathsf{Sib.Sign}_1(sk,\cdot)}$ and $\mathcal{O}^{\mathsf{Sib.Sign}_2(sk,\cdot)}$. When $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sib.Sign}_1(sk,\cdot)}$ on m, $\mathcal{B}$ queries $\sigma \leftarrow \mathcal{O}^{\mathsf{Grothb.Sign}(sk,\mathtt{m})}$ and returns σ. When $\mathcal{A}$ queries $\mathcal{O}^{\mathsf{Sib.Sign}_2(sk,\cdot)}$ on m, $\mathcal{B}$ simulates a Schnorr signature without knowledge of sk by programming the random oracle.

Finally, $\mathcal{A}$ outputs a forgery. Let us first consider the unforgeability-1 game, meaning that $\mathcal{A}$ outputs forgery σ^* on message m^*, such that σ^* is a valid Grothb signature on m^* and $\mathcal{O}^{\mathsf{Sib.Sign}_1(sk,\cdot)}$ was not queried on m^*. This means that $\mathcal{B}$ did not query $\mathcal{O}^{\mathsf{Grothb.Sign}(sk,\cdot)}$ on m^*, so $\mathcal{B}$ can break the unforgeability of Grothb by submitting forgery (σ^*, m^*).

Next, consider the unforgeability-2 game. Forgery σ^* is a Schnorr signature on m^* and $\mathcal{A}$ did not query $\mathcal{O}^{\mathsf{Sib.Sign}_1(sk,\cdot)}$ on m^*. This means that the Schnorr signature is not a simulated signature and we use the forking lemma [BN06b] to extract sk. Now, $\mathcal{B}$ picks a new message $\hat{m}^*$ for which it did not query $\mathcal{O}^{\mathsf{Grothb.Sign}(sk,\cdot)}$, and uses sk to create signature $\hat{\sigma}^*$ on $\hat{m}^*$. It submits $(\hat{\sigma}^*, \hat{m}^*)$ as its forgery to win the Grothb unforgeability game. ☐

4.4 A Generic Construction for Delegatable Credentials

In this section, we provide a generic construction for delegatable anonymous credentials with attributes. We first explain the intuition behind our construction, then present a construction based on sibling signatures defined in Section 4.3.2 and non-interactive zero-knowledge proofs. Then we prove that our generic construction securely realizes $\mathcal{F}_{\mathsf{dac}}$. We provide an efficient instantiation of our generic construction in the next section.

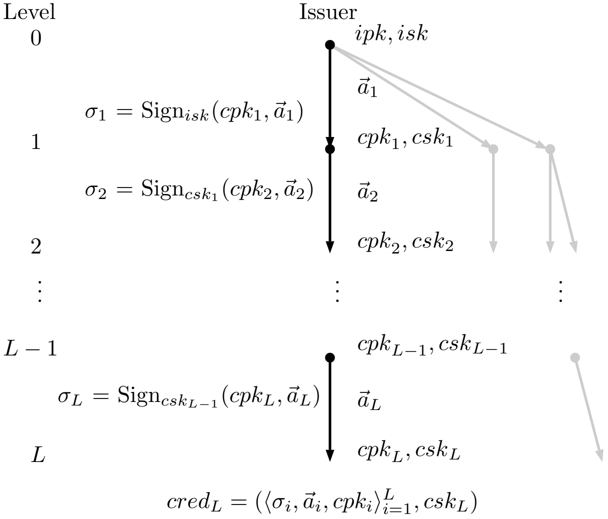


Figure 4.2: Our Generic Construction: Delegation

4.4.1 Construction Overview

Recall that our definition of delegatable credentials allows for multiple levels of delegation. There is a root delegator (also called issuer) that issues Level-1 credentials to users. Users can delegate their Level-L credential, resulting in a Level-$(L + 1)$ credential. We now explain on a high level how a user obtains a Level-1 credential and then that credential is delegated. It is then easy to see how a Level-L credential is delegated (this is also depicted in Fig. 4.2).

The issuer first generates a signing key isk and corresponding verification key ipk and publishes ipk, after which it can issue a Level-1 credential to a user. The user, to get Level-1 credential issued, generates a fresh secret and a public key (csk_1, cpk_1) for this credential and sends public key cpk_1 to the root delegator. The root delegator signs this public key together with a set of attributes $\vec{a}_1$ and sends the signature σ_1 back to the user. A Level-1 credential $cred_1$ consists of the signature σ_1, attributes $\vec{a}_1$, and credential keys (cpk_1, csk_1).

The user can delegate $cred_1$ further to another user by issuing a Level-2 credential. The receiver generates a fresh key pair (csk_2, cpk_2)

for the Level-2 credential. The delegation is done by signing public key cpk_2 and a set of attributes $\vec{a}_2$ (chosen by the delegator) with the Level-1 credential secret key csk_1. The resulting signature σ_2 is sent back together with the attributes $\vec{a}_2$ and the original signature σ_1, and the corresponding attributes $\vec{a}_1$. The Level-2 credential consists of both signatures σ_1, σ_2, attributes $\vec{a}_1, \vec{a}_2$, and keys cpk_1, cpk_2, csk_2. Note that the Level-2 credential is a chain of two so-called *credential links*. The first link, consisting of $(\sigma_1, \vec{a}_1, cpk_1)$ proves that the delegator has a Level-1 credential containing attributes $\vec{a}_1$. The second link, $(\sigma_2, \vec{a}_2, cpk_2)$, proves that this delegator issued attributes $\vec{a}_2$ to the owner of cpk_2. The key csk_2 allows the user to prove he is the owner of this Level-2 credential. Note, that the Level-1 credential secret key csk_1 is not sent together with the signature and the credential link, so that it is impossible for a user who owns the Level-2 credential to present or delegate the Level-1 credential.

The Level-2 credential can be delegated further in the analogous way by generating a signature on attributes and a public key and sending them together with lower-level credential links. A Level-L credential is therefore a chain of the L credential links, where every link adds a number of attributes $\vec{a}_i$, and a secret key csk_L that allows the owner to present the credential or to delegate it further.

A credential of any level can be presented by its owner by generating a NIZK proof proving a possession of all credential links back to the issuer and selectively disclosing attributes from the corresponding signatures. This proof, that we call an *attribute token*, can be verified with the public key of only the issuer. The public keys of all the credential links remain hidden in the zero-knowledge proof and, therefore, the identities of all the intermediate delegators are not revealed by the attribute token.

4.4.2 Generic Construction

Our generic construction Π_{dac} is based on sibling signature schemes, where Sign_1 signs vectors of messages, combined with non-interactive zero-knowledge proofs. We allow different sibling signature schemes to be used at different delegation levels. Let Sib_i denote the scheme used by the owners of Level-i credentials. As we sign public keys of another signature scheme and the attribute values, the different signature schemes must be compatible with each other: The public key space of Sib_{i+1} must be included in the message space $\mathcal{M}_1$ of Sib_i. It follows that the attribute space $\mathbb{A}_i$ is the message space of Sib_{i-1}. In addition, we require an IND-CPA public-key encryption scheme PKE compatible with Sib_0, i.e., it can encrypt the issuer secret key.

The required system parameters for the signature schemes and the non-interactive zero-knowledge proof system are taken from $\mathcal{F}_{\mathsf{crs}}$. In addition, $\mathcal{F}_{\mathsf{crs}}$ contains a public key epk for PKE such that nobody knows the corresponding secret key. We implicitly assume that every protocol participant queries $\mathcal{F}_{\mathsf{crs}}$ to retrieve the system parameters and that the system parameters are passed as an implicit input to every algorithm of the signature schemes. Moreover, every party must query $\mathcal{F}_{\mathsf{ca}}$ to retrieve the issuer public key and check its validity by verifying π_{isk}. Our generic construction allows the building blocks to be proven secure with respect to *local* random oracles, as this is sufficient to prove our overall construction to be secure with respect to strict global random oracles.

Setup. In the setup phase, the issuer $\mathcal{I}$ creates his key pair and registers this with the CA functionality $\mathcal{F}_{\mathsf{ca}}$.

1. $\mathcal{I}$, upon receiving input $(\mathsf{SETUP}, \mathsf{sid}, \langle n_i \rangle_i)$:
   - Check that $\mathsf{sid} = \mathcal{I}, \mathsf{sid}'$ for some sid'.
   - Run $(ipk, isk) \leftarrow \mathsf{Sib}_0.\mathsf{Gen}(1^\kappa)$, encrypt isk to the crs public key by computing $c_{isk} \leftarrow \mathsf{PKE.Enc}(epk, isk)$, and compute proof

     $$\pi_{isk} \leftarrow \mathsf{NIZK}\{isk : (ipk, isk) \in \mathsf{Sib}_0.\mathsf{Gen}(1^\kappa) \wedge c_{isk} \in \mathsf{PKE.Enc}(epk, isk)\}.$$

     Register public key $(ipk, c_{isk}, \pi_{isk})$ with $\mathcal{F}_{\mathsf{ca}}$. Let $cpk_0 \leftarrow ipk$.
   - Output $(\mathsf{SETUPDONE}, \mathsf{sid})$.

Delegate. Any user $\mathcal{P}_i$ with a Level-$L-1$ credential can delegate this credential to another user $\mathcal{P}_j$, giving $\mathcal{P}_j$ a Level-L credential. Delegator $\mathcal{P}_i$ can choose the attributes he adds in this delegation. Note that only the issuer $\mathcal{I}$ can issue a Level-1 credential, so we distinguish two cases: issuance (delegation of a Level-1 credential) and delegation of credential of level $L > 1$.

2. $\mathcal{P}_i$ on input $(\mathsf{DELEGATE}, \mathsf{sid}, ssid, \vec{a}_1, \ldots, \vec{a}_L, \mathcal{P}_j)$ with $\vec{a_L} \in \mathbb{A}_L^{n_L}$:
   - If $L = 1$, $\mathcal{P}_i$ only proceeds if he is the issuer $\mathcal{I}$ with $\mathsf{sid} = (\mathcal{I}, \mathsf{sid}')$. If $L > 1$, $\mathcal{P}_i$ checks that he possesses a credential chain that

signs $\vec{a}_1, \ldots, \vec{a}_{L-1}$. That is, he looks up $\mathit{cred} = (\langle \sigma_i, \vec{a}_i, \mathit{cpk}_i \rangle_{i=1}^{L-1}, \mathit{csk}_{L-1})$ in $\mathcal{L}_{\mathsf{cred}}$.
- Send $(\mathsf{sid}, \mathit{ssid}, \vec{a_1}, \ldots, \vec{a_L})$ to $\mathcal{P}_j$ over $\mathcal{F}_{\mathsf{smt}}$.
- $\mathcal{P}_j$, upon receiving $(\mathsf{sid}, \mathit{ssid}, \vec{a}_1, \ldots, \vec{a}_L)$ from $\mathcal{P}_i$ over $\mathcal{F}_{\mathsf{smt}}$, generate a fresh credential specific key pair $(\mathit{cpk}_L, \mathit{csk}_L) \leftarrow \mathsf{Sib}_L.\mathsf{Gen}(1^\kappa)$.
- Send cpk_L to $\mathcal{P}_i$ over $\mathcal{F}_{\mathsf{smt}}$.
- $\mathcal{P}_i$, upon receiving cpk_L from $\mathcal{P}_j$ over secure channel $\mathcal{F}_{\mathsf{smt}}$, computes $\sigma_L \leftarrow \mathsf{Sib}_{L-1}.\mathsf{Sign}_1(\mathit{csk}_{L-1}; \mathit{cpk}_L, \vec{a}_L)$ and sends $\langle \sigma_i, \mathit{cpk}_i \rangle_{i=1}^{L}$ to $\mathcal{P}_j$ over $\mathcal{F}_{\mathsf{smt}}$.
- $\mathcal{P}_j$, upon receiving $\langle \sigma_i, \mathit{cpk}_i \rangle_{i=1}^{L}$ from $\mathcal{P}_i$ over $\mathcal{F}_{\mathsf{smt}}$, verifies the credential by checking $\mathsf{Sib}_{i-1}.\mathsf{Verify}_1(\mathit{cpk}_{i-1}, \sigma_i, \mathit{cpk}_i, \vec{a}_i)$ for $i = 1, \ldots, L$. It stores $\mathit{cred} \leftarrow (\langle \sigma_i, \vec{a}_i, \mathit{cpk}_i \rangle_{i=1}^{L}, \mathit{csk}_L)$ in $\mathcal{L}_{\mathsf{cred}}$. Output $(\mathsf{DELEGATE}, \mathsf{sid}, \mathit{ssid}, \vec{a}_1, \ldots, \vec{a}_L, \mathcal{P}_i)$.

Present. A user can present a credential she owns, while also signing a message m. The disclosed attributes are described by $\vec{a}_1, \ldots, \vec{a}_L$. Let $\vec{a}_i = a_{i,1}, \ldots, a_{i,n} \in (\mathbb{A} \cup \bot)^n$. If $a_{i,j} \in \mathbb{A}$, the user shows it possesses this attribute. If $a_{i,j} = \bot$, the user does not show the attribute. Let D be the set of indices of disclosed attributes, i.e., the set of pairs (i, j) where $a_{i,j} \neq \bot$.

3. $\mathcal{P}_i$, upon receiving input $(\mathsf{PRESENT}, \mathsf{sid}, m, \vec{a}_1, \ldots, \vec{a}_L)$ with $\vec{a}_i \in (\mathbb{A}_i \cup \bot)^{n_i}$ for $i = 1, \ldots, L$:
   - Look up a credential $\mathit{cred} = (\langle \sigma_i, \vec{a}'_i, \mathit{cpk}_i \rangle_{i=1}^{L}, \mathit{csk}_L)$ in $\mathcal{L}_{\mathsf{cred}}$, such that $\vec{a_i} \preceq \vec{a_i}'$ for $i = 1, \ldots, L$. Abort if no such credential was found.
   - Create an attribute token by proving knowledge of the credential:

$$
\begin{aligned}
\mathit{at} \leftarrow \mathsf{NIZK}\big\{&(\sigma_1, \ldots, \sigma_L, \mathit{cpk}_1, \ldots, \mathit{cpk}_L, \langle a'_{i,j} \rangle_{i \notin D}, \mathit{tag}) : \\
&\bigwedge_{i=1}^{L} 1 = \mathsf{Sib}_{i-1}.\mathsf{Verify}_1(\mathit{cpk}_{i-1}, \sigma_i, \mathit{cpk}_i, a'_{i,1}, \ldots, a'_{i,n_i}) \\
&\qquad\qquad\qquad \wedge 1 = \mathsf{Sib}.\mathsf{Verify}_2(\mathit{cpk}_L, \mathit{tag}, m)\big\}
\end{aligned}
$$

   - Output $(\mathsf{TOKEN}, \mathsf{sid}, \mathit{at})$.

Verify. A user can verify an attribute token by verifying the zero knowledge proof.

4. $\mathcal{P}_i$, upon receiving input $(\mathsf{VERIFY}, \mathsf{sid}, \mathit{at}, m, \vec{a}_1, \ldots, \vec{a}_L)$:

- Verify the zero-knowledge proof at with respect to m and $\vec{a}_1, \ldots, \vec{a}_L$. Set $f \leftarrow 1$ if valid and $f \leftarrow 0$ otherwise.
- Output $(\mathsf{VERIFIED}, \mathsf{sid}, f)$.

Random Oracles.

This generic construction uses multiple building blocks that may use one or multiple random oracles. If the building blocks require no random oracles, our generic construction does not require any random oracles. If they do, then our generic construction will use a single global random oracle $\mathcal{G}_{\mathsf{sRO}}$. Let Sib_i and NIZK use local random oracles $RO_1, \ldots, RO_j$, mapping to sets $S_1, \ldots, S_j$ respectively. To work with $\mathcal{G}_{\mathsf{sRO}}$, our generic construction assumes the existence of efficiently computable probabilistic algorithms $\mathsf{Embed}_1, \ldots, \mathsf{Embed}_j$ and $\mathsf{Embed}_1^{-1}, \ldots, \mathsf{Embed}_j^{-1}$, such that for $h \overset{\$}{\leftarrow} \{0,1\}^{\ell(\kappa)}$, $\mathsf{Embed}(h)$ is computationally indistinguishable from uniform in $\mathbb{G}$, and for all $x \in \mathbb{G}$, $\mathsf{Embed}(\mathsf{Embed}^{-1}(x)) = x$ and for $x \overset{\$}{\leftarrow} \mathbb{G}$, $\mathsf{Embed}^{-1}(x)$ is computationally indistinguishable from uniform in $\{0,1\}^{\ell(\kappa)}$. Whenever one of the building blocks would query RO_i on m, expecting an element uniform in S_i, it instead queries $\mathcal{G}_{\mathsf{sRO}}$ on (i, m) and uses Embed_i to map the result to S_i. Note that we also apply domain separation by prepending i to the query, which allows us to use a single random oracle. The Embed^{-1} algorithms are only used in the security proof: If the simulator programmed $RO_i(m)$ to $x \in S_i$ in the security proof of one of the building blocks, then in a reduction proving equivalence between the real and ideal world of our DAA scheme, we can program $\mathcal{G}_{\mathsf{sRO}}$ on (i, m) to $\mathsf{Embed}_i^{-1}(x)$ to achieve the same programming.

4.4.3 Security of Π_{dac}

We now prove the security of our generic construction.

Theorem 8. *Our delegatable credentials protocol* Π_{dac} *GUC-realizes* $\mathcal{F}_{\mathsf{dac}}$ *(as defined in Section 5.3), in the* $(\mathcal{F}_{\mathsf{smt}}, \mathcal{F}_{\mathsf{ca}}, \mathcal{F}_{\mathsf{crs}}, \mathcal{G}_{\mathsf{sRO}})$*-hybrid model, provided that*

- Sib_i *is a secure sibling signature scheme (as defined in Section 4.3.2),*
- NIZK *is a simulation-sound zero-knowledge proof of knowledge,*
- Sib_i *and* NIZK *use local random oracles* $RO_1, \ldots, RO_j$*, mapping to* $S_1, \ldots, S_j$ *respectively, and efficiently computable probabilistic algo-*

rithms $\mathsf{Embed}_1, \ldots, \mathsf{Embed}_j$ *and* $\mathsf{Embed}_1^{-1}, \ldots, \mathsf{Embed}_j^{-1}$ *exist, such that*

- *for* $h \xleftarrow{\$} \{0,1\}^{\ell(\kappa)}$, $\mathsf{Embed}(h)$ *is computationally indistinguishable from uniform in* $\mathbb{G}$,
- *for all* $x \in \mathbb{G}$, $\mathsf{Embed}(\mathsf{Embed}^{-1}(x)) = x$ *and*
- *for* $x \xleftarrow{\$} \mathbb{G}$, $\mathsf{Embed}^{-1}(x)$ *is computationally indistinguishable from uniform in* $\{0,1\}^{\ell(\kappa)}$.

Proof. By Theorem 1, it is sufficient to show that Π_{dac} $\mathcal{G}_{\mathsf{sRO}}$-EUC-emulates $\mathcal{F}_{\mathsf{dac}}$ in the $\mathcal{F}_{\mathsf{smt}}, \mathcal{F}_{\mathsf{ca}}, \mathcal{F}_{\mathsf{crs}}$-hybrid model, meaning that we have to show that there exists a simulator $\mathcal{S}$ as a function of $\mathcal{A}$ such that no $\mathcal{G}_{\mathsf{sRO}}$-externally constrained environment can distinguish Π_{dac} and $\mathcal{A}$ from $\mathcal{F}_{\mathsf{dac}}$ and $\mathcal{S}$. We prove this using a sequence of games, starting with the real world protocol execution. In the next game we construct one entity $\mathcal{C}$ that runs the real world protocol for all honest parties. Then we split $\mathcal{C}$ into two pieces, a functionality $\mathcal{F}$ and a simulator $\mathcal{S}$, where $\mathcal{F}$ receives all inputs from honest parties and sends the outputs to honest parties. We start with a dummy functionality, and gradually change $\mathcal{F}$ and update $\mathcal{S}$ accordingly, to end up with the full $\mathcal{F}_{\mathsf{dac}}$ and a satisfying simulator. First, we show how we can reduce to the security of the building blocks (which were proven w.r.t. a local random oracle), and then we start the sequence of games.

The domain separation of $\mathcal{G}_{\mathsf{sRO}}$ and the availability of the Embed and Embed^{-1} algorithms allow us to reduce to the properties of the building blocks. If we want to reduce to, e.g., the unforgeability of Sib_i, which uses random oracle RO_i, then as shown in Figure 3.3, the reduction simulates $\mathcal{G}_{\mathsf{sRO}}$. It plays the unforgeability game of Sib_i against a challenger who also controls random oracle RO_i. We need to make sure that the global random oracle on points (i, m) agrees with RO_i. More precisely, we simulate $\mathcal{G}_{\mathsf{sRO}}$ on (i, m) by querying RO_i to obtain h. We then let $\mathcal{G}_{\mathsf{sRO}}$ return $\mathsf{Embed}_i^{-1}(h)$. Observe that for points $(i' \neq i, m')$, we can still freely choose $\mathcal{G}_{\mathsf{sRO}}$'s output, showing that in this setting, we can for example simulate NIZK proofs, as this is based on a different local random oracle, which we are simulating in $\mathcal{G}_{\mathsf{sRO}}$.

Game 1: This is the real world.

1. `Setup.` On input $(\mathsf{SETUP}, \mathsf{sid}, \langle n_i \rangle_i)$ from $\mathcal{I}$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SETUP}, \mathsf{sid}, \langle n_i \rangle_i), \mathcal{I})$ to $\mathcal{A}$.

2. `Delegate.` On input $(\mathsf{DELEGATE}, \mathsf{sid}, ssid, \vec{a_1}, \ldots, \vec{a_L}, \mathcal{P}_j)$ from some party $\mathcal{P}_i$, with $\vec{a_L} \in \mathbb{A}_L^{n_L}$.
   - Output $(\mathsf{FORWARD}, (\mathsf{DELEGATE}, \mathsf{sid}, ssid, \vec{a_1}, \ldots, \vec{a_L}, \mathcal{P}_j), \mathcal{P}_i)$ to $\mathcal{A}$.

3. `Present.` On input $(\mathsf{PRESENT}, \mathsf{sid}, m, \vec{a}_1, \ldots, \vec{a}_L)$ from some party $\mathcal{P}_i$, with $\vec{a}_i \in (\mathbb{A}_i \cup \perp)^{n_i}$ for $i = 1, \ldots, L$.
   - Output $(\mathsf{FORWARD}, (\mathsf{PRESENT}, \mathsf{sid}, m, \vec{a}_1, \ldots, \vec{a}_L), \mathcal{P}_i)$ to $\mathcal{A}$.

4. `Verify.` On input $(\mathsf{VERIFY}, \mathsf{sid}, at, m, \vec{a}_1, \ldots, \vec{a}_L)$ from some party $\mathcal{P}_i$.
   - Output $(\mathsf{FORWARD}, (\mathsf{VERIFY}, \mathsf{sid}, at, m, \vec{a}_1, \ldots, \vec{a}_L), \mathcal{P}_i)$ to $\mathcal{A}$.

Figure 4.3: Ideal functionality for Game 3 in the proof of Theorem 8

When a simulated party "$\mathcal{P}$" outputs m and no specific action is defined, send $(\mathsf{OUTPUT}, m, \mathcal{P})$ to $\mathcal{F}$.

Simulating $\mathcal{F}_{\mathsf{crs}}$

- $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{crs}}$ honestly, except that it chooses *epk* such that it knows the corresponding decryption key *esk*.

Forwarded Input

- On input $(\mathsf{FORWARD}, m, \mathcal{P})$.
  - Give "$\mathcal{P}$" input m.

Figure 4.4: Simulator for GAME 3 in the proof of Theorem 8

Game 2: We let the simulator $\mathcal{S}$ receive all inputs and generate all outputs by simulating the honest parties honestly. It also simulates the hybrid functionalities honestly, except that it simulates $\mathcal{F}_{\mathsf{crs}}$ in a way that it knows the decryption key *esk* corresponding to *epk*. Clearly, this is equal to the real world.

Game 3: We now start creating a functionality $\mathcal{F}$ that receives inputs from honest parties and generates the outputs for honest parties. It works together with a simulator $\mathcal{S}$. In this game, we simply let $\mathcal{F}$ forward all inputs to $\mathcal{S}$, who acts as before. When $\mathcal{S}$ would generate an output, it first forwards it to $\mathcal{F}$, who then outputs it. This game hop simply restructures GAME 2, we have GAME 3 = GAME 2.

Game 4: $\mathcal{F}$ now handles the setup queries, and lets $\mathcal{S}$ enter algorithms that $\mathcal{F}$ will store. Observe that $\mathcal{S}$ will always know *isk*: If $\mathcal{I}$ is honest, $\mathcal{S}$ simulates the issuer, and if $\mathcal{I}$ is corrupt, it can extract *isk* by using *esk* to decrypt *isk* from c_{isk}. $\mathcal{S}$ defines $\mathsf{Present}$ to create a fresh credential key pair, issue a level L credential to this key pair, and create an attribute token as in the real-world protocol. It defines Ver to be equal to the real-world protocol.

$\mathcal{F}$ checks the structure of sid, and aborts if it does not have the expected structure. This does not change the view of $\mathcal{E}$, as $\mathcal{I}$ in the protocol performs the same check, giving GAME 4 = GAME 3.

Game 5: $\mathcal{F}$ now handles the verification queries using the algorithm that $\mathcal{S}$ defined in GAME 4. In GAME 4, $\mathcal{S}$ defined the Ver algorithm as the real world verification algorithm so we have GAME 5 = GAME 4.

1. `Setup.` On input $(\mathsf{SETUP}, \mathsf{sid}, \langle n_i \rangle_i)$ from $\mathcal{I}$.
   - Verify that $\mathsf{sid} = (\mathcal{I}, \mathsf{sid}')$.
   - Output $(\mathsf{SETUP}, \mathsf{sid}, \langle n_i \rangle_i)$ to $\mathcal{A}$ and wait for response $(\mathsf{SETUP}, \mathsf{sid}, \mathsf{Present}, \mathsf{Ver}, \langle \mathbb{A}_i \rangle_i)$ from $\mathcal{A}$, where $\mathsf{Present}$ is a probablistic ITM Ver is a deterministic ITM, both interacting only with random oracle $\mathcal{G}_{\mathsf{sRO}}$.
   - Store algorithms $\mathsf{Present}$ and Ver and credential parameters $\langle \mathbb{A}_i \rangle_i, \langle n_i \rangle_i$, initialize $\mathcal{L}_{\mathsf{de}} \leftarrow \varnothing$; $\mathcal{L}_{\mathsf{at}} \leftarrow \varnothing$.
   - Output $(\mathsf{SETUPDONE}, \mathsf{sid})$ to $\mathcal{I}$.
2. `Delegate.` On input $(\mathsf{DELEGATE}, \mathsf{sid}, ssid, \vec{a_1}, \ldots, \vec{a_L}, \mathcal{P}_j)$ from some party $\mathcal{P}_i$, with $\vec{a_L} \in \mathbb{A}_L^{n_L}$.
   - Output $(\mathsf{FORWARD}, (\mathsf{DELEGATE}, \mathsf{sid}, ssid, \vec{a_1}, \ldots, \vec{a_L}, \mathcal{P}_j), \mathcal{P}_i)$ to $\mathcal{A}$.
3. `Present.` On input $(\mathsf{PRESENT}, \mathsf{sid}, m, \vec{a}_1, \ldots, \vec{a}_L)$ from some party $\mathcal{P}_i$, with $\vec{a}_i \in (\mathbb{A}_i \cup \bot)^{n_i}$ for $i = 1, \ldots, L$.
   - Output $(\mathsf{FORWARD}, (\mathsf{PRESENT}, \mathsf{sid}, m, \vec{a}_1, \ldots, \vec{a}_L), \mathcal{P}_i)$ to $\mathcal{A}$.
4. `Verify.` On input $(\mathsf{VERIFY}, \mathsf{sid}, at, m, \vec{a}_1, \ldots, \vec{a}_L)$ from some party $\mathcal{P}_i$.
   - Output $(\mathsf{FORWARD}, (\mathsf{VERIFY}, \mathsf{sid}, at, m, \vec{a}_1, \ldots, \vec{a}_L), \mathcal{P}_i)$ to $\mathcal{A}$.

Figure 4.5: Ideal functionality for Game 4 in the proof of Theorem 8

When a simulated party "$\mathcal{P}$" outputs m and no specific action is defined, send $(\mathsf{OUTPUT}, m, \mathcal{P})$ to $\mathcal{F}$.

Simulating $\mathcal{F}_{\mathsf{crs}}$

- $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{crs}}$ honestly, except that it chooses *epk* such that it knows the corresponding decryption key *esk*.

Setup

Honest $\mathcal{I}$

- On input $(\mathsf{SETUP}, \mathsf{sid}, \langle n_i \rangle_i)$ from $\mathcal{F}_{\mathsf{dac}}$.
  - Parse sid as $(\mathcal{I}, \mathsf{sid}')$ and give "$\mathcal{I}$" input $(\mathsf{SETUP}, \mathsf{sid}, \langle n_i \rangle_i)$.
  - When "$\mathcal{I}$" outputs $(\mathsf{SETUPDONE}, \mathsf{sid})$, $\mathcal{S}$ takes its public key *ipk* and secret key *isk* and defines $\mathsf{Present}$ and Ver, and the attribute spaces $\langle \mathbb{A}_i \rangle_i$.
    - ∗ Define $\mathsf{Present}(m, \vec{a}_1, \ldots, \vec{a}_L)$ as follows: Run $(cpk_i, csk_i) \leftarrow \mathsf{SIG}_i.\mathsf{Gen}(1^\kappa)$ for $i = 1, \ldots, L$. Compute $\sigma_1 \leftarrow \mathsf{SIG}_0.\mathsf{Sign}(isk; cpk_1, \vec{a}_1)$ and $\sigma_i \leftarrow \mathsf{SIG}_{i-1}.\mathsf{Sign}(csk_{i-1}, cpk_i, \vec{a}_i)$ for $i = 2, \ldots, L$. Next, compute *at* as in the real world protocol and return *at*.
    - ∗ Define $\mathsf{Ver}(at, m, \vec{a}_1, \ldots, \vec{a}_L)$ as the real world verification algorithm that verifies with respect to *ipk*.
    - ∗ Define $\mathbb{A}_i$ as $\mathbb{G}_1$ for odd i and as $\mathbb{G}_2$ for even i.

    $\mathcal{S}$ sends $(\mathsf{SETUP}, \mathsf{sid}, \mathsf{Present}, \mathsf{Ver}, \langle \mathbb{A}_i \rangle_i)$ to $\mathcal{F}_{\mathsf{dac}}$.

Corrupt $\mathcal{I}$

- $\mathcal{S}$ notices this setup as it notices $\mathcal{I}$ registering a public key with "$\mathcal{F}_{\mathsf{ca}}$" with $\mathsf{sid} = (\mathcal{I}, \mathsf{sid}')$.
  - If the registered key is of the form $(ipk, c_{isk}, \pi_{isk})$ and π_{isk} is valid, $\mathcal{S}$ obtains the issuer secret key $isk \leftarrow \mathsf{PKE.Dec}(esk, c_{isk})$.
  - $\mathcal{S}$ defines $\mathsf{Present}, \mathsf{Ver}$ and $\langle \mathbb{A}_i \rangle$ as when $\mathcal{I}$ is honest, but now depending on the extracted key.
  - $\mathcal{S}$ sends $(\mathsf{SETUP}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{dac}}$ on behalf of $\mathcal{I}$.
- On input $(\mathsf{SETUP}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{dac}}$.
  - $\mathcal{S}$ sends $(\mathsf{SETUP}, \mathsf{sid}, \mathsf{Present}, \mathsf{Ver}, \langle \mathbb{A}_i \rangle_i)$ to $\mathcal{F}_{\mathsf{dac}}$.
- On input $(\mathsf{SETUPDONE}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{dac}}$
  - $\mathcal{S}$ continues simulating "$\mathcal{I}$".

Forwarded Input

- On input $(\mathsf{FORWARD}, m, \mathcal{P})$.
  - Give "$\mathcal{P}$" input m.

Figure 4.6: Simulator for GAME 4 in the proof of Theorem 8

1. `Setup.` On input $(\mathsf{SETUP}, \mathsf{sid}, \langle n_i \rangle_i)$ from $\mathcal{I}$.
   - Verify that $\mathsf{sid} = (\mathcal{I}, \mathsf{sid}')$.
   - Output $(\mathsf{SETUP}, \mathsf{sid}, \langle n_i \rangle_i)$ to $\mathcal{A}$ and wait for response $(\mathsf{SETUP}, \mathsf{sid}, \mathsf{Present}, \mathsf{Ver}, \langle \mathbb{A}_i \rangle_i)$ from $\mathcal{A}$, where $\mathsf{Present}$ is a probablistic ITM Ver is a deterministic ITM, both interacting only with random oracle $\mathcal{G}_{\mathsf{sRO}}$.
   - Store algorithms $\mathsf{Present}$ and Ver and credential parameters $\langle \mathbb{A}_i \rangle_i, \langle n_i \rangle_i$, initialize $\mathcal{L}_{\mathsf{de}} \leftarrow \varnothing$; $\mathcal{L}_{\mathsf{at}} \leftarrow \varnothing$.
   - Output $(\mathsf{SETUPDONE}, \mathsf{sid})$ to $\mathcal{I}$.
2. `Delegate.` On input $(\mathsf{DELEGATE}, \mathsf{sid}, ssid, \vec{a_1}, \ldots, \vec{a_L}, \mathcal{P}_j)$ from some party $\mathcal{P}_i$, with $\vec{a_L} \in \mathbb{A}_L^{n_L}$.
   - Output $(\mathsf{FORWARD}, (\mathsf{DELEGATE}, \mathsf{sid}, ssid, \vec{a_1}, \ldots, \vec{a_L}, \mathcal{P}_j), \mathcal{P}_i)$ to $\mathcal{A}$.
3. `Present.` On input $(\mathsf{PRESENT}, \mathsf{sid}, m, \vec{a}_1, \ldots, \vec{a}_L)$ from some party $\mathcal{P}_i$, with $\vec{a}_i \in (\mathbb{A}_i \cup \perp)^{n_i}$ for $i = 1, \ldots, L$.
   - Output $(\mathsf{FORWARD}, (\mathsf{PRESENT}, \mathsf{sid}, m, \vec{a}_1, \ldots, \vec{a}_L), \mathcal{P}_i)$ to $\mathcal{A}$.
4. `Verify.` On input $(\mathsf{VERIFY}, \mathsf{sid}, at, m, \vec{a}_1, \ldots, \vec{a}_L)$ from some party $\mathcal{P}_i$.
   - Set $f \leftarrow \mathsf{Ver}(at, m, \vec{a}_1, \ldots, \vec{a}_L)$.
   - Output $(\mathsf{VERIFIED}, \mathsf{sid}, f)$ to $\mathcal{P}_i$.

Figure 4.7: Ideal functionality for Game 5 in the proof of Theorem 8

When a simulated party “$\mathcal{P}$” outputs m and no specific action is defined, send $(\mathsf{OUTPUT}, m, \mathcal{P})$ to $\mathcal{F}$.

Simulating $\mathcal{F}_{\mathsf{crs}}$

- $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{crs}}$ honestly, except that it chooses *epk* such that it knows the corresponding decryption key *esk*.

Setup

unchanged.

Verify

Nothing to simulate.

Forwarded Input

- On input $(\mathsf{FORWARD}, m, \mathcal{P})$.
  - Give “$\mathcal{P}$” input m.

Figure 4.8: Simulator for GAME 5 in the proof of Theorem 8

Game 6: $\mathcal{F}$ now also handles the delegation queries. If both the delegator and the delegatee are honest, $\mathcal{S}$ does not learn the attribute values and must simulate the real world protocol with dummy values. As all communication is over a secure channel, this difference is not noticable by the adversary.

If the delegatee is corrupt, $\mathcal{S}$ learns the attribute values $\mathcal{S}$ can simulate the real world protocol with the correct input. If the delegator is corrupt and the delegatee honest, $\mathcal{S}$ has to take more care: The corrupt delegator may have received delegated credentials from other corrupt users, without $\mathcal{S}$ and $\mathcal{F}$ knowing. If $\mathcal{S}$ would make a delegation query with $\mathcal{F}$ on the delegator's behalf, $\mathcal{F}$ would reject as it does not possess the required attributes for this delegation, invalidating the simulation. In this case, $\mathcal{S}$ first informs $\mathcal{F}$ of the missing delegations, such that $\mathcal{F}$'s records accept the delegation, and only then calls $\mathcal{F}$ on the delegator's behalf for this delegation.

As $\mathcal{S}$ only lacks information to simulate when both parties are honest, but this change is not noticable due to the use of a secure channel, GAME 6 $\approx$ GAME 5.

1. `Setup.` On input $(\mathsf{SETUP}, \mathsf{sid}, \langle n_i \rangle_i)$ from $\mathcal{I}$.
   - Verify that $\mathsf{sid} = (\mathcal{I}, \mathsf{sid}')$.
   - Output $(\mathsf{SETUP}, \mathsf{sid}, \langle n_i \rangle_i)$ to $\mathcal{A}$ and wait for response $(\mathsf{SETUP}, \mathsf{sid}, \mathsf{Present}, \mathsf{Ver}, \langle \mathbb{A}_i \rangle_i)$ from $\mathcal{A}$, where $\mathsf{Present}$ is a problistic ITM Ver is a deterministic ITM, both interacting only with random oracle $\mathcal{G}_{\mathsf{sRO}}$.
   - Store algorithms $\mathsf{Present}$ and Ver and credential parameters $\langle \mathbb{A}_i \rangle_i, \langle n_i \rangle_i$, initialize $\mathcal{L}_{\mathsf{de}} \leftarrow \varnothing$; $\mathcal{L}_{\mathsf{at}} \leftarrow \varnothing$.
   - Output $(\mathsf{SETUPDONE}, \mathsf{sid})$ to $\mathcal{I}$.
2. `Delegate.` On input $(\mathsf{DELEGATE}, \mathsf{sid}, ssid, \vec{a_1}, \ldots, \vec{a_L}, \mathcal{P}_j)$ from some party $\mathcal{P}_i$, with $\vec{a_L} \in \mathbb{A}_L^{n_L}$.
   - If $L = 1$, check $\mathsf{sid} = (\mathcal{P}_i, \mathsf{sid}')$ and add an entry $\langle \mathcal{P}_j, \vec{a_1} \rangle$ to $\mathcal{L}_{\mathsf{de}}$.
   - If $L > 1$, check that an entry $\langle \mathcal{P}_i, \vec{a_1}, \ldots, \vec{a_{L-1}} \rangle$ exists in $\mathcal{L}_{\mathsf{de}}$.
   - Output $(\mathsf{ALLOWDEL}, \mathsf{sid}, ssid, \mathcal{P}_i, \mathcal{P}_j, L)$ to $\mathcal{A}$ and wait for input $(\mathsf{ALLOWDEL}, \mathsf{sid}, ssid)$ from $\mathcal{A}$.
   - Add an entry $\langle \mathcal{P}_j, \vec{a_1}, \ldots, \vec{a_L} \rangle$ to $\mathcal{L}_{\mathsf{de}}$.
   - Output $(\mathsf{DELEGATE}, \mathsf{sid}, ssid, \vec{a_1}, \ldots, \vec{a_L}, \mathcal{P}_i)$ to $\mathcal{P}_j$.
3. `Present.` On input $(\mathsf{PRESENT}, \mathsf{sid}, m, \vec{a}_1, \ldots, \vec{a}_L)$ from some party $\mathcal{P}_i$, with $\vec{a}_i \in (\mathbb{A}_i \cup \perp)^{n_i}$ for $i = 1, \ldots, L$.
   - Output $(\mathsf{FORWARD}, (\mathsf{PRESENT}, \mathsf{sid}, m, \vec{a}_1, \ldots, \vec{a}_L), \mathcal{P}_i)$ to $\mathcal{A}$.
4. `Verify.` On input $(\mathsf{VERIFY}, \mathsf{sid}, at, m, \vec{a}_1, \ldots, \vec{a}_L)$ from some party $\mathcal{P}_i$.
   - Set $f \leftarrow \mathsf{Ver}(at, m, \vec{a}_1, \ldots, \vec{a}_L)$.
   - Output $(\mathsf{VERIFIED}, \mathsf{sid}, f)$ to $\mathcal{P}_i$.

