DISS. ETH NO. 23362

PRACTICAL COMPOSABLE CRYPTOGRAPHIC PROTOCOLS RESISTANT AGAINST ADAPTIVE ATTACKS

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

ROBERT R. ENDERLEIN

ing. sys. com. dipl. EPF
École Polytechnique Fédérale de Lausanne

born on 2 July 1987

citizen of Coppet VD and Germany

accepted on the recommendation of

Prof. Dr. Ueli Maurer, examiner
Dr. Jan Camenisch, co-examiner
Prof. Dr. Ralf Küsters, co-examiner

2016

To my father.

Abstract

In this thesis we are interested in practical cryptographic protocols that give strong security guarantees. This means that the protocols must remain secure in arbitrary contexts and against adversaries who adaptively attack these protocols during their execution, that they must have realistic prerequisites (setup assumptions, required resources), and that they be efficient enough to be used in a practical setting. The former can be achieved by designing the protocols in a security composition framework, however simultaneously making them *practical* is challenging.

The goal of this thesis is to simplify the design of *adaptively secure* and *practical* protocols. To that effect, we first design two protocols that satisfy the above properties, and thereby uncover design principles that make it easier to achieve the goal. These protocols, and many other adaptively secure protocols in the literature, assume the existence of *perfectly erasable memory*. This is not always available in practice: we therefore also study imperfectly erasable memory and how it can be improved. Finally, we have noticed that there were problems with the existing composition frameworks in which we designed our protocols: we have extended the basic IITM framework to provide a sound and unambiguous framework for future protocol design.

More precisely, the contributions of this thesis are as follows.

First, we present a protocol for two-party computation of arithmetic circuits, a primitive that is often useful in constructing higher-level protocols. Most *multi*-party computation protocols rely on an *honest majority* of parties to securely re-use outputs from one building block as inputs to another, however this technique is not applicable to *two*-party protocols; for that reason, existing two-party protocols cannot be used as building blocks when constructing higher-level protocols. We provide an efficient two-party arithmetic circuit protocol, and extend it so that it supports zero-knowledge proofs about previously stored values: higher-level protocols can then use commitments together with those zero-knowledge proofs to efficiently and securely transfer outputs from one block to another. Our protocol thereby simplifies the design of many higher-level protocols.

Second, we present a protocol for two-server password-authenticated secret-sharing. This protocol allows a user to store data on two servers protected by nothing more than a (possibly weak) password; she can use such a protocol to bootstrap her digital

identity. Even if one server is compromised, the user's password and data are still safe: the adversary learns no useful information and cannot for example mount a brute-force password-recovery attack. This protocol even allows compromised servers to recover to a secure state after being compromised. Previous work failed to provide security guarantees in the case of a server actually being compromised.

Third, we study how to properly model erasable memory in security composition frameworks. In practice, often only imperfectly erasable memory is available, e.g., memory that leaks some bits of the stored data even after the data has been erased. Efficient adaptively secure protocols, however, often assume *perfectly erasable* memory. We amplify the erasability of some types of memories in order to realize memories that leak less or do not leak at all. We further improve the cryptographic primitive, namely the all-or-nothing transform, that we use in such realizations. Our work is applicable to more types of erasable memories than prior work.

Fourth, we present conventions for writing adaptively secure protocols. Prior to our work, all existing composition frameworks had a number of issues: on the one hand the frameworks are either too general (requiring specification of too many irrelevant details) or too restricted (certain classes of protocols cannot be modelled at all); on the other hand, existing protocols in the literature are often underspecified. Protocols designed in different works can thus not be combined. We address these issues by giving an abstraction layer on top of the IITM framework and require the designer to specify only the relevant pieces: the resulting protocol will be sound and unambiguous.

Our work simplifies the design of practical and secure protocols, by providing a useful building block for computing arithmetic circuits, providing techniques to recover from adaptive attacks, showing how to work in the absence of perfectly erasable memory, and providing a sound composition framework that takes care of irrelevant details. However, secure protocol design still remains challenging and a largely manual process, and we hope that future work will further simplify it, by, for example, automating parts of the design processes.

Résumé

Dans cette thèse, nous nous intéressons aux protocoles cryptographiques pratiques qui donnent des garanties de sécurité solides. Cela signifie que ces protocoles doivent rester sécurisés dans des contextes arbitraires et contre des adversaires qui attaquent ces protocoles de manière adaptative au cours de leur exécution, qu'ils doivent avoir des prérequis réalistes (hypothèses de configuration, ressources requises), et qu'ils soient suffisamment efficaces pour être utilisés dans un cadre pratique. Le premier peut être obtenu en concevant ces protocoles dans un cadre de composition universelle, mais les rendre simultanément *pratiques* est difficile.

L'objectif de cette thèse est de simplifier la conception de protocoles *pratiques* et *résistants contre les attaques adaptatives*. À cet effet, en premier lieu, nous concevons deux protocoles qui satisfont les propriétés ci-dessus, et ainsi découvrons des principes de conception qui facilitent l'atteinte de notre objectif. Ces protocoles, et de nombreux autres protocoles résistants contre les attaques adaptatives dans la littérature, présupposent l'existence de *mémoire parfaitement effaçable*. Cette dernière n'est malheureusement pas toujours disponible dans la pratique : nous étudions donc également des mémoires imparfaitement effaçables et comment elles peuvent être améliorées. Enfin, nous avons remarqué l'existence de problèmes avec les cadres de composition existants dans lesquels nous avons conçu nos protocoles : nous avons élargi le cadre de composition IITM (machines de Turing interactives inexhaustibles) en développant un cadre correct et sans ambiguïté pour de futurs conceptions de protocoles.

Plus précisément, les contributions de cette thèse sont les suivantes.

Tout d'abord, nous présentons un protocole pour le calcul de circuits arithmétiques à deux joueurs, une primitive qui est souvent utile dans la construction de protocoles de plus haut niveau. La plupart des protocoles de calcul *multi*-joueurs supposent que *la majorité des joueurs soient honnêtes* afin de transférer les entrées et sorties d'une primitive à une autre de manière sécurisée, mais cette technique n'est pas applicable aux protocoles à *deux* joueurs ; pour cette raison, les protocoles à deux joueurs existants ne peuvent pas être utilisés comme primitives pour des protocoles de plus haut niveau. Nous fournissons un protocole efficace pour le calcul de circuits arithmétique pour deux joueurs, et nous l'améliorons afin qu'il supporte des preuves à divulgation nulle de connaissances sur les valeurs précédemment stockées : des protocoles de plus haut niveau

peuvent alors utiliser un schéma de mise en gage conjointement avec ces preuves de connaissances afin de transférer les sorties d'une primitive à une autre de manière efficace et sûre. De cette façon, notre protocole simplifie la conception de nombreux protocoles de plus haut niveau.

Deuxièmement, nous présentons un protocole de partage de secrets à base de mots de passe pour deux serveurs. Notre protocole permet à une utilisatrice de stocker des données sur deux serveurs protégés par rien de plus qu'un (petit) mot de passe ; elle peut utiliser un tel protocole comme tremplin pour stocker son identité numérique. Dans le cas où un serveur est compromis, le mot de passe et les données de l'utilisatrice restent protégés : l'adversaire n'apprend aucune information utile et ne peut pas, par exemple, retrouver le mot de passe avec une attaque par force brute. En outre, dans notre protocole, les serveurs peuvent retourner à un état sécurisé après avoir étés compromis. Les protocoles antérieurs ont omis de fournir des garanties de sécurité dans le cas où l'un des serveurs est compromis.

Troisièmement, nous étudions comment modéliser correctement une mémoire effaçable dans un cadre de composition universelle. Comme c'est souvent le cas en pratique, seulement une mémoire imparfaitement effaçable est disponible, par exemple, une mémoire qui divulgue certains bits des données stockées même après que ces derniers aient été effacés. Cependant, la plupart des protocoles pratiques qui résistent aux attaques adaptatives supposent que la mémoire disponsible soit *parfaitement effaçable*. Nous amplifions l'effaçabilité de certains types de mémoires afin de réaliser des mémoires qui divulguent moins de données qu'initialement ou ne divulguent absolument rien. De plus, nous améliorons la fonction cryptographique que nous utilisons dans ces réalisations : la transformation tout-ou-rien. Notre travail est applicable pour une plus grande variété de mémoires effaçables que le travail préalable.

Quatrièmement, nous présentons des conventions d'écriture pour des protocoles qui résistent aux attaques adaptatives. Avant notre travail, tous les cadres de composition universelle existants avaient un certain nombre de défauts : d'une part les cadres sont soit trop généraux (il est nécessaire de spécifier trop de détails inintéressants) soit trop restreints (certaines classes de protocoles ne peuvent pas être modélisées du tout) ; d'autre part, les protocoles existants dans la littérature sont souvent sous-spécifiés. Des protocoles conçus dans différents travaux ne peuvent donc pas être combinés. Nous abordons ces questions en donnant une couche d'abstraction au-dessus du cadre de composition IITM qui permet aux concepteurs de protocoles de ne préciser que les pièces pertinentes : le protocole résultant sera correct et sans ambiguïté.

Notre travail simplifie la conception de protocoles pratiques et sécurisés, en fournissant une primitive utile pour le calcul de circuits arithmétiques, en fournissant des techniques pour restorer la sécurité des joueurs après une attaque adaptative, en montrant comment travailler en l'absence de mémoire parfaitement effaçable, et en fournissant un cadre de composition correct et qui prend soins de détails inintéressants pour le concepteur. Cependant, la conception de protocoles sécurisés reste encore difficile et est un processus essentiellement manuel, et nous espérons que des travaux futurs vont la simplifier davantage, par exemple, en automatisant certaines parties du processus de conception de protocoles.

Contents

Acknowledgements xiii

1 Introduction 1

2 Preliminaries 9
2.1 Notation 9
2.2 Indistinguishable Distributions and Ensembles 10
2.3 The Decisional Diffie-Hellman (DDH) Assumption 11
2.4 Cryptographic Building Blocks 11
2.4.1 Public-Key Encryption Scheme 12
2.4.2 Signature Scheme 14
2.4.3 Commitment Scheme 15
2.4.4 Pseudo-Random Generator (PRG) 16
2.4.5 Linear Block Code (LBC) 17
2.4.6 Ramp Secret Sharing Scheme (SSS) 17
2.4.7 Exposure Resilient Function (ERF) 17
2.4.8 Universal Hash Function 18
2.4.9 All-or-Nothing Transform (AoNT) 18
2.5 Composability Frameworks 21
2.5.1 The UC Framework 21
2.5.2 The GNUC framework 24
2.5.3 The IITM Model with Responsive Environments 25
2.5.4 The Constructive Cryptography (CC) Model 30
2.6 Some Basic Ideal Functionalities 33
2.6.1 Authenticated Channels 33
2.6.2 One-sided–authenticated Channels 33
2.6.3 Zero-knowledge Proofs of Knowledge and Existence 34
2.6.4 Ideal Functionalities For the CRS and Random Oracle Models 34

3 Practical Two-Party Computation of Arithmetic Circuits 37
3.1 Homomorphic "Mixed" Trapdoor (HMT) Commitments 38
3.1.1 A Scheme for Messages in $\mathbb{Z}_n$ 38

3.1.2 A Scheme over a Prime Order Group . . . 39
3.2 Our Ideal Functionality $\mathcal{F}_{\text{abb}}$ . . . 39
3.2.1 Informal Definition of $\mathcal{F}_{\text{abb}}$ . . . 40
3.2.2 Formal Definition of $\mathcal{F}_{\text{abb}}$ . . . 41
3.3 Construction . . . 44
3.3.1 Realizing Π_{abb} . . . 45
3.3.2 The Π_{mul} Subroutine for Multiplication of Committed Inputs . . . 47
3.3.3 Efficiency Considerations for the Zero-Knowledge Proofs in Π_{abb} 49
3.4 Additional Instructions for $\mathcal{F}_{\text{abb}}$ . . . 50
3.4.1 Instructions as Part of a Higher-Level Protocol . . . 50
3.4.2 Modifying $\mathcal{F}_{\text{abb}}$ to Add New Instructions . . . 51
3.5 Security Proof . . . 52
3.5.1 Main Ideas . . . 52
3.5.2 Security Proof . . . 53
3.5.3 Proof of Lemma 3.5 . . . 54
3.6 Related Work and Comparison . . . 63
3.6.1 Efficiency Comparison . . . 64
3.6.2 Comments about the Efficiency of Related Work . . . 65
3.7 Example of a Useful Protocol Constructed with $\mathcal{F}_{\text{abb}}$ . . . 66
3.7.1 Ideal Functionality . . . 66
3.7.2 Construction . . . 67
3.7.3 Security . . . 67

4 Practical 2-Server Password-Authenticated Secret Sharing . . . 69
4.1 Corruption in the UC Model . . . 71
4.2 Our Ideal Functionality $\mathcal{F}_{\text{2pass}}$ . . . 72
4.2.1 Informal Definition of $\mathcal{F}_{\text{2pass}}$ . . . 72
4.2.2 Formal Definition of $\mathcal{F}_{\text{2pass}}$ . . . 74
4.3 Our Construction of TPASS Secure Against Transient Corruptions . . . 81
4.3.1 High Level Approach of our TPASS Protocol . . . 81
4.3.2 Key Ideas of our TPASS Protocol . . . 82
4.3.3 Detailed Construction of Π_{2pass} . . . 85
4.3.4 Computational and Communication Complexity . . . 91
4.3.5 Comparison with Related Work . . . 91
4.4 Security Proof . . . 94
4.4.1 Main Ideas . . . 94
4.4.2 Security Proof . . . 96

5 Memory Erasability Amplification . . . 107
5.1 Modelling Imperfectly Erasable Memory . . . 108
5.1.1 Specification of the General Imperfectly Erasable Memory Resource 109
5.1.2 Instantiations of $\mathsf{M}\langle\Sigma, \psi, \rho, \kappa\rangle$ . . . 110
5.2 Constructing Better Memory Resources . . . 111
5.2.1 Admissible Converters for Constructions using Erasable Memory 112
5.2.2 Memory Erasability Amplification . . . 113
5.2.3 Constructing a Large Perfectly Erasable Memory from a Small One 117