Figure 4.9: Ideal functionality for Game 6 in the proof of Theorem 8

When a simulated party "$\mathcal{P}$" outputs m and no specific action is defined, send $(\mathsf{OUTPUT}, m, \mathcal{P})$ to $\mathcal{F}$.

Simulating $\mathcal{F}_{\mathsf{crs}}$

- $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{crs}}$ honestly, except that it chooses *epk* such that it knows the corresponding decryption key *esk*.

Setup

unchanged.

Delegate

Honest $\mathcal{P}$, $\mathcal{P}'$

- $\mathcal{S}$ notices this delegation as it receives $(\mathsf{ALLOWDEL}, \mathsf{sid}, \mathit{ssid}, \mathcal{P}, \mathcal{P}', L)$ from $\mathcal{F}_{\mathsf{dac}}$.
  - $\mathcal{S}$ picks dummy attribute values $\vec{a_1}, \dots, \vec{a_L}$ and gives "$\mathcal{P}$" input $(\mathsf{DELEGATE}, \mathsf{sid}, \mathit{ssid}, \vec{a_1}, \dots, \vec{a_L}, \mathcal{P}')$.
  - When "$\mathcal{P}'$" outputs $(\mathsf{DELEGATE}, \mathsf{sid}, \mathit{ssid}, \vec{a_1}, \dots, \vec{a_L}, \mathcal{P})$, let $\mathcal{F}_{\mathsf{dac}}$ proceed by outputting $(\mathsf{ALLOWDEL}, \mathsf{sid}, \mathit{ssid})$ to $\mathcal{F}_{\mathsf{dac}}$.

Honest $\mathcal{P}$, corrupt $\mathcal{P}'$

- $\mathcal{S}$ notices this delegation as it receives $(\mathsf{ALLOWDEL}, \mathsf{sid}, \mathit{ssid}, \mathcal{P}, \mathcal{P}', L)$ from $\mathcal{F}_{\mathsf{dac}}$.
  - Output $(\mathsf{ALLOWDEL}, \mathsf{sid}, \mathit{ssid})$ to $\mathcal{F}_{\mathsf{dac}}$.
- $\mathcal{S}$ receives $(\mathsf{DELEGATE}, \mathsf{sid}, \mathit{ssid}, \vec{a_1}, \dots, \vec{a_L}, \mathcal{P})$ as $\mathcal{P}'$ is corrupt.
  - $\mathcal{S}$ gives "$\mathcal{P}$" input $(\mathsf{DELEGATE}, \mathsf{sid}, \mathit{ssid}, \vec{a_1}, \dots, \vec{a_L}, \mathcal{P}')$.

Honest $\mathcal{P}'$, corrupt $\mathcal{P}$

- $\mathcal{S}$ notices this delegation as "$\mathcal{P}$" outputs $(\mathsf{DELEGATE}, \mathsf{sid}, \mathit{ssid}, \vec{a_1}, \dots, \vec{a_L}, \mathcal{P})$.
  - If $L > 1$ and $\mathcal{S}$ has not simulated delegating attributes $\vec{a_1}, \dots, \vec{a_{L-1}}$ to $\mathcal{P}$, and there is a corrupt party $\mathcal{P}''$ that has attributes $\vec{a_1}, \dots, \vec{a_i}$ for $0 < i < L-1$ (note that if the root delegator $\mathcal{I}$ is corrupt, $i = 0$), $\mathcal{P}''$ may have delegated $\vec{a}_i, \dots, \vec{a}_{L-1}$ to $\mathcal{P}'$ without $\mathcal{S}$ noticing. Therefore, $\mathcal{S}$ needs to delegate attributes $\vec{a}_i, \dots, \vec{a}_{L-1}$ in the ideal world, which is possible as $\mathcal{P}''$ is corrupt: $\mathcal{S}$ sends $(\mathsf{DELEGATE}, \mathsf{sid}, \mathit{ssid}, \vec{a_1}, \dots, \vec{a_i}, \mathcal{P}')$ on $\mathcal{P}''$'s behalf to $\mathcal{F}_{\mathsf{dac}}$ and allows the delegation, and for $j = i+1, \dots, L-1$, sends $(\mathsf{DELEGATE}, \mathsf{sid}, \mathit{ssid}, \vec{a_1}, \dots, \vec{a_j}, \mathcal{P}')$ on $\mathcal{P}'$'s behalf to $\mathcal{F}_{\mathsf{dac}}$, allowing every delegation. Note that $\mathcal{P}'$ now possesses attributes $\vec{a_1}, \dots, \vec{a_{L-1}}$ in $\mathcal{F}_{\mathsf{dac}}$'s records.
  - Send $(\mathsf{DELEGATE}, \mathsf{sid}, \mathit{ssid}, \vec{a_1}, \dots, \vec{a_L}, \mathcal{P}')$ on $\mathcal{P}$'s behalf to $\mathcal{F}_{\mathsf{dac}}$.
- On input $(\mathsf{ALLOWDEL}, \mathsf{sid}, \mathit{ssid}, \mathcal{P}, \mathcal{P}', L)$ from $\mathcal{F}_{\mathsf{dac}}$.
  - Output $(\mathsf{ALLOWDEL}, \mathsf{sid}, \mathit{ssid})$ to $\mathcal{F}_{\mathsf{dac}}$.

Corrupt $\mathcal{P}$, $\mathcal{P}$'

Nothing to simulate.

Verify

Nothing to simulate.

Forwarded Input

- On input $(\mathsf{FORWARD}, m, \mathcal{P})$.
  - Give "$\mathcal{P}$" input m.

Figure 4.10: Simulator for GAME 6 in the proof of Theorem 8

1. `Setup.` On input $(\mathsf{SETUP}, \mathsf{sid}, \langle n_i \rangle_i)$ from $\mathcal{I}$.
    - Verify that $\mathsf{sid} = (\mathcal{I}, \mathsf{sid}')$.
    - Output $(\mathsf{SETUP}, \mathsf{sid}, \langle n_i \rangle_i)$ to $\mathcal{A}$ and wait for response $(\mathsf{SETUP}, \mathsf{sid}, \mathsf{Present}, \mathsf{Ver}, \langle \mathbb{A}_i \rangle_i)$ from $\mathcal{A}$, where $\mathsf{Present}$ is a probablistic ITM Ver is a deterministic ITM, both interacting only with random oracle $\mathcal{G}_{\mathsf{sRO}}$.
    - Store algorithms $\mathsf{Present}$ and Ver and credential parameters $\langle \mathbb{A}_i \rangle_i, \langle n_i \rangle_i$, initialize $\mathcal{L}_{\mathsf{de}} \leftarrow \varnothing$; $\mathcal{L}_{\mathsf{at}} \leftarrow \varnothing$.
    - Output $(\mathsf{SETUPDONE}, \mathsf{sid})$ to $\mathcal{I}$.
2. `Delegate.` On input $(\mathsf{DELEGATE}, \mathsf{sid}, \mathit{ssid}, \vec{a_1}, \ldots, \vec{a_L}, \mathcal{P}_j)$ from some party $\mathcal{P}_i$, with $\vec{a_L} \in \mathbb{A}_L^{n_L}$.
    - If $L = 1$, check $\mathsf{sid} = (\mathcal{P}_i, \mathsf{sid}')$ and add an entry $\langle \mathcal{P}_j, \vec{a_1} \rangle$ to $\mathcal{L}_{\mathsf{de}}$.
    - If $L > 1$, check that an entry $\langle \mathcal{P}_i, \vec{a_1}, \ldots, a_{\overrightarrow{L-1}} \rangle$ exists in $\mathcal{L}_{\mathsf{de}}$.
    - Output $(\mathsf{ALLOWDEL}, \mathsf{sid}, \mathit{ssid}, \mathcal{P}_i, \mathcal{P}_j, L)$ to $\mathcal{A}$ and wait for input $(\mathsf{ALLOWDEL}, \mathsf{sid}, \mathit{ssid})$ from $\mathcal{A}$.
    - Add an entry $\langle \mathcal{P}_j, \vec{a_1}, \ldots, \vec{a_L} \rangle$ to $\mathcal{L}_{\mathsf{de}}$.
    - Output $(\mathsf{DELEGATE}, \mathsf{sid}, \mathit{ssid}, \vec{a_1}, \ldots, \vec{a_L}, \mathcal{P}_i)$ to $\mathcal{P}_j$.
3. `Present.` On input $(\mathsf{PRESENT}, \mathsf{sid}, m, \vec{a}_1, \ldots, \vec{a}_L)$ from some party $\mathcal{P}_i$, with $\vec{a}_i \in (\mathbb{A}_i \cup \bot)^{n_i}$ for $i = 1, \ldots, L$.
    - Check that an entry $\langle \mathcal{P}_i, \vec{a}'_1, \ldots, \vec{a}'_L \rangle$ exists in $\mathcal{L}_{\mathsf{de}}$ such that $\vec{a_i} \preceq \vec{a_i}'$ for $i = 1, \ldots, L$.
    - Set $\mathit{at} \leftarrow \mathsf{Present}(m, \vec{a}_1, \ldots, \vec{a}_L)$ and abort if $\mathsf{Ver}(\mathit{at}, m, \vec{a}_1, \ldots, \vec{a}_L) = 0$.
    - Store $\langle m, \vec{a}_1, \ldots, \vec{a}_L \rangle$ in $\mathcal{L}_{\mathsf{at}}$.
    - Output $(\mathsf{TOKEN}, \mathsf{sid}, \mathit{at})$ to $\mathcal{P}_i$.
4. `Verify.` On input $(\mathsf{VERIFY}, \mathsf{sid}, \mathit{at}, m, \vec{a}_1, \ldots, \vec{a}_L)$ from some party $\mathcal{P}_i$.
    - Set $f \leftarrow \mathsf{Ver}(\mathit{at}, m, \vec{a}_1, \ldots, \vec{a}_L)$.
    - Output $(\mathsf{VERIFIED}, \mathsf{sid}, f)$ to $\mathcal{P}_i$.

Figure 4.11: Ideal functionality for GAME 7 in the proof of Theorem 8

Simulating $\mathcal{F}_{\mathsf{crs}}$

- $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{crs}}$ honestly, except that it chooses *epk* such that it knows the corresponding decryption key *esk*.

Setup

unchanged.

Delegate

unchanged.

Present

Nothing to simulate.

Verify

Nothing to simulate.

Figure 4.12: Simulator for GAME 7 in the proof of Theorem 8

Game 7: $\mathcal{F}$ now generates the attribute tokens for honest parties, using the Present algorithm that $\mathcal{S}$ defined in GAME 4. First, $\mathcal{F}$ checks whether the party is eligible to create such an attribute token, and aborts otherwise. This does not change $\mathcal{E}$'s view, as the real world protocol performs an equivalent check. Second, $\mathcal{F}$ tests whether attribute token *at* generated with Present is valid w.r.t. Ver before outputting *at*. $\mathcal{S}$ defined Present to sign a valid witness for the NIZK that *at* is, and Ver will verify the NIZK. By completeness of all the sibling signature schemes Sib and completeness of NIZK, *at* will be accepted by Ver. This shows that $\mathcal{F}$ outputs an attribute token if and only if the real world party would output an attribute token.

Next, we must show that the generated attribute token is indistinguishable between the real and ideal world. Both the real world protocol and the Present algorithm compute

$$\begin{aligned} at \leftarrow \mathsf{NIZK}\big\{&(\sigma_1, \dots, \sigma_L, cpk_1, \dots, cpk_L, \langle a'_{i,j} \rangle_{i \notin D}, tag) : \\ &\bigwedge_{i=1}^{L} 1 = \mathsf{Sib}_{i-1}.\mathsf{Verify}_1(cpk_{i-1}, \sigma_i, cpk_i, a'_{i,1}, \dots, a'_{i,n_i}) \wedge \\ &\qquad\qquad\qquad\qquad 1 = \mathsf{Sib}.\mathsf{Verify}_2(cpk_L, tag, m)\big\} \end{aligned}$$

but in the real world, a party uses his own credential every time he proves this statement, and $\mathcal{F}$ creates a fresh credential for every signature. Note that the credential only concerns the witness of the zero-knowledge proof. By the witness indistinguishability of the zero-knowledge proofs, this change is not noticable and we have GAME 7 $\approx$ GAME 6.

1. `Setup.` On input $(\mathsf{SETUP}, \mathsf{sid}, \langle n_i \rangle_i)$ from $\mathcal{I}$.
   - Verify that $\mathsf{sid} = (\mathcal{I}, \mathsf{sid}')$.
   - Output $(\mathsf{SETUP}, \mathsf{sid}, \langle n_i \rangle_i)$ to $\mathcal{A}$ and wait for response $(\mathsf{SETUP}, \mathsf{sid}, \mathsf{Present}, \mathsf{Ver}, \langle \mathbb{A}_i \rangle_i)$ from $\mathcal{A}$, where $\mathsf{Present}$ is a probablistic ITM Ver is a deterministic ITM, both interacting only with random oracle $\mathcal{G}_{\mathsf{sRO}}$.
   - Store algorithms $\mathsf{Present}$ and Ver and credential parameters $\langle \mathbb{A}_i \rangle_i, \langle n_i \rangle_i$, initialize $\mathcal{L}_{\mathsf{de}} \leftarrow \varnothing$; $\mathcal{L}_{\mathsf{at}} \leftarrow \varnothing$.
   - Output $(\mathsf{SETUPDONE}, \mathsf{sid})$ to $\mathcal{I}$.
2. `Delegate.` On input $(\mathsf{DELEGATE}, \mathsf{sid}, \mathit{ssid}, \vec{a_1}, \dots, \vec{a_L}, \mathcal{P}_j)$ from some party $\mathcal{P}_i$, with $\vec{a_L} \in \mathbb{A}_L^{n_L}$.
   - If $L = 1$, check $\mathsf{sid} = (\mathcal{P}_i, \mathsf{sid}')$ and add an entry $\langle \mathcal{P}_j, \vec{a_1} \rangle$ to $\mathcal{L}_{\mathsf{de}}$.
   - If $L > 1$, check that an entry $\langle \mathcal{P}_i, \vec{a_1}, \dots, \vec{a_{L-1}} \rangle$ exists in $\mathcal{L}_{\mathsf{de}}$.
   - Output $(\mathsf{ALLOWDEL}, \mathsf{sid}, \mathit{ssid}, \mathcal{P}_i, \mathcal{P}_j, L)$ to $\mathcal{A}$ and wait for input $(\mathsf{ALLOWDEL}, \mathsf{sid}, \mathit{ssid})$ from $\mathcal{A}$.
   - Add an entry $\langle \mathcal{P}_j, \vec{a_1}, \dots, \vec{a_L} \rangle$ to $\mathcal{L}_{\mathsf{de}}$.
   - Output $(\mathsf{DELEGATE}, \mathsf{sid}, \mathit{ssid}, \vec{a_1}, \dots, \vec{a_L}, \mathcal{P}_i)$ to $\mathcal{P}_j$.
3. `Present.` On input $(\mathsf{PRESENT}, \mathsf{sid}, m, \vec{a}_1, \dots, \vec{a}_L)$ from some party $\mathcal{P}_i$, with $\vec{a}_i \in (\mathbb{A}_i \cup \perp)^{n_i}$ for $i = 1, \dots, L$.
   - Check that an entry $\langle \mathcal{P}_i, \vec{a}'_1, \dots, \vec{a}'_L \rangle$ exists in $\mathcal{L}_{\mathsf{de}}$ such that $\vec{a_i} \preceq \vec{a_i}'$ for $i = 1, \dots, L$.
   - Set $at \leftarrow \mathsf{Present}(m, \vec{a}_1, \dots, \vec{a}_L)$ and abort if $\mathsf{Ver}(at, m, \vec{a}_1, \dots, \vec{a}_L) = 0$.
   - Store $\langle m, \vec{a}_1, \dots, \vec{a}_L \rangle$ in $\mathcal{L}_{\mathsf{at}}$.
   - Output $(\mathsf{TOKEN}, \mathsf{sid}, at)$ to $\mathcal{P}_i$.
4. `Verify.` On input $(\mathsf{VERIFY}, \mathsf{sid}, at, m, \vec{a}_1, \dots, \vec{a}_L)$ from some party $\mathcal{P}_i$.
   - If there is no record $\langle m, \vec{a}_1, \dots, \vec{a}_L \rangle$ in $\mathcal{L}_{\mathsf{at}}$, $\mathcal{I}$ is honest, and for $i = 1, \dots, L$, there is no corrupt $\mathcal{P}_j$ such that $\langle \mathcal{P}_j, \vec{a}'_1, \dots, \vec{a}'_i \rangle \in \mathcal{L}_{\mathsf{de}}$ with $\vec{a}_j \preceq \vec{a}'_j$ for $j = 1, \dots, i$, set $f \leftarrow 0$.
   - Else, set $f \leftarrow \mathsf{Ver}(at, m, \vec{a}_1, \dots, \vec{a}_L)$.
   - Output $(\mathsf{VERIFIED}, \mathsf{sid}, f)$ to $\mathcal{P}_i$.

Figure 4.13: Ideal functionality for GAME 8 in the proof of Theorem 8

Simulating $\mathcal{F}_{\mathsf{crs}}$
- $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{crs}}$ honestly, except that it chooses *epk* such that it knows the corresponding decryption key *esk*.

Setup
unchanged.

Delegate
unchanged.

Present
Nothing to simulate.

Verify
Nothing to simulate.

Figure 4.14: Simulator for GAME 8 in the proof of Theorem 8

Game 8: $\mathcal{F}$ now guarantees unforgeability of attribute tokens. We make this change gradually, where in the first intermediate game we guarantee unforgeability of level 1 attribute tokens, then of level 2, and so forth, and we prove that each game is indistinguishable from the previous.

If the unforgeability check for level L credentials triggers with non-negligible probability, there must be an attribute token *at* that was valid before but is rejected by the unforgeability check of $\mathcal{F}$. This means that one of the two statements must hold with non-negligible probability:

- *at* proves knowledge either of a public key cpk_L that belongs to an honest user with the correct attributes, but this user never signed m (as otherwise the unforgeability check would not trigger)
- *at* proves knowledge of a public key cpk_L that does not belong to an honest user.

We will reduce both cases to the unforgeability of Sib. Recall that we only assume Sib to be unforgeable with respect to a local random oracle, i.e., the security proof may use the observability and programmability of the random oracle to simulate signing queries, which our simulator does not have. However, since this is a security reduction, everything falls under control of the reduction (as depicted in Figure 3.3), including the global random oracle. This means that the reduction can observe and program the random oracle, and we can reduce to the security of Sib in this setting.

In the first case, we can reduce to the unforgeability-2 property of Sib: There can only be polynomially many delegations of a level L credential to an honest user. Pick a random one and simulate the receiving party with the public key pk as received from the unforgeability game of Sib. When the user delegates this credential, use the Sign_1 oracle, and when presenting the credential, use the Sign_2 oracle. Finally, when $\mathcal{F}$ sees an attribute token at that it considers a forgery, the soundness of NIZK allows us to extract from the zero-knowledge proof. With non-negligible probability, $cpk_L = pk$, and then tag is a Sib forgery.

In the second case, we can reduce to the unforgeability-1 property of Sib: If $L = 1$, simulate the issuer with $ipk \leftarrow pk$, where pk is taken from the Sib unforgeability game. As isk us not known to the simulator, we simulate π_{isk}, and define the Present algorithm to simulate the proof such that the issuer secret key is not needed. $\mathcal{I}$ uses the Sign_1 oracle to delegate. If a delegation was chosen, simulate the receiver using $cpk_i \leftarrow pk$. If $L > 1$, there can only be polynomially many delegations that give an honest user a credential of level $L - 1$. Pick a random one and simulate the receiving party with $cpk_{L-1} \leftarrow pk$. Use the Sign_1 oracle to delegate this credential, and the Sign_2 oracle to present this credential. Finally, when $\mathcal{F}$ sees an attribute token at that it considers a forgery, extract from the zero-knowledge proof. With non-negligible probability, $cpk_{L-1} = pk$, and then σ_L is a Sib forgery on message cpk_L.

$\mathcal{F}$ of GAME 8 of equal to $\mathcal{F}_{\mathsf{dac}}$, concluding our sequence of games.

□

4.5 A Concrete Instantiation using Pairings

We propose an efficient instantiation of our generic construction based on the Groth-Schnorr sibling signatures SibGS that we introduced in Sec. 4.3.2.

In the generic construction, we have a sibling signature scheme Sib_i for each delegation level i, where Sib_i must sign the public key of Sib_{i+1}. Groth signature scheme uses bilinear group $\Lambda = (q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, e, g_1, g_2)$. Recall that $\mathsf{Groth}_{\mathbb{G}_1}$ signs messages in $\mathbb{G}_1$ with a public key in $\mathbb{G}_2$, while $\mathsf{Groth}_{\mathbb{G}_2}$ signs messages in $\mathbb{G}_2$ with a public key in $\mathbb{G}_1$. Therefore, we set Sib_{2n} to SibGS1 and Sib_{2n+1} to SibGS2. This means that we have attribute sets[1] $\mathbb{A}_{2n} = \mathbb{G}_1$ and $\mathbb{A}_{2n+1} = \mathbb{G}_2$.

---

[1] Alternatively, one could define a single attribute set $\mathbb{A}$ for all levels and use injective functions $f_1 : \mathbb{A} \rightarrow \mathbb{G}_1$ and $f_2 : \mathbb{A} \rightarrow \mathbb{G}_2$, such as setting $\mathbb{A} = \mathbb{Z}_q$ and $f_1(a) =$

$$
\begin{gathered}
\mathsf{SPK}\Big\{(\langle s'_i, t'_{i,j}\rangle_{i=1,\dots,L,j=1,\dots,n_i}, \langle a_{i,j}\rangle_{i\notin D}, \langle cpk_i\rangle_{i=1,\dots,L-1}, csk_L) : \\
\bigwedge_{i=1,3,\dots}^{L} \Bigg(e(y_{1,1}, g_2)\big[e(g_1, ipk)\big]_{i=1} = e(\underline{s'_i}, r'_i)\big[e(g_1^{-1}, \underline{cpk_{i-1}})\big]_{i\neq 1} \wedge \\
1_{\mathbb{G}_\mathrm{t}}\big[e(y_{1,1}, ipk)\big]_{i=1} = e(\underline{t'_{i,1}}, r'_i)\big[e(\underline{cpk_i}, g_2^{-1})\big]_{i\neq L}\big[e(g_1, g_2^{-1})^{\underline{csk_i}}\big]_{i=L}\big[e(y_{1,1}^{-1}, \underline{cpk_{i-1}})\big]_{i\neq 1} \wedge \\
\bigwedge_{j:(i,j)\in D} e(a_{i,j}, g_2)\big[e(y_{1,j+1}, ipk)\big]_{i=1} = e(\underline{t'_{i,j+1}}, r'_i)\big[e(y_{1,j+1}^{-1}, \underline{cpk_{i-1}})\big]_{i\neq 1} \wedge \\
\bigwedge_{j:(i,j)\notin D} 1_{\mathbb{G}_\mathrm{t}}\big[e(y_{1,j+1}, ipk)\big]_{i=1} = e(\underline{t'_{i,j+1}}, r'_i)e(\underline{a_{i,j}}, g_2^{-1})\big[e(y_{1,j+1}^{-1}, \underline{cpk_{i-1}})\big]_{i\neq 1}\Bigg) \wedge \\
\bigwedge_{i=2,4,\dots}^{L} \Bigg(e(g_1, y_{2,1}) = e(r'_i, \underline{s'_i})e(\underline{cpk_{i-1}}, g_2^{-1}) \wedge \\
1_{\mathbb{G}_\mathrm{t}} = e(r'_i, \underline{t'_{i,1}})e(\underline{cpk_{i-1}}, y_{2,1}^{-1})\big[e(g_1^{-1}, \underline{cpk_i})\big]_{i\neq L}\big[e(g_1^{-1}, g_2)^{\underline{csk_i}}\big]_{i=L} \wedge \\
\bigwedge_{j:(i,j)\in D} e(g_1, a_{i,j}) = e(r'_i, \underline{t'_{i,j+1}})e(\underline{cpk_{i-1}}, y_{2,j+1}^{-1}) \wedge \\
\bigwedge_{j:(i,j)\notin D} 1_{\mathbb{G}_\mathrm{t}} = e(r'_i, \underline{t'_{i,j+1}})e(\underline{cpk_{i-1}}, y_{2,j+1}^{-1})e(g_1^{-1}, \underline{a_{i,j}})\Bigg)\Big\}(sp, r'_1, \dots, r'_L, m).
\end{gathered}
$$

Figure 4.15: Efficient instantiation of the NIZK used to generate attribute tokens (witness underlined for clarity).

In addition to the bilinear group, SibGS1 requires parameters $y_{1,1}, \dots, y_{1,n+1} \in \mathbb{G}_1$, where n is the maximum number of attributes signed at an odd level ($n = max_{i=1,3,\dots}(n_i)$), and SibGS2 requires parameters $y_{2,1}, \dots, y_{2,n+1} \in \mathbb{G}_2$, for n the maximum number of attributes signed at an even level ($n = max_{i=2,4,\dots}(n_i)$). $\mathcal{F}_{\mathsf{crs}}$ provides both the bilinear groups Λ and the $y_{i,j}$ values.

We consider Level-0 to be an even level and, therefore, the issuer key pair is $(ipk = g_2^{isk}, isk)$. The issuer must verifiably encrypt its secret key isk to a public key from the CRS. This can be achieved in the random-oracle model using the techniques of Camenisch and Shoup [CS03].

4.5.1 A Concrete Proof for the Attribute Tokens

What remains to show is how to efficiently instantiate the zero-knowledge proof that constitutes the attribute tokens. Since we instantiate

---

$g_1^a, f_2(a) = g_2^a$), but for ease of presentation we omit this step and work directly with attributes in $\mathbb{G}_1$ and $\mathbb{G}_2$.

Sib_{2i} with $\mathsf{SibGS1}$ and Sib_{2i+1} with $\mathsf{SibGS2}$, we can rewrite the proof we need to instantiate as follows.

$$
\begin{aligned}
at \leftarrow \mathsf{NIZK}\Big\{&(\sigma_1, \ldots, \sigma_L, cpk_1, \ldots, cpk_L, \langle a'_{i,j}\rangle_{i \notin D}, tag) : \\
&\bigwedge_{i=1,3,\ldots}^{L} 1 = \mathsf{SibGS1.Verify}_1(cpk_{i-1}, \sigma_i, cpk_i, a'_{i,1}, \ldots, a'_{i,n_i}) \\
&\bigwedge_{i=2,4,\ldots}^{L} 1 = \mathsf{SibGS2.Verify}_1(cpk_{i-1}, \sigma_i, cpk_i, a'_{i,1}, \ldots, a'_{i,n_i}) \\
&\qquad\qquad\qquad\qquad \wedge\, 1 = \mathsf{SibGSb.Verify}_2(cpk_L, tag, m)\Big\}
\end{aligned}
$$

The proof has three parts: First, it proves all the odd-level credential links by proving that σ_i is valid using $\mathsf{SibGS1.Verify}_1$. Second, it proves the even-level credential links by proving that σ_i verifies with $\mathsf{SibGS2.Verify}_1$. Finally, it proves that the user signed message m with $\mathsf{SibGSb.Verify}_2$, where b depends on whether L is even or odd.

The abstract zero-knowledge proof can be efficiently instantiated with a generalized Schnorr zero-knowledge proof. Let $\sigma_i = (r_i, s_i, t_{i,1}, \ldots, t_{i,n_i+1})$. First, we use the fact that Groth is randomizable and randomize each signature to $(r'_i, s'_i, t'_{i,1}, \ldots, t'_{i,n_i+1})$. As r'_i is now uniform in the group, we can reveal the value rather than proving knowledge of it. Next, we use a Schnorr-type proof depicted in Fig. 4.15 to prove knowledge of the s and t values of the signatures, the undisclosed attributes, the credential public keys, and the credential secret key. The concrete zero-knowledge proof contains the same parts as described for the abstract zero-knowledge proof. The third part, proving knowledge of tag, is somewhat hidden. Recall that we instantiate $\mathsf{SibGSb.Verify}_2$ with Schnorr signatures, which means the signature is a proof of knowledge of csk_L. This can efficiently be combined with other two parts of the proof: instead of proving knowledge of cpk_L, we prove knowledge of csk_L.

4.5.2 Optimizing Attribute Token Computation

There is a lot of room for optimization when computing zero-knowledge proofs such as the one depicted in Fig. 4.15. We describe how to efficiently compute this specific proof, but many of these optimizations will be applicable to other zero-knowledge proofs in pairing-based settings.

```
input: $\langle r_i, s_i, \langle t_{i,j}\rangle_{j=1}^{n_1+1}\rangle_{i=1}^{L}, csk_L, \langle cpk_i\rangle_{i=1}^{L}, \langle a_{i,j}\rangle_{i=1,\ldots,L,j=1,\ldots,n_i}, D, sp, m$
for $i = 1,\ldots,L$ do                                                    ▷ Randomize $\sigma_i$
    $\rho_{\sigma_i} \xleftarrow{\$} \mathbb{Z}_q$, $r'_1 \leftarrow r_i^{\rho_{\sigma_i}}$, $s'_i \leftarrow s_i^{1/\rho_{\sigma_i}}$
    for $j = 1,\ldots n_i + 1$ do
        $t'_{i,j} \leftarrow t_{i,j}^{1/\rho_{\sigma_i}}$
    end for
end for
$\langle \rho_{s_i}, \langle \rho_{t_{i,j}}\rangle_{j=1}^{n_i+1}, \langle \rho_{a_{i,j}}\rangle_{j=1}^{n_i}\rangle_{i=1}^{L}, \langle \rho_{cpk_i}\rangle_{i=1}^{L-1}, \rho_{csk_L} \xleftarrow{\$} \mathbb{Z}_q$
for $i = 1,3,\ldots,L$ do                          ▷ Compute com-values for odd-level $\sigma_i$
    $\mathsf{com}_{i,1} \leftarrow e(g_1,r_i)^{\rho_{\sigma_i}\cdot\rho_{s_i}}\big[\cdot e(g_1^{-1},g_2)^{\rho_{cpk_{i-1}}}\big]_{i\neq 1}$
    $\mathsf{com}_{i,2} \leftarrow e(g_1,r_i)^{\rho_{\sigma_i}\cdot\rho_{t_{i,1}}}\cdot e(g_1,g_2^{-1})^{\rho_{cpk_i}}\big[\cdot e(y_{1,1},g_2)^{\rho_{cpk_{i-1}}}\big]_{i\neq 1}$
    for $j = 1,\ldots,n_i$ do
        if $(i,j) \in D$ then                                ▷ Attribute $a_{i,j}$ is disclosed
            $\mathsf{com}_{i,j+2} \leftarrow e(g_1,r_i)^{\rho_{\sigma_i}\cdot\rho_{t_{i,j+1}}}\big[\cdot e(y_{1,j+1},g_2)^{\rho_{cpk_{i-1}}}\big]_{i\neq 1}$
        else                                                   ▷ Attribute $a_{i,j}$ is hidden
            $\mathsf{com}_{i,j+2} \leftarrow e(g_1,r_i)^{\rho_{\sigma_i}\cdot\rho_{t_{i,j+1}}}\cdot e(g_1,g_2^{-1})^{\rho_{a_{i,j}}}\big[\cdot e(y_{1,j+1},g_2)^{\rho_{cpk_{i-1}}}\big]_{i\neq 1}$
        end if
    end for
end for
for $i = 2,4,\ldots,L$ do                         ▷ Compute com-values for even-level $\sigma_i$
    $\mathsf{com}_{i,1} \leftarrow e(r_i,g_2)^{\rho_{\sigma_i}\cdot\rho_{s_i}} e(g_1,g_2^{-1})^{\rho_{cpk_{i-1}}}$
    $\mathsf{com}_{i,2} \leftarrow e(r_i,g_2)^{\rho_{\sigma_i}\cdot\rho_{t_{i,1}}} e(g_1,y_{2,1}^{-1})^{\rho_{cpk_{i-1}}} e(g_1^{-1},g_2)^{\rho_{cpk_i}}$
    for $j = 1,\ldots,n_i$ do
        if $(i,j) \in D$ then                                ▷ Attribute $a_{i,j}$ is disclosed
            $\mathsf{com}_{i,j+2} \leftarrow e(g_1,y_{2,j+1}^{-1})^{\rho_{cpk_{i-1}}}\cdot e(r_i,g_2)^{\rho_{\sigma_i}\cdot\rho_{t_{i,j+1}}}$
        else                                                   ▷ Attribute $a_{i,j}$ is hidden
            $\mathsf{com}_{i,j+2} \leftarrow e(g_1,y_{2,j+1}^{-1})^{\rho_{cpk_{i-1}}}\cdot e(r_i,g_2)^{\rho_{\sigma_i}\cdot\rho_{t_{i,j+1}}}\cdot e(g_1^{-1},g_2)^{\rho_{a_{i,j}}}$
        end if
    end for
end for
$c \leftarrow \mathsf{H}(sp, ipk, \langle r'_i, \langle \mathsf{com}_{i,j}\rangle_{j=1}^{n_i+2}\rangle_{i=1}^{L}, \langle a_{i,j}\rangle_{(i,j)\in D}, m)$       ▷ Fiat-Shamir hash
for $i = 1,3,\ldots,L$ do                          ▷ Compute res-values for odd-level $\sigma_i$
    $\mathsf{res}_{s_i} = g_1^{\rho_{s_i}} s_i^c$,  $\big[\mathsf{res}_{cpk_i} = g_1^{\rho_{cpk_i}} cpk_i^c\big]_{i\neq L}$, $\big[\mathsf{res}_{csk_i} = \rho_{cpk_i} + c\cdot csk_i\big]_{i=L}$
    for $j = 1,\ldots,n_i + 1$ do
        $\mathsf{res}_{t_{i,j}} = g_1^{\rho_{t_{i,j}}} t_{i,j}^c$
    end for
    for $j = 1,\ldots,n_i$ with $(i,j) \notin D$ do
        $\mathsf{res}_{a_{i,j}} = g_1^{\rho_{a_{i,j}}} a_{i,j}^c$
    end for
end for
for $i = 2,4,\ldots,L$ do                         ▷ Compute res-values for even-level $\sigma_i$
    $\mathsf{res}_{s_i} = g_2^{\rho_{s_i}} s_i^c$,  $\big[\mathsf{res}_{cpk_i} = g_2^{\rho_{cpk_i}} cpk_i^c\big]_{i\neq L}$, $\big[\mathsf{res}_{csk_i} = \rho_{cpk_i} + c\cdot csk_i\big]_{i=L}$
    for $j = 1,\ldots,n_i + 1$ do
        $\mathsf{res}_{t_{i,j}} = g_2^{\rho_{t_{i,j}}} t_{i,j}^c$
    end for
    for $j = 1,\ldots,n_i$ with $(i,j) \notin D$ do
        $\mathsf{res}_{a_{i,j}} = g_2^{\rho_{a_{i,j}}} a_{i,j}^c$
    end for
end for
output: $c, \langle r'_i, \mathsf{res}_{s_i}, \langle \mathsf{res}_{t_{i,j}}\rangle_{j=1}^{n_i+1}\rangle_{i=1}^{L}, \langle \mathsf{res}_{a_{i,j}}\rangle_{(i,j)\notin D}, \langle \mathsf{res}_{cpk_i}\rangle_{i=1}^{L-1}, \mathsf{res}_{csk_L}$
```

Figure 4.16: Pseudocode for efficiently computing attribute tokens.

Computing attribute tokens. The pairing operation is the most expensive operation in bilinear groups, so for the efficiency of the scheme it is beneficial to minimize the amount of pairings computed. We can use some optimizations in computing the zero-knowledge proof that remove the need to compute any pairings. As a small example, suppose we prove $\mathsf{SPK}\{x : z = e(x, b)\}$. The standard way to compute this is taking $r_x \xleftarrow{\$} \mathbb{G}_1$, computing $\mathsf{com} \leftarrow e(r_x, b)$, $c \leftarrow \mathsf{H}(\mathsf{com}, \ldots)$, and $\mathsf{res}_x \leftarrow r_x \cdot x^c$. We can compute the same values without computing the pairing by precomputing $e(g, b)$, taking $\rho \xleftarrow{\$} \mathbb{Z}_q$ and setting $\mathsf{com} \leftarrow e(g_1, b)^\rho$ and $\mathsf{res} \leftarrow g_1^\rho x^c$.

To prove knowledge of a Groth signature, we must prove $z = e(x, r')$, where r' is the randomized r-value of the Groth signature. If we try to apply the previous trick, we set $\rho_x \xleftarrow{\$} \mathbb{Z}_q$, $\mathsf{com}_x \leftarrow e(g_1, r')^{\rho_x}$. However, now we cannot precompute $e(g_1, r')$ since r' is randomized before every proof. We can solve this by remembering the randomness used to randomize the Groth signature. Let $r' = r^{\rho_\sigma}$, we can compute $\mathsf{com}_x \leftarrow e(g_1, r)^{\rho_\sigma \cdot \rho_x}$ by precomputing $e(g_1, r)$. The full pseudocode for computing the proofs using these optimizations is given in Fig. 4.16.

Verifying attribute tokens. In verification, computing pairings is unavoidable, but there are still tricks to keep verification efficient. The pairing function is typically instantiated with the tate pairing, which consists of two parts: Miller's algorithm $\hat{t}(\cdot)$ and the final exponentiation $\mathsf{fexp}(\cdot)$ [DSD07]. Both parts account for roughly half the time required to compute a pairing.[2] When computing the product of multiple pairings, we can compute the Miller loop for every pairing and then compute the final exponentiation only once for the whole product. This means that computing the product of three pairings is roughly equally expensive as computing two individual pairings.