5.3 New Realizations of All-or-Nothing Transforms 120
5.3.1 AoNT from a Protocol 120
5.3.2 Perfectly Secure AoNT Based on Matrices with Ramp Minimum Distance 123
5.3.3 Realizing a Perfectly Secure AoNT over a Small Field by Combining AoNTs 126
5.3.4 Computationally Secure AoNT over a Large Field from a PRG . . 127

6 Conventions for Usable Universal Composability 131
6.1 Templates for Real Protocols and Ideal Functionalities 132
6.1.1 Specifying Real Protocols 136
6.1.2 Specifying Ideal Functionalities 139
6.2 Mapping Templates to ITMs 142
6.2.1 Notation for the Formal Specification of ITMs 142
6.2.2 Real protocols 144
6.2.3 Ideal protocols 154
6.3 Programming Language for the Templates 158
6.4 An Example Functionality and its Realization: Digital Signatures 162
6.4.1 The Ideal Functionality $\mathcal{F}_{\text{sig}}$ 162
6.4.2 Realizing $\mathcal{F}_{\text{sig}}$ 163
6.5 Joint State 163
6.5.1 Conventions for Joint-State Protocols 166
6.5.2 Joint State Realization of Signatures 172

7 Concluding Remarks 175

References 177

Appendix

A Formal Definition of Ideal Functionalities 187
A.1 Common Reference Strings $\mathcal{F}_{\text{crs}}^{D}$ 187
A.2 Authenticated Channels $\mathcal{F}_{\text{ac}}$ 188
A.3 One-Sided–Authenticated Channels $\mathcal{F}_{\text{osac}}$ 189
A.4 Zero-Knowledge Proofs of Existence for One Verifier $\mathcal{F}_{\text{gzk}}$ 193
A.4.1 GNUC Formalism 195
A.5 Zero-Knowledge Proofs of Existence for Two Verifiers $\mathcal{F}_{\text{gzk}}^{\text{2v}}$ 195

Curriculum Vitae 199

Acknowledgements

First of all, I thank my supervisor at IBM, Dr. Jan Camenisch. I am grateful for his great patience and that he guided me through the realities of being a researcher. He always provided helpful advice on focusing on the truly important things. Furthermore, he was understanding and supportive during a difficult personal time.

I am truly indebted to my supervisor at ETH, Prof. Dr. Ueli Maurer. He has an incredible amount of patience, always had time for me, and taught me to question the conventions of the field of cryptography instead of blindly following them.

I am grateful to Prof. Dr. Ralf Küsters for agreeing to serve as my co-examiner.

I express my deep gratitude to my parents and my grand-father for their love, encouragement, and support. They have greatly inspired me to pursue a PhD, as they did in their time. I dedicate this thesis to my late father who would have been so proud to see it.

Sincere thanks also goes to Anne-Marie Cromack for proofreading my abstract, introduction, and conclusion; to Grégory Demay and Gregory Neven for proofreading my French abstract; and to Christian Badertscher for proofreading the acknowledgements.

It has been a great pleasure and a great honor to collaborate with Jan Camenisch, Maria Dubovitskaya, Stephan Krenn, Ralf Küsters, Anja Lehmann, Ueli Maurer, Gregory Neven, Franz-Stefan Preiss, Daniel Rausch, and Victor Shoup.

I was fortunate to have had wonderful colleagues at IBM Research. I thank especially Cecilia Boschini, Christian Cachin, Jan Camenisch, Manu Drijvers, Maria Dubovitskaya, Cedric Favre, Eduarda Freire, Alexandra Gorski, Nikola Knezevic, Daniel Kovacs, Stephan Krenn, Anja Lehmann, Mario Lucic, Vadim Lyubashevsky, Zoltan Nagy, Gregory Neven, Michael Osborne, Vit Pratzler, Franz-Stefan Preiss, Kai Samelin, Dieter Sommer, and Andreas Wespi, for making the lab such a fantastic place. I especially thank all those who contributed to my doctoral hat: I will cherish it as a memento to all the fun we had hiking, rafting, sledding, bowling, etc.

I am equally fortunate to have had great colleagues at ETH, specially Christian Badertscher, Sandro Coretti, Grégory Demay, Daniel Jost, Christian Matt, Gregor Seiler, Björn Tackmann, and Daniel Tschudi.

I am lucky to have such great friends. I thank Jonas Wagner in particular, for the amazing time we had together (hiking, skiing, paragliding, mountain biking, playing

Dominion) and for his excellent book gifts; and Pierluca Borsó, for board games and interesting discussions.

During my PhD, I had the honor of co-founding the ACM VIS commission, and made many friends while serving in it. I particularly thank Jan Doerrie, Sandro Feuz, Daniel Graf, Jan Hazla, Akaki Mamageishvili, Sebastian Millius, Rajko Nenadov, and Kieran Nikko, for the great time we had organizing the various programming competitions. A big thanks also to the VIS committees at large during those years, especially for organizing the awesome Snowdayz.

Before my PhD, I had the pleasure of co-founding the student association PolyProg and the Helvetic Coding Contest, and remained a member during all my PhD. I was amazed at how much we achieved (and how much time it took away from my PhD 😳). I am particularly grateful to Mohamed Abouhamra Abdel-Fattah, Pierluca Borsó, Titus Cieslewski, Christian Kauth, Oswald Maskens, Solal Pirelli, Jérémy Rabasco, Valérian Rousset, Yannick Schäffer, Benjamin Schubert, Jakub Tarnawski, Jonas Wagner, Jean-Paul Wenger, Johannes Würthrich, Joey Zenhäusern, and Christian Zommerfelds, all of which have become great friends.

Robert R. Enderlein
Zurich, May 2016

1

Introduction

Until around the middle of the 20th century, cryptography was used by a small number of people for protecting their sensitive communications, especially in military contexts. In those times, cryptography was an art, and offered little in the way of provable security: in fact, as late as World War II, ciphers were regularly broken [Kah96,Sin11]. Nowadays, cryptography is pervasively used in digital communication. Anybody who does online banking or shopping, uses e-mail or social media, and increasingly anybody who uses a search engine or an online encyclopedia is a regular user of cryptography. Modern cryptography is a science, and cryptographic protocols are now expected to come with rigorous security proofs. The scope of cryptography has also been expanded beyond ensuring the confidentiality of communication. It is now concerned with the design of systems that need to resist malicious attempts to abuse them [Gol01], examples include electronic auctions, electronic voting, digital cash, and securing distributed computation.

Unfortunately, in many distributed systems today, security is often an afterthought and considered secondary to cost effectiveness and user experience. Many high-profile security breaches reported in the media attest to this. This is especially worrisome in an era where our personal data (family pictures, backups), our electronic tax returns, and even our health records are being stored on remote servers, i.e., the "cloud", that are controlled by a third-party. This is exacerbated by the fact that many cloud computing providers are financed by advertising, and thus have an incentive to analyze their users' data.

Clearly, in today's world there is a need for cryptographic protocols that are both practical and secure. That is, they should be actually realizable in practice, be fast enough to run on resource-constrained devices such as smartphones, and present strong and provable security guarantees in the presence of untrusted or not well-protected cloud hosts and in case of theft of the user's smartphone or laptop.

Provable security. Designing and proving secure large and complex cryptographic protocols is very challenging. Today, the security proofs of most practical protocols consider only a single instance of the protocol and therefore all security guarantees are lost if such a protocol is run concurrently with other protocols or with itself, in other words, when used in practice. Better security guarantees can be obtained when using composability frameworks—the Universal Composability (UC) framework [Can00,Can01],

the similar GNUC framework [HS11], the Constructive Cryptography Framework [MR11, Mau10, Mau11b], the IITM framework [Kue06, KT13] and its extension to responsive environments [CEK+16a], and others [PW01, CDPW07, BPW07]—which ensure that protocols proved secure in the framework remain secure in arbitrary environments. This also simplifies the design of protocols: high-level protocols can be composed from lower-level "building block" protocols, and the security proofs of the higher-level protocols can be based on the security of the building blocks and so become modular and easier.

All of these frameworks define the security of a protocol by comparing its execution to an *ideal process*, where the output of the latter is computed by a trusted party that sees the inputs of all parties [GMW87]. That is, it is proven that any attack on the real protocol corresponds to an attack on the ideal process—which is secure by definition, thereby the real protocol is also secure.

Although this concept is very powerful, it was shown that many interesting protocols cannot be realized directly in such frameworks [CF01]. If one wishes to realize those, one must at least assume a secure setup [CKL06]. There are two popular such setup assumptions: in the so-called *standard-model*, one assumes that all protocol participants agree on a common reference string (CRS) with a specific distribution; whereas in the *random oracle model*, one assumes that all participants have access to a common random function.

Practical protocols. Unfortunately, protocols proven secure in such composability frameworks are typically one or two orders of magnitude less efficient than their traditional counterparts with "single-instance" security. This presents us with a dilemma: on the one hand we have secure but inefficient protocols, and on the other we have efficient protocols that do not come with realistic security guarantees. In this thesis we try to remedy this situation by considering *practical* protocols, which we define as follows.

- We eschew unrealistic setup assumptions. For example, many efficient protocols used today rely on the random oracle model. Unfortunately random oracles do not exist in practice and any workarounds will invalidate the security of such protocols [CGH98]. We thus require all our protocols to be in the *standard model* (with CRS). As it is impossible to achieve universal composability without some kind of setup assumption [CKL06], a CRS seems like a reasonable, pragmatic compromise.
- In the context of provable security, "efficient" is synonymous with polynomial time. By that definition, zero-knowledge protocols of Hamiltonicity that require one to transform an input to an instance of a particular NP problem (in that case, Hamiltonian circuits) and that use a "cut-and-choose" technique (repeating the protocol hundreds of times) are considered efficient. Likewise for protocols making use of such proofs [CLOS02]. However, such protocols are completely impractical. We thus require all our protocols to *preserve the algebraic structure of the data we are working with* and to *avoid cut-and-choose techniques*.
- Similarly, we strive to *minimize the actual number of expensive operations*, chiefly the number of exponentiations, that our protocols use. For example, it was observed that using zero-knowledge proofs of knowledge is a computationally expensive operation, as witnesses need to be verifiably encrypted as part of the

proof [CKS11]. By using zero-knowledge proofs of *existence* [CKS11] instead, one can significantly improve the runtime of protocols.
Of course, it is understood that in many cases, our protocols will still require many hundreds of exponentiations, hence there will always be a gap between the runtime of our practical protocols and the runtime of "efficient" (but not necessarily secure) protocols in use today.

- Finally, as observed before, we require our protocols to come with provable security guarantees and operate in arbitrary contexts, hence they *must be designed and proven secure in a composition framework*. Furthermore, as real computers can be compromised by hackers or even stolen at any time during protocol execution (and not only just before the protocol execution starts), we require all our protocols to be *resistant against adaptive attacks*.

Contribution and outline. The goal of this thesis is to simplify the design of adaptively secure cryptographic protocols that are also practical. We start by providing a practical protocol for arithmetic circuit evaluation which realizes a cryptographic primitive that is useful for constructing higher-level protocols, for example for constructing an OPRF [JL09], or credential authentication and key exchange protocols [CCGS10]. Circuit evaluation protocols are very general and can be used for realizing many additional cryptographic protocols, such as oblivious transfer with complex access policies or password-authenticated key exchange, but one can often achieve better constructions by re-designing a protocol from scratch. Next, we design a protocol for password-authenticated storage of data. Such a protocol is very relevant in practice, as users cannot be expected to remember secure cryptographic keys. This protocol also serves as a study for reasoning about adaptive corruption, and especially recovery from corruption. The two protocols above, and in fact many other practical adaptively secure protocols in the literature, assume the existence of perfectly erasable memory. Unfortunately, hard discs and SSDs, as well as various file systems, are designed to preserve data and be fast and not to reliably erase data. It is therefore unreasonable to assume the existence of perfectly erasable memory. We therefore study imperfectly erasable memory in a composition framework, for example memory that leaks a few bits of the stored data to the adversary even after an erasure, and how to improve such memory. The ultimate goal is of course to realize perfectly erasable memory from such imperfectly erasable ones. Finally, we have noticed that there are problems with the currently existing composition frameworks: such frameworks are either too general or too restricted. Frameworks of the former type place too much burden on the protocol designer, which in practice means that results are not stated in a precise manner (and are sometimes wrong, as noted by Camenisch et al. [CEK+16a]), and that different works may implement certain details in different ways and thereby prevent protocol designers from combining these works to build higher-level protocols. Frameworks of the latter type do not permit the designer to model certain classes of protocols at all. We provide a remedy by proposing an abstraction layer on top of the very general IITM model with responsive environments [CEK+16a]. It requires the designer to specify only the relevant pieces of his protocol; irrelevant details are taken care of by the conventions we propose.

More precisely, the outline and contributions of this thesis are as follows.

Preliminaries. In Chapter 2, we define the notation and recall background information used in this thesis.

Practical Two-Party Computation of Arithmetic Circuits. In Chapter 3, we present a set of new, efficient, universally composable two-party protocols for evaluating reactive arithmetic circuits modulo n, where n is a safe RSA modulus of unknown factorization. This protocol is in the standard model and assumes that secure erasures are available. Our main contribution is twofold.

First, we provide a mechanism for protocol designers to easily integrate our arithmetic circuit functionality as a building block in their higher-level protocol in a practical yet secure manner. Most UC-secure two-party schemes and protocols found in the literature can not be used as building blocks for higher-level protocols because they do not offer the proper interfaces. That is, in the case of two-party protocols it is typically not possible to ensure that a party's output from one building block is used as the party's input to another building block; this is unlike the case of multi-party protocols with honest majority, where it is possible to secret-share all input and output values and then, by virtue of the majority's honesty, it is ensured that the right outputs are used as inputs to the next building block. We solved this issue with an additional instruction in our circuit evaluation protocol that allows parties to do a zero-knowledge proof over values both internal and external to the circuit. By using commitments (or credentials), it is thereby possible to securely transfer values from the circuit to another building block and vice-versa.

Second, we provide a concrete construction of the circuit evaluation protocol that is in itself more efficient than prior work. We achieve the latter by using cryptographic primitives that work very well together. Additionally, the tools we use in our construction—especially our novel mixed trapdoor commitment scheme—may be of independent interest.

Our protocol can be extended with some features, such as generating random values and computing multiplicative inverses modulo n, using standard techniques. Other features require an extension of our ideal functionality, in particular, we add an instruction for exponentiated output, with which we can directly implement Jarecki and Liu's two-party protocol for computing the following oblivious pseudorandom function (OPRF) [JL09]:

$$f_y(x) = \begin{cases} g^{1/(y+x)} & \text{if } \gcd(y+x, n) = 1. \\ 1 & \text{otherwise.} \end{cases}$$

Here, Alice's private input is x, Bob's private input is y, and Alice's output is $f_y(x)$. As pointed out by Jarecki and Liu, OPRF's have many useful cryptographic applications.

Practical 2-Server Password-Authenticated Secret Sharing. In Chapter 4, we provide the first threshold password-authenticated secret sharing (TPASS) protocol that is provably secure against *adaptive* corruptions, assuming data can be securely erased. TPASS protocols enable users to share secret data among a set of servers so that they can later recover that data using a single password. No coalition of servers up to a certain threshold can learn anything about the data or perform an offline dictionary attack on the password. Our protocol is a two-server protocol in the public-key setting, meaning that servers have trusted public keys, but users do not. We also describe a *recovery*

procedure that servers can execute to recover from corruption and to renew their keys assuming a trusted backup is available. The security of the password and the stored secret is preserved as long as both servers are never corrupted simultaneously.

Our construction uses the same basic approach as the TPASS protocols of Brainard et al. [BJKS03] and Camenisch et al. [CLN12]. During the setup phase, the user generates shares of his key and password and sends them to the servers (together with some commitments that will later be used in the retrieve phase). During the retrieve phase, the servers run a subprotocol with the user to verify the latter's password attempt using the commitments and shares obtained during setup. If the verification succeeds, the servers send the shares of the key back to the user, who can then reconstruct the key. Furthermore, the correctness of all values exchanged is enforced by zero-knowledge proofs. Like the recent work of Camenisch et al. [CLLN14], we do not require the user to share the password during the retrieve phase but run a dedicated protocol to verify whether the provided password equals the one shared priorly. This offers additional protection for the user's password in case he mistakenly tries to recover his secret from servers different from the ones he initially shared his secret with. During setup, the user can be expected to carefully choose his servers, but retrieval happens more frequently and possibly from different devices, leaving more room for error.

The novelty of our protocol lies in how we transform the basic approach into an efficient protocol secure against an adaptive adversary. The crux here is that parties should never be committed to their inputs but at the same time must prove that they perform their computation correctly. We believe that the techniques we use in our protocol to achieve this are of independent interest when building other protocols that are UC-secure against adaptive corruptions. First, instead of using (binding) encryptions to transmit integers between parties, we use a variant of Beaver and Haber's non-committing encryption based on one-time pads (OTP) [BH93]: the sender first commits to a value with a mixed trapdoor commitment scheme (see Section 3.1) and then encrypts both the value and the opening with the OTP. This enables the recipient to later prove statements about the encrypted value. Second, our three-party password-checking protocol achieves efficiency by transforming commitments with shared opening information into an Elgamal-like encryption of the same value under a shared secret key. To be able to simulate the servers' state if they get corrupted during the protocol execution, each pair of parties needs to temporarily re-encrypt the ciphertext with a key shared between them.

Memory Erasability Amplification. In Chapter 5, based on the observation that perfectly erasable memory is not always available, we model a memory resource that is only *imperfectly* erasable in the Constructive Cryptography framework. The user of that memory resource can write data once in it, and later retrieve the data. The user may also erase the data. The resource may be attacked by an adversary: in a successful attack *before* the data was erased, the adversary obtains the entire data; *after* the data was erased, the adversary obtains some residual information about the previously written data, as determined by a *leakage class*. We study multiple such leakage classes. For example, one that returns a constant number of bits (chosen by the adversary) of the written data. We also consider leakage classes that leak bits of the data randomly, or return a noisy version of the written data. Another important leakage class is that of length-shrinking functions chosen by the adversary, however as we shall see, those are

only interesting in the case that they are applied to *a part* of the memory (the other part leaking fully to the adversary), as otherwise the adversary can simply choose a function that mimics the decoding algorithm used by the memory.

We then propose protocols to amplify the erasability of such memory resources. That is, we propose protocols that construct erasable memory with a given leakage class from a (weaker) memory with a different class. For example, a protocol that is essentially a thin wrapper around an All-or-Nothing Transform (AoNT) [CDH+00], constructs a perfectly erasable memory from a memory that leaks a constant number of bits. Protocols may use (perfectly erasable) temporary storage to perform computation, however such temporary storage is discarded as soon as they send a message to a non-memory resource, e.g., a communication channel—thereby we exclude trivial protocols that bypass the memory resources.

Finally, we propose better realizations of AoNT's. As discussed above, AoNT's are a useful tool to amplify the erasability of memory. We propose several AoNT's with better parameters than previously known. For example, we improve the standard construction of a perfectly-secure AoNT from a Linear Block Code (LBC), by observing that an LBC with a large minimum distance does not yield an AoNT with optimal privacy threshold. We propose the metric of *ramp minimum distance*: an LBC has ramp minimum distance d if its $k \times n$ generator matrix has the property that for all $r \in \{1, \ldots, k\}$: all $r \times (n - (d - r))$ sub-matrices have rank r; and show that LBC's optimized for that metric yield perfectly secure AoNT's with better parameters than can be achieved with the standard construction. Another example: we propose a computationally secure AoNT that operates over a large alphabet (large enough for one symbol to encode a cryptographic key) that is optimal: the actual data is just one symbol shorter than the encoded data, and it is secure even if all but one of the symbols of the encoded data leak. Such an AoNT can be realized from a Pseudo-Random Generator (PRG) with some specific properties.

Conventions for Usable Universal Composability. In Chapter 6, we address the shortcomings of existing composition frameworks by providing an abstraction layer with templates on top of the IITM model with responsive environments (see Section 2.5.3 and Camenisch et al. [CEK+16a]). Thereby we allow for concise yet flexible protocol design, including handling of joint state. Finally, we provide a precise mapping from the templates to our model for unambiguous and complete protocol specifications. More precisely, we provide the following.

We provide an abstraction layer and templates, including the treatment of joint state. Based on the IITM model with responsive environments/adversaries, we provide a carefully designed abstraction layer, including templates and syntax for specifying computations, which allow the protocol designer to specify ideal functionalities and protocols, including joint state realization, in an unambiguously, yet convenient and flexible way. The abstraction layer is flexible enough to allow the protocol designer to model various forms of corruption, including top-down, bottom-up, strong and weak static corruption as well as dynamic corruption. Unlike other frameworks, including the UC and GNUC models, our framework allows a party to take different roles (e.g., signer and verifier, or initiator and responder) within one session. In other models, one would have to merge roles for this, leading to artificial protocol specifications. Allowing one party to play different roles in one session is not only convenient but more importantly

it is also security relevant: by excluding the case that a party takes different roles in one session, attacks in the real world might be missed.

We provide a precise mapping to the responsive IITM model. Protocol templates are mapped to the responsive IITM model such that we obtain precisely specified real and ideal protocol systems, where we make use of restricting messages in different places and rely on the assumption that environments are responsive. In particular, the protocol designer can rely on this, leading to simpler and more natural specifications. Also, the mapping includes the specification of default behavior. This not only facilitates the specification task for the protocol designer (who, hence, does not have to define certain aspects), but importantly also guarantees that protocols are never underspecified.

Finally, we illustrate our framework by the ideal functionality $\mathcal{F}_{\text{sig}}$ for digital signatures. We specify this functionality, its realization, as well as a joint-state realization using the proposed abstraction layer and templates.

Concluding Remarks. Finally, in Chapter 7, we propose new research directions.

2

Preliminaries

This chapter defines the notation, and recalls the various cryptographic hardness assumptions, cryptographic building blocks and other schemes, composition frameworks, and basic ideal functionalities that are used throughout this thesis.

2.1 Notation

Let $\mathbb{N}_i$ denote the set of all natural numbers between 0 and $(i-1)$, let $\mathbb{Z}_i$ denote the ring of integers modulo i. Let $\mathbb{N}_i^*$ and $\mathbb{Z}_i^*$ denote $\mathbb{N}_i \setminus \{0\}$ and $\mathbb{Z}_i \setminus \{0\}$, respectively. Let $\mathbb{R}_+$ denote the set of all non-negative real numbers. Let $\mathrm{GF}(q)$ denote the Galois field of q elements, where q is a prime power.

A number p is a *safe prime* if p and $(p-1)/2$ are prime numbers. A number is a *safe semi-prime* if it is the product of two safe primes.

If L is a set of positive integers, let $[u]_L$ denote the subvector of u taken at all positions in L. If S is a set, then 2^S denotes the powerset of S (the set of all subsets of S).

The probability of an event E over a random space Ω is written as $\Pr[\Omega : E]$. We write $\Pr[E]$ if Ω is clear from the context.

If A is a deterministic Polynomial-Time algorithm, then $y \leftarrow \mathsf{A}(x)$ denotes the assignment of variable y to the output of $\mathsf{A}(x)$. If A is a Probabilistic Polynomial-Time (PPT) algorithm, then $y \xleftarrow{\$} \mathsf{A}(x)$ denotes the assignment of y to the output of $\mathsf{A}(x)$ when run with fresh random coins on input x. For a set $\mathbb{A}$: $x \xleftarrow{\$} \mathbb{A}$ denotes the assignment of x to a value chosen uniformly at random from $\mathbb{A}$. If $\mathcal{P}$ is a conditional probability distribution $\mathcal{P} \in (X \times Y \mapsto [0,1])$ with $\forall y \in Y : \sum_{x \in X} \mathcal{P}(x \mid y) = 1$, then we also write $\mathcal{P} \in Y \overset{\$}{\mapsto} X$; let $x \xleftarrow{\$} \mathcal{P}(y)$ denote the sampling of $x \in X$ conditioned on the event $Y = y$, i.e., the probability that a given $x \in X$ is chosen is $\mathcal{P}(x \mid y)$. If $\mathcal{U}$ and $\mathcal{P}$ are parties, and Sub is a two-party protocol, then let $(out_{\mathcal{U}}; out_{\mathcal{P}}) \xleftarrow{\$} \langle \mathcal{U}.\mathsf{Sub}(in_{\mathcal{U}}), \mathcal{P}.\mathrm{Sub}(in_{\mathcal{P}}) \rangle (in_{\mathcal{UP}})$ denote the simultaneous execution of the protocol by the two parties, on common input $in_{\mathcal{UP}}$, with $\mathcal{U}$'s additional private input $in_{\mathcal{U}}$, with $\mathcal{P}$'s additional private input $in_{\mathcal{P}}$, and where $\mathcal{U}$'s output is $out_{\mathcal{U}}$ and $\mathcal{P}$'s output is $out_{\mathcal{P}}$; we use an analogue notation for three-party protocols.

Let $\mathbf{I}_r$ denote the identity matrix of size $r \times r$, and let $\mathbf{0}$ denote the zero matrix of appropriate size.

If u is a vector or a list, let u_i or $u[i]$ denote the ith element of u. If V is an associative array, then $V[k] \leftarrow v$ denotes the insertion of the value v into the array under the identifier k. By $v' \leftarrow V[k]$, one denotes the retrieval of the value associated with identifier k, and storing that retrieved value in the variable v'. If A is a (mutable) set, $A \leftarrow k$ is a shorthand notation for inserting k into it.

Let $\{0,1\}^*$ denote the set of all finite bit strings, and let $\{0,1\}^+$ denote the set of all non-empty finite bit strings. Let ε denote the empty string. If s is a bitstring, then let $|s|$ denote the length of s.

Let Λ denote a fixed, finite alphabet of symbols (for example Unicode codepoints). Throughout this text we will use monospace fonts to denote characters in Λ, e.g.: `P` or `Q`. Let $\Lambda^\star$ denote the set of strings over Λ. We use the list-encoding function $\langle\cdot\rangle$ like in the GNUC paper [HS11]: If $a_1, \ldots, a_n \in \Lambda^\star$, then $\langle a_1, \ldots, a_n\rangle$ is a string over Λ that encodes the list $(a_1, \ldots, a_n)$ in some canonical way.

For a binary relation $R \subseteq \{0,1\}^+ \times \{0,1\}^+$, let $R[0]$ denote the language induced by R, i.e., $R[0] = \{m | \exists (m, m') \in R\}$.

Throughout this thesis we denote the security parameter by $\eta \in \mathbb{N}$. Let 1^η denote the string consisting of η ones. Unless otherwise noted, all algorithms in this thesis are PPT and take 1^η as extra (often implicit) input.

We use the following arrow-notation: $\overset{publicData}{\dashrightarrow}$ to denote the transmission of public data over a channel that two parties have already established between themselves (we discuss how such a channel is established in more detail later). When we write ($\diamondsuit$: *dataToErase*) next to such an arrow, we mean that the value *dataToErase* is securely erased before the public data is transmitted. When we write $\lfloor secretData \rceil_{\text{🔒}}$ on such an arrow, we mean that *secretData* is sent in a non-committing encrypted form, as will be made clear in the sequel. All these transmissions must be secure against adaptive corruptions in the erasure model.

2.2 Indistinguishable Distributions and Ensembles

This section recalls the notion of negligible functions, indistinguishable distributions, and indistinguishable distribution ensembles.