Fig. 4.17 shows how to verify attribute tokens efficiently using this observation. When we write $e(a, b)$ in the pseudocode, it means we can precompute the value.

4.5.3 Efficiency Analysis of Our Instantiation

We now analyze the efficiency of our construction. Namely, we calculate the number of pairing operations and (multi-) exponentiations

---

[2] We verified this by running `bench_pair.c` of the AMCL library (`github.com/miracl/amcl`) using the BN254 curve.

```
input: $c, \langle r'_i, \mathsf{res}_{s_i}, \langle \mathsf{res}_{t_{i,j}} \rangle_{j=1}^{n_i+1} \rangle_{i=1}^{L}, \langle \mathsf{res}_{a_{i,j}} \rangle_{(i,j) \notin D}, \langle \mathsf{res}_{cpk_i} \rangle_{i=1}^{L-1}, \mathsf{res}_{csk_L},$
    $\langle a_{i,j} \rangle_{(i,j \in D)}, D, sp, m$
for $i = 1, 3, \ldots, L$ do                                  ▷ Recompute com-values for odd-level $\sigma_i$
    $\mathsf{com}_{i,1} \leftarrow \mathsf{fexp}(\hat{t}(\mathsf{res}_{s_i}, r'_i) \big[ \cdot \hat{t}(g_1^{-1}, \mathsf{res}_{cpk_{i-1}}) \big]_{i \neq 1}) \cdot (e(y_{1,1}, g_2) \big[ \cdot e(g_1, ipk) \big]_{i=1})^{-c}$
    $\mathsf{com}_{i,2} \leftarrow \mathsf{fexp}(\hat{t}(\mathsf{res}_{t_{i,1}}, r'_i) \big[ \cdot \hat{t}(y_{1,1}, \mathsf{res}_{cpk_{i-1}}) \big]_{i \neq 1} \big[ \cdot \hat{t}(\mathsf{res}_{cpk_i}, g_2^{-1}) \big]_{i \neq L}) \big[ \cdot e(g_1, g_2^{-1})^{\mathsf{res}_{csk_i}} \big]_{i=L} \big[ \cdot e(y_{1,1}, ipk)^{-c} \big]_{i=1}$
    for $j = 1, \ldots, n_i$ do
        if $(i,j) \in D$ then                                 ▷ Attribute $a_{i,j}$ is disclosed
            $\mathsf{com}_{i,j+2} \leftarrow$
                $\mathsf{fexp}(\hat{t}(\mathsf{res}_{t_{i,j+1}}, r'_i) \big[ \cdot \hat{t}(y_{1,j+1}, \mathsf{res}_{cpk_{i-1}}) \big]_{i \neq 1}) \cdot (e(a_{i,j}, g_2) \big[ e(y_{1,j+1}, ipk) \big]_{i=1})^{-c}$
        else                                                  ▷ Attribute $a_{i,j}$ is hidden
            $\mathsf{com}_{i,j+2} \leftarrow$
                $\mathsf{fexp}(\hat{t}(\mathsf{res}_{t_{i,j+1}}, r'_i) \cdot \hat{t}(\mathsf{res}_{a_{i,j}}, g_2^{-1}) \big[ \cdot \hat{t}(y_{1,j+1}, \mathsf{res}_{cpk_{i-1}}) \big]_{i \neq 1}) \big[ \cdot e(y_{1,j+1}, ipk)^{-c} \big]_{i=1}$
        end if
    end for
end for
for $i = 2, 4, \ldots, L$ do                                  ▷ Compute com-values for even-level $\sigma_i$
    $\mathsf{com}_{i,1} \leftarrow \mathsf{fexp}(\hat{t}(r'_i, \mathsf{res}_{s_i}) \cdot \hat{t}(\mathsf{res}_{cpk_{i-1}}, g_2^{-1})) \cdot e(g_1, y_{2,1})^{-c}$
    $\mathsf{com}_{i,2} \leftarrow \mathsf{fexp}(\hat{t}(r'_i, \mathsf{res}_{t_{i,1}}) \cdot \hat{t}(\mathsf{res}_{cpk_{i-1}}, y_{2,1}^{-1}) \big[ \cdot \hat{t}(g_1^{-1}, \mathsf{res}_{cpk_i}) \big]_{i \neq L}) \big[ \cdot e(g_1^{-1}, g_2)^{\mathsf{res}_{csk_i}} \big]_{i=L}$
    for $j = 1, \ldots, n_i$ do
        if $(i,j) \in D$ then                                 ▷ Attribute $a_{i,j}$ is disclosed
            $\mathsf{com}_{i,j+2} \leftarrow \mathsf{fexp}(\hat{t}(\mathsf{res}_{cpk_{i-1}}, y_{2,j+1}^{-1}) \cdot \hat{t}(r'_i, \mathsf{res}_{t_{i,j+1}})) \cdot e(g_1, a_{i,j})^{-c}$
        else                                                  ▷ Attribute $a_{i,j}$ is hidden
            $\mathsf{com}_{i,j+2} \leftarrow \mathsf{fexp}(\hat{t}(\mathsf{res}_{cpk_{i-1}}, y_{2,j+1}^{-1}) \cdot \hat{t}(r'_i, \mathsf{res}_{t_{i,j+1}}) \cdot \hat{t}(g_1^{-1}, \mathsf{res}_{a_{i,j}}))$
        end if
    end for
end for
$c' \leftarrow \mathsf{H}(sp, ipk, \langle r'_i, \langle \mathsf{com}_{i,j} \rangle_{j=1}^{n_i+2} \rangle_{i=1}^{L}, \langle a_{i,j} \rangle_{(i,j) \in D}, m)$     ▷ Fiat-Shamir hash
output: $c = c'$
```

Figure 4.17: Pseudocode for efficiently verifying attribute tokens.

in different groups that is required to compute and verify attribute tokens. We also compute the size of credentials and attribute tokens with respect to a delegation level and number of attributes, and provide concrete timings for our prototype implementation in C that generates and verifies Level-2 attribute tokens.

Let d_i and u_i denote the amount of disclosed and undisclosed attributes at delegation level i, respectively, and we define $n_i = d_i + u_i$.

Computational efficiency.

Let us count the operations required to perform the recurring operations, namely delegating credentials, presenting credentials, and verifying attribute tokens. For operations we use the following notation. We use $X\{\mathbb{G}_1^j\}$, $X\{\mathbb{G}_2^j\}$, and $X\{\mathbb{G}_\mathrm{t}^j\}$ to denote X j-multi-exponentiations

Algorithm	**Operations**	**Total time estimate (ms)**
DELEGATE	For each odd Level-i: $1\{\mathbb{G}_2\} + (n_i + 2)\{\mathbb{G}_1\} + (n_i + 1)\{\mathbb{G}_1^2\}$	$2.96 + 1.21n_i$
	For each even Level-i: $1\{\mathbb{G}_1\} + (n_i + 2)\{\mathbb{G}_2\} + (n_i + 1)\{\mathbb{G}_2^2\}$	$5.27 + 3.52n_i$
PRESENT	$\sum_{i=1,3,..}^{L} \big(1\{\mathbb{G}_2\} + (n_i+2)\{\mathbb{G}_1\} + (1 + d_i)\{\mathbb{G}_{\mathrm{t}}^2\} + (1 + u_i)\{\mathbb{G}_{\mathrm{t}}^3\} + (2 + n_i)\{\mathbb{G}_1^2\}\big)$	$\sum_{i=1,3,..}^{L}(13.63 + 3.89d_i + 6.11u_i + 1.21n_i) +$
	$\sum_{i=2,4,..}^{L} \big(1\{\mathbb{G}_1\} + (n_i+2)\{\mathbb{G}_2\} + (1 + d_i)\{\mathbb{G}_{\mathrm{t}}^2\} + (1 + u_i)\{\mathbb{G}_{\mathrm{t}}^3\} + (2 + n_i)\{\mathbb{G}_2^2\}\big)$	$\sum_{i=2,4,..}^{L}(17.58 + 3.89d_i + 6.11u_i + 3.52n_i)$
VERIFY	$(1 + d_1)E + (3 + u_1 + d_L)E^2 + u_L E^3 + (4 + n_1 + d_L)\{\mathbb{G}_{\mathrm{t}}\} + \sum_{i=2,3,..}^{(L-1)} \big((1 + d_i)E^2 + (1 + u_i)E^3 + (1 + d_i)\{\mathbb{G}_{\mathrm{t}}\}\big)$	$21.65 + 2.36d_1 + 3.91u_1 + 1.89n_1 + 5.80d_L + 5.48u_L + \sum_{i=2,3,..}^{(L-1)} \big(11.28 + 5.80d_i + 5.48u_i\big)$

Table 4.1: Performance evaluation and timing estimations, where d_i and u_i denote the amount of disclosed and undisclosed attributes at delegation level i, respectively, and $n_i = d_i + u_i$; $X\{\mathbb{G}_1^j\}$, $X\{\mathbb{G}_2^j\}$, and $X\{\mathbb{G}_{\mathrm{t}}^j\}$ denote X j-multi-exponentiations in the respective group; $j = 1$ means a simple exponentiation. E^k denote a k-pairing product that we can compute with k-Miller loops and a single shared final exponentiation; $k = 1$ means a single pairing. Benchmarks are (all in ms): $1\{\mathbb{G}_1\} = 0.54; 1\{\mathbb{G}_1^2\} = 0.67; 1\{\mathbb{G}_2\} = 1.21; 1\{\mathbb{G}_2^2\} = 2.31; 1\{\mathbb{G}_{\mathrm{t}}\} = 1.89; 1\{\mathbb{G}_{\mathrm{t}}^2\} = 3.89; 1\{\mathbb{G}_{\mathrm{t}}^3\} = 6.11; 1E = 2.36; 1E^2 = 3.91; 1E^3 = 5.48$.

in the respective group; $j = 1$ means a simple exponentiation. We denote as E^k a k-pairing product that we can compute with k-Miller loops and a single shared final exponentiation.

Delegation. Delegation of a credential includes generating a key and a signature on the public key and a set of attributes:

- for even i the cost is $1\{\mathbb{G}_1\} + (n_i + 2)\{\mathbb{G}_2\} + (n_i + 1)\{\mathbb{G}_2^2\}$,
- for odd i the cost is $1\{\mathbb{G}_2\} + (n_i + 2)\{\mathbb{G}_1\} + (n_i + 1)\{\mathbb{G}_1^2\}$.

Signature verification for Level-i costs $n_i \cdot E^3$ plus E^2 or E^3, depending on if the pairing with the public key was pre-computed or not.

Computing attribute tokens (Presentation). Randomizing σ_i costs $(n_i+2) \cdot \{\mathbb{G}_1\} + 1\{\mathbb{G}_2\}$ for odd i and $1\{\mathbb{G}_1\} + (n_i+2)\{\mathbb{G}_2\}$ for even i. Computing the com-values for Level-1 costs $(1+d_i)\{\mathbb{G}_{\mathrm{t}}\} + (1+u_i)\{\mathbb{G}_{\mathrm{t}}^2\}$.

The com-values for Level-i for $i > 1$ cost $(1 + d_i)\{\mathbb{G}_t^2\} + (1 + u_i)\{\mathbb{G}_t^3\}$. Computing the res-values for odd i costs $(2 + n_i)\{\mathbb{G}_1^2\}$, and for even i it costs $(2 + n_i)\{\mathbb{G}_2^2\}$, except the last level, where $1\{\mathbb{G}_1^2\}$ or $1\{\mathbb{G}_2^2\}$ can be saved when L is even or odd, respectively.

If we consider a practical example, where we show Level-2 credentials with attributes only on Level-1 (meaning that $n_2 = 0$), computing the attribute token costs very roughly $3n_1 + 13$ exponentiations, and more precisely: $(3 + n_1)\{\mathbb{G}_1\} + (2 + n_1)\{\mathbb{G}_1^2\} + 3\{\mathbb{G}_2\} + 1\{\mathbb{G}_2^2\} + (1 + d_1)\{\mathbb{G}_t\} + (2 + u_1)\{\mathbb{G}_t^2\} + 1\{\mathbb{G}_t^3\}$.

Verifying attribute tokens. Verifying the first credential link costs $(1 + d_1)E + (1 + u_1)E^2 + (2 + n_1)\{\mathbb{G}_t\}$ and one final exponentiation. Every next level adds $(1 + d_i)E^2 + (1 + u_i)E^3 + (1 + d_i)\{\mathbb{G}_t\}$, except the last level, which costs $(2 + d_i)E^2 + u_iE^3 + (2 + d_i)\{\mathbb{G}_t\}$.

For the same practical example with two levels, to verify a Level-2 attribute token will cost very roughly $n_1 + 4$ pairings and $n_1 + 4$ exponentiations, and more precisely: $(1 + d_1)E + (3 + u_1)E^2 + (4 + n_1)\{\mathbb{G}_t\}$. We summarize the above efficiency analysis in Table 4.1.

Size of attribute tokens

To count the size of an attribute token we use the following notation. We use $X[\mathbb{G}_1]$ and $X[\mathbb{G}_2]$ to denote X group elements from the respective group. The attribute token proves knowledge of every credential link, so the token grows in the credential level.

First, we look at credential links without attributes. For every level a credential link adds 4 group elements: $3[\mathbb{G}_1] + 1[\mathbb{G}_2]$ for an odd and $1[\mathbb{G}_1] + 3[\mathbb{G}_2]$ for an even level, respectively. Additionally, a token has 2 elements from $\mathbb{Z}_q$. This means that for even L, an attribute token generated from a Level-L credential without attributes takes $(2L)[\mathbb{G}_1] + (2L - 1)[\mathbb{G}_2] + 2\mathbb{Z}_q$.

Every attribute added to an odd level credential link adds one group element, if it is disclosed, and two elements, if this attribute remains hidden. For the odd levels these are the elements from $[\mathbb{G}_1]$ and for even levels - from $[\mathbb{G}_2]$. This means that for even L, an attribute token generated from a Level-L credential takes $(2L + \sum_{i=1,3,\ldots}^{L-1}(n_i + u_i))[\mathbb{G}_1] + (2L - 1 + \sum_{i=2,4,\ldots}^{L}(n_i + u_i))[\mathbb{G}_2] + 2\mathbb{Z}_p$.

n_1	n_2	PRESENT	VERIFY	EST. PRES.	EST. VERIFY
0	0	26.9 ms	20.2 ms	31.21 ms	21.65 ms
1	0	32.7 ms	25.4 ms	38.53 ms	27.45 ms
2	0	38.1 ms	30.9 ms	45.85 ms	33.25 ms
3	0	44.0 ms	36.1 ms	53.17 ms	39.05 ms
4	0	49.5 ms	41.4 ms	60.49 ms	44.85 ms
0	1	38.6 ms	24.8 ms	40.84 ms	27.13 ms
0	2	49.4 ms	29.2 ms	50.47 ms	32.61 ms
0	3	61.5 ms	34.1 ms	60.10 ms	38.09 ms
0	4	72.6 ms	38.7 ms	69.73 ms	43.57 ms
1	1	43.7 ms	30.1 ms	48.16 ms	32.93 ms
2	1	49.3 ms	35.4 ms	55.48 ms	38.73 ms

Table 4.2: Performance measurements of presenting and verifying Level-2 credentials, and our estimated timings following the computation of Table 4.1. No attributes are disclosed.

Implementation and Performance Analysis.

We have implemented a prototype of our concrete instantiation for delegatable credentials in the C programming language, using the Apache Milagro Cryptographic Library (AMCL) with a 254-bit Barreto-Naehrig curve [BN06a]. This prototype generates and verifies Level-2 attribute tokens. The prototype shows the practicality of our construction: generating an attribute token without attributes takes only 27 ms, and verification requires only 20 ms, on a 3.1GHz Intel I7-5557U laptop CPU. Table 4.2 shows performance figures when presenting tokens with attributes. Adding undisclosed attributes in the first credential link (that is, increasing n_1) adds roughly 6 ms to the token generation time per attribute, while adding undisclosed attributes in the second link (thus increasing n_2) adds 11 ms. For verification, every added undisclosed attribute increases verification time by 5 ms. Table 4.2 also shows that our estimated timings in Table 4.1 are accurate: the estimated values are close to the measured timings and our estimates are even a bit conservative.

Chapter 5

Anonymous Attestation

The Trusted Platform Module (TPM) is a chip embedded in a host computer that can create trustworthy attestations about the host's state. This chapter focuses on Direct Anonymous Attestation (DAA), which is a protocol allowing attestations that are anonymous by using techniques from anonymous credentials. The main difference is that the role of user in anonymous credentials is split into a TPM and a host in DAA, where we require attestations to be trustworthy as long as the TPM and credential issuer are honest, while privacy should be the responsibility of the host. First, we show that previous security notions for DAA have shortcomings and put forth a new security definition. Second, we present a secure and efficient DAA scheme from a global random oracle, building on our results from Chapter 3.

5.1 Introduction

Direct Anonymous Attestation (DAA) allows a small chip, the Trusted Platform Module (TPM), that is embedded in a host computer to create attestations about the state of the host system. Such attestations, which can be seen as signatures on the current state under the TPM's secret key, convince a remote verifier that the system it is communicating with is running on top of certified hardware and is using the correct software. A crucial feature of DAA is that it performs such attestations in a privacy-friendly manner. That is, the user of the host system can choose to create attestations anonymously ensuring that her transactions are unlinkable and do not leak any information about the particular TPM being used.

DAA was introduced by Brickell, Camenisch, and Chen [BCC04] for the Trusted Computing Group and was standardized in the TPM 1.2 specification in 2004 [Tru04]. Their paper inspired a large body of work on DAA schemes [BCL08, CMS08b, CF08, BCL09, Che09, CPS10, BL10, BFG$^+$13b, CDL16b, CDL16a, CCD$^+$17], including more efficient schemes using bilinear pairings as well as different security definitions and proofs. One result of these works is the recent TPM 2.0 specification [Tru14, Int15] that includes support for multiple pairing-based DAA schemes, two of which are standardized by ISO [Int13]. Recently, the protocol has received renewed attention for authentication: An extension of DAA called EPID is used in Intel SGX [CD16], the most recent development in the area of trusted computing. Further, the FIDO alliance, an industry consortium designing standards for strong user authentication, is in the process of standardizing a specification using DAA to attest that authentication keys are securely stored [CDE$^+$18].

Existing Security Definitions. Interestingly, in spite of the large scale deployment and the long body of work on the subject, DAA still lacks a sound and comprehensive security definition. There exist a number of security definitions in the literature. Unfortunately all of them have rather severe shortcomings such as allowing for obviously broken schemes to be proven secure. This was recently discussed by Bernard et al. [BFG$^+$13b] who provide an analysis of existing security notions and also propose a new security definition. In a nutshell, the existing definitions that capture the desired security properties in the form of an ideal functionality either fail to treat signatures as concrete objects that can be output or stored by the verifier [BCC04] or are unrealizable [CMS08a, CMS09]. The difficulty in defining a proper ideal functionality for the complex DAA setting might not be all that surprising considering the numerous (failed) attempts in modeling the much simpler standard signature scheme in the universal composability framework [BH04, Can04].

Another line of work therefore aimed at capturing the DAA requirements in the form of game-based security properties [BCL09, Che09, BFG$^+$13b] as a more intuitive way of modeling. However, the first attempts [BCL09, Che09] have failed to cover some of the expected security properties and also have made unconventional choices when defining unforgeability (the latter resulting in schemes being considered secure that use a *constant* value as signatures).

Realizing that the previous definitions were not sufficient, Bernard et al. [BFG$^+$13b] provided an extensive set of property-based security games. The authors consider only a simplified setting which they call pre-DAA. The simplification is that the host and the TPM are considered as single entity (the platform), thus they are both either corrupt or honest. For properties such as anonymity and non-frameability this is sufficient as they protect against a corrupt issuer and assume both the TPM and the host to be honest. Unforgeability of a TPM attestation, however, should rely only on the TPM being honest but allow the host to be corrupt. This cannot be captured in their model. In fact, shifting the load of the computational work to the host without affecting security in case the host is corrupted is one of the main challenges when designing a DAA scheme. Therefore, a DAA security definition should allow one to formally analyze the setting of an honest TPM and a corrupt host.

This is also acknowledged by Bernard et al. [BFG$^+$13b] who, after proposing a pre-DAA secure protocol, argue how to obtain a protocol achieving full DAA security. Unfortunately, due to the absence of a full DAA security model, this argumentation is done only informally. In this paper we show that their argumentation is actually somewhat flawed: the given proof for unforgeability of the given pre-DAA proof can not be lifted (under the same assumptions) to the full DAA setting. This highlights the fact that an “almost matching” security model together with an informal argument of how to achieve the actually desired security does not provide sound guarantees beyond what is formally proved.

Thus still no satisfying security model for DAA exists to date. This lack of a sound security definition is not only a theoretic problem but has resulted in insecure schemes being deployed in practice. A DAA scheme that allows anyone to forge attestations (as it does not exclude the “trivial” TPM credential $(1, 1, 1, 1)$) has even been standardized in ISO/IEC 20008-2 [CPS10, Int13].[1]

Trusting Hardware for Privacy? The first version of the TPM specification and attestation protocol had received strong criticism from

---

[1] We have corrected this vulnerability by submitting a technical corrigendum which was approved by ISO/IEC and published in December 2017.

privacy groups and data protection authorities as it imposed linkability and full identification of all attestations. As a consequence, guaranteeing the privacy of the platform, i.e., ensuring that an attestation does not carry any identifier, became an important design criteria for such hardware-based attestation. Indeed, various privacy groups and data protection authorities had been consulted in the design process of DAA.

Surprisingly, despite the strong concerns of having to trust a piece of hardware when TPMs and hardware-based attestation were introduced, the problem of privacy-preserving attestation in the presence of fraudulent hardware has not been fully solved yet. The issue is that the original DAA protocol as well as all other DAA protocols crucially rely on the honesty of the entire platform, i.e., host and TPM, for guaranteeing privacy. Clearly, assuming that the host is honest is unavoidable for privacy, as it communicates directly with the outside world and can output any identifying information it wants. However, further requiring that the TPM behaves in a fully honest way and aims to preserve the host's privacy is an unnecessarily strong assumption and contradicts the initial design goal of not having to trust the TPM.

Even worse, it is impossible to verify this strong assumption as the TPM is a chip that comes with pre-installed software, to which the user only has black-box access. While black-box access might allow one to partly verify the TPM's functional correctness, it is impossible to validate its *privacy* guarantees. A compromised TPM manufacturer can ship TPMs that provide seemingly correct outputs, but that are formed in a way that allows dedicated entities (knowing some trapdoor) to trace the user, for instance by encoding an identifier in a nonce that is hashed as part of the attestation signature. It could further encode its secret key in attestations, allowing a fraudulent manufacturer to *frame* an honest host by signing a statement on behalf of the platform. We stress that such attacks are possible on all current DAA schemes, meaning that, by compromising a TPM manufacturer, all TPMs it produces can be used as mass surveillance devices. The revelations of subverted cryptographic standards [PLS13, BBG13] and tampered hardware [Gre14] indicate that such attack scenarios are very realistic.

In contrast to the TPM, the host software can be verified by the user, e.g., being compiled from open source, and will likely run on hardware that is not under the control of the TPM manufacturer. Thus, while the honesty of the host is vital for the platform's privacy and there are means to verify or enforce such honesty, requiring the TPM to be honest is neither necessary nor verifiable.

5.1.1 Our Contribution

In this chapter we address this problem of anonymous attestation without having to trust a piece of hardware, a problem which has been open for more than a decade. We exhibit a new DAA protocol that provides privacy even if the TPM is subverted. More precisely, our contributions are threefold: we first present a formal security model for DAA, then we show how to model subverted parties within the Universal Composability (UC) model, and finally propose a protocol that is secure against subverted TPMs. This chapter is based on [CDL17].

A Formal DAA Model. We tackle the challenge of formally defining Direct Anonymous Attestation and provide an ideal functionality for DAA in the Universal Composability (UC) framework [Can00]. Our functionality models hosts and TPMs as individual parties who can be in different corruption states and comprises all expected security properties such as unforgeability, anonymity, and non-frameability. The model also includes verifier-local revocation where a verifier, when checking the validity of a signature, can specify corrupted TPMs from which he no longer accepts signatures.

We choose to define a new model rather than addressing the weaknesses of one of the existing models. The latest DAA security model by Bernard et al. [BFG$^+$13b] seems to be the best starting point. However, as their model covers pre-DAA only, changing all their definitions to full DAA would require changes to almost every aspect of them. Furthermore, given the complexity of DAA, we believe that the simulation-based approach is more natural as one has a lower risk of overlooking security properties. A functionality provides a full description of security and no oracles have to be defined as the adversary simply gets full control over corrupt parties. Furthermore, the UC framework comes with strong composability guarantees that allow for protocols to be analyzed individually and preserve that security when being composed with other protocols.

Modeling Subversion Attacks in UC. Our DAA functionality provides privacy guarantees in the case where the TPM is corrupt and the host remains honest. Modeling corruption in the sense of subverted parties is not straightforward: if the TPM was simply controlled by the adversary, then, using the standard UC corruption model, only very limited privacy can be achieved. The TPM has to see and approve every message it signs but, when corrupted, all these messages are given to the adversary as well. In fact, the adversary will learn which particular TPM is asked to sign which message. That is, the adversary can later recognize a certain TPM attestation via its message, even if the signatures are anonymous.

Modeling corruption of TPMs like this gives the adversary much more power than in reality: even if a TPM is subverted and runs malicious algorithms, it is still embedded into a host who controls all communication with the outside world. Thus, the adversary cannot communicate directly with the TPM, but only via the (honest) host. To model such subversions more accurately, we introduce *isolated* corruptions in UC. When a TPM is corrupted like this, we allow the ideal-world adversary (simulator) to specify a piece of code that the isolated, yet subverted TPM will run. Other than that, the adversary has no control over the isolated corrupted party, i.e., it cannot directly interact with the isolated TPM and cannot see its state. Thus, the adversary will also not automatically learn anymore which TPM signed which message.

A New DAA Protocol with Optimal Privacy. We further discuss why the existing DAA protocols do not offer privacy when the TPM is corrupt and propose a new DAA protocol which we prove to achieve our strong security definition. In contrast to most existing schemes, we construct our protocol from generic building blocks which yields a more modular design. A core building block are *split signatures* which allow two entities – in our case the TPM and host – each holding a secret key share to jointly generate signatures. Using such split keys and signatures is a crucial difference compared with all existing schemes, where only the TPM contributed to the attestation key which inherently limits the possible privacy guarantees. We also redesign the overall protocol such that the main part of the attestation, namely proving knowledge of a membership credential on the attestation key, can be done by the host instead of the TPM.

By shifting more responsibility and computations to the host, we do not only increase privacy, but also achieve stronger notions of non-frameability and unforgeability than all previous DAA schemes. Interestingly, this design change also improves the efficiency of the TPM, which is usually the bottleneck in a DAA scheme. In fact, we propose a pairing-based instantiation of our generic protocol which, compared to prior DAA schemes, has the most efficient TPM signing operation. This comes for the price of higher computational costs for the host and verifier. However, we estimate signing and verification times of around 20ms, which is sufficiently fast for most practical applications.

5.1.2 Related Work

The idea of combining a piece of tamper-resistant hardware with a user-controlled device was first suggested by Chaum [Cha92] and applied to the context of e-cash by Chaum and Pedersen [CP93], which got later refined by Cramer and Pedersen [CP94] and Brands [Bra94]. A user-controlled wallet is required to work with a piece of hardware, the observer, to be able to withdraw and spend e-cash. The wallet ensures the user's privacy while the observer prevents a user from double-spending his e-cash. Later, Brands in 2000 [Bra00] considered the more general case of user-bound credentials where the user's secret key is protected by a smart card. Brands proposes to let the user's host add randomness to the smart card contribution as a protection against subliminal channels. All these works use a blind signature scheme to issue credentials to the observers and hence such credentials can only be used a single time.

Young and Yung further study the protection against subverted cryptographic algorithms with their work on kleptography [YY97a, YY97b] in the late 1990s. Recently, caused by the revelations of subverted cryptographic standards [PLS13, BBG13] and tampered hardware [Gre14] as a form of mass-surveillance, this problem has again gained substantial attention.

Subversion-Resilient Cryptography. Bellare et al. [BPR14] provided a formalization of algorithm-substitution attacks and considered the challenge of securely encrypting a message with an encryption algorithm that might be compromised. Here, the corruption is limited to attacks where the subverted party's behavior is indistinguishable from that of a correct implementation, which models the goal of the adversary to remain undetected. This notion of algorithm-substitution attacks was later applied to signature schemes, with the goal of preserving unforgeability in the presence of a subverted signing algorithm [AMV15].

However, these works on subversion-resilient cryptography crucially rely on honestly generated keys and aim to prevent key or information leakage when the algorithms using these keys get compromised.

Recently, Russell et al. [RTYZ16, RTYZ17] extended this line of work by studying how security can be preserved when *all* algorithms, including the key generation can be subverted. The authors also propose immunization strategies for a number of primitives such as one-way permutations and signature schemes. The approach of replacing a correct implementation with an indistinguishable yet corrupt one is similar to the approach in our work, and like Russell et al. we allow the subversion of all algorithms, and aim for security (or rather privacy) when the TPM behaves maliciously already when generating the keys.

The DAA protocol studied in this work is more challenging to protect against subversion attacks though, as the signatures produced by the TPM must not only be unforgeable and free of a subliminal channel which could leak the signing key, but also be anonymous and unlinkable, i.e., signatures must not leak any information about the signer even when the key is generated by the adversary. Clearly, allowing the TPM to run subverted keys requires another trusted entity on the user's side in order to hope for any privacy-protecting operations. The DAA setting naturally satisfies this requirement as it considers a platform to consist of two individual entities: the TPM and the host, where all of TPM's communication with the outside world is run via the host.

Reverse Firewalls. This two-party setting is similar to the concept of reverse firewalls recently introduced by Mironov and Stephens-Davidowitz [MS15]. A reverse firewall sits in between a user's machine and the outside world and guarantees security of a joint cryptographic operation even if the user's machine has been compromised. Moreover, the firewall-enhanced scheme should maintain the original functionality and security, meaning the part run on the user's computer must be fully functional and secure on its own without the firewall. Thus, the presence of a reverse firewall can enhance security if the machine is corrupt but is not the source of security itself. This concept has been proven very powerful and manages to circumvent the negative results of resilience against subversion-attacks [DMSD16, CMY$^+$16].

The DAA setting we consider in this chapter is not as symmetric as a reverse firewall though. While both parties contribute to the unforgeability of attestations, the privacy properties are only achievable if the host is honest. In fact, there is no privacy towards the host, as the host is fully aware of the identity of the embedded TPM. The requirement of privacy-protecting and unlinkable attestation only applies to the final output produced by the host.

Divertible Protocols & Local Adversaries. A long series of related work explores divertible and mediated protocols [BD95, OO90, BBS98, AsV08], where a special party called the mediator controls the communication and removes hidden information in messages by rerandomizing them. The host in our protocol resembles the mediator, as it adds randomness to every contribution to the signature from the TPM. However, in our case the host is a normal protocol participant, whereas the mediator's sole purpose is to control the communication.

Alwen et al. [AKMZ12] and Canetti and Vald [CV12] consider local adversaries to model isolated corruptions in the context of multi-party protocols. These works thoroughly formalize the setting of multi-party computations where several parties can be corrupted, but are controlled by different and non-colluding adversaries. In contrast, the focus of this work is to limit the communication channel that the adversary has to the corrupted party itself. We leverage the flexibility of the UC model to define such isolated corruptions.

Generic MPC. Multi-party computation (MPC) was introduced by Yao [Yao82] and allows a set of parties to securely compute any function on private inputs. Although MPC between the host and TPM could solve our problem, a negative result by Katz and Ostrovsky [KO04] shows that this would require at least five rounds of communication, whereas our tailored solution is much more efficient. Further, none of the existing MPC models considers the type of subverted corruptions that is crucial to our work, i.e., one first would have to extend the existing models and schemes to capture such isolated TPM corruption. This holds in particular for the works that model tamper-proof hardware [Kat07, HPV16], as therein the hardware is assumed to be "perfect" and unsubvertable.

5.2 Issues in Existing Security Definitions

In this section we briefly discuss why current security definitions do not properly capture the security properties one would expect from a DAA scheme. Some of the arguments were already pointed out by Bernhard et al. [BFG$^+$13b], who provide a thorough analysis of the existing DAA security definitions and also propose a new set of definitions. For the sake of completeness, we summarize and extend their findings and also give an assessment of the latest definition by Bernhard et al.

Before discussing the various security definitions and their limitation, we informally describe how DAA works and what are the desired security properties. In a DAA scheme, we have four main entities: a number of trusted platform modules (TPM), a number of hosts, an issuer, and a number of verifiers. A TPM and a host together form a platform which performs the *join protocol* with the issuer who decides if the platform is allowed to become a member. Once being a member, the TPM and host together can *sign* messages with respect to basenames bsn. If a platform signs with $bsn = \perp$ or a fresh basename, the signature must be anonymous and unlinkable to previous signatures. That is, any verifier can check that the signature stems from a legitimate platform via a deterministic *verify* algorithm, but the signature does not leak any information about the identity of the signer. Only when the platform signs repeatedly with the same basename $bsn \neq \perp$, it will be clear that the resulting signatures were created by the same platform, which can be publicly tested via a (deterministic) *link* algorithm.

One requires the typical completeness properties for signatures created by honest parties:

Completeness: When an honest platform successfully creates a signature on a message m w.r.t. a basename bsn, an honest verifier will accept the signature.

Correctness of Link: When an honest platform successfully creates two signatures, σ_1 and σ_2, w.r.t. the same basename $bsn \neq \perp$, an honest verifier running a link algorithm on σ_1 and σ_2 will output 1. To an honest verifier, it also does not matter in which order two signatures are supplied when testing linkability between the two signatures.

The more difficult part is to define the security properties that a

DAA scheme should provide in the presence of malicious parties. These properties can be informally described as follows:

Unforgeability-1: When the issuer and all TPMs are honest, no adversary can create a signature on a message m w.r.t. basename bsn when no platform signed m w.r.t. bsn.

Unforgeability-2: When the issuer is honest, an adversary can only sign in the name of corrupt TPMs. More precisely, if n TPMs are corrupt, the adversary can at most create n unlinkable signatures for the same basename $bsn \neq \bot$.

Anonymity: The standard anonymity property requires an adversary that is given two signatures, w.r.t. two different basenames or $bsn = \bot$, cannot distinguish whether both signatures were created by one honest platform, or whether two different honest platforms created the signatures. In this chapter, we will investigate a stronger notion of anonymity, requiring the anonymity to hold whenever the host is honest, even if the platform's TPM is corrupt.

Non-frameability: No adversary can create signatures on a message m w.r.t. basename bsn that links to a signature created by an honest platform, when this honest platform never signed m w.r.t. bsn. We require this property to hold even when the issuer is corrupt.

5.2.1 Simulation-Based Security Definitions

A simulation-based security definition defines an ideal functionality, which can be seen as a central trusted party that receives inputs from all parties and provides outputs to them. Roughly, a protocol is called secure if its behavior is indistinguishable from the functionality.

The Brickell, Camenisch, Chen definition [BCC04]. DAA was first introduced by Brickell, Camenisch, and Chen [BCC04] along with a simulation-based security definition. The functionality has a single procedure encompassing both signature generation and verification, meaning that a signature is generated for a specific verifier and will immediately be verified by that verifier. As the signature is never output to the verifier, he only learns that a message was correctly signed, but can neither forward signatures or verify them again. Clearly this limits the scenarios in which DAA can be applied.

Furthermore, linkability of signatures with the same basename was not defined explicitly in the security definition. In the instantiation it is handled by attaching pseudonyms to signatures, and when two signatures have the same pseudonym, they must have been created by the same platform.

The Chen, Morissey, Smart definitions [CMS08a,CMS09]. An extension to the security definition by Brickell et al. was later proposed by Chen, Morissey, and Smart [CMS08a]. It aims at providing linkability as an explicit feature in the functionality. To this end, the functionality is extended with a link interface that takes as input two signatures and determines whether they link. However, as discussed before, the sign and verify interfaces are interactive and thus signatures are never sent as output to parties, so it is not possible to provide them as input either. This was realized by the authors who later proposed a new simulation-based security definition [CMS09] that now separates the generation of signatures from their verification by outputting signatures. Unfortunately, the functionality models the signature generation in a too simplistic way: signatures are simply random values, even when the TPM is corrupt. Furthermore, the verify interface refuses all requests when the issuer is corrupt. Clearly, both these behaviours are not realizable by any protocol.

5.2.2 Property-Based Security Definitions

Given the difficulties in properly defining ideal functionalities, there is also a line of work that captures DAA features via property-based definitions. Such definitions capture every security property in a separate security game.

The Brickell, Chen, and Li security definition [BCL09]. The first property-based security definition is presented by Brickell, Chen, and Li [BCL09]. They define security games for anonymity, and “user-controlled traceability”. The latter aims to capture our unforgeability-1 and unforgeability-2 requirements. Unfortunately, this definition has several major shortcomings that were already discussed by Bernhard et al. [BFG$^+$13b].

The first problem is that the game for unforgeability-1 considers insecure schemes to be secure. The adversary in the unforgeability-1 game has oracle access to the honest parties from whom he can request signatures on messages and basenames of his choice. The adversary then wins if he can come up with a valid signature that is not a previous oracle response. This last requirement allows trivially insecure schemes to win the security game: assume a DAA scheme that outputs the hash of the TPM's secret key gsk as signature, i.e., the signature is independent of the message. Clearly, this should be an insecure scheme as the adversary, after having seen one signature can provide valid signatures on arbitrary messages of his choice. However, this scheme is secure according to the unforgeability-1 game, as there reused signatures are not considered a forgery.

Another issue is that the game for unforgeability-2 is not well defined. The goal of the adversary is to supply a signature σ, a message m, a basename $bsn \neq \perp$, and a signer's identity ID. The adversary wins if another signature "associated with the same ID" exists, but the signatures do not link. Firstly, there is no check on the validity of the supplied signature, which makes winning trivial for the adversary. Secondly, "another signature associated with the same ID" is not precisely defined, but we assume it to mean that the signature was the result of a signing query with that ID. However, then the adversary is limited to tamper with at most one of the signatures, whereas the second one is enforced to be honestly generated and unmodified. Thirdly, there is no check on the relation between the signature and the supplied ID. We expect that the intended behavior is that the supplied signature uses the key of ID, but there is no way to enforce this. Now an adversary can simply make a signing query with (m, bsn, ID_1), thus obtaining σ, and win the game with (σ, m, bsn, ID_2).

The definition further lacks a security game that captures the non-frameability requirement. This means a scheme with a link algorithm that always outputs 1 can be proven secure. Chen [Che09] extends the definition to add non-frameability, but this extension inherits all the aforementioned problems from [BCL09].

The Bernhard et al. security definition [BFG$^+$13b]. Realizing that the previous security definitions are not sufficient, Bernhard et al. [BFG$^+$13b] provide an extensive set of property-based security definitions covering all expected security requirements.