Definition 2.1 (Negligible function [KL15]). *A function $f : \mathbb{N} \mapsto \mathbb{R}_+$ is called* negligible *if for all $c \in \mathbb{N}$ there exists $\eta_0 \in \mathbb{N}$ such that for all $\eta > \eta_0$: $f(\eta) < \eta^{-c}$.*

A function $f(x)$ is called overwhelming *if the function $g(x) := 1 - f(x)$ is negligible.*

Definition 2.2 (Indistinguishable distributions [Gol01]). *Let A and B be two distributions. A and B are ϵ-*(statistically) indistinguishable *if their statistical distance is no more than ϵ, i.e.:*

$$\sum_{a \in \{0,1\}^*} \left| \Pr[A = a] - \Pr[B = a] \right| \leq \epsilon.$$

A and B are *perfectly indistinguishable* if they are 0-indistinguishable, i.e., they are identical.

Definition 2.3 (Computationally indistinguishable ensembles [Gol01]). *Two ensembles* $\{A(x)\}_{x\in\mathcal{L}}$ *and* $\{B(x)\}_{x\in\mathcal{L}}$ *indexed by elements of a language* $\mathcal{L} \subseteq \{0,1\}^*$ *are* computationally indistinguishable *if for every PPT algorithm* D *whose output is in* $\{0,1\}$*, there exists a negligible function* negl *such that for every* $x \in \mathcal{L}$*:*

$$\left| \Pr\left[D(x, A(x)) = 1\right] - \Pr\left[D(x, B(x)) = 1\right] \right| \leq \mathrm{negl}(|x|).$$

We denote the ensemble $\{A(x)\}_{x\in\{1^\eta|\eta\in\mathbb{N},\eta>\eta_0\}}$ by the shorthand $\{A(1^\eta)\}_{1^\eta}$ (where η_0 is implicit), or even $\{A\}_{1^\eta}$ (where the security parameter is implicitly given to A).

2.3 The Decisional Diffie-Hellman (DDH) Assumption

This section recalls the definition of the Decisional Diffie-Hellman Assumption (DDH). We have chosen not to recall the definition of other cryptographic hardness assumptions, as we will not directly need them in the rest of the thesis; instead we rely on the security of the cryptographic schemes that we use.

Definition 2.4 (DDH Assumption [KL15]). *Let* ggen *be a PPT algorithm that takes as input a security parameter* 1^η *and outputs the description of a group* $\mathbb{G}$*, the order* p *of the group, and a generator* g*.*

The DDH assumption for the group generator ggen *holds if the following two ensembles are computationally indistinguishable:*

- $\{(\mathbb{G}, p, g, g^x, g^y, g^z)\}_{1^\eta}$ *for* $(\mathbb{G}, p, g) \xleftarrow{\$} \mathsf{ggen}(1^\eta)$ *and* $x, y, z \xleftarrow{\$} \mathbb{Z}_p$*.*
- $\{(\mathbb{G}, p, g, g^x, g^y, g^{x\cdot y})\}_{1^\eta}$ *for* $(\mathbb{G}, p, g) \xleftarrow{\$} \mathsf{ggen}(1^\eta)$ *and* $x, y \xleftarrow{\$} \mathbb{Z}_p$*.*

DDH over prime-order groups. In Chapter 4, we consider the DDH assumption over prime-order groups. In that context, ggen generates a group of prime order: for example, a prime-order subgroup of $\mathbb{Z}_q$ where q is a large prime; here the size of p in bits should be approximately equal to 2η. Ecrypt-II has made recommendation on size of q relative to the security parameter [BCC+11].

DDH over safe-semiprime-order groups. In Chapter 3, we consider the DDH assumption over a subgroup of $\mathbb{Z}_q$ where the order is a safe semi-prime. We refer to the Ecrypt-II [BCC+11] recommendations for suggestion on the size of the two primes relative to η. We stress that in that context, the factorization of the group order is *not* sent to the distinguisher.

2.4 Cryptographic Building Blocks

This section recalls the definition of the various cryptographic schemes that we use throughout this thesis. It covers encryption schemes, signature schemes, commitment schemes, pseudo-random generators (PRG), linear block codes (LBC), secret sharing schemes (SSS), exposure-resilient functions (ERF), universal hash functions, and all-or-nothing transforms (AoNT).

2.4.1 Public-Key Encryption Scheme

A public-key encryption scheme (also called cryptosystem) is a triplet of PPT algorithms $(\mathsf{kgen}, \mathsf{enc}, \mathsf{dec})$. The algorithm $\mathsf{kgen} : 1^\eta \xmapsto{\$} (pk, sk)$ takes the security parameter as input and outputs a public and a secret key. The algorithm $\mathsf{enc} : (pk, pt) \xmapsto{\$} ct$ takes as input a public key and a plaintext, and outputs a ciphertext. The algorithm $\mathsf{dec} : (sk, ct) \mapsto (pt' \text{ or } \perp)$ takes as input a secret key and a ciphertext, and outputs a plaintext (or an error symbol).

Correctness. A cryptosystem is correct if, except with negligible probability over the key generation $(pk, sk) \xleftarrow{\$} \mathsf{kgen}(1^\eta)$, for all admissible plaintexts pt: $pt = \mathsf{dec}(sk, \mathsf{enc}(pk, pt))$.

Explicit randomness. It is sometimes necessary to make the randomness used during key generation or encryption explicit. To that effect, we sometimes write $\mathsf{kgen} : 1^\eta \xmapsto{\$} (pk, sk, r)$ and $\mathsf{enc} : (pk, pt) \xmapsto{\$} (ct, r)$; here r represents all random choices that were made in the algorithm. We also write $\mathsf{kgen} : (1^\eta, r) \mapsto (pk, sk)$ and $\mathsf{enc} : (pk, pt, r) \mapsto ct$ to denote the deterministic algorithms that take the randomness r as input.

Compatibility with zero-knowledge proofs. A cryptosystem is compatible with $\mathcal{F}_{\text{gzk}}$, if one can efficiently prove knowledge of sk given pk, one can prove knowledge of a plaintext given a ciphertext, and one can prove further equations involving such a plaintext. We will use a shorthand notation to denote such proofs, e.g.: $\between sk, pt : (pk, sk) \in \mathsf{kgen}() \wedge pt = \mathsf{dec}(ct, sk)$ denotes a proof that a (well-formed) secret key and plaintext is known such that a given ciphertext decrypts to that plaintext with the secret key corresponding to a given public key.

2.4.1.1 Security against Chosen-Plaintext Attacks (CPA)

Semantic security game. The CPA security game (also called semantic security game) G_{ss} is defined as follows:

- $\mathsf{G}_{\text{ss}}(1^\eta)$ starts by choosing a public and secret key $(pk, sk) \xleftarrow{\$} \mathsf{kgen}(1^\eta)$ and sends pk to the adversary.
- G_{ss} then expects two plaintexts (pt_0, pt_1) of equal length from the adversary. It then flips a coin $b \xleftarrow{\$} \{0, 1\}$ and encrypts pt_b: $ct \xleftarrow{\$} \mathsf{enc}(pk, pt_b)$. It then sends ct to the adversary.
- G_{ss} now expects a guess $b' \in \{0, 1\}$ from the adversary. It outputs 1 if $b = b'$ and 0 otherwise.

Definition 2.5 (CPA security [KL15]). *A cryptosystem is* CPA-secure *if for all PPT adversaries $\mathcal{A}$, there exists a negligible function* negl*, such that for all security parameters $\eta \in \mathbb{N}$ larger than some constant η_0: the probability that $\mathsf{G}_{\text{ss}}(1^\eta)$ outputs 1 when interacting with $\mathcal{A}(1^\eta)$ is no more than $1/2 + \text{negl}(\eta)$ (here the probability is taken over the random choices of both G_{ss} and $\mathcal{A}$).*

2.4.1.2 Controlled-Ring Homomorphic (CRH) Encryption Schemes

A "controlled ring" homomorphic (CRH) cryptosystem [IPS09] corresponding to a ring family $\mathcal{R}$ is a tuple of PPT algorithms $(\mathsf{rgen}, \mathsf{rkgen}, \mathsf{enc}, \mathsf{dec}, \mathsf{add}, \mathsf{mul})$, where $\mathsf{rgen} : 1^\eta \xmapsto{\$} id$ generates a ring identifier from a security parameter, and $\mathsf{rkgen} : (1^\eta, id) \xmapsto{\$} (pk, sk)$ generates a key pair from the security parameter and ring identifier, with the following properties:

- Let $\mathsf{kgen}(1^\eta) \xmapsto{\$} (pk, sk)$ be defined as follows: $id \xleftarrow{\$} \mathsf{rgen}(1^\eta)$ then output the result of $\mathsf{rkgen}(1^\eta, id)$. It is required that $(\mathsf{kgen}, \mathsf{enc}, \mathsf{enc})$ is a semantically secure cryptosystem. The set of values that can be encrypted by enc are the elements of $\mathcal{R}_{id}$.
- Let negl be a negligible function. For any security parameter $\eta \in \mathbb{N}$ larger than some constant η_0: for any two plaintexts pt_1, pt_2: let $(pk, sk) \xleftarrow{\$} \mathsf{kgen}(1^\eta)$, let ct_1, ct_2 be two ciphertexts such that $ct_i \xleftarrow{\$} \mathsf{enc}(pk, pt_i)$, and let $ct_3 \xleftarrow{\$} \mathsf{add}(pk, ct_1, ct_2)$, and let $ct_4 \xleftarrow{\$} \mathsf{enc}(pk, pt_1 + pt_2)$. The distributions of ct_3 and ct_4 must be $\mathrm{negl}(\eta)$-indistinguishable.
- Let negl be a negligible function. For any security parameter $\eta \in \mathbb{N}$ larger than some constant η_0: for any two plaintexts pt, α: let $(pk, sk) \xleftarrow{\$} \mathsf{kgen}(1^\eta)$, let ct be the ciphertext such that $ct \xleftarrow{\$} \mathsf{enc}(pk, pt)$, and let $ct' \xleftarrow{\$} \mathsf{mul}(pk, ct, \alpha)$, and let $ct'' \xleftarrow{\$} \mathsf{enc}(pk, pt \cdot \alpha)$. The distributions of ct' and ct'' must be $\mathrm{negl}(\eta)$-indistinguishable.

In the sequel, we drop pk from add and mul if the public key is clear from context.

Examples. An example of a CRH cryptosystem is Camenisch-Shoup [CS03, DJ03]. In Chapter 3, we will use Jarecki and Shmatikov's variant of the latter [JS07], described hereafter. The Pailler cryptosystem [Pai99], is an *uncontrolled*-ring homomorphic cryptosystem [IPS09], as the plaintext ring cannot be fixed before key generation. The ElGamal cryptosystem (and the variant thereof where messages are in the exponent) does not fit this definition.

Simplified Camenisch-Shoup Encryption with Short Randomness. An example of a CRH encryption scheme that is compatible with $\mathcal{F}_{\mathrm{gzk}}$ is the simplified version of Camenisch-Shoup encryption with a short private key and short randomness, described by Jarecki and Shmatikov [JS07]. It is semantically secure if Paillier's Decision Composite Residuosity Assumption [Pai99] holds.

The ring generation algorithm outputs is a safe semiprime $n := id$: messages are then in $\mathbb{Z}_n$. Note that the factorization of n is not output. We refer to the Ecrypt-II [BCC+11] recommendations for suggestion on the size of n relative to η.

The key generation algorithm computes $x \xleftarrow{\$} \mathbb{Z}_{\lfloor \sqrt{n} \rfloor}$, $g' \xleftarrow{\$} \mathbb{Z}^*_{n^2}$, $g \leftarrow g'^{2n}$, $y \leftarrow g^x$. It then outputs the public key $pk := (g, y)$ and the secret key $sk := x$.

To encrypt the message $pt \in \mathbb{Z}_n$: $r \xleftarrow{\$} \mathbb{Z}_{\lfloor \sqrt{n} \rfloor}$, $u \leftarrow g^r$, $e \leftarrow y^r(n+1)^{pt} \pmod{n^2}$; the ciphertext is $ct := (u, e)$.

To decrypt: $pt''' \leftarrow (e/u^x)^2$, $pt'' \leftarrow \frac{pt'''-1}{n}$ (over the integers), $pt' \leftarrow pt'' \cdot 2^{-1} \pmod{n}$; output pt'.

2.4.1.3 Security against Chosen-Ciphertext Attacks (CCA2)

Labels. Some cryptosystems allow one to attach a non-malleable label ℓ to ciphertexts. Thereby enc is now $(pk, pt, \ell) \overset{\$}{\mapsto} ct$ and dec is now $(sk, ct, \ell) \mapsto (pt' \text{ or } \perp)$. One can also make the randomness explicit in the obvious manner.

Such a crytosystem with labels is correct if for all labels ℓ: the (regular) cryptosystem where the label is fixed to ℓ is correct.

CCA-2 security game (with labels). The following CCA-2 security game G_{cca} for a cryptosystem with labels is adapted from Shoup [Sho01].

- $\mathsf{G}_{\text{cca}}(1^\eta)$ starts by choosing a public and secret key $(pk, sk) \overset{\$}{\leftarrow} \mathsf{kgen}(1^\eta)$ and sends pk to the adversary. The first phase now starts.
- Upon receiving $(\texttt{oracle}, ct, \ell)$ from the adversary during the first phase: G_{cca} decrypts the ciphertext $pt \leftarrow \mathsf{dec}(sk, ct, \ell)$ and sends pt to the adversary. This query may be repeated many times.
- Upon receiving $(\texttt{challenge}, pt_0, pt_1, \ell')$ from the adversary where pt_0 and pt_1 have the same size: G_{cca} flips a coin $b \overset{\$}{\leftarrow} \{0,1\}$, encrypts pt_b: $ct' \overset{\$}{\leftarrow} \mathsf{enc}(pk, pt_b, \ell')$, and remembers ℓ'. It then sends ct' to the adversary. The second phase now starts.
- Upon receiving $(\texttt{oracle}, ct, \ell)$ from the adversary during the second phase such that $(ct, \ell) \neq (ct', \ell')$: G_{cca} decrypts the ciphertext $pt \leftarrow \mathsf{dec}(sk, ct, \ell)$ and sends pt to the adversary. This query may be repeated many times.
- Upon receiving $(\texttt{guess}, b')$ from the adversary: G_{cca} outputs 1 if $b = b'$ and 0 otherwise.

Definition 2.6 (CCA-2 security [Sho01]). *A cryptosystem is* CCA-2-secure *if for all PPT adversaries $\mathcal{A}$, there exists a negligible function* negl*, such that for all security parameters $\eta \in \mathbb{N}$ larger than a constant η_0: the probability that $\mathsf{G}_{\text{cca}}(1^\eta)$ outputs 1 when interacting with $\mathcal{A}(1^\eta)$ is no more than $1/2 + \text{negl}(\eta)$ (here the probability is taken over the random choices of both G_{cca} and $\mathcal{A}$).*

Example. The Cramer-Shoup cryptosystem in a hybrid setting over a group $\mathbb{G}$ of prime order q [CS98, Section 5.2], modified so that it includes the label ℓ in the hash function used during encryption, is an example of a cryptosystem with labels. Such a cryptosystem is CCA-2 secure provided that the DDH assumption holds.

2.4.2 Signature Scheme

A signature scheme is a triplet of PPT algorithms $(\mathsf{gen}, \mathsf{sig}, \mathsf{ver})$. The algorithm $\mathsf{gen} : 1^\eta \overset{\$}{\mapsto} (pk, sk)$ takes a security parameter as input and outputs a public and secret key. The algorithm $\mathsf{sig} : (m, sk) \overset{\$}{\mapsto} \sigma$ takes as input a message and a secret key, and outputs a signature. The algorithm $\mathsf{ver} : (m, \sigma, pk) \mapsto b$ takes as input a message, a signature, and a public key, and outputs a bit that indicates whether the signature is valid.

Correctness. A signature scheme is correct if, except with negligible probability over the key generation $(pk, sk) \overset{\$}{\leftarrow} \mathsf{gen}(1^\eta)$, it holds that for every legal message m: $1 = \mathsf{ver}(m, \mathsf{sig}(m, sk), pk)$.

Existential unforgeability against chosen-message-attack (EUF-CMA) game. The existential unforgeability against chosen-message-attack (EUF-CMA) game $\mathsf{G}_{\text{eufcma}}$ is defined as follows:

- $\mathsf{G}_{\text{eufcma}}(1^\eta)$ starts by generating a key pair: $(pk, sk) \xleftarrow{\$} \mathsf{gen}(1^\eta)$. It initializes a list of messages $\mathsf{msglist}$. Finally, it sends pk to the adversary.
- Whenever receiving $(\texttt{sign}, m)$ from the adversary, sign the message $\sigma \leftarrow \mathsf{sig}(m, sk)$ and add m to $\mathsf{msglist}$. Send σ to the adversary.
- Upon receiving $(\texttt{forge}, m, \sigma)$ where $m \notin \mathsf{msglist}$, and such that $\mathsf{ver}(m, \sigma, pk) = 1$: output 1.

Definition 2.7 (EUF-CMA security [KL15]). *A correct signature scheme is* EUF-CMA secure *if for all PPT adversaries $\mathcal{A}$, there is a negligible function* negl*, such that for all security parameters η larger than a constant η_0, the probability that $\mathsf{G}_{\text{eufcma}}(1^\eta)$ outputs 1 when interacting with $\mathcal{A}(1^\eta)$ is no more than* $\text{negl}(\eta)$ *(here the probability is taken over the random choices of both $\mathsf{G}_{\text{eufcma}}$ and $\mathcal{A}$).*

2.4.3 Commitment Scheme

A commitment scheme is a triplet of PPT algorithms $(\mathsf{cgen}, \mathsf{com}, \mathsf{cvfy})$. The algorithm cgen takes as input a group description and outputs parameters pc. The algorithm $\mathsf{com}(pc, a) \xmapsto{\$} (ca, oa)$ takes as input these parameters and a value to commit, and outputs a commitment and an opening. The algorithm $\mathsf{cvfy}(pc, ca, oa, a) \mapsto \{0, 1\}$ verifies whether oa is a valid opening for commitment ca and the committed value a.

In the sequel, we drop pc from com and cvfy if it is clear from context which parameters are used. We also write $\mathsf{com}(a, oa) \mapsto ca$ when the opening is chosen outside of the commitment algorithm.

Correctness. A commitment scheme is correct, if except for negligible probability over the group generation and parameter generation $pc \xleftarrow{\$} \mathsf{cgen}(\mathsf{ggen}(1^\eta))$, for all admissible values a we have that $(ca, oa) \xleftarrow{\$} \mathsf{com}(pc, a)$ implies that $\mathsf{cvfy}(pc, ca, oa, a) = 1$.

Hiding game. The hiding game for commitments G_{ch} is defined as follows [KL15]:

- $\mathsf{G}_{\text{ch}}(1^\eta)$ starts by choosing parameters $pc \xleftarrow{\$} \mathsf{cgen}(\mathsf{ggen}(1^\eta))$ and sends pc to the adversary.
- G_{ch} then expects two messages a_0 and a_1 from the adversary. It then flips a coin $b \xleftarrow{\$} \{0, 1\}$ and commits to a_b: $(ca_b, oa_b) \xleftarrow{\$} \mathsf{com}(pc, a_b)$. It then sends ca_b to the adversary.
- G_{ss} now expects a guess $b' \in \{0, 1\}$ from the adversary. It outputs 1 if $b = b'$ and 0 otherwise.

Binding game. The binding game for commitments G_{cb} is defined as follows [KL15]:

- $\mathsf{G}_{\text{cb}}(1^\eta)$ starts by choosing parameters $pc \xleftarrow{\$} \mathsf{cgen}(\mathsf{ggen}(1^\eta))$ and sends pc to the adversary.
- G_{cb} then expects two messages a_0 and a_1, two openings oa_0 and oa_1, and a single commitment ca from the adversary. It outputs 1 if $a_0 \neq a_1$, $\mathsf{cvfy}(pc, ca, oa_0, a_0) = 1$ and $\mathsf{cvfy}(pc, ca, oa_1, a_1) = 1$ and 0 otherwise.

Definition 2.8 (Perfectly-hiding secure commitment scheme [KL15]). *A commitment scheme is perfectly hiding and secure for a given group generator* $\mathsf{ggen}(1^\eta)$ *if:*

- *It is correct.*
- *For all (not necessarily PPT) adversaries* $\mathcal{A}$ *and for all security parameters* $\eta \in \mathbb{N}$ *larger than some constant* η_0*: the probability that* $\mathsf{G}_{\mathrm{ch}}(1^\eta)$ *outputs 1 when interacting with* $\mathcal{A}$ *is exactly* $1/2$ *(here the probability is taken over the random choices of both* G_{ch} *and* $\mathcal{A}$*).*
- *For all PPT adversaries* $\mathcal{A}$*, there exists a negligible function* negl*, such that for all security parameters* $\eta \in \mathbb{N}$ *larger than some constant* η_0*: the probability that* $\mathsf{G}_{\mathrm{cb}}(1^\eta)$ *outputs 1 when interacting with* $\mathcal{A}$ *is no more than* $\mathrm{negl}(\eta)$ *(here the probability is taken over the random choices of both* G_{cb} *and* $\mathcal{A}$*).*

Definition 2.9 (Statistically-biding secure commitment scheme [KL15]). *A commitment scheme is statistically-binding and secure for a given group generator* $\mathsf{ggen}(1^\eta)$ *if:*

- *It is correct.*
- *For all PPT adversaries* $\mathcal{A}$*, there exists a negligible function* negl*, such that for all security parameters* $\eta \in \mathbb{N}$ *larger than some constant* η_0*: the probability that* $\mathsf{G}_{\mathrm{ch}}(1^\eta)$ *outputs 1 when interacting with* $\mathcal{A}$ *is no more than* $1/2 + \mathrm{negl}(\eta)$ *(here the probability is taken over the random choices of both* G_{ch} *and* $\mathcal{A}$*).*
- *For all (not necessarily PPT) adversaries* $\mathcal{A}$*, there exists a negligible function* negl*, such that for all security parameters* $\eta \in \mathbb{N}$ *larger than some constant* η_0*: the probability that* $\mathsf{G}_{\mathrm{cb}}(1^\eta)$ *outputs 1 when interacting with* $\mathcal{A}$ *is no more than* $\mathrm{negl}(\eta)$ *(here the probability is taken over the random choices of both* G_{cb} *and* $\mathcal{A}$*).*

Compatibility with zero-knowledge proofs. A commitment scheme is compatible with $\mathcal{F}_{\mathrm{gzk}}$, if one can efficiently prove knowledge of the committed value a and the opening oa given the commitment ca, and one can prove further equations involving a and oa. We will use a shorthand notation to denote such proofs, e.g.: $\nearrow a, oa : \mathsf{cvfy}(ca, oa, a)$.

Homomorphic commitments. A commitment scheme is homomorphic if there exists two PPT algorithms $\mathsf{cadd}(pc, ca_0, ca_1, \ldots) \mapsto cs$ and $\mathsf{cmul}(pc, ca, b) \mapsto cp$, such that: $(\forall i : 1 = \mathsf{cvfy}(pc, ca_i, oa_i, a_i))$ implies that $1 = \mathsf{cvfy}(pc, cs, \sum_i oa_i, \sum_i a_i)$; and $1 = \mathsf{cvfy}(pc, ca, oa, a)$ implies that $1 = \mathsf{cvfy}(pc, cp, oa \cdot b, a \cdot b)$. In the sequel, we drop pc from cadd and cmul if it is clear from context which parameters are used.

2.4.4 Pseudo-Random Generator (PRG)

An ℓ-pseudo-random generator (PRG) is a polynomial-time algorithm $\mathsf{prg}(\mathbf{a}) \mapsto \mathbf{x}$, that takes as input a seed $\mathbf{a} \in \{0,1\}^*$ and outputs a string $\mathbf{x} \in \{0,1\}^*$; let $\ell(n)$ be a polynomial with $\forall n \in \mathbb{N} : \ell(n) > n$, it is required that $|\mathbf{x}| = \ell(|\mathbf{a}|)$.

Definition 2.10 (Computationally secure PRG [KL15]). *An* ℓ*-PRG is secure if the following two ensembles are computationally indistinguishable:* $\{\mathbf{b}\}_{1^n}$ *for* $\mathbf{b} \xleftarrow{\$} \{0,1\}^{\ell(\eta)}$*; and* $\{\mathsf{prg}(\mathbf{a})\}_{1^n}$ *for* $\mathbf{a} \xleftarrow{\$} \{0,1\}^\eta$.

2.4.5 Linear Block Code (LBC)

A linear block code over a field Φ, with block length n and message length k is denoted by: (Φ, n, k)-LBC. Such an LBC is described by a $k \times n$ generator matrix $\mathbf{G}$ whose entries are in Φ. To encode a message $\mathbf{a}$ (a row vector of length n whose entries are in Φ), one multiplies it with the generator matrix: $\mathbf{x} \leftarrow \mathbf{aG}$.

An LBC has minimum distance d if one can still recover the original message $\mathbf{a}$ from $\mathbf{x}$ even if up to $(d-1)$ positions of $\mathbf{x}$ have been erased. This is equivalent to requiring that all $k \times (n-(d-1))$ sub-matrices of $\mathbf{G}$ have rank k.

2.4.6 Ramp Secret Sharing Scheme (SSS)

A (ramp) secret sharing scheme consists of two algorithms $\mathsf{senc} : \mathbf{a} \stackrel{\$}{\mapsto} \mathbf{x}$ and $\mathsf{sdec} : (\mathbf{x}, L) \mapsto \mathbf{a}$. The algorithm senc takes as input a message of length k symbols of some field Φ, and outputs m shares (each share is also an element of Φ). The algorithm sdec takes as input n shares and a list of the indexes of those shares, and outputs a message.

Reconstruction. The SSS has perfect reconstruction if the original message can be recovered by decoding with exactly n shares.

Privacy. The SSS has a privacy threshold of d if d shares reveal nothing about the message.

We denote a (ramp) SSS scheme over a field Φ, for message length k, outputting m shares, where decode expects n shares, with perfect reconstruction, and with privacy threshold d as: (Φ, m, n, d, k)-SSS. Statistically secure and computationally secure SSS also exist, but we will not need them for this thesis (but see Section 2.4.9).

Example of perfect SSS's are Shamir's secret sharing scheme [BM84] and variants thereof [FY92, CC06].

2.4.7 Exposure Resilient Function (ERF)

An exposure resilient function (ERF) [CDH+00] is an algorithm $\mathsf{erf}(\mathbf{b}) \mapsto \mathbf{x}$ that takes as input a random string and outputs a key. It is secure if the output is indistinguishable from random even if up to a certain number of symbols of the input are known.

Definition 2.11 (Statistically secure ERF [CDH+00]). *A d-ERF* $\mathsf{erf} \in \Phi^n \mapsto \Phi^k$ *over some field* Φ *is* ϵ*-secure if for any set* $L \subset \{1, \ldots n\}$ *of size at most d, the following two distributions are* ϵ*-indistinguishable:*

- $([\mathbf{b}]_L, \mathbf{x}_0)$ *for* $\mathbf{b} \stackrel{\$}{\leftarrow} \Phi^n$ *and* $\mathbf{x}_0 \leftarrow \mathsf{erf}(\mathbf{b})$.
- $([\mathbf{b}]_L, \mathbf{x}_1)$ *for* $\mathbf{b} \stackrel{\$}{\leftarrow} \Phi^n$ *and* $\mathbf{x}_1 \stackrel{\$}{\leftarrow} \Phi^k$.

We denote such an erf as (an ϵ-secure) (Φ, n, d, k)-ERF. Perfect security means that $\epsilon = 0$.