The main improvement is the way signatures are identified. An identify algorithm is introduced that takes a signature and a TPM key, and outputs whether the key was used to create the signature, which is possible as signatures are uniquely identifiable if the secret key is known. In all their game definitions, the keys of honest TPMs are known, allowing the challenger to identify which key was used to create the signature, solving the problems related to the imprecisely defined ID in the Brickell, Chen, and Li definition.

However, the security games make a simplifying assumption, namely that the platform, consisting of a host and a TPM, is considered as *one* party. This approach, termed "pre-DAA", suffices for anonymity and non-frameability, as there both the TPM and host have to be honest. However, for the unforgeability requirements it is crucial that the host does *not* have to be trusted. In fact, distributing the computational work between the TPM and the host, such that the load on the TPM is as small as possible and, at the same time, not requiring the host to be honest, is the main challenge in designing a DAA scheme. Therefore, a DAA security definition must be able to formally analyze this setting of an honest TPM working with a corrupt host.

The importance of such full DAA security is also acknowledged by Bernhard et al. [BFG$^+$13b]. After formally proving a proposed scheme secure in the pre-DAA setting, the authors bring the scheme to the full DAA setting where the TPM and host are considered as separate parties. To obtain full DAA security, the host randomizes the issuer's credential on the TPM's public key. Bernhard et al. then argue that this has no impact on the proven pre-DAA security guarantees as the host does not perform any action involving the TPM secret key. While this seems intuitively correct, it gives no guarantees whether the security properties are *provably* preserved in the full DAA setting. Indeed, the proof of unforgeability of the pre-DAA scheme, which is proven under the DL assumption, does not hold in the full DAA setting as a corrupt host could notice the simulation used in the security proof. More precisely, in the Bernhard et al. scheme, the host sends values (b, d) to the TPM which are the re-randomized part of the issued credential and are supposed to have the form $b^{gsk} = d$ with gsk being the TPM's secret key. The TPM then provides a signature proof of knowledge (SPK) of gsk to the host. The pre-DAA proof relies on the DL assumption and places the unknown discrete logarithm of the challenge DL instance as the TPM key gsk. In the pre-DAA setting, the TPM then simulates the proof of knowledge of gsk for any input

(b, d). This, however, is no longer possible in the full DAA setting. If the host is corrupt, he can send arbitrary values (b, d) with $b^{gsk} \neq d$ to the TPM. The TPM must only respond with a SPK if (b, d) are properly set, but relying only on the DL assumption does not allow the TPM to perform this check. Thus, the unforgeability can no longer be proven under the DL assumption. Note that the scheme could still be proven secure using the stronger static DH assumption, but the point is that a proof of pre-DAA security and a seemingly convincing but informal argument to transfer the scheme to the full DAA setting does not guarantee security in the full DAA setting.

Another peculiarity of the Bernhard et al. definition is that it makes some rather strong yet somewhat hidden assumptions on the adversary's behavior. For instance, in the traceability game showing unforgeability of the credentials, the adversary must not only output the claimed forgery but also the secret keys of all TPMs. For a DAA protocol this implicitly assumes that the TPM secret key can be extracted from every signature. Similarly, in games such as non-frameability or anonymity that capture security against a corrupt issuer, the issuer's key is generated honestly within the game, instead of being chosen by the adversary. For any realization this assumes either a trusted setup setting or an extractable proof of correctness of the issuer's secret key.

In the scheme proposed by Bernhard et al. [BFG+13b], none of these implicit assumptions hold though: the generation of the issuer key is not extractable or assumed to be trusted, and the TPM's secret key cannot be extracted from every signature, as the rewinding for this would require exponential time. Note that these assumptions are indeed necessary to guarantee security for the proposed scheme. If the non-frameability game would allow the issuer to choose its own key, it could choose $y = 0$ and win the game. Ideally, a security definition should not impose such assumptions or protocol details. If such assumptions are necessary though, then they should be made explicit to avoid pitfalls in the protocol design.

5.3 A Security Model for DAA with Optimal Privacy

This section presents our security definition for direct anonymous attestation with optimal privacy. First, we introduce our formal DAA

1. `Issuer Setup.` On input $(\mathsf{SETUP}, \mathsf{sid})$ from issuer $\mathcal{I}$.
   - Verify that $\mathsf{sid} = (\mathcal{I}, sid')$.
   - Output $(\mathsf{SETUP}, \mathsf{sid})$ to $\mathcal{A}$ and wait for input $(\mathsf{ALG}, \mathsf{sid}, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ from $\mathcal{A}$.
   - Check that ver, link and $\mathsf{identify}$ are deterministic, and check that sig, ver, link, $\mathsf{identify}$, ukgen interact only with random oracle $\mathcal{G}_{\mathsf{sRO}}$.
   - Store $(\mathsf{sid}, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ and output $(\mathsf{SETUPDONE}, \mathsf{sid})$ to $\mathcal{I}$.

2. `Join Request.` On input $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
   - Create a join session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status \leftarrow request$.
   - Output $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{H}_j)$ to $\mathcal{M}_i$.
3. $\mathcal{M}$ `Join Proceed.` On input $(\mathsf{JOIN}, \mathsf{sid}, jsid)$ from TPM $\mathcal{M}_i$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = request$ to *delivered*.
   - Output $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{A}$, wait for input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ from $\mathcal{A}$.
   - Output $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i)$ to $\mathcal{I}$.
4. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ from $\mathcal{I}$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = delivered$ to *complete*.
   - Output $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid, \tau)$ from $\mathcal{A}$.
   - Abort if $\mathcal{I}$ or $\mathcal{M}_i$ is honest and a record $\langle \mathcal{M}_i, *, *\rangle \in$ `Members` already exists.
   - If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \bot$.
   - Else, verify that the provided tracing trapdoor τ is eligible by checking $\mathsf{CheckTtdCorrupt}(\tau) = 1$.
   - Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau\rangle$ into `Members` and output $(\mathsf{JOINED}, \mathsf{sid}, jsid)$ to $\mathcal{H}_j$.

Figure 5.1: The Setup and Join interfaces of our ideal functionality $\mathcal{F}_{\mathsf{pdaa}}$ for DAA with optimal privacy.

model as ideal functionality $\mathcal{F}_{\mathsf{pdaa}}$, which slightly deviates from the informal properties defined in Section 5.2, by ommitting the $bsn = \bot$ option. This simplifies the security notion, and by choosing fresh bsn values, platforms are still completely unlinkable. Second, we elaborate on the inherent limitations the UC framework imposes on privacy in the presence of fully corrupted parties and introduce the concept of *isolated corruptions*, which allow one to overcome this limitation yet capture the power of subverted TPMs.

5. `Sign Request.` On input $(\mathsf{SIGN}, \mathsf{sid}, ssid, \mathcal{M}_i, m, bsn)$ from $\mathcal{H}_j$.
   - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, * \rangle$ exists in `Members`, abort.
   - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status \rangle$ with $status \leftarrow request$.
   - Output $(\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid, m, bsn)$ to $\mathcal{M}_i$.
6. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid)$ from $\mathcal{M}_i$.
   - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status \rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, * \rangle$ exists in `Members`.
   - Generate the signature for a fresh or established key:
     - Retrieve (gsk, τ) from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau \rangle \in \texttt{DomainKeys}$. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, check $\mathsf{CheckTtdHonest}(\tau) = 1$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau \rangle$ in `DomainKeys`.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn)$, check $\mathsf{ver}(\sigma, m, bsn) = 1$.
     - Check $\mathsf{identify}(\sigma, m, bsn, \tau) = 1$ and that there is no $(\mathcal{M}', \mathcal{H}') \neq (\mathcal{M}_i, \mathcal{H}_j)$ with tracing trapdoor τ' registered in `Members` or `DomainKeys` with $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j \rangle$ in `Signed` and output $(\mathsf{Signature}, \mathsf{sid}, ssid, \sigma)$ to $\mathcal{H}_j$.
7. `Verify.` On input $(\mathsf{VERIFY}, \mathsf{sid}, m, bsn, \sigma, \texttt{RL})$ from some party $\mathcal{V}$.
   - Retrieve all tuples $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, \tau_i \rangle \in \texttt{Members}$ and $\langle \mathcal{M}_i, \mathcal{H}_j, *, *, \tau_i \rangle \in \texttt{DomainKeys}$ where $\mathsf{identify}(\sigma, m, bsn, \tau_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:
     - More than one τ_i was found.
     - $\mathcal{I}$ is honest and no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found.
     - $\mathcal{M}_i$ or $\mathcal{H}_j$ is honest but no entry $\langle *, m, bsn, \mathcal{M}_i, \mathcal{H}_j \rangle \in \texttt{Signed}$ exists.
     - There is a $\tau' \in \texttt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$ and no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ for an honest $\mathcal{H}_j$ was found.
   - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn)$.
   - Add $\langle \sigma, m, bsn, \texttt{RL}, f \rangle$ to `VerResults` and output $(\mathsf{VERIFIED}, \mathsf{sid}, f)$ to $\mathcal{V}$.
8. `Link.` On input $(\mathsf{LINK}, \mathsf{sid}, \sigma, m, \sigma', m', bsn)$ from a party $\mathcal{V}$.
   - Output $\perp$ to $\mathcal{V}$ if at least one signature (σ, m, bsn) or (σ', m', bsn) is not valid (verified via the `verify` interface with $\texttt{RL} = \emptyset$).
   - For each τ_i in `Members` and `DomainKeys` compute $b_i \leftarrow \mathsf{identify}(\sigma, m, bsn, \tau_i)$ and $b_i' \leftarrow \mathsf{identify}(\sigma', m', bsn, \tau_i)$ and do the following:
     - Set $f \leftarrow 0$ if $b_i \neq b_i'$ for some i.
     - Set $f \leftarrow 1$ if $b_i = b_i' = 1$ for some i.
   - If f is not defined yet, set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
   - Output $(\mathsf{LINK}, \mathsf{sid}, f)$ to $\mathcal{V}$.

Figure 5.2: The Sign, Verify, and Link interfaces of our ideal functionality $\mathcal{F}_{\mathsf{pdaa}}$ for DAA with optimal privacy.

5.3.1 Ideal Functionality $\mathcal{F}_{\mathsf{pdaa}}$

We start by describing the interfaces and guaranteed security properties in an informal manner, and present the detailed definition of $\mathcal{F}_{\mathsf{pdaa}}$ in Figures 5.1 and 5.2.

Setup.

The SETUP interface on input $\mathsf{sid} = (\mathcal{I}, sid')$ initiates a new session for the issuer $\mathcal{I}$ and expects the adversary to provide algorithms (ukgen, $\mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}$) that will be used inside the functionality. ukgen creates a new key gsk and a tracing trapdoor τ that allows $\mathcal{F}_{\mathsf{pdaa}}$ to trace signatures generated with gsk. sig, ver, and link are used by $\mathcal{F}_{\mathsf{pdaa}}$ to create, verify, and link signatures, respectively. Finally, $\mathsf{identify}$ allows to verify whether a signature belongs to a certain tracing trapdoor. This allows $\mathcal{F}_{\mathsf{pdaa}}$ to perform multiple consistency checks and enforce the desired non-frameability and unforgeability properties.

Note that the ver and link algorithms assist the functionality only for signatures that are not generated by $\mathcal{F}_{\mathsf{pdaa}}$ itself. For signatures generated by the functionality, $\mathcal{F}_{\mathsf{pdaa}}$ will enforce correct verification and linkage using its internal records. While ukgen and sig are probabilistic algorithms, the other ones are required to be deterministic. The link algorithm also has to be symmetric, i.e., for all inputs it must hold that $\mathsf{link}(\sigma, m, \sigma', m', bsn) \leftrightarrow \mathsf{link}(\sigma', m', \sigma, m, bsn)$. To allow for instantiations based on global random oracles, $\mathcal{F}_{\mathsf{pdaa}}$ allows the algorithms to query the global random oracle $\mathcal{G}_{\mathsf{sRO}}$. As in the previous chapter, this random oracle needs to be global to allow $\mathcal{F}_{\mathsf{pdaa}}$ to interact with the global random oracle. $\mathcal{F}_{\mathsf{pdaa}}$ makes sure that the algorithms communicate with no other entity, as any communication with the adversary would break the anonymity guarantees.

Join.

A host $\mathcal{H}_j$ can request to join with a TPM $\mathcal{M}_i$ using the JOIN interface. If both the TPM and the issuer approve the join request, the functionality stores an internal membership record for $\mathcal{M}_i, \mathcal{H}_j$ in `Members` indicating that from now on that platform is allowed to create attestations.

If the host is corrupt, the adversary must provide $\mathcal{F}_{\mathsf{pdaa}}$ with a tracing trapdoor τ. This value is stored along in the membership record and allows the functionality to check via the $\mathsf{identify}$ function whether signatures were created by this platform. $\mathcal{F}_{\mathsf{pdaa}}$ uses these checks to ensure non-frameability and unforgeability whenever it creates or verifies signatures. To ensure that the adversary cannot provide bad trapdoors that would break the completeness or non-frameability properties, $\mathcal{F}_{\mathsf{pdaa}}$ checks the legitimacy of τ via the "macro" function $\mathsf{CheckTtdCorrupt}$. This function checks that for all previously generated or verified signatures for which $\mathcal{F}_{\mathsf{pdaa}}$ has already seen another matching tracing trapdoor $\tau' \neq \tau$, the new trapdoor τ is not also identified as a matching key. $\mathsf{CheckTtdCorrupt}$ is defined as follows:

$$\begin{aligned}\mathsf{CheckTtdCorrupt}(\tau) = \nexists(\sigma, m, bsn) : \Big(\\ \Big(\langle \sigma, m, bsn, *, * \rangle \in \texttt{Signed} \vee \langle \sigma, m, bsn, *, 1 \rangle \in \texttt{VerResults}\Big) \wedge \\ \exists \tau' : \Big(\tau \neq \tau' \wedge \big(\langle *, *, \tau' \rangle \in \texttt{Members} \vee \langle *, *, *, *, \tau' \rangle \in \texttt{DomainKeys}\big) \\ \wedge\, \mathsf{identify}(\sigma, m, bsn, \tau) = \mathsf{identify}(\sigma, m, bsn, \tau') = 1\Big)\Big)\end{aligned}$$

Sign.

After joining, a host $\mathcal{H}_j$ can request a signature on a message m with respect to basename bsn using the SIGN interface. The signature will only be created when the TPM $\mathcal{M}_i$ explicitly agrees to signing m w.r.t. bsn and a join record for $\mathcal{M}_i, \mathcal{H}_j$ in `Members` exists (if the issuer is honest).

When a platform wants to sign message m w.r.t. a fresh basename bsn, $\mathcal{F}_{\mathsf{pdaa}}$ generates a new key gsk (and tracing trapdoor τ) via ukgen and then signs m with that key. The functionality also stores the fresh key (gsk, τ) together with bsn in `DomainKeys`, and reuses the same key when the platform wishes to sign repeatedly under the same basename. Using fresh keys for every signature naturally enforces the desired privacy guarantees: the signature algorithm does not receive any identifying information as input, and thus the created signatures are guaranteed to be anonymous (or pseudonymous in case bsn is reused).

Our functionality enforces this privacy property whenever the host is honest. Note, however, that $\mathcal{F}_{\mathsf{pdaa}}$ does not behave differently when the host is corrupt, as in this case its output does not matter due to the way corruptions are handled in UC. That is, $\mathcal{F}_{\mathsf{pdaa}}$ always outputs anonymous signatures to the host, but if the host is corrupt, the signature is given to the adversary, who can choose to discard it and output anything else instead.

To guarantee non-frameability and completeness, our functionality further checks that every freshly generated key, tracing trapdoor and signature does not falsely match with any existing signature or key. More precisely, $\mathcal{F}_{\mathsf{pdaa}}$ first uses the CheckTtdHonest macro to verify whether the new key does not match to any existing signature. CheckTtdHonest is defined as follows:

$$\begin{aligned}&\mathsf{CheckTtdHonest}(\tau) = \\ &\quad \forall \langle \sigma, m, bsn, \mathcal{M}, \mathcal{H} \rangle \in \mathtt{Signed} : \mathsf{identify}(\sigma, m, bsn, \tau) = 0 \quad \wedge \\ &\quad \forall \langle \sigma, m, bsn, *, 1 \rangle \in \mathtt{VerResults} : \mathsf{identify}(\sigma, m, bsn, \tau) = 0\end{aligned}$$

Likewise, before outputting σ, the functionality checks that no one else already has a key which would match this newly generated signature.

Finally, for ensuring unforgeability, the signed message, basename, and platform are stored in `Signed` which will be used when verifying signatures.

Verify.

Signatures can be verified by any party using the VERIFY interface. $\mathcal{F}_{\mathsf{pdaa}}$ uses its internal `Signed`, `Members`, and `DomainKeys` records to enforce unforgeability and non-frameability. It uses the tracing trapdoors τ stored in `Members` and `DomainKeys` to find out which platform created this signature. If no match is found and the issuer is honest, the signature is a forgery and rejected by $\mathcal{F}_{\mathsf{pdaa}}$. If the signature to be verified matches the tracing trapdoor of some platform with an honest TPM or host, but the signing records do not show that they signed this message w.r.t. the basename, $\mathcal{F}_{\mathsf{pdaa}}$ again considers this to be a forgery and rejects. If the records do not reveal any issues with the signature, $\mathcal{F}_{\mathsf{pdaa}}$ uses the ver algorithm to obtain the final result.

The verify interface also supports verifier-local revocation. The verifier can input a revocation list `RL` containing tracing trapdoors, and signatures matching any of those trapdoors are no longer accepted.

Link.

Using the LINK interface, any party can check whether two signatures (σ, σ') on messages (m, m') respectively, generated with the same basename *bsn* originate from the same platform or not. $\mathcal{F}_{\mathsf{pdaa}}$ again uses the tracing trapdoors τ stored in `Members` and `DomainKeys` to check which platforms created the two signatures. If they are the same, $\mathcal{F}_{\mathsf{pdaa}}$ outputs that they are linked. If it finds a platform that signed one, but not the other, it outputs that they are unlinked, which prevents framing of platforms with an honest host.

The full definition of $\mathcal{F}_{\mathsf{pdaa}}$ is given in Figures 5.1 and 5.2. Note that when $\mathcal{F}_{\mathsf{pdaa}}$ runs one of the algorithms sig, ver, identify, link, and ukgen, it does so without maintaining state. This means all user keys have the same distribution, signatures are equally distributed for the same input, and ver, identify, and link invocations only depend on the current input, not on previous inputs.

5.3.2 Modeling Subverted Parties in the UC Framework

As just discussed, our functionality $\mathcal{F}_{\mathsf{pdaa}}$ guarantees that signatures created with an honest host are unlinkable and do not leak any information about the signing platform, even if the TPM is corrupt. However, the adversary still learns the message and basename when the TPM is corrupt, due to the way UC models corruptions. We discuss how this standard corruption model inherently limits the achievable privacy level, and then present our approach of isolated corruptions which allow one to overcome this limitation yet capture the power of subverted TPMs. While we discuss the modeling of isolated corruptions in the context of our DAA functionality, we consider the general concept to be of independent interest as it is applicable to any other scenario where such subversion attacks can occur.

Conditional Privacy under Full TPM Corruption.

In the standard UC corruption model, the adversary gains full control over a corrupted party, i.e., it receives all inputs to that party and can choose its responses. For the case of a corrupt TPM this means that the adversary sees the message m and basename *bsn* whenever the honest host wants to create a signature. In fact, the adversary will learn which particular TPM $\mathcal{M}_i$ is asked to sign m w.r.t. *bsn*. Thus, even though the signature σ on m w.r.t. *bsn* is then created by $\mathcal{F}_{\mathsf{pdaa}}$ and does not leak any information about the identity of the signing platform, the adversary might still be able to recognize the platform's identity via the signed values. That is, if a message m or basename *bsn* is unique, i.e., only a single (and corrupt) TPM has ever signed m w.r.t. *bsn*, then, when later seeing a signature on m w.r.t. *bsn*, the adversary can derive which platform had created the signature.

A tempting idea for better privacy would be to change the functionality such that the TPM does not receive the message and basename when asked to approve an attestation via the SIGNPROCEED message. As a result, this information will not be passed to the adversary if the TPM is corrupt. However, that would completely undermine the purpose of the TPM that is supposed to serve as a trust anchor: verifiers accept a DAA attestation because they know a trusted TPM has approved them. Therefore, it is essential that the TPM sees and acknowledges the messages it signs.

Thus, in the presence of a fully corrupt TPM, the amount of privacy that can be achieved depends which messages and basenames are being signed – the more unique they are, the less privacy $\mathcal{F}_{\mathsf{pdaa}}$ guarantees.

Optimal Privacy under *Isolated* TPM Corruption.

The aforementioned leakage of all messages and basenames that are signed by a corrupt TPM is a result of the standard UC corruption model. Modeling corruption of TPMs like this gives the adversary much more power than in reality: even if a TPM is subverted and runs malicious algorithms, it is still embedded into a host who controls all communication with the outside world. Thus, the adversary cannot communicate directly with the TPM, but only via the (honest) host.

To model such subversions more accurately and study the privacy achievable in the presence of subverted TPMs, we define a relaxed level of corruption that we call *isolated corruption*. When the adversary corrupts a TPM in this manner, it can specify code for the TPM but cannot directly communicate with the TPM.

We formally define such isolated corruptions via the body-shell paradigm used to model standard UC corruptions [Can00]. Recall that the body of a party defines its behavior, whereas the shell models the communication with that party. Thus, for our isolated corruptions,

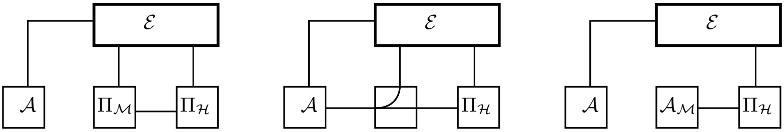


Figure 5.3: Modeling of corruption in the real world. Left: an honest TPM applies the protocol $\Pi_{\mathcal{M}}$, and communicates with the host running $\Pi_{\mathcal{H}}$. Middle: a corrupt TPM sends any input the adversary instructs it to, and forwards any messages received to the adversary. Right: an isolated corrupt TPM is controlled by an isolated adversary $\mathcal{A}_{\mathcal{M}}$, who can communicate with the host, but not with any other entities.

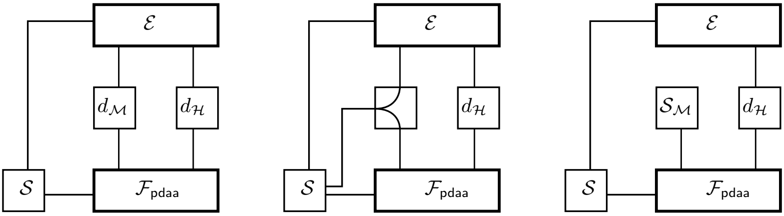


Figure 5.4: Modeling of corruption in the ideal world. Left: an honest TPM is a dummy party $d_{\mathcal{M}}$ that forwards inputs and outputs between the environment $\mathcal{E}$ and the functionality $\mathcal{F}_{\mathsf{pdaa}}$. Middle: a corrupt TPM sends any input the adversary instructs it to, and forwards any subroutine output to the adversary. Right: an isolated corrupt TPM is controlled by an isolated simulator $\mathcal{S}_{\mathcal{M}}$, who may send inputs and receive outputs from $\mathcal{F}_{\mathsf{pdaa}}$, but not communicate with any other entities.

the adversary gets control over the body but not the shell. Interestingly, this is exactly the inverse of honest-but-curious corruptions in UC, where the adversary controls the shell and thus sees all inputs and outputs, but cannot change the body, i.e., the parties behavior remains honest.

In our case, an adversary performing an isolated corruption can provide a body, which models the tampered algorithms that an isolated corrupt TPM may use. The shell remains honest though and handles inputs, and subroutine outputs, and only forwards the ones that are allowed to the body. In the real world, the shell would only allow communication with the host in which the TPM is embedded. In the ideal world, the shell allows inputs to and outputs from the functionality, and blocks anything else. For protocols making use of a global random oracle, the shell would also allow for communication with the global random oracle functionality in both the real and ideal world.

Figure 5.3 and Figure 5.4 depict the different levels of corruption in the real world and ideal world, respectively. In the ideal word, an isolated corruption of a TPM replaces the dummy TPM that forwards inputs and outputs between the environment and the ideal functionality with an *isolated simulator* comprising of the adversarial body and honest shell.

When designing a UC functionality, then all communication between a host and the "embedded" party that can get corrupted in such an isolated manner must be modeled as a direct channel (see e.g., the SIGN related interfaces in $\mathcal{F}_{\mathsf{pdaa}}$). Otherwise the simulator/adversary will be aware of the communication between both parties and can delay or block messages, which would contradict the concept of an isolated corruption where the adversary has no direct channel to the embedded party. Note that the perfect channel of course only holds if the host entity is honest, if it is corrupt (in the standard sense), the adversary can see and control all communication via the host anyway.

With such isolated adversaries we specify much stronger privacy. The adversary no longer automatically learns which isolated corrupt TPM signed which combination of messages and basenames, and the signatures created by $\mathcal{F}_{\mathsf{pdaa}}$ are guaranteed to be unlinkable. Of course the message m and basename bsn must not leak information about the identity of the platform. In certain applications, the platform would sign data generated or partially controlled by other functions contained in a TPM. This is out of scope of the attestation scheme, but the higher level scheme using $\mathcal{F}_{\mathsf{pdaa}}$ should ensure that this does not happen, by, e.g., letting the host randomize or sanitize the message.

5.4 Insufficiency of Existing DAA Schemes

Our functionality $\mathcal{F}_{\mathsf{pdaa}}$ requires all signatures on a message m with a fresh basename bsn to have the same distribution, even when the TPM is corrupt. None of the existing DAA schemes can be used to realize $\mathcal{F}_{\mathsf{pdaa}}$ when the TPM is corrupted (either fully or isolated). The reason is inherent to the common protocol design that underlies all DAA schemes so far, i.e., there is no simple patch that would allow upgrading the existing solutions to achieve optimal privacy.

In a nutshell, in all existing DAA schemes, the TPM chooses a secret key *gsk* for which it blindly receives a membership credential of a trusted issuer. To create a signature on message m with basename *bsn*, the platform creates a signature proof of knowledge signing message m and proving knowledge of *gsk* and the membership credential.

In the original RSA-based DAA scheme [BCC04], and the more recent qSDH-based schemes [CF08, BL11, BL10, CDL16a], the proof of knowledge of the membership credential is created jointly by the TPM and host. After jointly computing the commitment values of the proof, the host computes the hash over these values and sends the hash c to the TPM. To prevent leaking information about its key, the TPM must ensure that the challenge is a hash of fresh values. In all the aforementioned schemes this is done by letting the TPM choose a fresh nonce n and computing the final hash as $c' \leftarrow \mathsf{H}(n, c)$. An adversarial TPM can embed information in n instead of taking it uniformly at random, clearly altering the distribution of the proof and thus violating the desired privacy guarantees.

At a first glance, deriving the hash for the proof in a more robust manner might seem a viable solution to prevent such leakage. For instance, setting the nonce as $n \leftarrow n_t \oplus n_h$, with n_t being the TPM's and n_h the host's contribution, and letting the TPM commit to n_t before receiving n_h. While this indeed removes the leakage via the nonce, it still reveals the hash value $c' \leftarrow \mathsf{H}(n, c)$ to the TPM with the hash becoming part of the completed signature. Thus, the TPM can base its behavior on the hash value and, e.g., only sign messages for hashes that start with a 0-bit.

The same argument applies to the existing LRSW-based schemes [CPS10, BFG$^+$13b, CDL16b], where the proof of a membership credential is done solely by the TPM, and thus can leak information via the Fiat-Shamir hash output again. The general problem is that the signature proofs of knowledge are not randomizable. If the TPM would create a randomizable proof of knowledge, e.g., a Groth-Sahai proof [GS08], the host could randomize the proof to remove any hidden information, but this would yield a highly inefficient signing protocol for the TPM.

5.5 Building Blocks

In this section we introduce the building blocks for our DAA scheme. In addition to standard components such as additively homomorphic encryption, we introduce two non-standard types of signature schemes. One signature scheme we require is for the issuer to blindly sign the public key of the TPM and host. The second signature scheme is needed for the TPM and host to jointly create signed attestations, which we term *split signatures*.

The approach of constructing a DAA scheme from modular building blocks rather than basing it on a concrete instantiation was also used by Bernhard et al. [BFG+13b, BFG13a]. As they considered a simplified setting, called pre-DAA, where the host and platform have a joint corruption state, and we aim for much stronger privacy, their "linkable indistinguishable tag" is not sufficient for our construction. We replace this with our split signatures.

As our protocol requires "compatible" building blocks, i.e., the different schemes have to work in the same group, we assume the availability of public system parameters $spar \overset{\$}{\leftarrow} \mathsf{SParGen}(1^\kappa)$ generated for security parameter κ. We give *spar* as dedicated input to the individual key generation algorithms instead of the security parameter κ. For the sake of simplicity, we omit the system parameters as dedicated input to all other algorithms and assume that they are given as implicit input.

5.5.1 Homomorphic Encryption Schemes

We require an encryption scheme $(\mathsf{EncKGen}, \mathsf{Enc}, \mathsf{Dec})$ that is semantically secure and that has a cyclic group $\mathbb{G} = \langle g \rangle$ of order q as message space. It consists of a key generation algorithm $(epk, esk) \overset{\$}{\leftarrow} \mathsf{EncKGen}(spar)$, where *spar* defines the group $\mathbb{G}$, an encryption algorithm $C \overset{\$}{\leftarrow} \mathsf{Enc}(epk, m)$, with $m \in \mathbb{G}$, and a decryption algorithm $m \leftarrow \mathsf{Dec}(esk, C)$.

We further require that the encryption scheme has an appropriate *homomorphic property*, namely that there is an efficient operation $\odot$ on ciphertexts such that, if $C_1 \in \mathsf{Enc}(epk, m_1)$ and $C_2 \in \mathsf{Enc}(epk, m_2)$, then $C_1 \odot C_2 \in \mathsf{Enc}(epk, m_1 \cdot m_2)$. We will also use exponents to denote the repeated application of $\odot$, e.g., C^2 to denote $C \odot C$.

ElGamal Encryption. We use ElGamal encryption [ElG86], which is homomorphic and chosen plaintext secure. The semantic security is sufficient for our construction, as the parties always prove to each other that they formed the ciphertexts correctly. Let *spar* define a group $\mathbb{G} = \langle g \rangle$ of order q such that the DDH problem is hard.

$\mathsf{EncKGen}(spar)$: Pick $x \xleftarrow{\$} \mathbb{Z}_q$, compute $y \leftarrow g^x$, and output $esk \leftarrow x$, $epk \leftarrow y$.

$\mathsf{Enc}(epk, m)$: To encrypt a message $m \in \mathbb{G}$ under $epk = y$, pick $r \xleftarrow{\$} \mathbb{Z}_q$ and output the ciphertext $(C_1, C_2) \leftarrow (y^r, g^r m)$.

$\mathsf{Dec}(esk, C)$: On input the secret key $esk = x$ and a ciphertext $C = (C_1, C_2) \in \mathbb{G}^2$, output $m' \leftarrow C_2 \cdot C_1^{-1/x}$.

5.5.2 Signature Schemes for Encrypted Messages

We need a signature scheme that supports the signing of encrypted messages and must allow for (efficient) proofs proving that an encrypted value is correctly signed and proving knowledge of a signature that signs an encrypted value. Dual-mode signatures [CL15] satisfy these properties, as therein signatures on plaintext as well as on encrypted messages can be obtained. As we do not require signatures on plaintexts, though, we can use a simplified version.

A signature scheme for encrypted messages consists of the algorithms $(\mathsf{SigKGen}, \mathsf{EncSign}, \mathsf{DecSign}, \mathsf{Vf})$ and uses an encryption scheme $(\mathsf{EncKGen}, \mathsf{Enc}, \mathsf{Dec})$ that is compatible with the message space of the signature scheme. In particular, the algorithms working with encrypted messages or signatures also get the keys $(epk, esk) \xleftarrow{\$} \mathsf{EncKGen}(spar)$ of the encryption scheme as input.

$\mathsf{SigKGen}(spar)$: On input the system parameters, this algorithm outputs a public verification key spk and secret signing key ssk.

$\mathsf{EncSign}(ssk, epk, C)$: On input signing key ssk, a public encryption key epk, and ciphertext $C = \mathsf{Enc}(epk, m)$, outputs an "encrypted" signature $\overline{\sigma}$ of C.

$\mathsf{DecSign}(esk, spk, \overline{\sigma})$: On input an "encrypted" signature $\overline{\sigma}$, secret decryption key esk and public verification key spk, outputs a standard signature σ.

$\mathsf{Vf}(spk, \sigma, m)$: On input a public verification key spk, signature σ and message m, outputs 1 if the signature is valid and 0 otherwise.

For correctness, we require that any message encrypted with honestly generated keys that is honestly signed decrypts to a valid signature. More precisely, for any message m, we require

```
Experiment $\mathsf{Exp}^{\mathsf{ESIG\text{-}forge}}_{\mathcal{A},\mathsf{ESIG},\mathsf{Enc}}(\mathbb{G},\tau)$:
  $spar \leftarrow \mathsf{SParGen}(\tau)$
  $(spk, ssk) \xleftarrow{\$} \mathsf{SigKGen}(spar)$
  $\mathbf{L} \leftarrow \varnothing$
  $(m^*, \sigma^*) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{EncSign}(ssk,\cdot,\cdot)}}(spar, spk)$
      where $\mathcal{O}^{\mathsf{EncSign}}$ on input $(epk_i, m_i)$:
          add $m_i$ to the list of queried messages $\mathbf{L} \leftarrow \mathbf{L} \cup m_i$
          compute $C_i \xleftarrow{\$} \mathsf{Enc}(epk_i, m_i)$
          return $\overline{\sigma} \xleftarrow{\$} \mathsf{EncSign}(ssk, epk_i, C_i)$
  return 1 if $\mathsf{Vf}(spk, \sigma^*, m^*) = 1$ and $m^* \notin \mathbf{L}$
```

Figure 5.5: Unforgeability experiment for signatures on encrypted messages.

$$\mathsf{Pr}\Big[\mathsf{Vf}(spk, \sigma, m) = 1 \;\;|\;\; spar \leftarrow \mathsf{SParGen}(\tau), \\ (spk, ssk) \xleftarrow{\$} \mathsf{SigKGen}(spar), (epk, esk) \leftarrow \mathsf{EncKGen}(spar), \\ C \leftarrow \mathsf{Enc}(epk, m), \bar{\sigma} \leftarrow \mathsf{EncSign}(ssk, epk, c), \\ \sigma \leftarrow \mathsf{DecSign}(esk, spk, \bar{\sigma})\Big].$$

We use the unforgeability definition of [CL15], but omit the oracle for signatures on plaintext messages. Note that the oracle $\mathcal{O}^{\mathsf{EncSign}}$ will only sign correctly computed ciphertexts, which is modeled by providing the message and public encryption key as input and let the oracle encrypt the message itself before signing it. When using the scheme, this can easily be enforced by asking the signature requester for a proof of correct ciphertext computation, and, indeed, in our construction such a proof is needed for other reasons as well.

Definition 14. (UNFORGEABILITY OF SIGNATURES FOR ENCRYPTED MESSAGES). *We say a signature scheme for encrypted messages is* unforgeable *if for any efficient algorithm $\mathcal{A}$ the probability that the experiment given in Figure 5.5 returns 1 is negligible (as a function of τ).*

AGOT+ Signature Scheme. To instantiate the building block of signatures for encrypted messages we will use the AGOT+ scheme of [CL15], which was shown to be a secure instantiation of a dual-mode signature, hence is also secure in our simplified setting. Again, as we do not require signatures on plaintext messages we omit the standard signing algorithm. The AGOT+ scheme is based on the structure-preserving signature scheme by Abe et al. [AGOT14], which is proven to be unforgeable in the generic group model.

The AGOT+ scheme assumes the availability of system parameters $(q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathrm{t}, e, g_1, g_2, x)$, where $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathrm{t}$ are groups of prime order q generated by g_1, g_2, and $e(g_1, g_2)$ respectively, e is a non-degenerate bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_\mathrm{t}$, and x is an additional random group element in $\mathbb{G}_1$.

$\mathsf{SigKGen}(spar)$: Draw $v \xleftarrow{\$} \mathbb{Z}_q$, compute $y \leftarrow g_2^v$, and return $spk = y$, $ssk = v$.

$\mathsf{EncSign}(ssk, epk, M)$: On input a proper encryption $M = \mathsf{Enc}(epk, m)$ of a message $m \in \mathbb{G}_1$ under epk, and secret key $ssk = v$, choose a random $u \xleftarrow{\$} \mathbb{Z}_q^*$, and output the (partially) encrypted signature $\bar{\sigma} = (r, S, T, w)$, with $r \leftarrow g_2^u$, $S \leftarrow (M^v \odot \mathsf{Enc}(epk, x))^{1/u}$, $T \leftarrow (S^v \odot \mathsf{Enc}(epk, g_1))^{1/u}$, $w \leftarrow g_1^{1/u}$.

$\mathsf{DecSign}(esk, spk, \overline{\sigma})$: Parse $\overline{\sigma} = (r, S, T, w)$, compute $s \leftarrow \mathsf{Dec}(esk, S)$, $t \leftarrow \mathsf{Dec}(esk, T)$ and output $\sigma = (r, s, t, w)$.

$\mathsf{Vf}(spk, \sigma, m)$: Parse $\sigma = (r, s, t, w')$ and $spk = y$ and output 1 iff $m, s, t \in \mathbb{G}_1$, $r \in \mathbb{G}_2$, $e(s, r) = e(m, y) \cdot e(x, g_2)$, and $e(t, r) = e(s, y) \cdot e(g_1, g_2)$.

Note that for notational simplicity, we consider w part of the signature, i.e., $\sigma = (r, s, t, w)$, altough signature verification will ignore w. As pointed out by Abe et al., a signature $\sigma = (r, s, t)$ can be randomized using the randomization token w to obtain a signature $\sigma' = (r', s', t')$ by picking a random $u' \xleftarrow{\$} \mathbb{Z}_q^*$ and computing $r' \leftarrow r^{u'}, \quad s' \leftarrow s^{1/u'}, \quad t' \leftarrow (tw^{(u'-1)})^{1/u'^2}$.

For our construction, we also require the host to prove that it knows an encrypted signature on an encrypted message. In Section 5.7 we describe how such a proof can be done.

5.5.3 Split Signatures

The second signature scheme we require must allow two different parties, each holding a share of the secret key, to jointly create signatures. Our DAA protocol performs the joined public key generation and the signing operation in a strict sequential order. That is, the first party creates his part of the key, and the second party receiving the 'pre-public key' generates a second key share and completes the joined public key. Similarly, to sign a message the first signer creates a 'pre-signature' and the second signer completes the signature. We model the new signature scheme for that particular sequential setting rather than aiming for a more generic building block in the spirit of threshold or multi-signatures, as the existence of a strict two-party order allows for substantially more efficient constructions.