Computational security. In the context of computational security, the function erf takes as additional input a (usually implicit) security parameter; Φ, n, k, and d may depend on that security parameter.

Definition 2.12 (Computationally secure ERF [CDH+00]). *A (PPT) $d(\eta)$-ERF $\mathsf{erf}(1^\eta, \mathbf{b}) \mapsto \mathbf{x}$ where $\mathbf{b} \in \Phi(\eta)^{n(\eta)}$ is (computationally) secure if for any set $L \subset \{1, \dots n(\eta)\}$ of size at most $d(\eta)$, the following two ensembles are computationally indistinguishable:*

- $\{([\mathbf{b}]_L, \mathbf{x}_0)\}_{1^\eta}$ *for* $\mathbf{b} \xleftarrow{\$} \Phi(\eta)^{n(\eta)}$ *and* $\mathbf{x}_0 \leftarrow \mathsf{erf}(1^\eta, \mathbf{b})$.
- $\{([\mathbf{b}]_L, \mathbf{x}_1)\}_{1^\eta}$ *for* $\mathbf{b} \xleftarrow{\$} \Phi(\eta)^{n(\eta)}$ *and* $\mathbf{x}_1 \xleftarrow{\$} \Phi(\eta)^{k(\eta)}$.

In the sequel we also denote such a secure ERF as (Φ, n, d, k)-ERF, where the security parameter is implicit.

2.4.8 Universal Hash Function

Definition 2.13 (Universal hash function [CW79]). *A function $\mathsf{h} : \Sigma \times \Phi \mapsto \Phi'$ is called an ϵ-*almost universal hash function *if:*

$$\forall \mathbf{x}, \mathbf{x}' \in \Phi \text{ such that } \mathbf{x} \neq \mathbf{x}' : \Pr\left[\mathbf{k} \xleftarrow{\$} \Sigma; \mathsf{h}(\mathbf{k}, \mathbf{x}) = \mathsf{h}(\mathbf{k}, \mathbf{x}')\right] \leq \epsilon.$$

If $\epsilon = 1/|\Phi'|$, one simply says that h is a *universal hash function.*

2.4.9 All-or-Nothing Transform (AoNT)

An all-or-nothing transform (AoNT) [CDH+00] is similar to an SSS with $m = n$, and consists of two algorithms $\mathsf{aenc}(\mathbf{a}) \xmapsto{\$} \mathbf{x}$ and $\mathsf{adec}(\mathbf{x}) \mapsto \mathbf{a}$.

Definition 2.14 (Statistically secure AoNT [CDH+00]). *A d-AoNT with $\mathsf{aenc} \in \Phi^k \xmapsto{\$} \Phi^n$ and $\mathsf{adec} \in \Phi^n \mapsto \Phi^k$ over some field Φ is ϵ-secure if:*

- *For all messages $\mathbf{a} \in \Phi^k$, $\mathbf{a} = \mathsf{adec}(\mathsf{aenc}(\mathbf{a}))$.*
- *For any set $L \subset \{1, \dots n\}$ of size at most d, and for any two messages $\mathbf{a}_0, \mathbf{a}_1 \in \Phi^k$ the following two distributions are ϵ-indistinguishable:*
  $(\mathbf{a}_0, \mathbf{a}_1, [\mathsf{aenc}(\mathbf{a}_0)]_L)$ and $(\mathbf{a}_0, \mathbf{a}_1, [\mathsf{aenc}(\mathbf{a}_1)]_L)$.

We denote such an AoNT as (an ϵ-secure) (Φ, n, d, k)-AoNT. Perfect security means that $\epsilon = 0$.

Computational security. In the context of computational security, the two functions aenc and adec take as additional input a (usually implicit) security parameter; Φ, n, k, and d may depend on that security parameter.

Definition 2.15 (Computationally secure AoNT [CDH+00]). *A $d(\eta)$-AoNT with (PPT) $\mathsf{aenc}(1^\eta, \mathbf{a}) \xmapsto{\$} \mathbf{x}$ and (PPT) $\mathsf{adec}(1^\eta, \mathbf{x}) \mapsto \mathbf{a}$ where $\mathbf{a} \in \Phi(\eta)^{k(\eta)}$ and $\mathbf{x} \in \Phi(\eta)^{n(\eta)}$ is (computationally) secure if:*

- *For all security parameters η larger than a constant η_0 and for all messages $\mathbf{a} \in \Phi(\eta)^{k(\eta)}$:* $\mathbf{a} = \mathsf{adec}(\mathsf{aenc}(m))$.
- *For any set $L \subset \{1, \ldots n(\eta)\}$ of size at most $d(\eta)$, and for any two messages $\mathbf{a}_0, \mathbf{a}_1 \in \Phi^{k(\eta)}$ the following two ensembles are computationally indistinguishable:* $\{(\mathbf{a}_0, \mathbf{a}_1, [\mathsf{aenc}(1^\eta, \mathbf{a}_0)]_L)\}_{1^\eta}$ *and* $\{(\mathbf{a}_0, \mathbf{a}_1, [\mathsf{aenc}(1^\eta, \mathbf{a}_1)]_L)\}_{1^\eta}$.

In the sequel we also denote such a secure AoNT as (Φ, n, d, k)-AoNT, where the security parameter is implicit.

AoNT with public part. A $(\Phi, n+\nu, d, k)$-AoNT has a ν-public part, if in the above definitions the last ν symbols of $\mathsf{aenc}(\mathbf{a})$ are output in addition to $[\mathsf{aenc}(\mathbf{a})]_L$.

Realization from a SSS. It is easy to realize a perfect (Φ, n, d, k)-AoNT from any (Φ, m, n, d, k)-SSS, by simply ignoring all shares after the first n ones. This technique also works in the statistical and computational case.

Realization from an ERF. It is easy to realize an ϵ-secure $(\Phi, n+k, d, k)$-AoNT with a k-public part from any ϵ-secure (Φ, n, d, k)-ERF [CDH+00]: $\mathsf{aenc}(\mathbf{a}) \overset{\$}{\mapsto} \mathbf{b}||(\mathsf{erf}(\mathbf{b}) + \mathbf{a})$ where $\mathbf{b} \overset{\$}{\leftarrow} \Phi^n$; and $\mathsf{adec}(\mathbf{b}||\mathbf{x}) \mapsto \mathbf{x} - \mathsf{erf}(\mathbf{b})$. This technique also works in the computational case.

2.4.9.1 Perfect AoNTs based on Shamir's Secret Sharing

Blakely and Meadows's secret sharing scheme [BM84] can be used to directly realize a perfect $(\Phi, (k+d), d, k)$-AoNT for all $(k, d) \in \mathbb{N}^2$, all fields Φ, and where $(2k+d) < |\Phi|$. It is based on Shamir's secret sharing scheme.

Franklin and Young's ramp secret sharing scheme [FY92] can be used to directly realize an AoNT with the same parameters, but the bound on the field size is now improved to $(k+d) < |\Phi|$. Their scheme uses polynomial interpolation over $\mathrm{GF}(|\Phi|)$. In a nutshell, their scheme works as follows. To encode a message: choose a polynomial of degree $(k+d-1)$ over $\mathrm{GF}(|\Phi|)$; set the first k coefficients to be equal to the message, and the other d coefficients randomly. Evaluate the polynomial at $(k+d)$ distinct non-zero locations. To decode, use Lagrange interpolation to recover the coefficients of the polynomial.

Working in small fields. The two schemes above require the field size $|\Phi|$ to depend on the parameters k and d. If one needs an AoNT that operates on a smaller field Σ, e.g., $\mathrm{GF}(2)$, one can simply encode each element of Φ as multiple elements of Σ. Thereby one immediately gets an ϵ-secure $(\Sigma, (\alpha \cdot (k+d)), d, \alpha k)$-AoNT from an ϵ-secure $(\Phi, (k+d), d, k)$-AoNT where $\alpha = \log(|\Phi|)/\log(|\Sigma|) \in \mathbb{N}$. Notice that the realized AoNT only achieves a privacy threshold of d and not αd: intuitively, if parts of a symbol is leaked, one must consider the whole symbol to be compromised.

There exist more complex secret sharing scheme's that can be use to realize AoNTs with better parameters, such as Chen and Cramer's ramp secret sharing scheme [CC06] based on curves of high genus. We do not consider their results further in this chapter.

2.4.9.2 Perfect AoNTs from Linear Block Codes

Linear block codes can be used to create perfect ERFs, and thus by using the standard transformation by Canetti et al. [CDH+00], can be used to create perfect AoNTs.

Let $\mathbf{G}$ be the $k \times n$ matrix with elements in GF(q) with minimum distance d (i.e., $\mathbf{G}$ is the generator matrix of a linear block code with minimum distance d).

Let $\mathbf{M}$ be the following $(n+k) \times (n+k)$ matrix:

$$\mathbf{M} := \begin{bmatrix} \mathbf{I}_n & \mathbf{0} \\ \mathbf{G} & \mathbf{I}_k \end{bmatrix}.$$

To encode the data column-vector $\mathbf{a} \in \mathrm{GF}(q)^k$, $\mathsf{aenc}(\mathbf{a})$ selects a random column-vector $\mathbf{b} \leftarrow \mathrm{GF}(q)^n$, and returns the vector

$$\mathbf{y} \leftarrow \mathbf{M} \begin{bmatrix} \mathbf{b} \\ \mathbf{a} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \mathbf{G}\mathbf{b} + \mathbf{a} \end{bmatrix} := \begin{bmatrix} \mathbf{b} \\ \mathbf{x} \end{bmatrix}.$$

To reconstruct the data, $\mathsf{adec}(\mathbf{y})$ computes

$$\begin{bmatrix} \mathbf{b} \\ \mathbf{a} \end{bmatrix} \leftarrow \mathbf{M}^{-1}\mathbf{y} = \begin{bmatrix} \mathbf{b} \\ \mathbf{x} - \mathbf{G}\mathbf{b} \end{bmatrix}$$

and outputs $\mathbf{a}$. Hence this AoNT is a perfect $(\Phi, (n+k), d, k)$-AoNT.

2.4.9.3 Statistical AoNTs from Universal Hash Functions

Universal hash functions can be used to create very good statistical ERFs, and hence (using the standard transformation) very good AoNTs [CEGL08a, Lim08, CDH+00, BBCM95]. Given a $\{0,1\}^\nu \times \{0,1\}^n \mapsto \{0,1\}^k$ universal hash function h, one can realize the following $(\{0,1\}, n + (\nu + k), d, k)$-AoNT with a $(\nu + k)$-public part:

- $\mathsf{aenc}(\mathbf{a})$: Choose $\mathbf{b} \xleftarrow{\$} \{0,1\}^n$ and $\mathbf{k} \xleftarrow{\$} \{0,1\}^\nu$. Set $\mathbf{x} \leftarrow \mathsf{h}(\mathbf{k}, \mathbf{b}) + \mathbf{a}$. Return $\mathbf{b}||\mathbf{k}||\mathbf{x}$. (Here $\mathbf{k}$ and $\mathbf{x}$ are in the public part.)
- $\mathsf{adec}(\mathbf{b}||\mathbf{k}||\mathbf{x}) \mapsto \mathbf{x} - \mathsf{h}(\mathbf{k}, \mathbf{b})$.

The AoNT is $(2 \cdot 2^{(k+d-n)/2})$-secure [CDH+00, CEGL08b].

2.4.9.4 Computational AoNTs

Canetti et al. [CDH+00] showed how to stretch the output of an ERF with a PRG to obtain an ERF with larger output size (but the privacy threshold will remain unchanged). Let erf be a computationally secure (Φ, n, d, k)-ERF and let prg be a secure (Φ, k, m)-PRG. Then the ERF: $\mathbf{b} \mapsto \mathsf{prg}(\mathsf{erf}(\mathbf{b}))$ is a computationally secure (Φ, n, d, m)-ERF. From that, one can realize a computationally secure AoNT.

2.5 Composability Frameworks

Protocols constructed for and proven secure in a composability framework can be securely composed in arbitrary ways. Many such frameworks have been developed [Can00, Can01, PW00, MR11, HS11, KT13, CDPW07].

All of these frameworks define the security and functional properties of a protocol by specifying an *ideal process*, where the output of the latter is computed by a trusted party that sees the inputs of all parties [GMW87]. A real protocol realizes the ideal process if no efficient distinguisher can decide whether it is interacting with the ideal process plus a *simulator* or the real protocol plus an adversary.

The composition theorem in the various frameworks then guarantees that a secure protocol π can be used in arbitrary contexts: more precisely, a secure higher-level protocol that uses the ideal version of π as a subroutine remains secure when it uses π directly.

In this thesis we use the UC framework [Can00, Can01], the GNUC framework [HS11, HS15], the IITM framework [Kue06, KT13] with responsive environments [CEK+16a], and the constructive cryptography (CC) framework [MR11, Mau11b, Mau10]. A short summary of these frameworks follows.

2.5.1 The UC Framework

We now give an informal overview of the UC framework (version of 2013) [Can00], discuss how corruption is handled in that framework, give a brief overview of universal composition with joint state, and finally discuss some of the problems with the framework.

2.5.1.1 The UC Framework in a Nutshell

The UC framework is the most popular framework for representing cryptographic protocols and analysing their security. The framework defines a model of protocol execution, where programs are modelled as interactive Turing machines (ITM). The adversary is modelled as an additional ITM, and has some control over the scheduling of messages between individual machines. Informally, a real protocol is said to realize an ideal protocol (and is thus *secure*), if for all adversaries there exists an ideal adversary (or simulator) such that no environment can distinguish a protocol execution of the real protocol with the adversary (real world) from an execution of the ideal protocol with the simulator. The security of protocols is preserved when the protocol is combined with itself or other protocols that are run in an adversarially controlled manner, thanks to the *universal composition* operation. The following overview is adapted from Section 2 of [Can00] (version of 2013).

The computational model. The basic computing unit, called a machine, represents a running instance of an algorithm. Such a machine can be formalized as an interactive Turing machine.