We term this new building block *split signatures* partially following the notation by Bellare and Sandhu [BS01] who formalized different two-party settings for RSA-based signatures where the signing key is split between a client and server. Therein, the case "MSC" where the first signature contribution is produced by an external server and then completed by the client comes closest to out setting.

Formally, we define a split signature scheme SSIG as a tuple of algorithms $(\mathsf{PreKeyGen}, \mathsf{CompleteKeyGen}, \mathsf{VerKey}, \mathsf{PreSign}, \mathsf{CompleteSign}, \mathsf{Vf})$:

$\mathsf{PreKeyGen}(spar)$: On input the system parameters, this algorithm outputs the pre-public key ppk and the first share of the secret signing key ssk_1.

$\mathsf{CompleteKeyGen}(ppk)$: On input the pre-public key, this algorithm outputs a public verification key spk and the second secret signing key ssk_2.

$\mathsf{VerKey}(ppk, spk, ssk_2)$: On input the pre-public key ppk, the full public key spk, and a secret key share ssk_2, this algorithm outputs 1 iff the pre-public key combined with secret key part ssk_2 leads to full public key spk.

$\mathsf{PreSign}(ssk_1, m)$: On input a secret signing key share ssk_1, and message m, this algorithm outputs a pre-signature σ'.

$\mathsf{CompleteSign}(ppk, ssk_2, m, \sigma')$: On input the pre-public key ppk, the second signing key share ssk_2, message m, and pre-signature σ', this algorithm outputs the completed signature σ.

$\mathsf{Vf}(spk, \sigma, m)$: On input the public key spk, signature σ, and message m, this algorithm outputs a bit b indicating whether the signature is valid or not.

A split signature scheme must satisfy correctness, meaning that honestly generated signatures will pass verification.

Definition 15. *A split signature scheme is correct if we have*

$$\mathsf{Pr}\Big[\mathsf{Vf}(spk, \sigma, m) = 1 \;\;|\;\; spar \leftarrow \mathsf{SParGen}(\kappa),$$
$$(ppk, spk_1) \xleftarrow{\$} \mathsf{PreKeyGen}(spar), (spk, ssk_2) \xleftarrow{\$} \mathsf{CompleteKeyGen}(ppk),$$
$$\sigma' \xleftarrow{\$} \mathsf{PreSign}(ssk_1, m), \sigma' \leftarrow \mathsf{CompleteSign}(ppk, ssk_2, m, \sigma')\Big].$$

We further require a number of security properties from our split signatures. The first one is unforgeability which must hold if at least one of the two signers is honest. This is captured in two security experiments: type-1 unforgeability allows the first signer to be corrupt, and type-2 unforgeability considers a corrupt second signer. Our definitions are similar to the ones by Bellare and Sandhu, with the difference that we do not assume a trusted dealer creating *both* secret key shares. Instead, we let the adversary output the key share of the party he controls. For type-2 unforgeability we must ensure, though, that the adversary indeed integrates the honestly generated pre-key *ppk* when producing the completed public key *spk*, which we verify via VerKey. Formally, unforgeability for split signatures is defined as follows.

Definition 16. (TYPE-1/2 UNFORGEABILITY OF SSIG). *A split signature scheme is* type-1/2 unforgeable *if for any efficient algorithm $\mathcal{A}$ the probability that the experiments given in Figure 5.6 return 1 is negligible (as a function of κ).*

Further, we need a property that we call *key-hiding*, which ensures that signatures do not leak any information about the public key for which they are generated. This is needed in the DAA scheme to get unlinkability even in the presence of a corrupt TPM that contributes to the signatures and knows part of the secret key, yet should not be able to recognize “his” signatures afterwards. Our key-hiding notion is somewhat similar in spirit to key-privacy for encryption schemes as defined by Bellare et al. [BBDP01], which requires that a ciphertext should not leak anything about the public key under which it is encrypted.

Formally, this is captured by giving the adversary a challenge signature for a chosen message either under the real or a random public key. Clearly, the property can only hold as long as the real public key *spk*

```
**Experiment** $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{Unforgeability\text{-}1}}(\kappa)$:
  $spar \xleftarrow{\$} \mathsf{SParGen}(1^\kappa)$
  $(ppk, state) \leftarrow \mathcal{A}(spar)$
  $(spk, ssk_2) \leftarrow \mathsf{CompleteKeyGen}(ppk)$
  $\mathbf{L} \leftarrow \emptyset$
  $(m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{CompleteSign}(ppk, ssk_2, \cdot, \cdot)}}(state, spk)$
    where $\mathcal{O}^{\mathsf{CompleteSign}}$ on input $(m_i, \sigma_i')$:
      set $\mathbf{L} \leftarrow \mathbf{L} \cup m_i$
      return $\sigma_i \leftarrow \mathsf{CompleteSign}(ppk, ssk_2, m_i, \sigma_i')$
  return 1 if $\mathsf{Vf}(spk, \sigma^*, m^*) = 1$ and $m^* \notin \mathbf{L}$
```

```
**Experiment** $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{Unforgeability\text{-}2}}(\kappa)$:
  $spar \xleftarrow{\$} \mathsf{SParGen}(1^\kappa)$
  $(ppk, ssk_1) \leftarrow \mathsf{PreKeyGen}(spar)$
  $\mathbf{L} \leftarrow \emptyset$
  $(m^*, \sigma^*, spk, ssk_2) \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{PreSign}(ssk_1, \cdot)}}(spar, ppk)$
    where $\mathcal{O}^{\mathsf{PreSign}}$ on input $m_i$:
      set $\mathbf{L} \leftarrow \mathbf{L} \cup m_i$
      return $\sigma_i' \leftarrow \mathsf{PreSign}(ssk_1, m_i)$
  return 1 if $\mathsf{Vf}(spk, \sigma^*, m^*) = 1$, and $m^* \notin \mathbf{L}$
      and $\mathsf{VerKey}(ppk, spk, ssk_2) = 1$
```

Figure 5.6: Unforgeability-1 (1st signer is corrupt) and unforgeability-2 (2nd signer is corrupt) experiments.

is not known to the adversary, as otherwise he can simply verify the challenge signature. As we want the property to hold even when the first party is corrupt, the adversary can choose the first part of the secret key and also contribute to the challenge signature. The adversary is also given oracle access to $\mathcal{O}^{\mathsf{CompleteSign}}$ again, but is not allowed to query the message used in the challenge query, as he could win trivially otherwise (by the requirement of signature-uniqueness defined below and the determinism of CompleteSign). The formal experiment for our key-hiding property is given below. The oracle $\mathcal{O}^{\mathsf{CompleteSign}}$ is defined analogously as in type-1 unforgeability.

Definition 17. (KEY-HIDING PROPERTY OF SSIG). *We say a split signature scheme is* key-hiding *if for any efficient algorithm $\mathcal{A}$ the probability that the experiment given in Figure 5.7 returns 1 is negligible*

```
Experiment $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{Key\text{-}Hiding}}(\kappa)$:
  $spar \xleftarrow{\$} \mathsf{SParGen}(1^{\kappa})$
  $(ppk, state) \xleftarrow{\$} \mathcal{A}(spar)$
  $(spk, ssk_2) \xleftarrow{\$} \mathsf{CompleteKeyGen}(ppk)$
  $\mathbf{L} \leftarrow \varnothing$
  $(m, \sigma', state') \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{CompleteSign}(ppk, ssk_2, \cdot, \cdot)}}(state)$
  $b \xleftarrow{\$} \{0, 1\}$
  if $b = 0$ (signature under spk):
      $\sigma \leftarrow \mathsf{CompleteSign}(ppk, ssk_2, m, \sigma')$
  if $b = 1$ (signature under random key):
      $(ppk^*, ssk_1^*) \xleftarrow{\$} \mathsf{PreKeyGen}(spar)$
      $(spk^*, ssk_2^*) \xleftarrow{\$} \mathsf{CompleteKeyGen}(ppk^*)$
      $\sigma' \xleftarrow{\$} \mathsf{PreSign}(ssk_1^*, m)$
      $\sigma \leftarrow \mathsf{CompleteSign}(ppk^*, ssk_2^*, m, \sigma')$
  $b' \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{CompleteSign}(ppk, ssk_2, \cdot, \cdot)}}(state', \sigma)$
  return 1 if $b = b'$, $m \notin \mathbf{L}$, and $\mathsf{Vf}(spk, \sigma, m) = 1$
```

Figure 5.7: Key-hiding experiment for split signatures.

```
Experiment $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{Key\text{-}Uniqueness}}(\kappa)$:
  $spar \xleftarrow{\$} \mathsf{SParGen}(1^{\kappa})$
  $(\sigma, spk_0, spk_1, m) \xleftarrow{\$} \mathcal{A}(spar)$
  return 1 if $spk_0 \neq spk_1$, $\mathsf{Vf}(spk_0, \sigma, m) = 1$, and $\mathsf{Vf}(spk_1, \sigma, m) = 1$
```

Figure 5.8: Key-uniqueness experiment for split signatures.

(as a function of κ).

We also require two uniqueness properties for our split signatures. The first is *key-uniqueness*, which states that every signature is only valid under one public key.

Definition 18. (KEY-UNIQUENESS OF SPLIT SIGNATURES). *We say a split signature scheme has* key-uniqueness *if for any efficient algorithm $\mathcal{A}$ the probability that the experiment given in Figure 5.8 returns 1 is negligible (as a function of κ).*

The second uniqueness property required is *signature-uniqueness*, which guarantees that one can compute only a single valid signature on a certain message under a certain public key.

Experiment $\mathsf{Exp}_{\mathcal{A}}^{\mathsf{Signature\text{-}Uniqueness}}(\kappa)$:
$spar \xleftarrow{\$} \mathsf{SParGen}(1^{\kappa})$
$(\sigma_0, \sigma_1, spk, m) \xleftarrow{\$} \mathcal{A}(spar)$
return 1 if $\sigma_0 \neq \sigma_1$, $\mathsf{Vf}(spk, \sigma_0, m) = 1$, and $\mathsf{Vf}(spk, \sigma_1, m) = 1$

Figure 5.9: Signature-uniqueness experiment for split signatures.

Definition 19. (SIGNATURE-UNIQUENESS OF SPLIT SIGNATURES). *We say a split signature scheme has* signature uniqueness *if for any efficient algorithm $\mathcal{A}$ the probability that the experiment given in Figure 5.9 returns 1 is negligible (as a function of κ).*

Instantiation of split signatures (split-BLS). To instantiate split signatures, we use a modified BLS signature [BLS04]. Let H be a hash function $\{0,1\} \rightarrow \mathbb{G}_1^*$ and the public system parameters be the description of a bilinear map, i.e., $spar = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, g_1, g_2, e, q)$.

$\mathsf{PreKeyGen}(spar)$: Take $ssk_1 \xleftarrow{\$} \mathbb{Z}_q^*$, set $ppk \leftarrow g_2^{ssk_1}$, and output (ppk, ssk_1).

$\mathsf{CompleteKeyGen}(spar, ppk)$: Check $ppk \in \mathbb{G}_2$ and $ppk \neq 1_{\mathbb{G}_2}$. Take $ssk_2 \xleftarrow{\$} \mathbb{Z}_q^*$ and compute $spk \leftarrow ppk^{ssk_2}$. Output (spk, ssk_2).

$\mathsf{VerKey}(spar, ppk, spk, ssk_2)$: Output 1 iff $ppk \neq 1_{\mathbb{G}_2}$ and $spk = ppk^{ssk_2}$.

$\mathsf{PreSign}(spar, ssk_1, m)$: Output $\sigma' \leftarrow \mathsf{H}(m)^{ssk_1}$.

$\mathsf{CompleteSign}(spar, ppk, ssk_2, m, \sigma')$: If $e(\sigma', g_2) = e(\mathsf{H}(m), ppk)$, output $\sigma \leftarrow \sigma'^{ssk_2}$, otherwise $\perp$.

$\mathsf{Vf}(spar, spk, \sigma, m)$: Output 1 iff $\sigma \neq 1_{\mathbb{G}_1}$ and $e(\sigma, g_2) = e(\mathsf{H}(m), spk)$.

Theorem 9. *Split-BLS is a secure multisignature scheme under the co-CDH and XDH assumptions in the random-oracle model.*

This theorem is a result of the following lemmas.

Lemma 2. *Split-BLS is correct, as defined in Definition 15.*

Proof. From running $\mathsf{PreKeyGen}$, we get $ssk_1 \xleftarrow{\$} \mathbb{Z}_q^*$ and $ppk \leftarrow g_2^{ssk_1}$. $\mathsf{CompleteKeyGen}$ will check that $ppk \neq 1_{\mathbb{G}_2}$, which holds as ssk_1 is taken from $\mathbb{Z}_q^*$. It then takes $ssk_2 \xleftarrow{\$} \mathbb{Z}_q^*$ and $spk \leftarrow ppk^{ssk_2}$.

When signing, $\mathsf{PreSign}$ sets $\sigma' \leftarrow \mathsf{H}(m)^{ssk_1}$. $\mathsf{CompleteSign}$ checks $e(\sigma', g_2) \overset{?}{=} e(\mathsf{H}(m), ppk)$ which holds for this σ', and computes $\sigma \leftarrow \sigma'^{ssk_2}$.

Verification checks $e(\sigma, g_2) = e(\mathsf{H}(m), spk)$, which holds as $\sigma = \mathsf{H}(m)^{ssk_1 \cdot ssk_2}$ and $spk = g_2^{ssk_1 \cdot ssk_2}$. Since both ssk_1 and ssk_2 are taken from $\mathbb{Z}_q^*$, they are both unequal to 0 and $ssk_1 \cdot ssk_2 \neq 0$. As H maps to $\mathbb{G}_1^*$, this means $\sigma \neq 1_{\mathbb{G}_1}$. □

Lemma 3. *The split-BLS signature scheme satisfies unforgeability-1, as defined in Definition 16, under the co-CDH assumption, in the random-oracle model.*

Proof. Assume that adversary $\mathcal{A}$ breaks unforgeability-1 with non-negligible probability, then we construct reduction $\mathcal{B}$ that breaks co-CDH with non-negligible probability. $\mathcal{B}$ takes as input the groups and g_1^α, g_1^β, g_2^β, and must compute $g_1^{\alpha\cdot\beta}$. If $g_1^\alpha = 1_{\mathbb{G}_1}$ or $g_1^\beta = 1_{\mathbb{G}_1}$, $\mathcal{B}$ outputs $1_{\mathbb{G}_1}$ to solve the co-CDH problem directly. Otherwise, $\mathcal{B}$ initializes $\mathcal{A}$ on the parameters and receives the pre-key ppk from $\mathcal{A}$. For some unknown ssk_1, $ppk = g_2^{ssk_1}$. $\mathcal{B}$ simulates the (unknown) second key $ssk_2 = \beta / ssk_1$ by setting $spk \leftarrow g_2^\beta = ppk^{ssk_2}$. Random oracle queries are answered with g_1^r for $r \xleftarrow{\$} \mathbb{Z}_q^*$, while maintaining consistency, except for a random query $\bar{m}$, where it returns g_1^α. When $\mathcal{A}$ makes a $\mathsf{CompleteSign}$ query on a message $m \neq \bar{m}$ and pre-signature σ', first check $e(\sigma', g_2) \overset{?}{=} e(\mathsf{H}(m), ppk)$, and return $\perp$ if this does not hold. Otherwise, return $\sigma \leftarrow \mathsf{H}(m)^\beta = (g_1^\beta)^r$, where the reduction knows r from simulating the random oracle.

When $\mathcal{A}$ outputs forgery (m^*, σ^*). With non-negligible probability, $m^* = \bar{m}$, and we have $e(\sigma^*, g_2) = e(\mathsf{H}(m), spk) = e(g_1^\alpha, g_2^\beta)$, showing that σ^* solves the co-CDH instance. □

Lemma 4. *The split-BLS signature scheme satisfies unforgeability-2, as defined in Definition 16, under the co-CDH assumption, in the random-oracle model.*

Proof. Assume that adversary $\mathcal{A}$ breaks unforgeability-2 with non-negligible probability, then we construct reduction $\mathcal{B}$ that breaks co-CDH with non-negligible probability. $\mathcal{B}$ takes as input the groups and g_1^α,

g_1^β, g_2^β, and must compute $g_1^{\alpha\cdot\beta}$. If $g_1^\alpha = 1_{\mathbb{G}_1}$ or $g_1^\beta = 1_{\mathbb{G}_1}$, $\mathcal{B}$ outputs $1_{\mathbb{G}_1}$ to solve the co-CDH problem directly. $\mathcal{B}$ runs $\mathcal{A}$ on input the parameters and $ppk = g_2^\beta$. When $\mathcal{A}$ makes random oracle queries, $\mathcal{B}$ answers them with g_1^r with $r \xleftarrow{\$} \mathbb{Z}_q^*$, while maintaining consistency, except for a random query $\bar{m}$, where it returns g_1^α. When $\mathcal{A}$ makes a $\mathsf{PreSign}$ query on m, $\mathcal{B}$ looks up r such that $\mathsf{H}(m) = g_1^r$ from simulating the random oracle and output signature $\sigma \leftarrow (g_1^\beta)^r$. If $\mathcal{A}$ makes a query with $m = \bar{m}$, the reduction fails.

When $\mathcal{A}$ outputs $(m^*, \sigma^*, spk, ssk_2)$ with $\mathsf{VerKey}(ppk, spk, ssk_2) = 1$, $\mathsf{Vf}(spk, \sigma^*, m^*) = 1$, and m was not queried, with non-negligible probability we have $m^* = \bar{m}$ and therefore $e(\sigma^*, g_2) = e(g_1^\alpha, spk)$. Since $spk = g_2^{\beta^{ssk_2}}$, we have $e(\sigma^*, g_2) = e(g_1^\alpha, g_2^{\beta^{ssk_2}})$. As $\sigma^* \neq 1_{\mathbb{G}_1}$, we have $ssk_2 \neq 0$ and $e({\sigma^*}^{1/ssk_2}, g_2) = e(g_1^\alpha, g_2^\beta)$, so ${\sigma^*}^{1/ssk_2} = g_1^{\alpha\cdot\beta}$ solves the co-CDH instance. □

Lemma 5. *The split-BLS signature scheme is key-hiding, as defined in Definition 17, under the XDH assumption, in the random-oracle model.*

Proof. Assume that adversary $\mathcal{A}$ breaks the key-hiding property with non-negligible probability, then we construct reduction $\mathcal{B}$ that breaks XDH with non-negligible probability.

$\mathcal{B}$ receives input the groups and $g_1^\alpha, g_1^\beta, g_1^\gamma$. If $g_1^\alpha = 1_{\mathbb{G}_1}$, $g_1^\beta = 1_{\mathbb{G}_1}$, or $\gamma = 1_{\mathbb{G}_1}$, the reduction fails. It receives $ppk \in \mathbb{G}_2$ from $\mathcal{A}$, after initializing it on the system parameters. When $\mathcal{A}$ makes random oracle queries, $\mathcal{B}$ answers with g_1^r with $r \xleftarrow{\$} \mathbb{Z}_q^*$, while maintaining consistency, except for a random query $\bar{m}$, where it returns g_1^β. When $\mathcal{A}$ makes a $\mathsf{CompleteSign}$ query on a message $m \neq \bar{m}$ and pre-signature σ', $\mathcal{B}$ first checks $e(\sigma', g_2) \stackrel{?}{=} e(\mathsf{H}(m), ppk)$, and returns $\perp$ if this does not hold. Otherwise, return $\mathsf{H}(m)^\alpha = (g_1^\alpha)^r$, where the reduction knows r from simulating the random oracle. $\mathcal{A}$ then outputs the challenge message, which with nonnegligible probability is $\bar{m}$, and a presignature σ'. $\mathcal{B}$ checks $e(\sigma', g_2) \stackrel{?}{=} e(\mathsf{H}(m), ppk)$ and returns $\sigma \leftarrow g_1^\gamma$. $\mathcal{A}$ finally outputs a bit indicating whether the real key was used or a random key, which exactly corresponds to the XDH instance being a DDH tuple or not. $\mathcal{B}$ can therefore copy $\mathcal{A}$'s output to break the XDH instance with nonnegligible probability. □

Lemma 6. *The split-BLS signature scheme satisfies key-uniqueness, as defined in Definition 18.*

Proof. As we work in prime order groups, every element has a unique discrete logarithm in $\mathbb{Z}_q$. Assume for contradiction that a signature σ on message m verifies under two keys $spk_0 \neq spk_1$. We then have $e(\sigma, g_2) = e(\mathsf{H}(m), spk_b)$ for $b \in \{0, 1\}$. Let $\mathsf{H}(m)$ be g_1^r for some $r \in \mathbb{Z}_q^*$, and let $spk_b = g_2^{x_b}$ for some $x_b \in \mathbb{Z}_q^*$. This gives $e(\sigma, g_2) = e(g_1, g_2)^{r \cdot x_b}$. Let s be the discrete log of σ, this means $s = r \cdot x_0$ and $s = r \cdot x_1$, and since $r \in \mathbb{Z}_q^*$, $s/r = x_0 = x_1$, contradicting $spk_0 \neq spk_1$. □

Lemma 7. *The split-BLS signature scheme satisfies signature uniqueness, as defined in Definition 19.*

Proof. Assume for contradiction that two signatures $\sigma_0 \neq \sigma_1$ on message m both verify under key spk, we have $e(\sigma_b, g_2) = e(\mathsf{H}(m), spk)$ for $b \in \{0, 1\}$. Let $\mathsf{H}(m)$ be g_1^r, $\sigma_b = g_1^{s_b}$, and $spk = g_2^x$, for some $r, s_0, s_1 \in \mathbb{Z}_q^*$ and $x \in \mathbb{Z}_q$. This gives $s_0 = s_1 = r \cdot x$, which contradicts $s_0 \neq s_1$. □

5.6 Construction

This section describes our DAA protocol achieving optimal privacy. On a very high level, the protocol follows the core idea of existing DAA protocols: The platform, consisting of the TPM and a host, first generates a secret key *gsk* that gets blindly certified by a trusted issuer. Subsequently, the platform can use the key *gsk* to sign attestations and basenames and then prove that it has a valid credential on the signing key, certifying the trusted origin of the attestation.

This high-level procedure is the main similarity to existing schemes though, as we significantly change the role of the host to satisfy our notion of optimal privacy. First, we no longer rely on a single secret key *gsk* that is fully controlled by the TPM. Instead, both the TPM and host generate secret shares, *tsk* and *hsk* respectively, that lead to a joint public key *gpk*. For privacy reasons, we cannot reveal this public key to the issuer in the join protocol, as any exposure of the joint public key would allow to trace any subsequent signed attestations of the platform. Thus, we let the issuer sign only an encryption of the public key, using the signature scheme for encrypted messages. When creating this membership credential *cred* the issuer is assured that the blindly signed key is formed correctly and the credential is strictly bound to that unknown key.

After having completed the JOIN protocol, the host and TPM can together sign a message m with respect to a basename bsn. Both parties use their individual key shares and create a split signature on the message and basename (denoted as tag), which shows that the platform intended to sign this message and basename, and a split signature on only the basename (denoted as nym), which is used as a pseudonym. Recall that attestations from one platform with the same basename should be linkable. By the uniqueness of split signatures, nym will be constant for one platform and basename and allow for such linkability. Because split signatures are key-hiding, we can reveal tag and nym while preserving the unlinkability of signatures with different basenames.

When signing, the host proves knowledge of a credential that signs gpk. Note that the host can create the full proof of knowledge because the membership credential signs a joint public key. In existing DAA schemes, the membership credential signs a TPM secret, and therefore the TPM must always be involved to prove knowledge of the credential, which prevents optimal privacy as we argued in Section 5.4.

5.6.1 Our DAA Protocol with Optimal Privacy Π_{pdaa}

We now present our generic DAA protocol with optimal privacy Π_{pdaa} in detail. Let $\mathsf{SSIG} = (\mathsf{PreKeyGen}, \mathsf{CompleteKeyGen}, \mathsf{VerKey}, \mathsf{PreSign}, \mathsf{CompleteSign}, \mathsf{Vf})$ denote a secure split signature scheme, as defined in Section 5.5.3, and let $\mathsf{ESIG} = (\mathsf{SigKGen}, \mathsf{EncSign}, \mathsf{DecSign}, \mathsf{Vf})$ denote a secure signature scheme for encrypted messages, as defined in Section 5.5.2. In addition, we use a CPA secure encryption scheme $\mathsf{ENC} = (\mathsf{EncKGen}, \mathsf{Enc}, \mathsf{Dec})$. We require all these algorithms to be compatible, meaning they work with the same system parameters.

We further assume that functionalities $(\mathcal{F}_{\mathsf{crs}}, \mathcal{F}_{\mathsf{ca}}, \mathcal{F}_{\mathsf{auth*}})$ are available to all parties. The certificate authority functionality $\mathcal{F}_{\mathsf{ca}}$ allows the issuer to register his public key, and we assume that parties call $\mathcal{F}_{\mathsf{ca}}$ to retrieve the public key whenever needed. As the issuer key (ipk, π_{ipk}) also contains a proof of well-formedness, we also assume that each party retrieving the key will verify π_{ipk}.

The common reference string functionality $\mathcal{F}_{\mathsf{crs}}$ provides all parties with the system parameters $spar$ generated via $\mathsf{SParGen}(1^{\tau})$. All the algorithms of the building blocks take $spar$ as an input, which we omit – except for the key generation algorithms – for ease of presentation.

For the communication between the TPM and issuer (via the host) in the join protocol, we use our semi-authenticated channel $\mathcal{F}_{\mathsf{auth*}}$, as introduced in Section 2.3.2. This functionality abstracts the different options on how to realize the authenticated channel between the TPM and issuer that is established via an unauthenticated host. We assume the host and TPM can communicate directly, meaning that they have an authenticated and perfectly secure channel. This models the physical proximity of the host and TPM forming the platform: if the host is honest an adversary can neither alter nor read their internal communication, or even notice that communication is happening. To make the protocol more readable, we omit the explicit calls to the sub-functionalities with sub-session IDs and simply say e.g., issuer $\mathcal{I}$ registers its public key with $\mathcal{F}_{\mathsf{ca}}$.

`1. Issuer Setup.`

In the setup phase, the issuer $\mathcal{I}$ creates a key pair of the signature scheme for encrypted messages and registers the public key with $\mathcal{F}_{\mathsf{ca}}$.

(a) $\mathcal{I}$ upon input $(\mathsf{SETUP}, \mathsf{sid})$ generates his key pair:

- Check that $\mathsf{sid} = (\mathcal{I}, \mathsf{sid}')$ for some sid'.
- Get $(ipk, isk) \xleftarrow{\$} \mathsf{ESIG.SigKGen}(spar)$ and prove knowledge of the secret key via

$$\pi_{ipk} \leftarrow \mathsf{NIZK}\{(\boxed{isk}) : (ipk, isk) \in \mathsf{ESIG.SigKGen}(spar)\}(\mathsf{sid}).$$

- Initiate $\mathcal{L}_{\mathsf{JOINED}} \leftarrow \emptyset$.
- Register the public key (ipk, π_{ipk}) at $\mathcal{F}_{\mathsf{ca}}$ and store $(isk, \mathcal{L}_{\mathsf{JOINED}})$.
- Output $(\mathsf{SETUPDONE}, \mathsf{sid})$.

Join Protocol.

The join protocol runs between the issuer $\mathcal{I}$ and a platform, consisting of a TPM $\mathcal{M}_i$ and a host $\mathcal{H}_j$. The platform authenticates to the issuer and, if the issuer allows the platform to join, obtains a credential *cred* that subsequently enables the platform to create signatures. The credential is a signature on the encrypted joint public key *gpk* to which the host and TPM each hold a secret key share. To show the issuer that a TPM has contributed to the joint key, the TPM reveals an authenticated version of his (public) key contribution to the issuer and the host proves that it correctly incorporated that share in *gpk*. A unique sub-session identifier $jsid$ distinguishes several join sessions that might run in parallel.

2. Join Request.

The join request is initiated by the host.

(a) Host $\mathcal{H}_j$, on input $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i)$ parses $\mathsf{sid} = (\mathcal{I}, \mathsf{sid}')$ and sends $(\mathsf{sid}, jsid)$ to $\mathcal{M}_i$.[2]

(b) TPM $\mathcal{M}_i$, upon receiving $(\mathsf{sid}, jsid)$ from a party $\mathcal{H}_j$, outputs $(\mathsf{JOIN}, \mathsf{sid}, jsid)$.

3. $\mathcal{M}$-Join Proceed.

The join session proceeds when the TPM receives an explicit input telling him to proceed with the join session $jsid$.

(a) TPM $\mathcal{M}_i$, on input $(\mathsf{JOIN}, \mathsf{sid}, jsid)$ creates a key share for the split signature and sends it authenticated to the issuer (via the host):

- Run $(tpk, tsk) \xleftarrow{\$} \mathsf{SSIG.PreKeyGen}(spar)$.
- Send tpk over $\mathcal{F}_{\mathsf{auth}*}$ to $\mathcal{I}$ via $\mathcal{H}_j$, and store the key $(\mathsf{sid}, \mathcal{H}_j, tsk)$.

(b) When $\mathcal{H}_j$ notices $\mathcal{M}_i$ sending tpk over $\mathcal{F}_{\mathsf{auth}*}$ to the issuer, it generates its key share for the split signature and appends an encryption of the jointly produced gpk to the message sent towards the issuer.

- Complete the split signature key as

$$(gpk, hsk) \xleftarrow{\$} \mathsf{SSIG.CompleteKeyGen}(tpk).$$

- Create an ephemeral encryption key pair

$$(epk, esk) \xleftarrow{\$} \mathsf{EncKGen}(spar).$$

- Encrypt gpk under epk as $C \xleftarrow{\$} \mathsf{Enc}(epk, gpk)$.

---

[2] Recall that we use direct communication between a TPM and host, i.e., this message is authenticated and unnoticed by the adversary.

- Prove that C is an encryption of a public key gpk that is correctly derived from the TPM public key share tpk:

$$\pi_{\mathsf{JOIN},\mathcal{H}} \leftarrow \mathsf{NIZK}\{(\boxed{gpk}, hsk) : C \in \mathsf{Enc}(epk, gpk) \wedge \\ \mathsf{SSIG.VerKey}(tpk, gpk, hsk) = 1\}(\mathsf{sid}, jsid).$$

- Append $(\mathcal{H}_j, epk, C, \pi_{\mathsf{JOIN},\mathcal{H}})$ to the message $\mathcal{M}_i$ is sending to $\mathcal{I}$ over $\mathcal{F}_{\mathsf{auth}*}$ and store $(\mathsf{sid}, jsid, \mathcal{M}_i, esk, hsk, gpk)$.

(c) $\mathcal{I}$, upon receiving a message over $\mathcal{F}_{\mathsf{auth}*}$, receiving tpk authenticated by $\mathcal{M}_i$ and $(\mathcal{H}_j, epk, C, \pi_{\mathsf{JOIN},\mathcal{H}})$ in the unauthenticated part, verifies that the request is legitimate:

- Verify $\pi_{\mathsf{JOIN},\mathcal{H}}$ w.r.t. the authenticated tpk and check that $\mathcal{M}_i \notin \mathcal{L}_{\mathsf{JOINED}}$.
- Store $(\mathsf{sid}, jsid, \mathcal{H}_j, \mathcal{M}_i, epk, C)$ and output $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i)$.

`4. ` $\mathcal{I}$`-Join Proceed.`

The join session is completed when the issuer receives an explicit input telling him to proceed with join session $jsid$.

(a) $\mathcal{I}$ upon input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ signs the encrypted public key C using the signature scheme for encrypted messages:

- Retrieve $(\mathsf{sid}, jsid, \mathcal{H}_j, \mathcal{M}_i, epk, C)$ and set $\mathcal{L}_{\mathsf{JOINED}} \leftarrow \mathcal{L}_{\mathsf{JOINED}} \cup \mathcal{M}_i$.
- Sign C as $cred' \overset{\$}{\leftarrow} \mathsf{ESIG.EncSign}(isk, epk, C)$ and prove that it did so correctly. (This proof is required to allow verification in the security proof: ENC is only CPA-secure and thus we cannot decrypt $cred'$.)

$$\pi_{\mathsf{JOIN},\mathcal{I}} \leftarrow \mathsf{NIZK}\{isk : cred' \in \mathsf{ESIG.EncSign}(isk, epk, C) \wedge \\ (ipk, isk) \in \mathsf{ESIG.SigKGen}(spar)\}(\mathsf{sid}, jsid).$$

- Send $(\mathsf{sid}, jsid, cred', \pi_{\mathsf{JOIN},\mathcal{I}})$ to $\mathcal{H}_j$ (via the network).

(b) Host $\mathcal{H}_j$, upon receiving $(\mathsf{sid}, jsid, cred', \pi_{\mathsf{JOIN},\mathcal{I}})$ decrypts and stores the membership credential:

- Retrieve the session record $(\mathsf{sid}, jsid, \mathcal{M}_i, esk, hsk, gpk)$.
- Verify proof $\pi_{\mathsf{JOIN},\mathcal{I}}$ w.r.t. ipk, $cred'$, C, and decrypt the credential as $cred \leftarrow \mathsf{ESIG.DecSign}(esk, cred')$.
- Store the completed key record $(\mathsf{sid}, hsk, tpk, gpk, cred, \mathcal{M}_i)$ and output $(\mathsf{JOINED}, \mathsf{sid}, jsid)$.

Sign Protocol.

The sign protocol runs between a TPM $\mathcal{M}_i$ and a host $\mathcal{H}_j$. After joining, together they can sign a message m w.r.t. a basename bsn using the split signature. Sub-session identifier $ssid$ distinguishes multiple sign sessions.

`5. Sign Request.`

The signature request is initiated by the host.

(a) $\mathcal{H}_j$ upon input $(\mathsf{SIGN}, \mathsf{sid}, ssid, \mathcal{M}_i, m, bsn)$ prepares the signature process:

- Check that it joined with $\mathcal{M}_i$ (i.e., a completed key record for $\mathcal{M}_i$ exists).
- Create signature record $(\mathsf{sid}, ssid, \mathcal{M}_i, m, bsn)$.
- Send $(\mathsf{sid}, ssid, m, bsn)$ to $\mathcal{M}_i$.

(b) $\mathcal{M}_i$, upon receiving $(\mathsf{sid}, ssid, m, bsn)$ from $\mathcal{H}_j$, stores $(\mathsf{sid}, ssid, \mathcal{H}_j, m, bsn)$ and outputs $(\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid, m, bsn)$.

`6. Sign Proceed.`

The signature is completed when $\mathcal{M}_i$ gets permission to proceed for $ssid$.

(a) $\mathcal{M}_i$ on input $(\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid)$ creates the first part of the split signature on m w.r.t. bsn:

- Retrieve the signature request $(\mathsf{sid}, ssid, \mathcal{H}_j, m, bsn)$ and key $(\mathsf{sid}, \mathcal{H}_j, tsk)$.
- Set $tag' \xleftarrow{\$} \mathsf{SSIG.PreSign}(tsk, (0, m, bsn))$.
- Set $nym' \xleftarrow{\$} \mathsf{SSIG.PreSign}(tsk, (1, bsn))$.
- Send $(\mathsf{sid}, ssid, tag', nym')$ to $\mathcal{H}_j$.

(b) $\mathcal{H}_j$ upon receiving $(\mathsf{sid}, ssid, tag', nym')$ from $\mathcal{M}_i$ completes the signature:

- Retrieve the signature request $(\mathsf{sid}, ssid, \mathcal{M}_i, m, bsn)$ and key $(\mathsf{sid}, hsk, tpk, gpk, cred, \mathcal{M}_i)$.
- Compute $tag \leftarrow \mathsf{SSIG.CompleteSign}(hsk, tpk, (0, m, bsn), tag')$.
- Compute $nym \leftarrow \mathsf{SSIG.CompleteSign}(hsk, tpk, (1, bsn), nym')$.
- Prove that tag and nym are valid split signatures under public key gpk and that it owns a valid issuer credential $cred$ on gpk, without revealing gpk or $cred$.

$$\begin{aligned}\pi_{\mathsf{SIGN}} \leftarrow \mathsf{NIZK}\{(gpk, cred) : \mathsf{ESIG.Vf}(ipk, cred, gpk) = 1 \wedge \\ \mathsf{SSIG.Vf}(gpk, tag, (0, m, bsn)) = 1 \wedge \\ \mathsf{SSIG.Vf}(gpk, nym, (1, bsn)) = 1\}\end{aligned}$$

- Set $\sigma \leftarrow (tag, nym, \pi_{\mathsf{SIGN}})$ and output $(\mathsf{Signature}, \mathsf{sid}, ssid, \sigma)$.

Verify & Link.

Any party can use the following verify and link algorithms to determine the validity of a signature and whether two signatures for the same basename were created by the same platform.

`7. Verify.`

The verify algorithm allows one to check whether a signature σ on message m w.r.t. basename bsn and private key revocation list $\mathtt{RL}$ is valid.

(a) $\mathcal{V}$ upon input $(\mathsf{VERIFY}, \mathsf{sid}, m, bsn, \sigma, \mathtt{RL})$ verifies the signature:

- Parse σ as $(tag, nym, \pi_{\mathsf{SIGN}})$.
- Verify π_{SIGN} with respect to m, bsn, tag, and nym.
- For every $gpk_i \in \mathtt{RL}$, check that $\mathsf{SSIG.Vf}(gpk_i, nym, (1, bsn)) \neq 1$.
- If all tests pass, set $f \leftarrow 1$, otherwise $f \leftarrow 0$.
- Output $(\mathsf{VERIFIED}, \mathsf{sid}, f)$.

8. `Link.`

The link algorithm allows one to check whether two signatures σ and σ', on messages m and m' respectively, that were generated for the same basename bsn were created by the same platform.

(a) $\mathcal{V}$ upon input $(\mathsf{LINK}, \mathsf{sid}, \sigma, m, \sigma', m', bsn)$ verifies the signatures and compares the pseudonyms contained in σ, σ':

- Check that both signatures σ and σ' are valid with respect to (m, bsn) and (m', bsn) respectively, using the `Verify` algorithm with $\texttt{RL} \leftarrow \varnothing$. Output $\perp$ if they are not both valid.
- Parse the signatures as $(tag, nym, \pi_{\mathsf{SIGN}})$ and $(tag', nym', \pi'_{\mathsf{SIGN}})$.
- If $nym = nym'$, set $f \leftarrow 1$, otherwise $f \leftarrow 0$.
- Output $(\mathsf{LINK}, \mathsf{sid}, f)$.

Random Oracles.

As our generic construction for delegatable anonymous credentials, this generic construction may use building blocks that assume one or more random oracles. Our generic construction assumes Embed and Embed^{-1} algorithms and applies domain separation as described in Section 4.4.2.

5.6.2 Security

We now prove that that our generic protocol is a secure DAA scheme with optimal privacy under isolated TPM corruptions (and also achieves conditional privacy under full TPM corruption) as defined in Section 5.3.