Several such machines may run alongside each other and provide information to each other. A machine M can provide information to a machine M' in three ways: either by providing *input*, providing *subroutine output*, or by *sending a message*.

An *execution* of several such machines $M_1, M_2, \ldots$ on input x starts by running the initial machine M_1 with input x. Thereafter, whenever a machine M provides information to a machine M', M is suspended, and the execution of M' begins. Thereby, at any single point in time, only a single machine is active.

Protocols. A *protocol* is an algorithm for a distributed system, i.e., a collection of computer programs that are to be run by different participants. An *instance* of a protocol within a system of machines is a sequence of machines that relate to each other as part of a protocol execution.

Some machines in a protocol may be *subroutines* of other machines. If a machine M' is a subroutine of machine M, then M will provide input to M' and M' will provide subroutine output to M. In an instance of a protocol π, a machine that is not the subroutine of any other machine is also called a *main party* of that instance.

Polynomial time machines and protocols. The UC framework restricts its attention to systems where each of the machines runs in probabilistic polynomial time (PPT) to the difference between the number of bits it received as input and the number of bits it output to subroutines. As pointed out by the Hofheinz and Shoup [HS11], there are issues with this definition, especially since formally one needs to provide variable amounts of padding to the input of a protocol depending on its intended realization (number of subroutines).

Protocol execution (bare model). The "real-world" execution of a protocol π consists of a system of machines $(\mathcal{E}, \mathcal{A}, \pi_1, \ldots, \pi_m)$, where $\mathcal{E}$ is called the *environment*, $\mathcal{A}$ the *adversary*, and where $\pi_1, \ldots, \pi_m$ are then machines of the instance of π.

The machines can interact as follows. The environment $\mathcal{E}$ can provide inputs to the adversary and to the main parties of the instance. The adversary $\mathcal{A}$ can provide subroutine output to the environment or send a message to any of the machines π_i. A machine π_i may send messages to the adversary, provide inputs to its subroutines, and provide subroutine output to the machine it is a subroutine of (i.e., to its parent). Main parties may additionally provide subroutine output to the environment.

Let $\mathrm{Exec}(\pi, \mathcal{A}, \mathcal{E})(z)$ denote the random variable (over the local random choices of all machines) describing the output of the environment $\mathcal{E}$ that is given input z in a system of machines $(\mathcal{E}, \mathcal{A}, \pi_1, \ldots, \pi_m)$ as described above. Without loss of generality, such output is a single bit. Let $\mathrm{Exec}(\pi, \mathcal{A}, \mathcal{E})$ denote the ensemble $\{\mathrm{Exec}(\pi, \mathcal{A}, \mathcal{E})(1^\eta)\}_{1^\eta}$ of distributions over $\{0,1\}$.[1]

Ideal functionalities and protocols. An ideal process is a special protocol in the above model of protocol execution. Such an ideal protocol consists of an ideal functionality $\mathcal{F}$ that interacts with a certain number of *dummy* (main) parties; this ideal protocol is denoted $\textsc{ideal}_{\mathcal{F}}$. The ideal functionality can be though of as a program for a trusted party, and is a subroutine of all the dummy parties.

The dummy parties forward all input to $\mathcal{F}$ and forward all subroutine output from $\mathcal{F}$ to the environment as subroutine output.

[1] In the non-uniform model of computation, one would consider the following ensemble instead: $\{\mathrm{Exec}(\pi, \mathcal{A}, \mathcal{E})(z)\}_{z \in \{0,1\}^*}$.

Definition 2.16 (Protocol emulation). *A protocol* π (UC-)emulates *protocol* ϕ *if for any adversary* $\mathcal{A}$*, there exists an adversary* $\mathcal{S}$*—also called* simulator*—such that, for any environment* $\mathcal{E}$*, the ensembles* $\text{Exec}(\pi, \mathcal{A}, \mathcal{E})$ *and* $\text{Exec}(\phi, \mathcal{S}, \mathcal{E})$ *are (computationally) indistinguishable.*

Definition 2.17 (Realizing functionalities). *Protocol* π (UC-)realizes *an ideal functionality* $\mathcal{F}$ *if* π *emulates* $\text{IDEAL}_{\mathcal{F}}$*.*

Using protocols as subroutines. A protocol ρ uses protocol ϕ as a subroutine if some or all the programs in ρ use programs of ϕ as a subroutine. Stated differently, a protocol instance $(\phi_1, \phi_2, \ldots)$ is a subroutine of protocol instance $(\rho_1, \rho_2, \ldots)$, if each machine ϕ_i appears as a machine in $(\rho_1, \rho_2, \ldots)$ and is a subroutine of some other machine ρ_j. An instance of ρ may use multiple subroutine instances of ϕ.

Subroutine substitution. Let ρ be a protocol that uses ϕ as subroutine, and let π be a protocol that emulates ϕ. Then by $\rho^{\phi \to \pi}$ one denotes the protocol that is identical to ρ except that: for each instance of ϕ that is a subroutine of this instance of ρ, and for all i, the ith machine of the instance of ϕ is replaced by the ith machine of π. In case ρ uses multiple instances of ϕ, $\rho^{\phi \to \pi}$ uses multiple instances of π.

Theorem 2.18 (Universal composition theorem). *Let* ρ, ϕ, π *be protocols such that* ρ *uses* ϕ *as subroutine and such that* π *emulates* ϕ*. Then* $\rho^{\phi \to \pi}$ *emulates* ρ*.*

2.5.1.2 Adaptive Corruption in the UC Framework

The UC model defines several types of party corruptions, the most important being *static*, *adaptive*, and *transient* corruptions. In protocols secure against static party corruptions, parties are either honest or corrupt from the start of the protocol and do not change their corruption status. In protocols secure against adaptive corruptions, parties can become corrupted at any time; once corrupted, they remain so for the rest of the protocol. Finally, transient corruptions [Can00] are similar to adaptive corruptions, but parties can *recover* from corruption and regain their security. At all times, the environment is aware of the corruption status of all parties.[2]

Corruption of a party. When a party becomes corrupted, all of its internal state excluding the parts that were explicitly erased (🗑) is handed over to the adversary $\mathcal{A}$. $\mathcal{A}$ then controls that party. The ideal functionalities that were used as subroutines are notified of the corruption, and may provide additional information or capabilities to $\mathcal{A}$. Note that $\mathcal{A}$ can always choose to let a corrupted party follow the honest protocol, but passively monitor the party's internal state.

2.5.1.3 Joint State

The composition theorem of UC assumes that the local state of and randomness used by each protocol instance is independent from that of each other protocol instance. It is

[2] This can be achieved with the help of special messages that are sent to the environment upon each party corruption or recovery from corruption.

therefore not possible for the components to have any amount of joint state. However, in practice, it is desirable for protocols to have some amount of joint state, for example by sharing instances of encryption or signature schemes.

The UC model with joint state (JUC) [CR03] fixes the above issue by providing a new composition theorem. In a nutshell, let ρ be a high-level protocol that uses multiple instances of a sub-protocol π; let $\hat{\pi}$ be a protocol such that one instance of $\hat{\pi}$ emulates multiples instances of π; then the JUC theorem guarantees that the protocol ρ where all instances of π are replaced by a single instance of $\hat{\pi}$, emulates ρ.

Thereby, one can analyze the protocol ρ with the assumption that all instances of the sub-protocol π are independent, and be guaranteed that the protocol remains secure even if the sub-protocols have some joint state.

2.5.1.4 Problems with the UC Framework

Over the years, several problems were discovered in the UC framework and its multiple updates [HS11, KT13, CEK+16b]. The chief problems are that formally the composition theorem does not hold and that the definition of runtime is flawed. Furthermore, many aspects of the UC model are only described informally: see [CEK+16b] for a list. The existence of these problems was one of the reasons why alternative frameworks, such as GNUC, IITM, and Constructive Cryptography, were developed.

Cryptographers continue to use the UC framework to design protocols, even if, formally, security proofs in that framework cannot be sound due to the above-mentioned problems. One possible explanation for this is that the problems of the framework are limited to edge cases: it is hoped that distinguishing attacks due to the flaws of the framework do not translate to any attacks in reality. Furthermore, an overwhelming majority of these security proofs are not fully formal themselves, and it is expected that any omissions or flaws in the framework could be fixed, or that one can easily translate the protocol to another framework. Thereby these security proofs are not formal proofs, but rather compelling arguments.

2.5.2 The GNUC framework

The GNUC framework [HS11, HS15] fixes a number of flaws of the UC framework while trying to retain as much compatibility with UC as possible, thereby making it easier for protocol designers familiar with UC to start using GNUC. We therefore do not describe the GNUC framework in full, but only in comparison to UC.

To fix UC's flaws, GNUC imposes a strict hierarchical structure on protocols, requires hierarchical corruption, and changes the definition of runtime. In GNUC, unlike UC, it is the environment that corrupts top-level machines (and corrupted machines can corrupt their subroutines), thereby enforcing top-down corruption. However a number of protocols cannot be realized in GNUC. For example, the adversary cannot send messages to any machine from which it did not yet receive any messages: this requires somewhat unnatural definitions for, e.g., channel functionalities, where the recipient must first send a message to the functionality to indicate that he is ready before he can receive messages. The discussion by Küsters and Tuengerthal [KT13] shows additional examples.

2.5.3 The IITM Model with Responsive Environments

The IITM model was introduced in [Kue06] and revised in [KT13] with a more general runtime notion. The IITM model addresses UC's flaws in a different way than GNUC, thereby achieving formal correctness without GNUC's limitations. The main difference between UC/GNUC and IITM is that machines now perform computation in two modes: a deterministic CheckAddress mode where the machine determines whether to accept a message, and a Compute mode where the actual computation happens. The IITM model also has simpler runtime requirements for protocols compared to UC/GNUC. Thereby a machine cannot exhaust its runtime if it receives too many garbage messages from the adversary. However, this comes at a cost: the composition of two protocol systems satisfying the runtime requirements does not necessarily result in a protocol that does (in practice however, it is often easy to prove that a given composed protocol satisfies the requirements).

A short overview of the parts of the IITM model, and the parts of the IITM model with responsive environments [CEK+16a], that are needed for this thesis follow.

2.5.3.1 The IITM Model

Inexhaustible interactive Turing machines. An inexhaustible interactive Turing machine (IITM or simply ITM) is a probabilistic Turing machine with a number of named input and output tapes which determine how different ITMs are connected in a system of ITMs. There might exist several instances of an ITM, called ITIs, in a run of a system of ITMs. As detailed below, an instance of an ITM M runs in one of two modes: **CheckAddress** and **Compute**. The former is used to address the different instances of an ITM in a run, while in the latter the actual computation is performed. In **CheckAddress** mode the runtime of the ITM is bounded by a (fixed) polynomial in the length of the security parameter, the current input message, and the configuration of the machine. The runtime limit in **Compute** will be discussed in the sequel.

Systems of ITMs. A *system* $\mathcal{Q}$ of ITMs is a set $\mathcal{Q} = \{M_1, \ldots, M_k\}$ of ITMs $M_1, \ldots, M_k$, where the way ITMs in this system are connected is defined by the names of the tapes of these machines. More specifically, for every tape named t, it is required that at most two of these ITMs have a tape named t and, if two ITMs, say M_i and M_j have a tape named t, then one must be an input tape, say of M_i, and the other one an output tape of M_j. In other words, the two machines are connected via t, and hence an instance of M_j can send messages to an instance of M_i. In this thesis, we use the convention that M_i and M_j are then also connected in the other direction, i.e., M_i has an output tape and M_j an input tape named t'. One often refers to t' by t^{-1}, and call it the corresponding tape to t (with opposite direction). Also, one often refers to the bidirectional pair (t, t') simply by t. Tapes in $\mathcal{Q}$ that connect two machines are called *internal tapes* of $\mathcal{Q}$ and all others are called *external tapes* of $\mathcal{Q}$. The latter are further grouped into *I/O tapes* to communicate with other protocols, and *network tapes* to communicate with the adversary. When talking about the I/O and network *interfaces* of the system one refers to the set of external I/O and network tapes, respectively.

There are two special tapes, named start and decision, respectively. A system may have at most one machine with an input tape named start; start must not be an output

tape. This machine is called the *master ITM*. The tape decision may occur as an (external) output tape in a system only, not as an input tape.

A system $\mathcal{Q}_2$ is said to be *connectable* to a system $\mathcal{Q}_1$ if $\mathcal{Q}_2$ connects to the external tapes of $\mathcal{Q}_1$ only, i.e., tapes with the same name in $\mathcal{Q}_2$ and $\mathcal{Q}_1$ are external tapes of $\mathcal{Q}_2$ and $\mathcal{Q}_1$, respectively, and they have opposite directions, i.e., an input tape in one system is an output tape in the other. By $\{\mathcal{Q}_1, \mathcal{Q}_2\}$ one denotes the composition of the connectable systems $\mathcal{Q}_1$ and $\mathcal{Q}_2$, defined in the obvious way. Note that $\{\mathcal{Q}_1, \mathcal{Q}_2\}$ again is a system of ITMs as defined above. For example, if $\mathcal{Q}_1 = \{M_1, M_2\}$ and $\mathcal{Q}_2 = \{M_3, M_4, M_5\}$, then $\{\mathcal{Q}_1, \mathcal{Q}_2\} = \{M_1, \ldots, M_5\}$.

Running a system. In a run of a system $\mathcal{Q}$, an unbounded number of instances of each ITM in $\mathcal{Q}$ may be spawned. An instance of a machine, say M_i in $\mathcal{Q}$, can send a message to an instance of another machine M_j in $\mathcal{Q}$ if and only if M_i and M_j are connected via tapes. Which instance of M_j gets to process the message sent by the instance of M_i is determined by running the instances of M_j in mode **CheckAddress**.