Theorem 10. *Our protocol* Π_{pdaa} *described in Section 5.6, GUC-realizes* $\mathcal{F}_{\mathsf{pdaa}}$ *defined in Section 5.3, in the* $(\mathcal{F}_{\mathsf{auth}*}, \mathcal{F}_{\mathsf{ca}}, \mathcal{F}_{\mathsf{crs}}, \mathcal{G}_{\mathsf{sRO}})$*-hybrid model, provided that*

- SSIG *is a secure split signature scheme (as defined in Section 5.5.3),*
- ESIG *is a secure signature scheme for encrypted messages,*
- ENC *is a CPA-secure encryption scheme, and*
- NIZK *is a zero-knowledge, simulation-sound and online-extractable (for the underlined values) proof system,*

- SSIG*,* ESIG*,* ENC*, and* NIZK *use local random oracles* $RO_1, \ldots, RO_j$*, mapping to* $S_1, \ldots, S_j$ *respectively, and efficiently computable probabilistic algorithms* $\mathsf{Embed}_1, \ldots, \mathsf{Embed}_j$ *and* $\mathsf{Embed}_1^{-1}, \ldots, \mathsf{Embed}_j^{-1}$ *exist, such that*
  - *for* $h \xleftarrow{\$} \{0,1\}^{\ell(\kappa)}$*,* $\mathsf{Embed}(h)$ *is computationally indistinguishable from uniform in* $\mathbb{G}$*,*
  - *for all* $x \in \mathbb{G}$*,* $\mathsf{Embed}(\mathsf{Embed}^{-1}(x)) = x$ *and*
  - *for* $x \xleftarrow{\$} \mathbb{G}$*,* $\mathsf{Embed}^{-1}(x)$ *is computationally indistinguishable from uniform in* $\{0,1\}^{\ell(\kappa)}$*.*

By Theorem 1, it is sufficient to show that Π_{pdaa} $\mathcal{G}_{\mathsf{sRO}}$-EUC-emulates $\mathcal{F}_{\mathsf{pdaa}}$ in the $\mathcal{F}_{\mathsf{auth}*}, \mathcal{F}_{\mathsf{ca}}, \mathcal{F}_{\mathsf{crs}}$-hybrid model, meaning that we have to show that there exists a simulator $\mathcal{S}$ as a function of $\mathcal{A}$ such that no $\mathcal{G}_{\mathsf{sRO}}$-externally constrained environment can distinguish Π_{pdaa} and $\mathcal{A}$ from $\mathcal{F}_{\mathsf{pdaa}}$ and $\mathcal{S}$. We let the adversary perform both isolated corruptions and full corruptions on TPMs, showing that this proof both gives optimal privacy with respect to adversaries that only perform isolated corruptions on TPMs, and conditional privacy otherwise. The full proof is given in the full version of [CDL17], we present a proof sketch below.

Proof Sketch

Setup. For the setup, the simulator has to provide the functionality the required algorithms $(\mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$, where $\mathsf{sig}, \mathsf{ver}, \mathsf{link}$, and ukgen simply reflect the corresponding real-world algorithms. The signing algorithm also includes the issuer's secret key. When the issuer is corrupt, $\mathcal{S}$ can learn the issuer secret key by extracting from the proof π_{ipk}. When the issuer is honest, it is simulated by $\mathcal{S}$ in the real-world and thus $\mathcal{S}$ knows the secret key.

The algorithm $\mathsf{identify}(\sigma, m, bsn, \tau)$ that is used by $\mathcal{F}_{\mathsf{pdaa}}$ to internally ensure consistency and non-frameability is defined as follows: parse σ as $(tag, nym, \pi_{\mathsf{SIGN}})$ and output $\mathsf{SSIG.Vf}(\tau, nym, (1, bsn))$. Recall that τ is a tracing trapdoor that is either provided by the simulator (when the host is corrupt) or generated internally by $\mathcal{F}_{\mathsf{pdaa}}$ whenever a new gpk is generated.

Join. The join-related interfaces of $\mathcal{F}_{\mathsf{pdaa}}$ notify $\mathcal{S}$ about any triggered join request by a platform consisting of host $\mathcal{H}_j$ and TPM $\mathcal{M}_i$ such that $\mathcal{S}$ can simulate the real-world protocol accordingly. If the host is corrupt, the simulator also has to provide the functionality with the tracing trapdoor τ. For our scheme the joint key *gpk* of the split signature serves that purpose. For privacy reasons the key is never revealed, but the host proves knowledge and correctness of the key in $\pi_{\mathsf{JOIN},\mathcal{H}}$. Thus, if the host is corrupt, the simulator extracts *gpk* from this proof and gives it $\mathcal{F}_{\mathsf{pdaa}}$.

Sign. For platforms with an honest host, $\mathcal{F}_{\mathsf{pdaa}}$ creates anonymous signatures using the sig algorithm $\mathcal{S}$ defined in the setup phase. Thereby, $\mathcal{F}_{\mathsf{pdaa}}$ enforces unlinkability by generating and using fresh platform keys via ukgen whenever a platform requests a signature for a new basename. For signature requests where a platform repeatedly uses the same basename, $\mathcal{F}_{\mathsf{pdaa}}$ re-uses the corresponding key accordingly. We now briefly argue that no environment can notice this difference. Recall that signatures consist of signatures *tag* and *nym*, and a proof π_{SIGN}, with the latter proving knowledge of the platform's key *gpk* and credential *cred*, such that *tag* and *nym* are valid under *gpk* which is in turn certified by *cred*. Thus, for every new basename, the credential *cred* is now based on different keys *gpk*. However, as we never reveal these values but only prove knowledge of them in π_{SIGN}, this change is indistinguishable to the environment.

The signature *tag* and pseudonym *nym*, that are split signatures on the message and basename, are revealed in plain though. For repeated attestations under the same basename, $\mathcal{F}_{\mathsf{pdaa}}$ consistently re-uses the same key, whereas the use of a fresh basename will now lead to the disclosure of split signatures under different keys. The key-hiding property of split signatures guarantees that this change is unnoticeable, even when the TPM is corrupt and controls part of the key.[3] Note that the key-hiding property requires that the adversary does not know the joint public key *gpk*, which we satisfy as *gpk* is never revealed in our scheme; the host only proves knowledge of the key in $\pi_{\mathsf{JOIN},\mathcal{H}}$ and π_{SIGN}.

Verify. For the verification of DAA signatures $\mathcal{F}_{\mathsf{pdaa}}$ uses the provided ver algorithm but also performs additional checks that enforce the desired non-frameability and unforgeability properties. We show that these additional checks will fail with negligible probability only,

---

[3] Notice that this proof step is a reduction, in which $\mathcal{G}_{\mathsf{sRO}}$ is in the reduction's control, meaning we can reduce to the key-hiding property which was proven w.r.t. a *local* random oracle.

and therefore do not noticeably change the verification outcome.

First, $\mathcal{F}_{\mathsf{pdaa}}$ uses the identify algorithm and the tracing trapdoors τ_i to check that there is only a unique signer that matches to the signature that is to be verified. Recall that we instantiated the identify algorithm with the verification algorithm of the split signature scheme SSIG and $\tau = gpk$ are the (hidden) joint platform keys. By the key-uniqueness property of SSIG the check will fail with negligible probability only.

Second, $\mathcal{F}_{\mathsf{pdaa}}$ rejects the signature when no matching tracing trapdoor was found and the issuer is honest. For platforms with an honest hosts, theses trapdoors are created internally by the functionality whenever a signature is generated, and $\mathcal{F}_{\mathsf{pdaa}}$ immediately checks that the signature matches to the trapdoor (via the identify algorithm). For platforms where the host is corrupt, our simulator $\mathcal{S}$ ensures that a tracing trapdoor is stored in $\mathcal{F}_{\mathsf{pdaa}}$ as soon as the platform has joined (and received a credential). If a signature does not match any of the existing tracing trapdoors, it must be under a $gpk = \tau$ that was neither created by $\mathcal{F}_{\mathsf{pdaa}}$ nor signed by the honest issuer in the real-world. The proof π_{SIGN} that is part of every signature σ proves knowledge of a valid issuer credential on gpk. Thus, by the unforgeability of the signature scheme for encrypted messages ESIG, such invalid signatures can occur only with negligible probability.

Third, if $\mathcal{F}_{\mathsf{pdaa}}$ recognizes a signature on message m w.r.t. basename bsn that matches the tracing trapdoor of a platform with an honest TPM or honest host, but that platform has never signed m w.r.t. bsn, it rejects the signature. This can be reduced to unforgeability-1 (if the host is honest) or unforgeability-2 (if the TPM is honest) of the split signature scheme SSIG.

The fourth check that $\mathcal{F}_{\mathsf{pdaa}}$ makes corresponds to the revocation check in the real-world verify algorithm, i.e., it does not impose any additional check.

Link. Similar as for verification, $\mathcal{F}_{\mathsf{pdaa}}$ is not relying solely on the provided link algorithm but performs some extra checks when testing for the linkage between two signatures σ and σ'. It again uses identify and the internally stored tracing trapdoor to derive the final linking output. If there is one tracing trapdoor matching one signature but not the other, it outputs that they are not linked. If there is one tracing trapdoor matching both signatures, it enforces the output that they are linked. Only if no matching tracing trapdoor is found, $\mathcal{F}_{\mathsf{pdaa}}$

derives the output via link algorithm.

We now show that the two checks and decisions imposed by $\mathcal{F}_{\mathsf{pdaa}}$ are consistent with the real-world linking algorithm. In the real world, signatures $\sigma = (tag, nym, \pi_{\mathsf{SIGN}})$ and $\sigma' = (tag', nym', \pi'_{\mathsf{SIGN}})$ w.r.t basename bsn are linked iff $nym = nym'$. Tracing trapdoors are instantiated by the split signature scheme public keys gpk, and identify verifies nym under the key gpk. If one key matches one signature but not the other, then by the fact that the verification algorithm of the split signatures is deterministic, we must have $nym \neq nym'$, showing that the real world algorithm also outputs unlinked. If one key matches both signatures, we have $nym = nym'$ by the signature-uniqueness of split signatures, so the real-world algorithm also outputs linked.

Global Random Oracle. In the above paragraphs, we sketch reducing to the security properties of our building blocks, which were preven with respect to *local* random oracles that can be programmed and observed, whereas our protocol works with strict global random oracle $\mathcal{G}_{\mathsf{sRO}}$ that does not give the simulator such powers. However, we only do so in reductions showing that no environment can distinguish two worlds. In that setting, everything except the environment is internal to the reduction, including $\mathcal{G}_{\mathsf{sRO}}$, as is depicted in Figure 3.3. This shows that in such reductions, we can program and observe $\mathcal{G}_{\mathsf{sRO}}$, allowing us to reduce to the security of the underlying building blocks. The local random oracles may map to sets other than $\{0,1\}^{\ell(\kappa)}$. However, we can use the Embed algorithms to obtain elements in the right set. In a security reduction, the security game offers local random oracles RO_i. Whenever a party queries $\mathcal{G}_{\mathsf{sRO}}$ on (i, m), we query RO_i on m to obtain x, and simulate $\mathcal{G}_{\mathsf{sRO}}$ to return $\mathsf{Embed}^{-1}(x)$.

Proof of Theorem 10

We now formallly prove Theorem 10, by showing that for every adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that for every $\mathcal{G}_{\mathsf{sRO}}$-externally constrained environment $\mathcal{E}$ we have $\mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{Z}} \approx \mathrm{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$.

Proof. To show that no environment $\mathcal{E}$ can distinguish the real world, in which it is working with Π_{pdaa} and adversary $\mathcal{A}$, from the ideal world, in which it uses $\mathcal{F}_{\mathsf{pdaa}}$ with simulator $\mathcal{S}$, we use a sequence of games. We start with the real world protocol execution. In the next game we construct one entity $\mathcal{C}$ that runs the real world protocol for all honest parties. Then we split $\mathcal{C}$ into two pieces, a functionality $\mathcal{F}$ and a simulator $\mathcal{S}$, where $\mathcal{F}$ receives all inputs from honest parties and sends the outputs to honest parties. We start with a dummy functionality, and gradually change $\mathcal{F}$ and update $\mathcal{S}$ accordingly, to end up with the full $\mathcal{F}_{\mathsf{pdaa}}$ and a satisfying simulator. First we define all intermediate functionalities and simulators, and then we prove that they are all indistinguishable from each other.

Setup

1. `Issuer Setup.` On input $(\mathsf{SETUP}, \mathsf{sid})$ from issuer $\mathcal{I}$
   - Output $(\mathsf{FORWARD}, (\mathsf{SETUP}, \mathsf{sid}), \mathcal{I})$ to $\mathcal{S}$.

Join

2. `Join Request.` On input $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
   - Output $(\mathsf{FORWARD}, (\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i), \mathcal{H}_j)$ to $\mathcal{S}$.
3. $\mathcal{M}$ `Join Proceed.` On input $(\mathsf{JOIN}, \mathsf{sid}, jsid)$ from TPM $\mathcal{M}_i$.
   - Output $(\mathsf{FORWARD}, (\mathsf{JOIN}, \mathsf{sid}, jsid), \mathcal{M}_i)$ to $\mathcal{S}$.
4. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ from $\mathcal{I}$.
   - Output $(\mathsf{FORWARD}, (\mathsf{JOINPROCEED}, \mathsf{sid}, jsid), \mathcal{I})$ to $\mathcal{S}$.

Sign

5. `Sign Request.` On input $(\mathsf{SIGN}, \mathsf{sid}, ssid, \mathcal{M}_i, m, bsn)$ from $\mathcal{H}_j$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGN}, \mathsf{sid}, ssid, \mathcal{M}_i, m, bsn), \mathcal{H}_j)$ to $\mathcal{S}$.
6. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid)$ from $\mathcal{M}_i$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid), \mathcal{M}_i)$ to $\mathcal{S}$.

Verify

7. `Verify.` On input $(\mathsf{VERIFY}, \mathsf{sid}, m, bsn, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.
   - Output $(\mathsf{FORWARD}, (\mathsf{VERIFY}, \mathsf{sid}, m, bsn, \sigma, \mathtt{RL}), \mathcal{V})$ to $\mathcal{S}$.

Link

8. `Link.` On input $(\mathsf{LINK}, \mathsf{sid}, \sigma, m, \sigma', m', bsn)$ from a party $\mathcal{V}$.
   - Output $(\mathsf{FORWARD}, (\mathsf{LINK}, \mathsf{sid}, \sigma, m, \sigma', m', bsn), \mathcal{V})$ to $\mathcal{S}$.

Output

9. `Output.` On input $(\mathsf{OUTPUT}, \mathsf{sid}, \mathcal{P}, m)$ from $\mathcal{S}$.
   - Output m to $\mathcal{P}$.

Figure 5.10: $\mathcal{F}$ for GAME 3

When a simulated party "$\mathcal{P}$" outputs m and no specific action is defined, send $(\mathsf{OUTPUT}, \mathcal{P}, m)$ to $\mathcal{F}$. On input $(\mathsf{FORWARD}, m, \mathcal{P})$, give "$\mathcal{P}$" input m.

Figure 5.11: Simulator for GAME 3

Setup

1. `Issuer Setup.` On input $(\mathsf{SETUP}, \mathsf{sid})$ from issuer $\mathcal{I}$
    - Verify that $\mathsf{sid} = (\mathcal{I}, sid')$.
    - Output $(\mathsf{SETUP}, \mathsf{sid})$ to $\mathcal{A}$ and wait for input $(\mathsf{ALG}, \mathsf{sid}, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ from $\mathcal{A}$.
    - Check that ver, link and $\mathsf{identify}$ are deterministic, and check that sig, ver, link, $\mathsf{identify}$, ukgen interact only with random oracle $\mathcal{G}_{\mathsf{sRO}}$.
    - Store $(\mathsf{sid}, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ and output $(\mathsf{SETUPDONE}, \mathsf{sid})$ to $\mathcal{I}$.

Join

2. `Join Request.` On input $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
    - Output $(\mathsf{FORWARD}, (\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i), \mathcal{H}_j)$ to $\mathcal{S}$.
3. $\mathcal{M}$ `Join Proceed.` On input $(\mathsf{JOIN}, \mathsf{sid}, jsid)$ from TPM $\mathcal{M}_i$.
    - Output $(\mathsf{FORWARD}, (\mathsf{JOIN}, \mathsf{sid}, jsid), \mathcal{M}_i)$ to $\mathcal{S}$.
4. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ from $\mathcal{I}$.
    - Output $(\mathsf{FORWARD}, (\mathsf{JOINPROCEED}, \mathsf{sid}, jsid), \mathcal{I})$ to $\mathcal{S}$.

Sign

5. `Sign Request.` On input $(\mathsf{SIGN}, \mathsf{sid}, ssid, \mathcal{M}_i, m, bsn)$ from $\mathcal{H}_j$.
    - Output $(\mathsf{FORWARD}, (\mathsf{SIGN}, \mathsf{sid}, ssid, \mathcal{M}_i, m, bsn), \mathcal{H}_j)$ to $\mathcal{S}$.
6. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid)$ from $\mathcal{M}_i$.
    - Output $(\mathsf{FORWARD}, (\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid), \mathcal{M}_i)$ to $\mathcal{S}$.

Verify

7. `Verify.` On input $(\mathsf{VERIFY}, \mathsf{sid}, m, bsn, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.
    - Output $(\mathsf{FORWARD}, (\mathsf{VERIFY}, \mathsf{sid}, m, bsn, \sigma, \mathtt{RL}), \mathcal{V})$ to $\mathcal{S}$.

Link

8. `Link.` On input $(\mathsf{LINK}, \mathsf{sid}, \sigma, m, \sigma', m', bsn)$ from a party $\mathcal{V}$.
    - Output $(\mathsf{FORWARD}, (\mathsf{LINK}, \mathsf{sid}, \sigma, m, \sigma', m', bsn), \mathcal{V})$ to $\mathcal{S}$.

Output

9. `Output.` On input $(\mathsf{OUTPUT}, \mathsf{sid}, \mathcal{P}, m)$ from $\mathcal{S}$.
    - Output m to $\mathcal{P}$.

Figure 5.12: $\mathcal{F}$ for GAME 4

When a simulated party "$\mathcal{P}$" outputs m and no specific action is defined, send $(\mathsf{OUTPUT}, \mathcal{P}, m)$ to $\mathcal{F}$. On input $(\mathsf{FORWARD}, m, \mathcal{P})$, give "$\mathcal{P}$" input m.

Setup

Honest $\mathcal{I}$

- On input $(\mathsf{SETUP}, \mathsf{sid})$ from $\mathcal{F}$.
  - Parse sid as $(\mathcal{I}, \mathsf{sid}')$ and give "$\mathcal{I}$" input $(\mathsf{SETUP}, \mathsf{sid})$.
  - When "$\mathcal{I}$" outputs $(\mathsf{SETUPDONE}, \mathsf{sid})$, $\mathcal{S}$ takes its secret key isk and defines the following algorithms.
    - ∗ Define $\mathsf{sig}(((tsk, hsk), gpk), m, bsn)$ as follows: First, create a credential by taking encryption key $(epk, esk) \leftarrow \mathsf{EncKGen}()$. Encrypt the credential with $C \leftarrow \mathsf{Enc}(epk, gpk)$, and sign the ciphertext with $cred' \leftarrow \mathsf{EncSign}(isk, epk, C)$., and decrypt credential $cred \leftarrow \mathsf{DecSign}(esk, cred')$. Next, the algorithm performs the real world signing algorithm (performing both the tasks from the host and the TPM).
    - ∗ Define $\mathsf{ver}(\sigma, m, bsn)$ as the real world verification algorithm, except that the private-key revocation check is ommitted.
    - ∗ Define $\mathsf{link}(\sigma, m, \sigma', m', bsn)$ as the real world linking algorithm.
    - ∗ Define $\mathsf{identify}(\sigma, m, bsn, \tau)$ as follows: parse σ as $(tag, nym, \pi_{\mathsf{SIGN}})$ and check $\mathsf{SSIG.Vf}(\tau, nym, (1, bsn))$. If so, output 1, otherwise 0.
    - ∗ Define ukgen as follows: Let $(tpk, tsk) \leftarrow \mathsf{SSIG.PreKeyGen}()$, $(gpk, hsk) \leftarrow \mathsf{SSIG.CompleteKeyGen}(tpk)$, and output $((tsk, hsk), gpk)$.

    $\mathcal{S}$ sends $(\mathsf{ALG}, \mathsf{sid}, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ to $\mathcal{F}$.

Corrupt $\mathcal{I}$

- $\mathcal{S}$ notices this setup as it notices $\mathcal{I}$ registering a public key with "$\mathcal{F}_{\mathsf{ca}}$" with $\mathsf{sid} = (\mathcal{I}, \mathsf{sid}')$.
  - If the registered key is of the form (ipk, π_{isk}) and π is valid, $\mathcal{S}$ extracts isk from π_{isk}.
  - $\mathcal{S}$ defines the algorithms $\mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen}$ as when $\mathcal{I}$ is honest, but now depending on the extracted key.
  - $\mathcal{S}$ sends $(\mathsf{SETUP}, \mathsf{sid})$ to $\mathcal{F}$ on behalf of $\mathcal{I}$.
- On input $(\mathsf{SETUP}, \mathsf{sid})$ from $\mathcal{F}$.
  - $\mathcal{S}$ sends $(\mathsf{ALG}, \mathsf{sid}, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ to $\mathcal{F}$.
- On input $(\mathsf{SETUPDONE}, \mathsf{sid})$ from $\mathcal{F}$
  - $\mathcal{S}$ continues simulating "$\mathcal{I}$".

Figure 5.13: Simulator for GAME 4

Setup

Unchanged.

Join

2. `Join Request.` On input $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
   - Output $(\mathsf{FORWARD}, (\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i), \mathcal{H}_j)$ to $\mathcal{S}$.
3. `$\mathcal{M}$ Join Proceed.` On input $(\mathsf{JOIN}, \mathsf{sid}, jsid)$ from TPM $\mathcal{M}_i$.
   - Output $(\mathsf{FORWARD}, (\mathsf{JOIN}, \mathsf{sid}, jsid), \mathcal{M}_i)$ to $\mathcal{S}$.
4. `$\mathcal{I}$ Join Proceed.` On input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ from $\mathcal{I}$.
   - Output $(\mathsf{FORWARD}, (\mathsf{JOINPROCEED}, \mathsf{sid}, jsid), \mathcal{I})$ to $\mathcal{S}$.

Sign

5. `Sign Request.` On input $(\mathsf{SIGN}, \mathsf{sid}, ssid, \mathcal{M}_i, m, bsn)$ from $\mathcal{H}_j$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGN}, \mathsf{sid}, ssid, \mathcal{M}_i, m, bsn), \mathcal{H}_j)$ to $\mathcal{S}$.
6. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid)$ from $\mathcal{M}_i$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid), \mathcal{M}_i)$ to $\mathcal{S}$.

Verify

7. `Verify.` On input $(\mathsf{VERIFY}, \mathsf{sid}, m, bsn, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.
   - Set $f \leftarrow 0$ if at least one of the following conditions hold:
     - There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn)$.
   - Add $\langle \sigma, m, bsn, \mathtt{RL}, f \rangle$ to `VerResults` and output $(\mathsf{VERIFIED}, \mathsf{sid}, f)$ to $\mathcal{V}$.

Link

8. `Link.` On input $(\mathsf{LINK}, \mathsf{sid}, \sigma, m, \sigma', m', bsn)$ from a party $\mathcal{V}$.
   - Output $\perp$ to $\mathcal{V}$ if at least one signature (σ, m, bsn) or (σ', m', bsn) is not valid (verified via the `verify` interface with $\mathtt{RL} = \varnothing$).
   - Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
   - Output $(\mathsf{LINK}, \mathsf{sid}, f)$ to $\mathcal{V}$.

Output

9. `Output.` On input $(\mathsf{OUTPUT}, \mathsf{sid}, \mathcal{P}, m)$ from $\mathcal{S}$.
   - Output m to $\mathcal{P}$.

Figure 5.14: $\mathcal{F}$ for GAME 5

When a simulated party "$\mathcal{P}$" outputs m and no specific action is defined, send $(\mathsf{OUTPUT}, \mathcal{P}, m)$ to $\mathcal{F}$. On input $(\mathsf{FORWARD}, m, \mathcal{P})$, give "$\mathcal{P}$" input m.

Setup

Honest $\mathcal{I}$

- On input $(\mathsf{SETUP}, \mathsf{sid})$ from $\mathcal{F}$.
  - Parse sid as $(\mathcal{I}, \mathsf{sid}')$ and give "$\mathcal{I}$" input $(\mathsf{SETUP}, \mathsf{sid})$.
  - When "$\mathcal{I}$" outputs $(\mathsf{SETUPDONE}, \mathsf{sid})$, $\mathcal{S}$ takes its secret key isk and defines the following algorithms.
    - Define $\mathsf{sig}(((tsk, hsk), gpk), m, bsn)$ as follows: First, create a credential by taking encryption key $(epk, esk) \leftarrow \mathsf{EncKGen}()$. Encrypt the credential with $C \leftarrow \mathsf{Enc}(epk, gpk)$, and sign the ciphertext with $cred' \leftarrow \mathsf{EncSign}(isk, epk, C)$., and decrypt credential $cred \leftarrow \mathsf{DecSign}(esk, cred')$. Next, the algorithm performs the real world signing algorithm (performing both the tasks from the host and the TPM).
    - Define $\mathsf{ver}(\sigma, m, bsn)$ as the real world verification algorithm, except that the private-key revocation check is ommitted.
    - Define $\mathsf{link}(\sigma, m, \sigma', m', bsn)$ as the real world linking algorithm.
    - Define $\mathsf{identify}(\sigma, m, bsn, \tau)$ as follows: parse σ as $(tag, nym, \pi_{\mathsf{SIGN}})$ and check $\mathsf{SSIG.Vf}(\tau, nym, (1, bsn))$. If so, output 1, otherwise 0.
    - Define ukgen as follows: Let $(tpk, tsk) \leftarrow \mathsf{SSIG.PreKeyGen}()$, $(gpk, hsk) \leftarrow \mathsf{SSIG.CompleteKeyGen}(tpk)$, and output $((tsk, hsk), gpk)$.

    $\mathcal{S}$ sends $(\mathsf{ALG}, \mathsf{sid}, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ to $\mathcal{F}$.

Corrupt $\mathcal{I}$

- $\mathcal{S}$ notices this setup as it notices $\mathcal{I}$ registering a public key with "$\mathcal{F}_{\mathsf{ca}}$" with $\mathsf{sid} = (\mathcal{I}, \mathsf{sid}')$.
  - If the registered key is of the form (ipk, π_{isk}) and π is valid, $\mathcal{S}$ extracts isk from π_{isk}.
  - $\mathcal{S}$ defines the algorithms $\mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen}$ as when $\mathcal{I}$ is honest, but now depending on the extracted key.
  - $\mathcal{S}$ sends $(\mathsf{SETUP}, \mathsf{sid})$ to $\mathcal{F}$ on behalf of $\mathcal{I}$.
- On input $(\mathsf{SETUP}, \mathsf{sid})$ from $\mathcal{F}$.
  - $\mathcal{S}$ sends $(\mathsf{ALG}, \mathsf{sid}, \mathsf{sig}, \mathsf{ver}, \mathsf{link}, \mathsf{identify}, \mathsf{ukgen})$ to $\mathcal{F}$.
- On input $(\mathsf{SETUPDONE}, \mathsf{sid})$ from $\mathcal{F}$
  - $\mathcal{S}$ continues simulating "$\mathcal{I}$".

Figure 5.15: Simulator for GAME 5

Setup

Unchanged.

Join

2. `Join Request.` On input $(\mathsf{JOIN}, \mathsf{sid}, \mathit{jsid}, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
   - Create a join session record $\langle \mathit{jsid}, \mathcal{M}_i, \mathcal{H}_j, \mathit{status} \rangle$ with $\mathit{status} \leftarrow \mathit{request}$.
   - Output $(\mathsf{JOIN}, \mathsf{sid}, \mathit{jsid}, \mathcal{H}_j)$ to $\mathcal{M}_i$.
3. $\mathcal{M}$ `Join Proceed.` On input $(\mathsf{JOIN}, \mathsf{sid}, \mathit{jsid})$ from TPM $\mathcal{M}_i$.
   - Update the session record $\langle \mathit{jsid}, \mathcal{M}_i, \mathcal{H}_j, \mathit{status} \rangle$ with $\mathit{status} = \mathit{request}$ to $\mathit{delivered}$.
   - Output $(\mathsf{JOINPROCEED}, \mathsf{sid}, \mathit{jsid}, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINPROCEED}, \mathsf{sid}, \mathit{jsid})$ from $\mathcal{A}$.
   - Output $(\mathsf{JOINPROCEED}, \mathsf{sid}, \mathit{jsid}, \mathcal{M}_i)$ to $\mathcal{I}$.
4. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, \mathsf{sid}, \mathit{jsid})$ from $\mathcal{I}$.
   - Update the session record $\langle \mathit{jsid}, \mathcal{M}_i, \mathcal{H}_j, \mathit{status} \rangle$ with $\mathit{status} = \mathit{delivered}$ to $\mathit{complete}$.
   - Output $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, \mathit{jsid})$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, \mathit{jsid}, \tau)$ from $\mathcal{A}$.
   - Abort if $\mathcal{I}$ or $\mathcal{M}_i$ is honest and a record $\langle \mathcal{M}_i, *, * \rangle \in$ `Members` already exists.
   - If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \perp$.
   - Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau \rangle$ into `Members` and output $(\mathsf{JOINED}, \mathsf{sid}, \mathit{jsid})$ to $\mathcal{H}_j$.

Sign

5. `Sign Request.` On input $(\mathsf{SIGN}, \mathsf{sid}, \mathit{ssid}, \mathcal{M}_i, m, \mathit{bsn})$ from $\mathcal{H}_j$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGN}, \mathsf{sid}, \mathit{ssid}, \mathcal{M}_i, m, \mathit{bsn}), \mathcal{H}_j)$ to $\mathcal{S}$.
6. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, \mathsf{sid}, \mathit{ssid})$ from $\mathcal{M}_i$.
   - Output $(\mathsf{FORWARD}, (\mathsf{SIGNPROCEED}, \mathsf{sid}, \mathit{ssid}), \mathcal{M}_i)$ to $\mathcal{S}$.

Verify

Unchanged.

Link

Unchanged.

Output

Unchanged.

Figure 5.16: $\mathcal{F}$ for GAME 6

When a simulated party "$\mathcal{P}$" outputs m and no specific action is defined, send $(\mathsf{OUTPUT}, \mathcal{P}, m)$ to $\mathcal{F}$. On input $(\mathsf{FORWARD}, m, \mathcal{P})$, give "$\mathcal{P}$" input m.

Isolated Corrupt TPM

When a TPM $\mathcal{M}_i$ becomes isolated corrupted in the simulated real world, $\mathcal{S}$ defines a local simulator $\mathcal{S}_{\mathcal{M}_i}$ that simulates an honest host with the isolated corrupt $\mathcal{M}_i$. Note that $\mathcal{M}_i$ only talks to one host, who's identity is fixed upon receiving the first message. $\mathcal{S}_{\mathcal{M}_i}$ is defined as follows.

- When $\mathcal{S}_{\mathcal{M}_i}$ receives $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{H}_j)$ as $\mathcal{M}_i$ is isolated corrupt.
  - Give "$\mathcal{H}_j$" input $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i)$.
  - When "$\mathcal{H}_j$" outputs $(\mathsf{JOINED}, \mathsf{sid}, jsid)$, send $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.

Setup

Unchanged.

Join

Honest $\mathcal{M}$, $\mathcal{H}$, $\mathcal{I}$

- On input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - Give "$\mathcal{H}_j$" input $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i)$.
  - When "$\mathcal{M}_i$" outputs $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{H}_j)$, give "$\mathcal{M}_i$" input $(\mathsf{JOIN}, \mathsf{sid}, jsid)$.
  - When "$\mathcal{I}$" outputs $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i)$, output $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ to $\mathcal{F}$.
- On input $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid)$.
  - Give "$\mathcal{I}$" input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$.
  - When "$\mathcal{H}_j$" outputs $(\mathsf{JOINED}, \mathsf{sid}, jsid)$, output $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid, \bot)$ to $\mathcal{F}$.

Honest $\mathcal{H}$, $\mathcal{I}$, Corrupt $\mathcal{M}$

- When $\mathcal{S}$ receives $(\mathsf{JOIN}, \mathsf{sid}, jsid)$ from $\mathcal{F}$ as $\mathcal{M}_i$ is corrupt.
  - Give "$\mathcal{H}_j$" input $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i)$.
  - When "$\mathcal{I}$" outputs $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i)$, send $(\mathsf{JOIN}, \mathsf{sid}, jsid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.
- On input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - Output $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ to $\mathcal{F}$.
- On input $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid)$.
  - Give "$\mathcal{I}$" input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$.
  - When "$\mathcal{H}_j$" outputs $(\mathsf{JOINED}, \mathsf{sid}, jsid)$, output $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid, \bot)$ to $\mathcal{F}$.

Honest $\mathcal{M}$, $\mathcal{H}$, Corrupt $\mathcal{I}$

- On input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - Give "$\mathcal{H}_j$" input $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i)$.
  - When "$\mathcal{M}_i$" outputs $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{H}_j)$, give "$\mathcal{M}_i$" input $(\mathsf{JOIN}, \mathsf{sid}, jsid)$.
  - When "$\mathcal{H}_j$" outputs $(\mathsf{JOINED}, \mathsf{sid}, jsid)$, output $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ to $\mathcal{F}$.
- When $\mathcal{S}$ receives $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i)$ from $\mathcal{F}$ as $\mathcal{I}$ is corrupt.
  - Send $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ on $\mathcal{I}$'s behalf to $\mathcal{F}$.
- On input $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid)$.
  - output $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid, \bot)$ to $\mathcal{F}$.

Figure 5.17: First part of Simulator for GAME 6

Honest $\mathcal{M}$, $\mathcal{I}$, Corrupt $\mathcal{H}$

- $\mathcal{S}$ notices this join as "$\mathcal{M}_i$" outputs $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{H}_j)$.
  - Send $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i)$ on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.
- On input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - Continue simulating "$\mathcal{M}_i$" by giving it input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$.
  - When "$\mathcal{I}$" outputs $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i)$, extract gpk from $\pi_{\mathsf{JOIN},\mathcal{H}}$ and output $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ to $\mathcal{F}$.
- On input $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid)$ from $\mathcal{F}$.
  - output $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid, gpk)$ to $\mathcal{F}$.
- When $\mathcal{S}$ receives $(\mathsf{JOINED}, \mathsf{sid}, jsid)$ from $\mathcal{F}$ as $\mathcal{H}_j$ is corrupt.
  - Continue simulating "$\mathcal{I}$" by giving it input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$.

Honest $\mathcal{H}$, Corrupt $\mathcal{M}$, $\mathcal{I}$

- When $\mathcal{S}$ receives $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i)$ as $\mathcal{M}_i$ is corrupt.
  - Send $(\mathsf{JOIN}, \mathsf{sid}, jsid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.
- On input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - Give "$\mathcal{H}_j$" input $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i)$.
  - When "$\mathcal{H}_j$" outputs $(\mathsf{JOINED}, \mathsf{sid}, jsid)$, output $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ to $\mathcal{F}$.
- When $\mathcal{S}$ receives $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i)$ as $\mathcal{I}$ is corrupt.
  - Send $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ on $\mathcal{I}$'s behalf to $\mathcal{F}$.
- On input $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid)$ from $\mathcal{F}$.
  - Output $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid, \bot)$ to $\mathcal{F}$.

Honest $\mathcal{I}$, Corrupt $\mathcal{M}$, $\mathcal{H}$

- $\mathcal{S}$ notices this join as "$\mathcal{I}$" outputs $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i)$.
  - Extract gpk from $\pi_{\mathsf{JOIN},\mathcal{H}}$ and output $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ to $\mathcal{F}$.
  - Pick some corrupt identity $\mathcal{H}_j$, and send $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i)$ on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.
- When $\mathcal{S}$ receives $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{H}_j)$ as $\mathcal{M}_i$ is corrupt.
  - Send $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.
- On input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - Output $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ to $\mathcal{F}$.
- On input $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid, gpk)$ from $\mathcal{F}$.
  - Output $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid)$ to $\mathcal{F}$.
- When $\mathcal{S}$ receives $(\mathsf{JOINED}, \mathsf{sid}, jsid)$ as $\mathcal{H}_j$ is corrupt.
  - Give "$\mathcal{I}$" input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$.

Honest $\mathcal{M}$, Corrupt $\mathcal{H}$, $\mathcal{I}$

- $\mathcal{S}$ notices this join as "$\mathcal{M}_i$" outputs $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{H}_j)$.
  - Send $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i)$ on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.
- On input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i, \mathcal{H}_j)$ from $\mathcal{F}$.
  - Output $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ to $\mathcal{F}$.
- When $\mathcal{S}$ receives $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i)$ as $\mathcal{I}$ is corrupt.
  - Send $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ on $\mathcal{I}$'s behalf to $\mathcal{F}$.
- On input $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid)$ from $\mathcal{F}$.
  - Output $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid, \bot)$ to $\mathcal{F}$.
- When $\mathcal{S}$ receives $(\mathsf{JOINED}, \mathsf{sid}, jsid)$ as $\mathcal{H}_j$ is corrupt.
  - Give "$\mathcal{M}_i$" input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$.

Figure 5.18: Second part of Simulator for GAME 6

Honest $\mathcal{H}$, $\mathcal{I}$, Isolated corrupt $\mathcal{M}$

- On input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i, \mathcal{H}_j)$.
  - Give "$\mathcal{H}_j$" input $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i)$.
  - When "$\mathcal{M}_i$" outputs $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{H}_j)$, Give "$\mathcal{M}_i$" input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$.
  - When "$\mathcal{I}$" outputs $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i)$, output $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ to $\mathcal{F}$.
- On input $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid)$.
  - Give "$\mathcal{I}$" input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$.
  - When "$\mathcal{H}_j$" outputs $(\mathsf{JOINED}, \mathsf{sid}, jsid)$, output $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid, \perp)$ to $\mathcal{F}$.

Honest $\mathcal{H}$, Isolated corrupt $\mathcal{M}$, Corrupt $\mathcal{I}$

- On input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i, \mathcal{H}_j)$.
  - Give "$\mathcal{H}_j$" input $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i)$.
  - When "$\mathcal{M}_i$" outputs $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{H}_j)$, Give "$\mathcal{M}_i$" input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$.
  - When "$\mathcal{H}_j$" outputs $(\mathsf{JOINED}, \mathsf{sid}, jsid)$, output $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ to $\mathcal{F}$.
- When $\mathcal{S}$ receives $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i)$ as $\mathcal{I}$ is corrupt.
  - Send $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ on $\mathcal{I}$'s behalf to $\mathcal{F}$.
- On input $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid)$ from $\mathcal{F}$.
  - output $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid, \perp)$ to $\mathcal{F}$.

Figure 5.19: Third part of Simulator for GAME 6

Setup

Unchanged.

Join

Unchanged.

Sign

5. `Sign Request`. On input $(\mathsf{SIGN}, \mathsf{sid}, ssid, \mathcal{M}_i, m, bsn)$ from $\mathcal{H}_j$.
   - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in `Members`, abort.
   - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status \leftarrow request$.
   - Output $(\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid, m, bsn)$ to $\mathcal{M}_i$.
6. `Sign Proceed`. On input $(\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid)$ from $\mathcal{M}_i$.
   - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status\rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, *\rangle$ exists in `Members`.
   - Generate the signature for a fresh or established key:
     - Retrieve (gsk, τ) from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle \in$ `DomainKeys`. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau\rangle$ in `DomainKeys`.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn)$, check $\mathsf{ver}(\sigma, m, bsn) = 1$.
   - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j\rangle$ in `Signed` and output $(\mathsf{Signature}, \mathsf{sid}, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

7. `Verify`. On input $(\mathsf{VERIFY}, \mathsf{sid}, m, bsn, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.
   - Set $f \leftarrow 0$ if at least one of the following conditions hold:
     - There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn)$.
   - Add $\langle \sigma, m, bsn, \mathtt{RL}, f\rangle$ to `VerResults` and output $(\mathsf{VERIFIED}, \mathsf{sid}, f)$ to $\mathcal{V}$.

Link

8. `Link`. On input $(\mathsf{LINK}, \mathsf{sid}, \sigma, m, \sigma', m', bsn)$ from a party $\mathcal{V}$.
   - Output $\perp$ to $\mathcal{V}$ if at least one signature (σ, m, bsn) or (σ', m', bsn) is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
   - Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
   - Output $(\mathsf{LINK}, \mathsf{sid}, f)$ to $\mathcal{V}$.

Figure 5.20: $\mathcal{F}$ for GAME 7

Isolated Corrupt TPM

When a TPM $\mathcal{M}_i$ becomes isolated corrupted in the simulated real world, $\mathcal{S}$ defines a local simulator $\mathcal{S}_{\mathcal{M}_i}$ that simulates an honest host with the isolated corrupt $\mathcal{M}_i$. Note that $\mathcal{M}_i$ only talks to one host, who's identity is fixed upon receiving the first message. $\mathcal{S}_{\mathcal{M}_i}$ is defined as follows.

- When $\mathcal{S}_{\mathcal{M}_i}$ receives $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{H}_j)$ as $\mathcal{M}_i$ is isolated corrupt.
  - Give "$\mathcal{H}_j$" input $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i)$.
  - When "$\mathcal{H}_j$" outputs $(\mathsf{JOINED}, \mathsf{sid}, jsid)$, send $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.
- When $\mathcal{S}_{\mathcal{M}_i}$ receives $(\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid, m, bsn)$ as $\mathcal{M}_i$ is isolated corrupt.
  - Give "$\mathcal{H}_j$" input $(\mathsf{SIGN}, \mathsf{sid}, ssid, \mathcal{M}_i, m, bsn)$.
  - When "$\mathcal{H}_j$" outputs $(\mathsf{Signature}, \mathsf{sid}, ssid, \sigma)$, send $(\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.

Setup

Unchanged.

Join

Unchanged.

Sign

Honest $\mathcal{M}$, $\mathcal{H}$

Nothing to simulate.

Honest $\mathcal{H}$, Corrupt $\mathcal{M}$

- When $\mathcal{S}$ receives $(\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid, m, bsn)$ as $\mathcal{M}_i$ is corrupt.
  - Give "$\mathcal{H}_j$" input $(\mathsf{SIGN}, \mathsf{sid}, ssid, \mathcal{M}_i, m, bsn)$.
  - When "$\mathcal{H}_j$" outputs $(\mathsf{Signature}, \mathsf{sid}, ssid, \sigma)$, send $(\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid)$ on $\mathcal{M}_i$'s behalf to $\mathcal{F}$.

Honest $\mathcal{H}$, Isolated corrupt $\mathcal{M}$

Nothing to simulate.

Honest $\mathcal{M}$, Corrupt $\mathcal{H}$

- When "$\mathcal{M}_i$" outputs $(\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid, m, bsn)$.
  - Send $(\mathsf{SIGN}, \mathsf{sid}, ssid, \mathcal{M}_i, m, bsn)$ on $\mathcal{H}_j$'s behalf to $\mathcal{F}$.
  - When $\mathcal{S}$ receives $(\mathsf{Signature}, \mathsf{sid}, ssid, \sigma)$ from $\mathcal{F}$ as $\mathcal{H}_j$ is corrupt, give "$\mathcal{M}_i$" input $(\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid)$.

Figure 5.21: Simulator for GAME 7

Setup
Unchanged.
Join

2. `Join Request.` On input $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{M}_i)$ from host $\mathcal{H}_j$.
   - Create a join session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status \leftarrow request$.
   - Output $(\mathsf{JOIN}, \mathsf{sid}, jsid, \mathcal{H}_j)$ to $\mathcal{M}_i$.
3. $\mathcal{M}$ `Join Proceed.` On input $(\mathsf{JOIN}, \mathsf{sid}, jsid)$ from TPM $\mathcal{M}_i$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = request$ to $delivered$.
   - Output $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i, \mathcal{H}_j)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ from $\mathcal{A}$.
   - Output $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid, \mathcal{M}_i)$ to $\mathcal{I}$.
4. $\mathcal{I}$ `Join Proceed.` On input $(\mathsf{JOINPROCEED}, \mathsf{sid}, jsid)$ from $\mathcal{I}$.
   - Update the session record $\langle jsid, \mathcal{M}_i, \mathcal{H}_j, status\rangle$ with $status = delivered$ to $complete$.
   - Output $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid)$ to $\mathcal{A}$ and wait for input $(\mathsf{JOINCOMPLETE}, \mathsf{sid}, jsid, \tau)$ from $\mathcal{A}$.
   - Abort if $\mathcal{I}$ or $\mathcal{M}_i$ is honest and a record $\langle \mathcal{M}_i, *, *\rangle \in$ `Members` already exists.
   - If $\mathcal{H}_j$ is honest, set $\tau \leftarrow \bot$.
   - Else, verify that the provided tracing trapdoor τ is eligible by checking $\mathsf{CheckTtdCorrupt}(\tau) = 1$.
   - Insert $\langle \mathcal{M}_i, \mathcal{H}_j, \tau\rangle$ into `Members` and output $(\mathsf{JOINED}, \mathsf{sid}, jsid)$ to $\mathcal{H}_j$.

Sign
Unchanged.
Verify
Unchanged.
Link
Unchanged.

Figure 5.22: $\mathcal{F}$ for GAME 8

Isolated corrupt TPM
Unchanged.
Setup
Unchanged.
Join
Unchanged.
Sign
Unchanged.

Figure 5.23: Simulator for GAME 8

Setup

Unchanged.

Join

Unchanged.

Sign

5. `Sign Request.` On input $(\mathsf{SIGN}, \mathsf{sid}, ssid, \mathcal{M}_i, m, bsn)$ from $\mathcal{H}_j$.
    - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, * \rangle$ exists in `Members`, abort.
    - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status \rangle$ with $status \leftarrow request$.
    - Output $(\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid, m, bsn)$ to $\mathcal{M}_i$.
6. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid)$ from $\mathcal{M}_i$.
    - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status \rangle$ with $status = request$ and update it to $status \leftarrow complete$.
    - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, * \rangle$ exists in `Members`.
    - Generate the signature for a fresh or established key:
        - Retrieve (gsk, τ) from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau \rangle \in$ `DomainKeys`. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, check $\mathsf{CheckTtdHonest}(\tau) = 1$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau \rangle$ in `DomainKeys`.
        - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn)$, check $\mathsf{ver}(\sigma, m, bsn) = 1$.
    - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j \rangle$ in `Signed` and output $(\mathsf{Signature}, \mathsf{sid}, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

7. `Verify.` On input $(\mathsf{VERIFY}, \mathsf{sid}, m, bsn, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.
    - Set $f \leftarrow 0$ if at least one of the following conditions hold:
        - There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
    - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn)$.
    - Add $\langle \sigma, m, bsn, \mathtt{RL}, f \rangle$ to `VerResults` and output $(\mathsf{VERIFIED}, \mathsf{sid}, f)$ to $\mathcal{V}$.

Link

8. `Link.` On input $(\mathsf{LINK}, \mathsf{sid}, \sigma, m, \sigma', m', bsn)$ from a party $\mathcal{V}$.
    - Output $\perp$ to $\mathcal{V}$ if at least one signature (σ, m, bsn) or (σ', m', bsn) is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
    - Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
    - Output $(\mathsf{LINK}, \mathsf{sid}, f)$ to $\mathcal{V}$.

Figure 5.24: $\mathcal{F}$ for GAME 9

Isolated corrupt TPM
Unchanged.
Setup
Unchanged.
Join
Unchanged.
Sign
Unchanged.

Figure 5.25: Simulator for GAME 9

Setup

Unchanged.

Join

Unchanged.

Sign

5. `Sign Request.` On input $(\mathsf{SIGN}, \mathsf{sid}, ssid, \mathcal{M}_i, m, bsn)$ from $\mathcal{H}_j$.
   - If $\mathcal{H}_j$ is honest and no entry $\langle \mathcal{M}_i, \mathcal{H}_j, * \rangle$ exists in `Members`, abort.
   - Create a sign session record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status \rangle$ with $status \leftarrow request$.
   - Output $(\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid, m, bsn)$ to $\mathcal{M}_i$.
6. `Sign Proceed.` On input $(\mathsf{SIGNPROCEED}, \mathsf{sid}, ssid)$ from $\mathcal{M}_i$.
   - Look up record $\langle ssid, \mathcal{M}_i, \mathcal{H}_j, m, bsn, status \rangle$ with $status = request$ and update it to $status \leftarrow complete$.
   - If $\mathcal{I}$ is honest, check that $\langle \mathcal{M}_i, \mathcal{H}_j, * \rangle$ exists in `Members`.
   - Generate the signature for a fresh or established key:
     - Retrieve (gsk, τ) from $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau \rangle \in \mathtt{DomainKeys}$. If no such entry exists, set $(gsk, \tau) \leftarrow \mathsf{ukgen}()$, check $\mathsf{CheckTtdHonest}(\tau) = 1$, and store $\langle \mathcal{M}_i, \mathcal{H}_j, bsn, gsk, \tau \rangle$ in `DomainKeys`.
     - Compute signature $\sigma \leftarrow \mathsf{sig}(gsk, m, bsn)$, check $\mathsf{ver}(\sigma, m, bsn) = 1$.
     - Check $\mathsf{identify}(\sigma, m, bsn, \tau) = 1$ and that there is no $(\mathcal{M}', \mathcal{H}') \neq (\mathcal{M}_i, \mathcal{H}_j)$ with tracing trapdoor τ' registered in `Members` or `DomainKeys` with $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - Store $\langle \sigma, m, bsn, \mathcal{M}_i, \mathcal{H}_j \rangle$ in `Signed` and output $(\mathsf{Signature}, \mathsf{sid}, ssid, \sigma)$ to $\mathcal{H}_j$.

Verify

7. `Verify.` On input $(\mathsf{VERIFY}, \mathsf{sid}, m, bsn, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.
   - Set $f \leftarrow 0$ if at least one of the following conditions hold:
     - There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn)$.
   - Add $\langle \sigma, m, bsn, \mathtt{RL}, f \rangle$ to `VerResults` and output $(\mathsf{VERIFIED}, \mathsf{sid}, f)$ to $\mathcal{V}$.

Link

8. `Link.` On input $(\mathsf{LINK}, \mathsf{sid}, \sigma, m, \sigma', m', bsn)$ from a party $\mathcal{V}$.
   - Output $\perp$ to $\mathcal{V}$ if at least one signature (σ, m, bsn) or (σ', m', bsn) is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
   - Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
   - Output $(\mathsf{LINK}, \mathsf{sid}, f)$ to $\mathcal{V}$.

Figure 5.26: $\mathcal{F}$ for GAME 10

Isolated corrupt TPM
Unchanged.
Setup
Unchanged.
Join
Unchanged.
Sign
Unchanged.

Figure 5.27: Simulator for GAME 10

Setup
Unchanged.
Join
Unchanged.
Sign
Unchanged.
Verify

7. `Verify`. On input $(\mathsf{VERIFY}, \mathsf{sid}, m, bsn, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.
   - Retrieve all tuples $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, \tau_i \rangle \in$ `Members` and $\langle \mathcal{M}_i, \mathcal{H}_j, *, *, \tau_i \rangle \in$ `DomainKeys` where $\mathsf{identify}(\sigma, m, bsn, \tau_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:
     - More than one τ_i was found.
     - There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn)$.
   - Add $\langle \sigma, m, bsn, \mathtt{RL}, f \rangle$ to `VerResults` and output $(\mathsf{VERIFIED}, \mathsf{sid}, f)$ to $\mathcal{V}$.

Link

8. `Link`. On input $(\mathsf{LINK}, \mathsf{sid}, \sigma, m, \sigma', m', bsn)$ from a party $\mathcal{V}$.
   - Output $\perp$ to $\mathcal{V}$ if at least one signature (σ, m, bsn) or (σ', m', bsn) is not valid (verified via the `verify` interface with $\mathtt{RL} = \varnothing$).
   - Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
   - Output $(\mathsf{LINK}, \mathsf{sid}, f)$ to $\mathcal{V}$.

Figure 5.28: $\mathcal{F}$ for GAME 11

Isolated corrupt TPM
Unchanged.
Setup
Unchanged.
Join
Unchanged.
Sign
Unchanged.

Figure 5.29: Simulator for GAME 11

Setup
Unchanged.
Join
Unchanged.
Sign
Unchanged.
Verify

7. `Verify`. On input ($\mathsf{VERIFY}, \mathsf{sid}, m, bsn, \sigma, \mathtt{RL}$) from some party $\mathcal{V}$.
    - Retrieve all tuples $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, \tau_i \rangle \in$ `Members` and $\langle \mathcal{M}_i, \mathcal{H}_j, *, *, \tau_i \rangle \in$ `DomainKeys` where $\mathsf{identify}(\sigma, m, bsn, \tau_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:
        - More than one τ_i was found.
        - $\mathcal{I}$ is honest and no tuple $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found.
        - There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
    - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn)$.
    - Add $\langle \sigma, m, bsn, \mathtt{RL}, f \rangle$ to `VerResults` and output ($\mathsf{VERIFIED}, \mathsf{sid}, f$) to $\mathcal{V}$.

Link

8. `Link`. On input ($\mathsf{LINK}, \mathsf{sid}, \sigma, m, \sigma', m', bsn$) from a party $\mathcal{V}$.
    - Output $\perp$ to $\mathcal{V}$ if at least one signature (σ, m, bsn) or (σ', m', bsn) is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
    - Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
    - Output ($\mathsf{LINK}, \mathsf{sid}, f$) to $\mathcal{V}$.

Figure 5.30: $\mathcal{F}$ for GAME 12

Isolated corrupt TPM
Unchanged.
Setup
Unchanged.
Join
Unchanged.
Sign
Unchanged.

Figure 5.31: Simulator for GAME 12

Setup

Unchanged.

Join

Unchanged.

Sign

Unchanged.

Verify

7. `Verify.` On input $(\mathsf{VERIFY}, \mathsf{sid}, m, bsn, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.
   - Retrieve all tuples $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, \tau_i \rangle \in$ `Members` and $\langle \mathcal{M}_i, \mathcal{H}_j, *, *, \tau_i \rangle \in$ `DomainKeys` where $\mathsf{identify}(\sigma, m, bsn, \tau_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:
     - More than one τ_i was found.
     - $\mathcal{I}$ is honest and no tuple $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found.
     - $\mathcal{M}_i$ or $\mathcal{H}_j$ is honest but no entry $\langle *, m, bsn, \mathcal{M}_i, \mathcal{H}_j \rangle \in$ `Signed` exists.
     - There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$.
   - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn)$.
   - Add $\langle \sigma, m, bsn, \mathtt{RL}, f \rangle$ to `VerResults` and output $(\mathsf{VERIFIED}, \mathsf{sid}, f)$ to $\mathcal{V}$.

Link

8. `Link.` On input $(\mathsf{LINK}, \mathsf{sid}, \sigma, m, \sigma', m', bsn)$ from a party $\mathcal{V}$.
   - Output $\perp$ to $\mathcal{V}$ if at least one signature (σ, m, bsn) or (σ', m', bsn) is not valid (verified via the `verify` interface with $\mathtt{RL} = \varnothing$).
   - Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
   - Output $(\mathsf{LINK}, \mathsf{sid}, f)$ to $\mathcal{V}$.

Figure 5.32: $\mathcal{F}$ for GAME 13

Isolated corrupt TPM

Unchanged.

Setup

Unchanged.

Join

Unchanged.

Sign

Unchanged.

Figure 5.33: Simulator for GAME 13

Setup

Unchanged.

Join

Unchanged.

Sign

Unchanged.

Verify

7. `Verify`. On input $(\mathsf{VERIFY}, \mathsf{sid}, m, bsn, \sigma, \mathtt{RL})$ from some party $\mathcal{V}$.
   - Retrieve all tuples $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ from $\langle \mathcal{M}_i, \mathcal{H}_j, \tau_i \rangle \in$ `Members` and $\langle \mathcal{M}_i, \mathcal{H}_j, *, *, \tau_i \rangle \in$ `DomainKeys` where $\mathsf{identify}(\sigma, m, bsn, \tau_i) = 1$. Set $f \leftarrow 0$ if at least one of the following conditions hold:
     - More than one τ_i was found.
     - $\mathcal{I}$ is honest and no tuple $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ was found.
     - $\mathcal{M}_i$ or $\mathcal{H}_j$ is honest but no entry $\langle *, m, bsn, \mathcal{M}_i, \mathcal{H}_j \rangle \in$ `Signed` exists.
     - There is a $\tau' \in \mathtt{RL}$ where $\mathsf{identify}(\sigma, m, bsn, \tau') = 1$, and no pair $(\tau_i, \mathcal{M}_i, \mathcal{H}_j)$ for an honest $\mathcal{H}_j$ was found.
   - If $f \neq 0$, set $f \leftarrow \mathsf{ver}(\sigma, m, bsn)$.
   - Add $\langle \sigma, m, bsn, \mathtt{RL}, f \rangle$ to `VerResults` and output $(\mathsf{VERIFIED}, \mathsf{sid}, f)$ to $\mathcal{V}$.

Link

8. `Link`. On input $(\mathsf{LINK}, \mathsf{sid}, \sigma, m, \sigma', m', bsn)$ from a party $\mathcal{V}$.
   - Output $\perp$ to $\mathcal{V}$ if at least one signature (σ, m, bsn) or (σ', m', bsn) is not valid (verified via the `verify` interface with $\mathtt{RL} = \emptyset$).
   - Set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
   - Output $(\mathsf{LINK}, \mathsf{sid}, f)$ to $\mathcal{V}$.

Figure 5.34: $\mathcal{F}$ for GAME 14

Isolated corrupt TPM

Unchanged.

Setup

Unchanged.

Join

Unchanged.

Sign

Unchanged.

Figure 5.35: Simulator for GAME 14

Setup
Unchanged.
Join
Unchanged.
Sign
Unchanged.
Verify
Unchanged.
Link

8. `Link.` On input (LINK, sid, $\sigma, m, \sigma', m', bsn$) from a party $\mathcal{V}$.
    - Output $\perp$ to $\mathcal{V}$ if at least one signature (σ, m, bsn) or (σ', m', bsn) is not valid (verified via the `verify` interface with $\texttt{RL} = \varnothing$).
    - For each τ_i in `Members` and `DomainKeys` compute $b_i \leftarrow \mathsf{identify}(\sigma, m, bsn, \tau_i)$ and $b'_i \leftarrow \mathsf{identify}(\sigma', m', bsn, \tau_i)$ and do the following:
        - Set $f \leftarrow 0$ if $b_i \neq b'_i$ for some i.
        - Set $f \leftarrow 1$ if $b_i = b'_i = 1$ for some i.
    - If f is not defined yet, set $f \leftarrow \mathsf{link}(\sigma, m, \sigma', m', bsn)$.
    - Output $(\mathsf{LINK}, \mathsf{sid}, f)$ to $\mathcal{V}$.

Figure 5.36: $\mathcal{F}$ for GAME 15

Isolated corrupt TPM
Unchanged.
Setup
Unchanged.
Join
Unchanged.
Sign
Unchanged.

Figure 5.37: Simulator for GAME 15

We now show that every game hop is indistinguishable from the previous. Note that although we separate $\mathcal{F}$ and $\mathcal{S}$, in reductions we can consider them to be one entity, as this does not affect $\mathcal{A}$ and $\mathcal{E}$.

Game 1: This is the real world.

Game 2: We let the simulator $\mathcal{S}$ receive all inputs and generate all outputs. It does so by simulating all honest parties honestly. It simulates the oracles honestly, except that it chooses encryption keys in the crs of which it knows corresponding secret keys, allowing it to decrypt messages encrypted to the crs. Clearly, this is equal to the real world.

Game 3: We now start creating a functionality $\mathcal{F}$ that receives inputs from honest parties and generates the outputs for honest parties. It works together with a simulator $\mathcal{S}$. In this game, we simply let $\mathcal{F}$ forward all inputs to $\mathcal{S}$, who acts as before. When $\mathcal{S}$ would generate an output, it first forwards it to $\mathcal{F}$, who then outputs it. This game hop simply restructures GAME 2, we have GAME 3 = GAME 2.

Game 4: $\mathcal{F}$ now handles the setup queries, and lets $\mathcal{S}$ enter algorithms that $\mathcal{F}$ will store. $\mathcal{F}$ checks the structure of sid, and aborts if it does not have the expected structure. This does not change the view of $\mathcal{E}$, as $\mathcal{I}$ in the protocol performs the same check, giving GAME 4 = GAME 3.

Game 5: $\mathcal{F}$ now handles the verify and link queries using the algorithms that $\mathcal{S}$ defined in GAME 4. In GAME 4, $\mathcal{S}$ defined the ver algorithm as the real world with the revocation check ommitted. As $\mathcal{F}$ performs this check separately. The link algorithm is equal to the real world algorithm, showing that using these algorithms does not change the verification or linking outcome, so GAME 5 = GAME 4.

Game 6: We now let $\mathcal{F}$ handle the join queries. $\mathcal{S}$ receives enough information from $\mathcal{F}$ to correctly simulate the real world protocol. Only when a join query with honest issuer and corrupt TPM and host takes place, $\mathcal{S}$ misses some information. It must make a join query with $\mathcal{F}$ on the host's behalf, but it does not know the identity of the host. However, it is sufficient to choose an arbitrary corrupt host. This results in a different host registered in `Members`, but $\mathcal{F}$ will not use this information when the registered host is corrupt. Since $\mathcal{S}$ can always simulate the real world protocol, we have GAME 6 = GAME 5.

Game 7: $\mathcal{F}$ now handles the sign queries. When one party creates two signatures with different basenames, $\mathcal{F}$ signs with different keys, showing that the signatures are unlinkable. $\mathcal{S}$ can simulate the real world protocol and block any signatures that would not be successfully generated in the real world. $\mathcal{F}$ may prevent a signature from being output, when the TPM and host did not yet join, or when the signature generated by $\mathcal{F}$ does not pass verification. If the TPM and host did not join, and the host is honest, the real world would also not output a signature, as the host performs this check. The signatures $\mathcal{F}$ generate will always pass verification, as the algorithms that $\mathcal{S}$ set in GAME 4 will only create valid signatures (by completeness of the split signatures, signatures on encrypted messages, and zero-knoweldge proofs). This shows that $\mathcal{F}$ outputs a signature if and only if the real world would outputs a signature.

What remains to show is that the signatures that $\mathcal{F}$ outputs are indistinguishable from the real world signatures. We make this change gradually. First, all signatures come from the real world, and then we let $\mathcal{F}$ gradually create more signatures, until all signatures come from $\mathcal{F}$. Let GAME $7.i.j$ denote the game in which $\mathcal{F}$ creates all signatures for platforms with TPMs $\mathcal{M}_{i'}$ with $i' < i$, lets $\mathcal{S}$ create the signatures if $i' > i$, and for the platform with TPM $\mathcal{M}_i$, the first j distinct basenames are signed. We show that GAME $7.i.j$ is indistinguishable from GAME $7.i.(j+1)$, and by repeating this argument, we have GAME $7 \approx$ GAME 6.

Proof of Game $7.i.j \approx$ Game $7.i.(j+1)$ We make small changes to GAME $7.i.j$ and GAME $7.i.(j+1)$, and then show that the remaining difference can be reduced to the key hiding property of the split signatures.

As we are in a reduction, where we play the key hiding game with a challenger, and we have access to some local random oracle RO_i. $\mathcal{G}_{\mathsf{sRO}}$ is simulated, meaning we are free to observe and simulate, except that we need to keep $\mathcal{G}_{\mathsf{sRO}}(i, \cdot)$ in sync with RO_i. We simulate $\mathcal{G}_{\mathsf{sRO}}(i, m)$ by quering $h \leftarrow RO_i(m)$ and using $\mathsf{Embed}^{-1}(h)$ as $\mathcal{G}_{\mathsf{sRO}}$'s output.

First, we let the NIZK proofs in join and in the signatures be simulated (as we simulate the random oracle), which is indistinguishable by the zero-knowledge property of the proofs. Second, we encrypt dummy values in join and sign, instead of encrypting *cred* and *gpk*. Under the

CPA security of the encryption scheme, this is indistinguishable.[4] Note that the host cannot decrypt his credential while reducing to the CPA security, which means he cannot verify the credential and he cannot later use it to sign. Proof $\pi_{\mathsf{JOIN},\mathcal{I}}$ guarantees that the encrypted credential is valid, so it still aborts when the issuer tries to send a invalid credential. The simulator simulating the honest host can solve the second problem: since GAME 4, the simulator knows the issuer secret key and can therefore create an equivalent credential.

Now, the only remaning difference is the computation of *tag* and *nym*. In GAME $7.i.j$, $\mathcal{S}$ computes these values using the same key as it joined with, and in GAME $7.i.(j+1)$, $\mathcal{F}$ uses a fresh key.

We first show that the difference in *nym* is indistinguishable under the key hiding property of the split signatures. $\mathcal{S}$ simulates the honest host without knowing *gpk*. In the join, it uses a dummy ciphertext and simulates the proof. Signatures with basename $bsn_{j'}$ are handled as follows.

- $j' \leq j$: these signatures are created by $\mathcal{F}$.
- $j' = j + 1$: $\mathcal{S}$ gives the challenger of the key hiding game of split signatures message $bsn_{j'}$, giving it the pseudonym for $bsn_{j'}$. As the split signatures are unique, we can use this pseudonym for every signature with $bsn_{j'}$.
- $j' > j + 1$: $\mathcal{S}$ uses $\mathcal{O}^{\mathsf{CompleteSign}}$ to compute *tag* and *nym*.

If the bit in the key hiding game is zero, *nym* is computed like in GAME $7.i.j$, and if one, *nym* is computed like in GAME $7.i.(j+1)$, so any environment distinguishing the different ways to compute *nym* can break the key hiding property of the split signatures.

What remains to show is that using a fresh key for every basename in the computation of *tag* is also indistinguishable. Here we make the same reduction to the key hiding property of split signatures, but now we make a reduction per message that the platform signs with this basename.

Game 8: $\mathcal{F}$ now runs the CheckTtdCorrupt algorithm when $\mathcal{S}$ gives the extracted *gpk* from platforms with a corrupt host. This checks

---

[4] Note that $\mathcal{S}$ previously held the trapdoor to the crs encryption key. $\mathcal{S}$ only uses this to extract *gpk* in the join and gives it to $\mathcal{F}$. Since $\mathcal{F}$ does not use this extracted value yet, we can omit these extractions here, and use the CPA property of the encryption scheme.

that $\mathcal{F}$ has not seen valid signatures yet that match both this key and existing key. If this happens, we break the key-uniqueness property of the split signatures, so Game 8 $\approx$ Game 7.

Game 9: When $\mathcal{F}$ creates fresh domain keys when signing for honest platforms, it checks that there are no signatures that match this key. Since $\mathcal{S}$ instantiated the $\mathsf{identify}$ algorithm with the verification algorithm of the split signatures, this would mean there already exists a valid signature under the freshly generated key. Clearly, this breaks the unforgeability-1 property of the split signatures, so Game 9 $\approx$ Game 8.

Game 10: $\mathcal{F}$ now performs additional tests on the signatures it creates, and if any fails, it aborts. First, it checks whether the generated signature matches the key it was generated with. With the algorithms $\mathcal{S}$ defined in Game 4, this always holds. Second, $\mathcal{F}$ checks that there is no other platform with a key that matches this signature. We can reduce this check occuring to the key-hiding property of SSIG using a hybrid argument. In Game 10.i, $\mathcal{F}$ performs this check for the first i entries in `DomainKeys`.

The proof of Game 7 shows that signing under a different key is indistinguishable, meaning that the environment gains no information on $\tau = spk$ and we only have to worry about collisions. As any unforgeable split-signature scheme must have an exponentially large key space, the chance that a collission occurs is negligible.

Game 11: In verification, $\mathcal{F}$ now checks whether it knows multiple tracing keys that match one signature. As $\mathcal{S}$ instantiated the $\mathsf{identify}$ with the verification of split signatures, this cannot happen with non-neglibible probability by the key-uniqueness property of the split signatures, Game 11 $\approx$ Game 10.

Game 12: When $\mathcal{I}$ is honest, $\mathcal{F}$ verifying a signature now checks whether the signature matches some key of a platform that joined, and if not, rejects the signature. Under the unforgeability of the signature scheme for encrypted messages, this check will trigger only with negligible probability.

When reducing to the unforgeability of the signature scheme for encrypted messages, we do not know the issuer secret key isk. $\mathcal{S}$ simulating $\mathcal{I}$ therefore simulates proof π in the public key of the issuer. When $\mathcal{S}$ must create a credential while simulating the join protocol, it now uses the signing oracle. From C_2, it can extract gpk using its knowledge of the crs trapdoor. It passes gpk to the signing oracle, along with the ephemeral encryption key epk, which allows simulation without knowing isk. $\mathcal{F}$'s algorithms used to be based on the issuer secret key, which we do not know in this reduction. We let sig now also use the signing oracle. Instead of encrypting gpk with epk, it passes these two values to the signing oracle, and continues as before. Note that any gpk we pass to the signing oracle is stored in `Members` or `DomainKeys`. Now, when we see a valid signature that does not match any of the gpk values stored, we can extract a forgery: Signatures have structure $(tag, nym, \pi_{\mathsf{SIGN}})$, with

$$\pi_{\mathsf{SIGN}} \leftarrow NIZK\{(gpk, cred) : \mathsf{ESIG.Vf}(ipk, cred, gpk) = 1 \wedge \\ \mathsf{SSIG.Vf}(gpk, tag, (0, m, bsn)) = 1 \wedge \mathsf{SSIG.Vf}(gpk, nym, (1, bsn)) = 1\}$$

If the signature does not match any of the keys (using the $\mathsf{identify}$ algorithm), it means that nym is not a valid split signature under any of the gpk values for which an oracle query has been made. By soundness of the proof, $\mathcal{S}$ can extract a credential on the gpk value used, which will be a forgery. Note that as we perform this extraction only in reductions, online extractability is not required.

As the signature scheme for encrypted messages is unforgeable, we have Game 12 $\approx$ Game 11.

Game 13: $\mathcal{F}$ now rejects signatures on message m with basename bsn that match the key of a platform with an honest TPM or honest host, but that platform never signed m w.r.t. bsn. If signatures that would previously have been accepted are now no longer accepted, we can break the unforgeability of the split signatures.

We distinguish two cases: the host is honest, which means gpk is found in `DomainKeys` (as for honest hosts, we do not register a τ value in `Members`), or the TPM is honest and the host is corrupt, which means the matching key is found in `Members`. the matching key gpk is found in `Members` and the host is honest, the matchking key is found in `Members` and the host is corrupt, or gpk is found in `DomainKeys`.

[Case 1 – *gpk* in `DomainKeys`, honest host]. Let Game 13.i.j denote the game in which $\mathcal{F}$ prevents forgeries for keys in `DomainKeys` of the platform with TPM $\mathcal{M}_{i'}$ and $i' < i$, and prevents forgery under the keys in domainkeys with $bsn_{j'}$, $j' < j$ of the platform with TPM $\mathcal{M}_i$ lets $\mathcal{S}$ create the signatures if $i' > i$, and for the platform with TPM $\mathcal{M}_i$, the first j distinct basenames are signed. We show that GAME 13.i.j is indistinguishable from GAME 13.i.$(j+1)$ under unforgeability-1 of the split signatures.

$\mathcal{S}$ receives the system parameters, which it puts in the crs. $\mathcal{S}$ now changes the algorithms it gives to $\mathcal{F}$, such that on input bsn_j, it runs $(ppk, tsk) \leftarrow \mathsf{PreKeyGen}(spar)$ and gives ppk to the challenger. $\mathcal{S}$ receives gpk, for which it does not know the full secret key. When $\mathcal{F}$ wants to sign using gpk, it must create tag and nym without knowing the second part of the secret key. It creates the pre-signature using tsk, and completes the signature using $\mathcal{O}^{\mathsf{CompleteSign}}$. Now, when $\mathcal{F}$ notices a signature on message m w.r.t. basename bsn that the platform never signed, it means it did not query $\mathcal{O}^{\mathsf{CompleteSign}}$ on $(0, m, bsn)$, so we can extract tag which is a forgery on $(0, m, bsn)$.

[Case 2 – gpk in `Members`, honest TPM, corrupt host]. We make this change gradually, for each TPM $\mathcal{M}_i$ individually.

$\mathcal{S}$ receives the system parameters, which it puts in the crs. When $\mathcal{S}$ simulates $\mathcal{M}_i$ joining, instead of running $\mathsf{PreKeyGen}$, it uses the ppk as received from the challenger. When $\mathcal{S}$ simulating the issuer receives gpk and π_1 from the platform with $\mathcal{M}_i$, it extracts hsk such that $\mathsf{VerKey}(spar, ppk, spk, hsk) = 1$. Observe that we do not need online extractability of hsk, as in this reduction we extract from just one proof, and rewinding would be acceptable. Whenever $\mathcal{S}$ must pre-sign using the unknown tsk, it calls $\mathcal{O}^{\mathsf{PreSign}}$. When $\mathcal{F}$ sees a signature matching this platform's key gpk on message m w.r.t. basename bsn that $\mathcal{M}_i$ never signed, extract tag, which is a valid signature on $(0, m, bsn)$ under gpk. Now the unforgeability-2 game is won by submitting $((0, m, bsn), tag, gpk, hsk)$.

Game 14: $\mathcal{F}$ now prevents revocation of platforms with an honest host. Note that revocation requires a gpk value of the platform to be placed on the revocation list. We now show that no environment has nonnegligible probability of entering these values.

For platforms with an honest host, we can remove all information on gpk. First, when we encrypt gpk, tag, or $cred$, we encrypt dummy values instead and simulate the proofs. Second, we can replace the nym values by signatures under different keys, by the key hiding property of the split signatures. Now, the environment must simply guess gpk. As any unforgeable split-signature scheme must have an exponentially large key space, the chance that a collission occurs is negligible.

Game 15: $\mathcal{F}$ answering linking queries now uses its tracing information to answer the queries. Previously, it compared nym and nym', valid split signatures on bsn under keys gpk and gpk' respectively. If $nym = nym'$, $\mathcal{F}$ answered 1, and otherwise 0.

$\mathcal{F}$ now takes all the gpk values it knows and if it finds some gpk such that one nym is a valid split signature under gpk, but nym' is not, it outputs that the signatures are not linked. Clearly, in this case we must have $nym \neq nym'$, so the linking decision does not change. If $\mathcal{F}$ finds some gpk such that both nym and nym' are valid signatures on bsn under gpk, it outputs that the signatures are linked. By signature uniqueness, we have $nym = nym'$, so again, the linking decision does not change. This shows GAME 15 $\approx$ GAME 14. □

5.7 Concrete Instantiation and Efficiency

In this section we describe on a high level how to efficiently instantiate the generic building blocks to instantiate our generic DAA scheme presented in Section 5.6.

The split signature scheme is instantiated with the split-BLS signatures (as described in Section 5.5.3), the signatures for encrypted messages with the AGOT+ signature scheme (as described in Section 5.5.2) and the encryption scheme with ElGamal, both working in $\mathbb{G}_2$. All the zero-knowledge proofs are instantiated with non-interactive Schnorr-type proofs about discrete logarithms, and witnesses that have to be online extractable are encrypted using ElGamal for group elements and Camenisch-Shoup encryption [CS03] for exponents. Note that the latter is only used by the issuer to prove that its key is correctly formed, i.e., every participant will only work with Camenisch-Shoup ciphertexts once. The shared system parameters $spar$ then consist of a security parameter κ, a bilinear group $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t$ of prime order q with generators g_1 and g_2 and bilinear map e, as generated by $\mathsf{PairGen}(1^\kappa)$. Further, the system parameters contain an additional random group element $x \xleftarrow{\$} \mathbb{G}_2$ for the AGOT+ signature and an ElGamal encryption key $epk_{\mathsf{crs}} \xleftarrow{\$} \mathbb{G}_2$. This crs-key allows for efficient online extractability in the security proof, as the simulator will be privy of the corresponding secret key. Finally, let $\mathsf{H} : \{0,1\}^* \rightarrow \mathbb{G}_1^*$ be a hash function, that we model as a random oracle in the security proof.

Setup. The issuer registers the AGOT+ key $ipk = g_1^{isk}$ along with a proof π_{ipk} that ipk is well-formed. For universal composition, we need isk to be online-extractable, which can be achieved by verifiable encryption. To this end, we let the crs additionally contain a public key $(\mathsf{n}, \mathsf{y}, \mathsf{g}, \mathsf{h})$ for the CPA version of the Camenisch-Shoup encryption scheme and an additional element g to make the verifiable encryption work [CS03]. We thus instantiate the proof

$$\pi_{ipk} \leftarrow \mathsf{NIZK}\{(\boxed{isk}) : (ipk, isk) \in \mathsf{ESIG.SigKGen}(spar)\}(\mathsf{sid})$$

as follows:

$$\begin{aligned}\pi_{ipk} \leftarrow \mathsf{SPK}\{(isk, r) : ipk = g_1^{isk} \ \wedge \ \hat{\mathsf{g}}^r \mathsf{g}^{isk} \bmod \mathsf{n} \wedge \\ \mathsf{g}^r \bmod \mathsf{n} \ \wedge \ \mathsf{y}^r \mathsf{h}^{isk} \bmod \mathsf{n} \ \wedge \ isk \in [-\mathsf{n}/4, \mathsf{n}/4]\}(\mathsf{sid})\end{aligned}$$

Join. Using the split-BLS signature, the TPM has a secret key $tsk \in \mathbb{Z}_q^*$ and public key $tpk = g_2^{tsk}$, the host has secret key $hsk \in \mathbb{Z}_q^*$, and together they have created the public key $gpk = g_2^{tsk \cdot hsk}$.

We now show how to instantiate the proof $\pi_{\mathsf{JOIN},\mathcal{H}}$ where the host proves that C is an encryption of a correctly derived gpk. Recall that the issuer receives the M_i's public key contribution tpk authenticated from the TPM.

$$\begin{aligned}\pi_{\mathsf{JOIN},\mathcal{H}} \leftarrow \mathsf{NIZK}\{(\boxed{gpk}, hsk) : C \in \mathsf{Enc}(epk, gpk) \ \wedge \\ \mathsf{SSIG.VerKey}(tpk, gpk, hsk) = 1\}(\mathsf{sid}, jsid).\end{aligned}$$

The joint public key gpk is encrypted under an ephemeral key epk using ElGamal with crs trapdoor epk_{crs}. We set $\rho \xleftarrow{\$} \mathbb{Z}_q$, $C_1 \leftarrow epk_{\mathsf{crs}}^{\rho}$, $C_2 \leftarrow epk^{\rho}$, $C_3 \leftarrow g_2^{\rho} \cdot gpk$ and prove:

$$\begin{aligned}\pi'_{\mathsf{JOIN},\mathcal{H}} \leftarrow \mathsf{SPK}\{(hsk, \rho) : C_1 = epk_{\mathsf{crs}}^{\rho} \ \wedge \\ C_2 = epk^{\rho} \ \wedge \ C_3 = g_2^{\rho} \cdot tpk^{hsk}\}(\mathsf{sid}, jsid).\end{aligned}$$

The host sets $\pi_{\mathsf{JOIN},\mathcal{H}} \leftarrow (C_1, C_2, C_3, \pi'_{\mathsf{JOIN},\mathcal{H}})$ as the final proof. Note that gpk is online-extractable as it is encrypted under epk_{crs}. The issuer checks $tpk \neq 1_{\mathcal{G}_2}$ and verifies $\pi'_{\mathsf{JOIN},\mathcal{H}}$.

Next, the issuer places an AGOT+ signature on gpk. Since $gpk \in \mathbb{G}_2$, the decrypted credential has the form (r, s, t, w) which is an element of $\mathbb{G}_1 \times \mathbb{G}_2^3$. The issuer computes the credential on ciphertext (C_1, C_2, C_3) as follows: Choose a random $u, \rho_1, \rho_2 \xleftarrow{\$} \mathbb{Z}_q^*$, and compute the (partially) encrypted signature $\bar{\sigma} = (r, (S_1, S_2, S_3), (T_1, T_2, T_3), w)$:

$$\begin{aligned} r &\leftarrow g_2^u, & S_1 &\leftarrow C_2^{v/u} epk^{\rho_1}, & S_2 &\leftarrow (C_3^v x)^{1/u} g_2^{\rho_1}, \\ T_1 &\leftarrow S_2^{v/u} epk^{\rho_2}, & T_2 &\leftarrow (S_2^v g_2)^{1/u} g_2^{\rho_2}, & w &\leftarrow g_2^{1/u}. \end{aligned}$$

Then, with $\pi_{\mathsf{JOIN},\mathcal{I}}$ it proves that it signed the ciphertext correctly:

$$\begin{aligned} \pi_{\mathsf{JOIN},\mathcal{I}} \leftarrow \mathsf{NIZK}\{ isk : cred' \in \mathsf{ESIG.EncSign}(isk, epk, C) \wedge \\ (ipk, isk) \in \mathsf{ESIG.SigKGen}(spar) \}(\mathsf{sid}, jsid). \end{aligned}$$

To instantiate this, we let the issuer create $\pi'_{\mathsf{JOIN},\mathcal{I}}$ as follows, using witness $u' = \frac{1}{u}$ and $isk' = \frac{isk}{u}$:

$$\begin{aligned} \pi'_{\mathsf{JOIN},\mathcal{I}} \leftarrow \mathsf{SPK}\{ (u', isk', \rho_1, \rho_2) : g_2 = r^{u'} \ \wedge \ S_1 = C_2^{isk'} epk^{\rho_1} \ \wedge \\ S_2 = C_3^{isk'} x^{u'} g_2^{\rho_1} \ \wedge \ T_1 = S_1^{isk'} epk^{\rho_2} \ \wedge \ T_2 = S_2^{isk'} g_2^{u'} g_2^{\rho_2} \ \wedge \ w = g_2^{u'} \ \wedge \\ 1 = ipk^{-isk'} g_1^{u'} \}(\mathsf{sid}, jsid). \end{aligned}$$

The issuer outputs $\pi_{\mathsf{JOIN},\mathcal{I}} = (r, S_1, S_2, T_1, T_2, w, \pi'_{\mathsf{JOIN},\mathcal{I}})$.

Sign. In our concrete instantiation, signatures on messages and basenames are split-BLS signatures, i.e., the TPM and host jointly compute BLS signatures $tag \leftarrow \mathsf{H}(0, m, bsn)^{tsk \cdot hsk}$ and $nym \leftarrow \mathsf{H}(1, bsn)^{tsk \cdot hsk}$. Recall that we cannot reveal the joint public key gpk or the credential $cred$. Instead the host provides the proof π_{SIGN} that tag and nym are valid split signatures under public key gpk and that it owns a valid issuer credential $cred$ on gpk, without disclosing gpk and $cred$:

$$\begin{aligned} \pi_{\mathsf{SIGN}} \leftarrow \mathsf{NIZK}\{ (gpk, cred) : \mathsf{ESIG.Vf}(ipk, cred, gpk) = 1 \wedge \\ \mathsf{SSIG.Vf}(gpk, tag, (0, m, bsn)) = 1 \wedge \mathsf{SSIG.Vf}(gpk, nym, (1, bsn)) = 1 \} \end{aligned}$$

This proof can be realized as follows: First, the host randomizes the AGOT+ credential (r, s, t, w) to (r', s', t', w) using the randomization token w. Note that this randomization allows the host to release r' (instead of encrypting it) without becoming linkable. The host then proves knowledge of the rest of the credential and *gpk*, such that the credential is valid under the issuer public key and signs *gpk*, that *tag* is a valid split-BLS signature on $(0, m, bsn)$ under *gpk*, and that *nym* is a valid split-BLS signature on $(1, bsn)$ under *gpk*. It computes the following proof:

$$\begin{aligned}
&\pi'_{\mathsf{SIGN}} \leftarrow \mathsf{SPK}\{(gpk, s', t') : \\
&e(g_1, x) = e(r', s')e(V^{-1}, gpk) \ \wedge \ e(g_1, g_2) = e(r', t')e(V^{-1}, gpk) \ \wedge \\
&e(tag, g_2) = e(\mathsf{H}(0, m, bsn), gpk) \ \wedge \ e(nym, g_2) = e(\mathsf{H}(1, bsn), gpk)\}
\end{aligned}$$

The host finally sets $\pi_{\mathsf{SIGN}} \leftarrow (r', \pi'_{\mathsf{SIGN}})$.

Verify. A verifier receiving $(tag, nym, \pi_{\mathsf{SIGN}})$ verifies π'_{SIGN} and checks $nym \neq 1_{\mathbb{G}_1}$ and $tag \neq 1_{\mathbb{G}_1}$.

Embedding functions for $\mathcal{G}_{\mathsf{sRO}}$. In addition to secure instantiations of the primitives, Theorem 10 states that we also need Embed functions for any type of random oracle used by the primitives.

The verifiable encryption by Camenisch and Shoup requires a random oracle mapping to $\{0,1\}^k$. If we have $\mathcal{G}_{\mathsf{sRO}}$ output $\{0,1\}^{\ell(\kappa)}$ such that $\ell(\kappa)$ is much larger than k, we can let Embed simply reduce modulo 2^k, and $h \xleftarrow{\$} \mathsf{Embed}^{-1}(m)$ lets m define the k least-significant bits of h, and chooses the $\ell(\kappa) - k$ most significant bits uniformly at random.

For the SPK proofs, we use a random oracle mapping to $\mathbb{Z}_q$, allowing us to make similar Embed and Embed^{-1} functions by letting $\{0,1\}^{\ell(\kappa)}$ to be exponentially larger than $\mathbb{Z}_q$. We let Embed simply reduce modulo q, and $\mathsf{Embed}^{-1}(m)$ takes a value r uniformly at random in $\mathbb{Z}_{2^{\ell(\kappa)}-q}$ and returns $r + m$ (computed over the integers).

Finally, split-BLS uses a random oracle mapping to $\mathbb{G}_1^*$. Without making assumptions on the groups that PairGen generates, it seems impossible to construct appropriate Embed functions. In practice, however, $\mathbb{G}_1$ is typically an elliptic curve, and appropriate Embed and Embed^{-1} functions are described in [BLS04, Section 3.2].

5.7.1 Security

When using the concrete instantiations as presented above we can derive the following corollary from Theorem 10 and the required security assumptions of the deployed building blocks. We have opted for a highly efficient instantiation of our scheme, which comes for the price of stronger assumptions such as the generic group (for AGOT+ signatures) and random oracle model (for split-BLS signatures and Fiat-Shamir NIZKs). We would like to stress that our generic scheme based on abstract building blocks, presented in Section 5.6, does not require either of the models, and one can use less efficient instantiations to avoid these assumptions.

Corollary 1. *Our protocol* Π_{pdaa} *described in Section 5.6 and instantiated as described above, GUC-realizes* $\mathcal{F}_{\mathsf{pdaa}}$ *in the* $(\mathcal{F}_{\mathsf{auth}*}, \mathcal{F}_{\mathsf{ca}}, \mathcal{F}_{\mathsf{crs}}, \mathcal{G}_{\mathsf{sRO}})$*-hybrid model under the following assumptions:*

Primitive	*Instantiation*	*Assumption*
SSIG	*split-BLS*	*co-CDH, XDH, ROM*
ESIG	*AGOT+*	*generic group model*
ENC	*ElGamal*	*SXDH*
NIZK	*ElGamal, Fiat-Shamir, Camenisch-Shoup*	*SXDH, Strong RSA [FO97], ROM*

5.7.2 Efficiency

We now give an overview of the efficiency of our protocol when instantiated as described above. Our analysis focuses on signing and verification, which will be used the most and thus have the biggest impact on the performance of the scheme. We now discuss the efficiency of our protocol when instantiated as described above. Our analysis focuses on the signing protocol and verification, which will be used the most and thus have the biggest impact on the performance of the scheme.

TPM. Given the increased "responsibility" of the host, our protocol is actually very lightweight on the TPM's side. When signing, the TPM only performs two exponentiations in $\mathbb{G}_1$. In fact, according to the efficiency overview by Camenisch et al. [CDL16a], our scheme has the most efficient signing operation for the TPM to date. Since the TPM is typically orders of magnitude slower than the host, minimizing the TPM's workload is key to achieve an efficient scheme.

Host. The host performs more tasks than in previous DAA schemes, but remains efficient. The host runs CompleteSign twice, which costs 4 pairings and 2 exponentiations in $\mathbb{G}_1$. Next, it constructs π_{SIGN}.

This involves randomizing the AGOT credential, which costs 1 exponentiation in $\mathbb{G}_1$ and 3 in $\mathbb{G}_2$. It then constructs π'_{SIGN}, which costs 3 exponentiations in $\mathbb{G}_2$ and 6 pairings. This results in total signing cost of $3\mathbb{G}_1, 6\mathbb{G}_2, 10P$ for a host.

Verifier. The verification checks the validity of $(tag, nym, \pi_{\mathsf{SIGN}})$ by verifying π'_{SIGN}. Computing the left-hand sides of the euqations in π'_{SIGN} costs two pairings, as $e(g_1, g_2)$ and $e(g_1, x)$ can be precomputed. Verifying the rest of the proof costs 6 pairings and 4 exponentiations in $\mathbb{G}_\mathrm{t}$. The revocation check with a revocation list of n elements costs $n+1$ pairings.

Estimated Performance. We measured the speed of the Apache Milagro Cryptographic Library (AMCL)[5] and found that exponentiations in $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_\mathrm{t}$ require 0.6ms, 1.0ms, and 1.4ms respectively. A pairing costs 1.6ms. Using these numbers, we estimate a signing time of 23.8ms for the host, and a verification time of 18.4ms, showing that also for the host our protocol is efficient enough to be used in practice. Table 5.1 gives an overview of the efficiency of our concrete instantiation.

	$\mathcal{M}$ Sign	$\mathcal{H}$ Sign	Verify
Operations	$2\mathbb{G}_1$	$3\mathbb{G}_1$, $6\mathbb{G}_2$, $10P$	$4\mathbb{G}_\mathrm{t}$, $8P$
Est. Time		23.8ms	18.4ms

Table 5.1: Efficiency of our concrete DAA scheme ($n\mathbb{G}$ indicates n exponentiations in group $\mathbb{G}$, and nP indicates n pairing operations).

---

[5] See `https://github.com/miracl/amcl`. We used the C-version of the library, configured to use the BN254 curve. The program `benchtest_pair.c` has been used to retrieve the timings, executed on an Intel i5-4300U CPU.

Chapter 6

Concluding Remarks

Authentication is a key aspect of digital security, but revealing too much information while authenticating will harm the users' privacy and security due to an increased risk of identity theft. Anonymous credentials enable privacy-friendly authentication by revealing as little as possible. In this dissertation, we have taken multiple steps towards obtaining practical and composable anonymous credentials. First, we have shown that global random oracles are much more powerful than was known before, by presenting different notions of global random oracles, and presenting very efficient protocols with these notions. This allows us to correctly model composition with random oracles, by capturing the fact that multiple protocols typically use the same random oracle. Second, building on our results on global random oracles, we presented a delegatable anonymous credential scheme which is the first to be composable and the first to include attributes, while also being very efficient. This allows anonymous credentials to be used in a setting where credentials are hierarchically issued, which is a typical setting in today's public key infrastructure. Third, we presented a formal security model for direct anonymous attestation, a form of anonymous credentials where a TPM holds a part of the signing key. DAA has been lacking a formal security model that captures all desired properties for the last decade Our model captures very strong privacy guarantees, by preserving privacy of a host even if it signs with a subverted TPM. These steps allow anonymous credentials to be used for attested computing without affecting the privacy of users.

Some relevant questions require further investigation. In Chapter 3, we did not prove that adding restricted programmability to a non-programmable global random oracle maintains security, while intuitively this should not help the adversary and therefore be possible. It seems promising to further investigate this subject, ideally proving that adding restricted programmability maintains security, potentially for a restricted class of protocols.

Our definition of delegatable credentials focuses on a single root issuer per protocol instance. While this allows for a simple and understandable ideal functionality, it does not allow for proving statements about multiple credentials from different root issuers. It would therefore be useful to extend $\mathcal{F}_{\mathsf{dac}}$ to work with multiple root issuers and extend our protocol to that setting. Another simplification in $\mathcal{F}_{\mathsf{dac}}$ is that it does not require anonymity during issuance or delegation, which again helps achieving a simple definition and an efficient realizations. The lack of anonymity during issuance is not a problem for many use cases, but certain applications would require anonymous issuance. One could extend $\mathcal{F}_{\mathsf{dac}}$ to optionally support anonymous issuance, which will be harder to realize, but then the higher level protocol designer can choose the level of anonymity required for the specific setting.

The constructions for delegatable credentials and anonymous attestation are only secure with respect to static adversaries. Future work could investigate how one can construct delegatable credentials and anonymous attestation secure against an adaptive adversary. For anonymous attestation, this would also allow us to reason about forward anonymity, which requires signatures from an honest host to remain anonymous even after the host becomes corrupted.

Our generic constructions for delegatable anonymous credentials and direct anonymous attestation are built from primitives defined in a property-based manner, rather than using hybrid functionalities as one would expect in the UC framework. Unfortunately, using hybrid functionalities is not always possible. In some cases one could choose between a property-based notion and a UC functionality, but the UC functionality is much stronger and therefore harder to realize, such that building on the hybrid functionality would yield a less efficient overall protocol. An example is a cryptographic commitment scheme, where the property-based definition is easier to achieve than the UC notion. In other cases, it is simply not possible to use hybrid functionalties to capture the building blocks. For example, anonymous credentials typically use zero-knowledge proofs to prove possession of a signature from the issuer. It is not clear how the zero-knowledge functionality

$\mathcal{F}_{\mathsf{zk}}$, which expects a statement and a witness as input, can prove that the signature functionality $\mathcal{F}_{\mathsf{sig}}$ would consider a certain signature to be valid. A more fundamental direction of future work would aim to remove these road blocks and find ways to combine functionalities in the same way as their property-based counterparts, which would unluck the full potential of the UC framework.

Bibliography

[AB13] Prabhanjan Ananth and Raghav Bhaskar. Non observability in the random oracle model. In Willy Susilo and Reza Reyhanitabar, editors, *ProvSec 2013*, volume 8209 of *LNCS*, pages 86–103. Springer, Heidelberg, October 2013.

[AGOT14] Masayuki Abe, Jens Groth, Miyako Ohkubo, and Mehdi Tibouchi. Unified, minimal and selectively randomizable structure-preserving signatures. In Yehuda Lindell, editor, *TCC 2014*, volume 8349 of *LNCS*, pages 688–712. Springer, Heidelberg, February 2014.

[AKMZ12] Joël Alwen, Jonathan Katz, Ueli Maurer, and Vassilis Zikas. Collusion-preserving computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 124–143. Springer, Heidelberg, August 2012.

[AMV15] Giuseppe Ateniese, Bernardo Magri, and Daniele Venturi. Subversion-resilient signature schemes. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 364–375. ACM Press, October 2015.

[AsV08] Joël Alwen, abhi shelat, and Ivan Visconti. Collusion-free protocols in the mediated model. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 497–514. Springer, Heidelberg, August 2008.

[BBDP01] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 566–582. Springer, Heidelberg, December 2001.

[BBG13] James Ball, Julian Borger, and Glenn Greenwald. Revealed: how US and UK spy agencies defeat internet privacy and security. Guardian Weekly, September 2013.

[BBS98] Matt Blaze, Gerrit Bleumer, and Martin Strauss. Divertible protocols and atomic proxy cryptography. In Kaisa Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 127–144. Springer, Heidelberg, May / June 1998.

[BCC04] Ernest F. Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In Vijayalakshmi Atluri, Birgit Pfitzmann, and Patrick McDaniel, editors, *ACM CCS 04*, pages 132–145. ACM Press, October 2004.

[BCC$^+$09] Mira Belenkiy, Jan Camenisch, Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Hovav Shacham. Randomizable proofs and delegatable anonymous credentials. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 108–125. Springer, Heidelberg, August 2009.

[BCG$^+$15] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *2015 IEEE Symposium on Security and Privacy*, pages 287–304. IEEE Computer Society Press, May 2015.

[BCL08] Ernie Brickell, Liqun Chen, and Jiangtao Li. A new direct anonymous attestation scheme from bilinear maps. In Peter Lipp, Ahmad-Reza Sadeghi, and Klaus-Michael Koch, editors, *Trusted Computing - Challenges and Applications, First International Conference on Trusted Computing and Trust in Information Technologies, Trust 2008, Villach, Austria, March 11-12, 2008, Proceedings*, volume 4968 of *Lecture Notes in Computer Science*, pages 166–178. Springer, 2008.

[BCL09] Ernie Brickell, Liqun Chen, and Jiangtao Li. Simplified security notions of direct anonymous attestation and a concrete scheme from pairings. *Int. J. Inf. Sec.*, 8(5):315–330, 2009.

[BD95] Mike Burmester and Yvo Desmedt. A secure and efficient conference key distribution system (extended abstract). In Alfredo De Santis, editor, *EUROCRYPT'94*, volume 950 of *LNCS*, pages 275–286. Springer, Heidelberg, May 1995.

[Bea92] Donald Beaver. Foundations of secure interactive computing. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 377–391. Springer, Heidelberg, August 1992.

[BFG13a] David Bernhard, Georg Fuchsbauer, and Essam Ghadafi. Efficient signatures of knowledge and DAA in the standard model. In Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *ACNS 13*, volume 7954 of *LNCS*, pages 518–533. Springer, Heidelberg, June 2013.

[BFG$^+$13b] David Bernhard, Georg Fuchsbauer, Essam Ghadafi, Nigel P. Smart, and Bogdan Warinschi. Anonymous attestation with user-controlled linkability. *Int. J. Inf. Sec.*, 12(3):219–249, 2013.

[BH04] Michael Backes and Dennis Hofheinz. How to break and repair a universally composable signature functionality. In Kan Zhang and Yuliang Zheng, editors, *Information Security, 7th International Conference, ISC 2004, Palo Alto, CA, USA, September 27-29, 2004, Proceedings*, volume 3225 of *Lecture Notes in Computer Science*, pages 61–72. Springer, 2004.

[BL10] Ernie Brickell and Jiangtao Li. A pairing-based DAA scheme further reducing TPM resources. In Alessandro Acquisti, Sean W. Smith, and Ahmad-Reza Sadeghi, editors, *Trust and Trustworthy Computing, Third International Conference, TRUST 2010, Berlin, Germany, June 21-23, 2010. Proceedings*, volume 6101 of *Lecture Notes in Computer Science*, pages 181–195. Springer, 2010.

[BL11] Ernie Brickell and Jiangtao Li. Enhanced privacy ID from bilinear pairing for hardware authentication and attestation. *IJIPSI*, 1(1):3–33, 2011.

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, September 2004.

[BM15] Rishiraj Bhattacharyya and Pratyay Mukherjee. Non-adaptive programmability of random oracle. *Theor. Comput. Sci.*, 592:97–114, 2015.

[BN06a] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In Bart Preneel and Stafford Tavares, editors, *SAC 2005*, volume 3897 of *LNCS*, pages 319–331. Springer, Heidelberg, August 2006.

[BN06b] Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 06*, pages 390–399. ACM Press, October / November 2006.

[BPR14] Mihir Bellare, Kenneth G. Paterson, and Phillip Rogaway. Security of symmetric encryption against mass surveillance. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 1–19. Springer, Heidelberg, August 2014.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 93*, pages 62–73. ACM Press, November 1993.

[BR95] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In Alfredo De Santis, editor, *EUROCRYPT'94*, volume 950 of *LNCS*, pages 92–111. Springer, Heidelberg, May 1995.

[BR96] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures: How to sign with RSA and Rabin. In Ueli M. Maurer, editor, *EUROCRYPT'96*, volume 1070 of *LNCS*, pages 399–416. Springer, Heidelberg, May 1996.

[Bra94] Stefan Brands. Untraceable off-line cash in wallets with observers (extended abstract). In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 302–318. Springer, Heidelberg, August 1994.

[Bra00] Stefan A. Brands. *Rethinking Public Key Infrastructures and Digital Certificates: Building in Privacy.* MIT Press, Cambridge, MA, USA, 2000.

[BS01] Mihir Bellare and Ravi Sandhu. The security of practical two-party RSA signature schemes. Cryptology ePrint Archive, Report 2001/060, 2001. `http://eprint.iacr.org/2001/060`.

[Can00] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. `http://eprint.iacr.org/2000/067`.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[Can04] Ran Canetti. Universally composable signature, certification, and authentication. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, page 219. IEEE Computer Society, 2004.

[CCD$^+$17] Jan Camenisch, Liqun Chen, Manu Drijvers, Anja Lehmann, David Novick, and Rainer Urian. One TPM to bind them all: Fixing TPM 2.0 for provably secure anonymous attestation. In *2017 IEEE Symposium on Security and Privacy*, pages 901–920. IEEE Computer Society Press, May 2017.

[CD16] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. `http://eprint.iacr.org/2016/086`.

[CDD17] Jan Camenisch, Manu Drijvers, and Maria Dubovitskaya. Practical UC-secure delegatable credentials with attributes and their application to blockchain. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 683–699. ACM Press, October / November 2017.

[CDE$^+$14] Jan Camenisch, Maria Dubovitskaya, Robert R. Enderlein, Anja Lehmann, Gregory Neven, Christian Paquin, and Franz-Stefan Preiss. Concepts and languages for privacy-preserving attribute-based authentication. *J. Inf. Sec. Appl.*, 19(1):25–44, 2014.

[CDE$^+$18] Jan Camenisch, Manu Drijvers, Alec Edgington, Anja Lehmann, Rolf Lindemann, and Rainer Urian. FIDO ECDAA algorithm, implementation draft. `https://fidoalliance.org/specs/fido-uaf-v1.2-id-20180220/fido-ecdaa-algorithm-v1.2-id-20180220.pdf`, 2018.

[CDG$^+$18] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 280–312. Springer, Heidelberg, April / May 2018.

[CDL16a] Jan Camenisch, Manu Drijvers, and Anja Lehmann. Anonymous attestation using the strong diffie hellman assumption revisited. In Michael Franz and Panos Papadimitratos, editors, *Trust and Trustworthy Computing - 9th International Conference, TRUST 2016, Vienna, Austria, August 29-30, 2016, Proceedings*, volume 9824 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2016.

[CDL16b] Jan Camenisch, Manu Drijvers, and Anja Lehmann. Universally composable direct anonymous attestation. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016, Part II*, volume 9615 of *LNCS*, pages 234–264. Springer, Heidelberg, March 2016.

[CDL17] Jan Camenisch, Manu Drijvers, and Anja Lehmann. Anonymous attestation with subverted TPMs. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 427–461. Springer, Heidelberg, August 2017.

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 61–85. Springer, Heidelberg, February 2007.

[CDR16] Jan Camenisch, Maria Dubovitskaya, and Alfredo Rial. UC commitments for modular protocol design and applications to revocation and attribute tokens. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 208–239. Springer, Heidelberg, August 2016.

[CEK$^+$16] Jan Camenisch, Robert R. Enderlein, Stephan Krenn, Ralf Küsters, and Daniel Rausch. Universal composition with responsive environments. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 807–840. Springer, Heidelberg, December 2016.

[CF01] Ran Canetti and Marc Fischlin. Universally composable commitments. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 19–40. Springer, Heidelberg, August 2001.

[CF08] Xiaofeng Chen and Dengguo Feng. Direct anonymous attestation for next generation TPM. *JCP*, 3(12):43–50, 2008.

[CGH98] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited (preliminary version). In *30th ACM STOC*, pages 209–218. ACM Press, May 1998.

[Cha85] David Chaum. Security without identification: Transaction systems to make big brother obsolete. *Commun. ACM*, 28(10):1030–1044, 1985.

[Cha92] David Chaum. Achieving electronic privacy. *Scientific american*, 267(2):96–101, 1992.

[Che09] Liqun Chen. A DAA scheme requiring less TPM resources. In Feng Bao, Moti Yung, Dongdai Lin, and Jiwu Jing, editors, *Information Security and Cryptology - 5th International Conference, Inscrypt 2009, Beijing, China, December 12-15, 2009. Revised Selected Papers*, volume 6151 of *Lecture Notes in Computer Science*, pages 350–365. Springer, 2009.

[CJS14] Ran Canetti, Abhishek Jain, and Alessandra Scafuro. Practical UC security with a global random oracle. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 597–608. ACM Press, November 2014.

[CKLM12] Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Sarah Meiklejohn. Malleable proof systems and applications. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 281–300. Springer, Heidelberg, April 2012.

[CKLM13a] Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Sarah Meiklejohn. Malleable signatures: Complex unary transformations and delegatable anonymous credentials. Cryptology ePrint Archive, Report 2013/179, 2013. `http://eprint.iacr.org/2013/179`.

[CKLM13b] Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Sarah Meiklejohn. Succinct malleable NIZKs and an application to compact shuffles. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 100–119. Springer, Heidelberg, March 2013.

[CKLM14] Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Sarah Meiklejohn. Malleable signatures: New definitions and delegatable anonymous credentials. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 199–213. IEEE Computer Society, 2014.

[CKY09] Jan Camenisch, Aggelos Kiayias, and Moti Yung. On the portability of generalized Schnorr proofs. In Antoine Joux, editor, *EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 425–442. Springer, Heidelberg, April 2009.

[CL04] Jan Camenisch and Anna Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In

Matthew Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 56–72. Springer, Heidelberg, August 2004.

[CL06] Melissa Chase and Anna Lysyanskaya. On signatures of knowledge. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 78–96. Springer, Heidelberg, August 2006.

[CL15] Jan Camenisch and Anja Lehmann. (Un)linkable pseudonyms for governmental databases. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 1467–1479. ACM Press, October 2015.

[CLNS17] Jan Camenisch, Anja Lehmann, Gregory Neven, and Kai Samelin. UC-secure non-interactive public-key encryption. In *IEEE CSF*, 2017.

[CMS08a] Liqun Chen, Paul Morrissey, and Nigel P. Smart. On proofs of security for DAA schemes. In Joonsang Baek, Feng Bao, Kefei Chen, and Xuejia Lai, editors, *ProvSec 2008*, volume 5324 of *LNCS*, pages 156–175. Springer, Heidelberg, October / November 2008.

[CMS08b] Liqun Chen, Paul Morrissey, and Nigel P. Smart. Pairings in trusted computing (invited talk). In Steven D. Galbraith and Kenneth G. Paterson, editors, *PAIRING 2008*, volume 5209 of *LNCS*, pages 1–17. Springer, Heidelberg, September 2008.

[CMS09] L Chen, P. Morrissey, and N.P. Smart. DAA: Fixing the pairing based protocols. Cryptology ePrint Archive, Report 2009/198, 2009. `http://eprint.iacr.org/2009/198`.

[CMY$^+$16] Rongmao Chen, Yi Mu, Guomin Yang, Willy Susilo, Fuchun Guo, and Mingwu Zhang. Cryptographic reverse firewall via malleable smooth projective hash functions. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 844–876. Springer, Heidelberg, December 2016.

[Cor00] Jean-Sébastien Coron. On the exact security of full domain hash. In Mihir Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 229–235. Springer, Heidelberg, August 2000.

[CP93] David Chaum and Torben P. Pedersen. Wallet databases with observers. In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 89–105. Springer, Heidelberg, August 1993.

[CP94] Ronald Cramer and Torben P. Pedersen. Improved privacy in wallets with observers (extended abstract). In Tor Helleseth, editor, *EUROCRYPT'93*, volume 765 of *LNCS*, pages 329–343. Springer, Heidelberg, May 1994.

[CPS10] Liqun Chen, Dan Page, and Nigel P. Smart. On the design and implementation of an efficient DAA scheme. In Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010, Passau, Germany, April 14-16, 2010. Proceedings*, volume 6035 of *Lecture Notes in Computer Science*, pages 223–237. Springer, 2010.

[CS97] Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups (extended abstract). In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 410–424. Springer, Heidelberg, August 1997.

[CS03] Jan Camenisch and Victor Shoup. Practical verifiable encryption and decryption of discrete logarithms. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 126–144. Springer, Heidelberg, August 2003.

[CV12] Ran Canetti and Margarita Vald. Universally composable security with local adversaries. In Ivan Visconti and Roberto De Prisco, editors, *SCN 12*, volume 7485 of *LNCS*, pages 281–301. Springer, Heidelberg, September 2012.

[DMSD16] Yevgeniy Dodis, Ilya Mironov, and Noah Stephens-Davidowitz. Message transmission with reverse firewalls—secure communication on corrupted machines. In Matthew

Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 341–372. Springer, Heidelberg, August 2016.

[DSD07] Augusto Jun Devegili, Michael Scott, and Ricardo Dahab. Implementing cryptographic pairings over Barreto-Naehrig curves (invited talk). In Tsuyoshi Takagi, Tatsuaki Okamoto, Eiji Okamoto, and Takeshi Okamoto, editors, *PAIRING 2007*, volume 4575 of *LNCS*, pages 197–207. Springer, Heidelberg, July 2007.

[DSW08] Yevgeniy Dodis, Victor Shoup, and Shabsi Walfish. Efficient constructions of composable commitments and zero-knowledge proofs. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 515–535. Springer, Heidelberg, August 2008.

[ElG86] Taher ElGamal. On computing logarithms over finite fields. In Hugh C. Williams, editor, *CRYPTO'85*, volume 218 of *LNCS*, pages 396–402. Springer, Heidelberg, August 1986.

[FFS88] Uriel Feige, Amos Fiat, and Adi Shamir. Zero-knowledge proofs of identity. *Journal of Cryptology*, 1(2):77–94, 1988.

[FLR$^+$10] Marc Fischlin, Anja Lehmann, Thomas Ristenpart, Thomas Shrimpton, Martijn Stam, and Stefano Tessaro. Random oracles with(out) programmability. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 303–320. Springer, Heidelberg, December 2010.

[FO97] Eiichiro Fujisaki and Tatsuaki Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 16–30. Springer, Heidelberg, August 1997.

[FO13] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *Journal of Cryptology*, 26(1):80–101, January 2013.

[FP09] Georg Fuchsbauer and David Pointcheval. Anonymous consecutive delegation of signing rights: Unifying group and proxy signatures. In Véronique Cortier, Claude Kirchner, Mitsuhiro Okada, and Hideki Sakurada, editors, *Formal to Practical Security - Papers Issued from the 2005-2008 French-Japanese Collaboration*, volume 5458 of *Lecture Notes in Computer Science*, pages 95–115. Springer, 2009.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987.

[Fuc11] Georg Fuchsbauer. Commuting signatures and verifiable encryption. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 224–245. Springer, Heidelberg, May 2011.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *17th ACM STOC*, pages 291–304. ACM Press, May 1985.

[Gre14] Glenn Greenwald. No place to hide: Edward snowden, the nsa, and the u.s. surveillance state. Metropolitan Books, May 2014.

[Gro15] Jens Groth. Efficient fully structure-preserving signatures for large messages. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part I*, volume 9452 of *LNCS*, pages 239–259. Springer, Heidelberg, November / December 2015.

[GS08] Jens Groth and Amit Sahai. Efficient non-interactive proof systems for bilinear groups. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 415–432. Springer, Heidelberg, April 2008.

[HPV16] Carmit Hazay, Antigoni Polychroniadou, and Muthuramakrishnan Venkitasubramaniam. Composable security in the tamper-proof hardware model under minimal complexity. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 367–399. Springer, Heidelberg, October / November 2016.

[Int13] International Organization for Standardization. ISO/IEC 20008-2: Information technology - Security techniques - Anonymous digital signatures - Part 2: Mechanisms using a group public key, 2013.

[Int15] International Organization for Standardization. ISO/IEC 11889: Information technology - Trusted platform module library, 2015.

[Kat07] Jonathan Katz. Universally composable multi-party computation using tamper-proof hardware. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 115–128. Springer, Heidelberg, May 2007.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition.* CRC Press, 2014.

[KO04] Jonathan Katz and Rafail Ostrovsky. Round-optimal secure two-party computation. In Matthew Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 335–354. Springer, Heidelberg, August 2004.

[MS15] Ilya Mironov and Noah Stephens-Davidowitz. Cryptographic reverse firewalls. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 657–686. Springer, Heidelberg, April 2015.

[Nie02] Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 111–126. Springer, Heidelberg, August 2002.

[OO90] Tatsuaki Okamoto and Kazuo Ohta. Divertible zero knowledge interactive proofs and commutative random self-reducibility. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *EUROCRYPT'89*, volume 434 of *LNCS*, pages 134–148. Springer, Heidelberg, April 1990.

[PLS13] Nicole Perlroth, Jeff Larson, and Scott Shane. N.S.A. able to foil basic safeguards of privacy on web. The New York Times, September 2013.

[PS00] David Pointcheval and Jacques Stern. Security arguments for digital signatures and blind signatures. *Journal of Cryptology*, 13(3):361–396, 2000.

[RTYZ16] Alexander Russell, Qiang Tang, Moti Yung, and Hong-Sheng Zhou. Cliptography: Clipping the power of kleptographic attacks. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 34–64. Springer, Heidelberg, December 2016.

[RTYZ17] Alexander Russell, Qiang Tang, Moti Yung, and Hong-Sheng Zhou. Generic semantic security against a kleptographic adversary. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 907–922. ACM Press, October / November 2017.

[Sch90] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 239–252. Springer, Heidelberg, August 1990.

[Sch91] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[Tru04] Trusted Computing Group. TPM main specification version 1.2, 2004.

[Tru14] Trusted Computing Group. Trusted platform module library specification, family "2.0", 2014.

[TW05] Mårten Trolin and Douglas Wikström. Hierarchical group signatures. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *ICALP 2005*, volume 3580 of *LNCS*, pages 446–458. Springer, Heidelberg, July 2005.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, November 1982.

[YY97a] Adam Young and Moti Yung. Kleptography: Using cryptography against cryptography. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 62–74. Springer, Heidelberg, May 1997.

[YY97b] Adam Young and Moti Yung. The prevalence of kleptographic attacks on discrete-log based cryptosystems. In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 264–276. Springer, Heidelberg, August 1997.

Curriculum Vitae

Manu Drijvers, born February 7th, 1991, citizen of the Netherlands.

Education

2015 - 2018	**ETH Zurich**. Ph.D. student in the computer science department.
2012 - 2014	**Radboud University Nijmegen**. Master of Science in Computing Science.
2009 - 2012	**Radboud University Nijmegen**. Bachelor of Science in Computing Science.
2003 - 2009	**Stedelijk Gymnasium Nijmegen**. Pre-university Secondary Education (VWO).

Work Experience

2018 - present	**DFINITY**. Cryptography Researcher.
2014 - 2018	**IBM Research – Zurich**. Predoctoral Researcher in Security & Privacy group.
2012 - 2014	**Drijvers-IT**. Freelance Software Developer.
2009 - 2012	**Utrecht University**. Java Developer.

Publications

1. Dan Boneh, Manu Drijvers, and Gregory Neven. Compact Multi-Signatures for Smaller Blockchains. *ASIACRYPT 2018.*

2. Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. In Jesper Buus Nielsen and Vincent Rijmen, editors,

EUROCRYPT 2018, Part I, volume 10820 of *LNCS*, pages 280–312. Springer, Heidelberg, April / May 2018.

3. Jan Camenisch, Manu Drijvers, and Maria Dubovitskaya. Practical UC-secure delegatable credentials with attributes and their application to blockchain. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 683–699. ACM Press, October / November 2017.
4. Jan Camenisch, Manu Drijvers, and Anja Lehmann. Anonymous attestation with subverted TPMs. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 427–461. Springer, Heidelberg, August 2017.
5. Jan Camenisch, Liqun Chen, Manu Drijvers, Anja Lehmann, David Novick, and Rainer Urian. One TPM to bind them all: Fixing TPM 2.0 for provably secure anonymous attestation. In *2017 IEEE Symposium on Security and Privacy*, pages 901–920. IEEE Computer Society Press, May 2017.
6. Jan Camenisch, Manu Drijvers, and Jan Hajny. Scalable revocation scheme for anonymous credentials based on n-times unlinkable proofs. In Edgar R. Weippl, Stefan Katzenbeisser, and Sabrina De Capitani di Vimercati, editors, *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society, WPES@CCS 2016, Vienna, Austria, October 24 - 28, 2016*, pages 123–133. ACM, 2016.
7. Jan Camenisch, Manu Drijvers, and Anja Lehmann. Anonymous attestation using the strong diffie hellman assumption revisited. In Michael Franz and Panos Papadimitratos, editors, *Trust and Trustworthy Computing - 9th International Conference, TRUST 2016, Vienna, Austria, August 29-30, 2016, Proceedings*, volume 9824 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2016.
8. Jan Camenisch, Manu Drijvers, and Anja Lehmann. Universally composable direct anonymous attestation. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016, Part II*, volume 9615 of *LNCS*, pages 234–264. Springer, Heidelberg, March 2016.