More specifically, in a run of a system $\mathcal{Q}(1^\eta)$ with security parameter η, only one ITI is active at any time and all other ITIs wait for new input. The first machine to be activated is the master ITM in $\mathcal{Q}$, by writing the empty message on start;[3] if no master ITM exists, the run of $\mathcal{Q}$ terminates immediately. If a message m is written by some ITI on one of its output tapes, say on t (initially, as mentioned, the empty message is written on start), and there is a machine, say M, in $\mathcal{Q}$, with an input tape named t, then it is decided as follows which instance of M gets to process m.

The instances of M are run in **CheckAddress** mode in the order of their creation, until one instance accepts m. This instance (if any) then runs in **Compute** mode with input m written on its input tape t. If no instance accepted m, a fresh instance of M is spawned and run in mode **CheckAddress** and if it accepts m, it gets to process m on its input tape t in **Compute** mode. Otherwise, the freshly created instance is deleted again, m is dropped, and the empty message is written on start in order to trigger the master ITM (of which there might be several instances as well, where again their **CheckAddress** is used to decide which one gets to process the message). After running an ITI in mode **CheckAddress**, the configuration is set back to the state before it was run in **CheckAddress**; thus, this mode does not and cannot change the configuration of a machine.

When an instance of M processes a message in mode **Compute**, it may write at most one message, say m', on one of its output tapes, say t', and then stop. If there is an ITM with an input tape named t' in the system, the message m' is delivered to one instance of that ITM on tape t' as described above. If the instance of M stops without output or there is no ITM with an input tape t', then (an instance of) the master ITM is activated. A run stops as soon as a message is written on decision, no master instance accepted the incoming message, or in mode **Compute** a master ITI did not produce output. The *overall output* of a run is defined to be the one-bit message that is output on decision, or zero if decision was not written to. The probability that

[3] If a system is run with external input, then this input is written on start. Note that the IITM model supports both uniform and non-uniform environments/machines. For ease of presentation, only the former setting is considered.

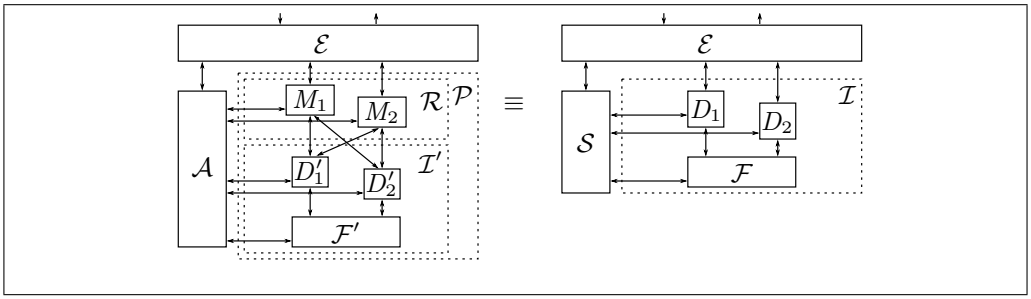


Fig. 2.1: The setup for the universal composability experiment ($\mathcal{P} \leq \mathcal{I}$) and internal structure of protocols. Here $\mathcal{E}$ is an environmental system, $\mathcal{A}$ and $\mathcal{S}$ are adversarial systems, and $\mathcal{P}$ and $\mathcal{I}$ are protocol systems. Horizontal arrows correspond to pairs of network tapes, and vertical/oblique arrows to pairs of I/O tapes.

the overall output of a run of $\mathcal{Q}(1^\eta)$ is $b \in \{0,1\}$ is denoted by $\Pr[\mathcal{Q}(1^\eta) = b]$, where the probability is taken over the random choices of all the ITIs in runs of $\mathcal{Q}$.

Equivalent systems. Two systems that produce overall output 1 with almost the same probability are called equivalent: two systems $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are *equivalent* ($\mathcal{Q}_1 \equiv \mathcal{Q}_2$) if and only if $|\Pr[\mathcal{Q}_1(1^\eta) = 1] - \Pr[\mathcal{Q}_1(1^\eta) = 1]|$ is negligible in η.

Types of systems. To define simulation, one distinguishes between *protocol systems*, *adversarial systems*, and *environmental systems*. These are arbitrary systems (in the sense defined above), but where only environmental systems may have start and decision tapes; in particular, only the environment may contain the master ITM. Adversarial systems $(\mathcal{A}, \mathcal{S})$, environmental systems $(\mathcal{E})$, and protocol systems $(\mathcal{P}, \mathcal{I})$ are connected as illustrated in Figure 2.1. In the IITM model neither any specific internal structure of $\mathcal{P}$ or $\mathcal{I}$ nor any specific addressing mechanism or corruption behavior is fixed; $\mathcal{P}$ and $\mathcal{I}$ are arbitrary systems which can be freely specified by the protocol designer.

Runtime requirements for environmental and protocol systems. Compared to other frameworks, the IITM model uses very general and simple runtime notions. An environmental system $\mathcal{E}$ has to be *universally bounded*, i.e., there exists a polynomial p such that for every system $\mathcal{Q}$ connectable to $\mathcal{E}$ the overall runtime of $\mathcal{E}$ in mode **Compute** is bounded by $p(\eta)$ in every run of $\mathcal{E} \mid \mathcal{Q}$ with security parameter η. Given a system $\mathcal{Q}$, one denotes by $\mathsf{Env}(\mathcal{Q})$ the set of all environmental systems that can be connected to $\mathcal{Q}$. A protocol system $\mathcal{P}$ has to be *environmentally bounded*, i.e., for every environmental system $\mathcal{E}$ there exists a polynomial p such that for every η the overall runtime of $\mathcal{P}$ in mode **Compute** is bounded by $p(\eta)$ in every run of $\mathcal{E} \mid \mathcal{P}$ with security parameter η, except for a negligible set of runs.

Simulation and Universal Composability. Informally, $\mathcal{P}$ *realizes* or *emulates* $\mathcal{I}$, denoted by $\mathcal{P} \leq \mathcal{I}$, if and only if for all adversarial systems $\mathcal{A}$ connectable to $\mathcal{P}$ there exists an adversarial system $\mathcal{S}$ (the *simulator*) connectable to $\mathcal{I}$ such that for all $\mathcal{E}$ one has that $\{\mathcal{E}, \mathcal{A}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{I}\}$. Intuitively, $\mathcal{I}$ is an *ideal protocol* which formally specifies a cryptographic task in an ideal way, while $\mathcal{P}$ is a *real protocol* which tries to realize this task in a real setting such that for all attacks on $\mathcal{P}$ there is one on $\mathcal{I}$ such that

both attacks are indistinguishable for any environment, and hence, $\mathcal{P}$ is as secure as $\mathcal{I}$, where the latter is secure by definition. The above security notion corresponds to the classical notion of universally composable security. In the IITM model also the notions dummy UC, strong simulatability, black-box simulatability, and reactive simulatability have been formulated and shown to be equivalent [KT13], which is an important sanity check for a UC-like model.

The IITM model enjoys general composition theorems, which, for instance, allow one to replace an ideal protocol $\mathcal{I}$ by its realization $\mathcal{P}$, and hence, allow for modular protocol design.

2.5.3.2 IITM with Responsive Environments

When modeling protocols in the IITM model, it is often necessary to initialize new instances of ITMs with, e.g., their corruption status, keys, or algorithms. For this purpose, the adversary is asked to provide this information. This information is provided with meta messages that are not present in reality, but are used for modeling purposes only. A protocol designer would typically expect the adversary to provide the necessary information right away. However, in *all* current UC-like frameworks the adversary is not bound to provide an answer right away, but he could activate other machines and influence their state before answering the request. This often leads to unintended behavior and artificial situations, which the protocol designer has to handle in protocol specifications and which make the protocol specifications unnecessarily complex and cumbersome, and possibly even unreasonable.

To circumvent that problem, Camenisch et al. [CEK+16a] have introduced the concept of *responsive environments*: the adversary and the environment must immediately answer certain meta messages and thereby provide certain important meta information right away.

Responsive environments. To define responsive environments (and later in the security notions also responsive adversaries), one first needs to define restrictions. Restrictions capture the messages that the environment/adversaries need to answer immediately. For our conventions (see Section 6.2), we fix a specific restriction, which, however, allows for a flexible use of restricting messages.

Definition 2.19 (Restriction [CEK+16a]). *Let $R \subseteq \{0,1\}^+ \times \{0,1\}^+$ be a set of tuples of non-empty messages. Recall that $R[0] := \{m | \exists m' : (m, m') \in R\}$. The set R is called a* restriction *if and only if the following holds true:*
There exists an algorithm A which for all inputs of the form (m, m') runs in at most polynomial time in the length of m' and outputs 1 iff $m \in R[0]$ and $(m, m') \notin R$. In all other cases, A outputs 0.
A message $m \in R[0]$ is called a restricting message.

Responsive environments can now be defined. Note that given any system $\mathcal{Q}$, an environmental system may connect to all external (network and I/O) tapes of $\mathcal{Q}$.

Definition 2.20 (Responsive environments [CEK+16a]). *An environmental system $\mathcal{E}$ is called* responsive *for a system $\mathcal{Q}$ with respect to a restriction R, if in an*

overwhelming set of runs of $\{\mathcal{E}, \mathcal{Q}\}$ *every restricting message from* $\mathcal{Q}$ *(on a network tape) is immediately answered, i.e., for any (restricting) message* $m \in R[0]$ *sent by* $\mathcal{Q}$ *on a network tape, the first message* m' *that* $\mathcal{E}$ *sends back to* $\mathcal{Q}$ *is written on the same bidirectional network tape and satisfies* $(m, m') \in R$*. By* $\mathsf{Env}_R(\mathcal{Q}) \subseteq \mathsf{Env}(\mathcal{Q})$ *one denotes the set of responsive environmental systems for* $\mathcal{Q}$*.*

Security Notions for Responsive Environments. The notion of strong simulatability in the context of responsive environments now follows. Roughly speaking, $\mathcal{P}$ realizes $\mathcal{I}$ w.r.t. strong simulatability if and only if there exists a simulator $\mathcal{S}$ such that for all environments $\mathcal{E}$ one has that $\{\mathcal{E}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{I}\}$. (Hence, the real adversary is dropped completely and $\mathcal{E}$ may connect directly to the network tapes of $\mathcal{P}$.)

From now on, we consider responsive environments only. Consequently, we require protocol systems $\mathcal{Q}$ to be environmentally bounded only for all environments in $\mathsf{Env}_R(\mathcal{Q})$. One call such systems R-environmentally bounded.

Definition 2.21 (Responsive simulators [CEK+16a]). *Let* $\mathcal{P}$ *and* $\mathcal{I}$ *be (R-environmentally bounded) protocol systems. Let* $\mathcal{S}$ *be an adversarial system such that* $\mathcal{S}$ *can be connected to* $\mathcal{I}$*, the set of external tapes of* $\mathcal{S}$ *is disjoint from the set of I/O-tapes of* $\mathcal{I}$*,* $\{\mathcal{S}, \mathcal{I}\}$ *and* $\mathcal{P}$ *have the same external interface, and* $\{\mathcal{S}, \mathcal{I}\}$ *is R-environmentally bounded. Then,* $\mathcal{S}$ *is called a* responsive simulator *if in an overwhelming set of runs of* $\{\mathcal{E}, \mathcal{S}, \mathcal{I}\}$ *every restricting message from* $\mathcal{I}$ *(on a network tape) is immediately answered (in the sense of Definition 2.20). One denotes the set of all such simulators for protocol systems* $\mathcal{P}$ *and* $\mathcal{I}$ *by* $\mathrm{Sim}_R^{\mathcal{P}}(\mathcal{I})$*.*

This definition ensures that restricting messages from $\mathcal{I}$ are answered without activating another machine of $\mathcal{I}$ (and with an expected response), even if $\mathcal{I}$ is connected to a simulator (on its network interface). This allows us to consider $\{\mathcal{E}, \mathcal{S}\}$ to be a responsive environment for $\mathcal{I}$. This is crucial to prove the composition theorems and to prove that strong simulatability is a transitive relation.

Definition 2.22 (Strong simulatability with responsive environments [CEK+16a]). *Let* $\mathcal{P}$ *and* $\mathcal{I}$ *be protocol systems, the* real *and* ideal protocol*, respectively. Then,* $\mathcal{P}$ realizes $\mathcal{I}$ with respect to responsive environments *(*$\mathcal{P} \leq_R^{SS} \mathcal{I}$ *or simply* $\mathcal{P} \leq \mathcal{I}$*) if and only if there exists* $\mathcal{S} \in \mathrm{Sim}_R^{\mathcal{P}}(\mathcal{I})$ *such that* $\{\mathcal{E}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{I}\}$ *for every* $\mathcal{E} \in \mathsf{Env}_R(\mathcal{P})$*.*

Camenisch et al. have shown that $\leq$ is a reflexive and transitivity relation. They also analogously define UC security (UC), dummy UC ($dumUC$), black-box simulatability (BB), and reactive simulatability (RS) for responsive environments, and prove that these notions are equivalent to the above notion.

Composition Theorems for Responsive Environments. The composition theorems proved in the IITM model (with general environments) can be generalized to responsive environments as well. The following theorem handles the concurrent composition of any (fixed) number of potentially different protocols.

Theorem 2.23 (Composition theorem [CEK+16a]). *Let* R *be a restriction. Let* $k \geq 1$*,* $\mathcal{Q}$ *be a system of IITMs without* start *and* decision *tape, and* $\mathcal{P}_1, \ldots, \mathcal{P}_k, \mathcal{I}_1, \ldots, \mathcal{I}_k$ *be protocol systems such that all systems have pairwise disjoint sets of network tapes and the following conditions are satisfied:*

1. For all $j \leq k$: $\mathcal{P}_j \leq_R \mathcal{I}_j$
2. $\mathcal{Q}, \mathcal{P}_1, \ldots, \mathcal{P}_k$ are connectable and $\{\mathcal{Q}, \mathcal{P}_1, \ldots, \mathcal{P}_k\}$ is R-environmentally bounded.

Then, $\{\mathcal{Q}, \mathcal{P}_1, \ldots, \mathcal{P}_k\} \leq_R \{\mathcal{Q}, \mathcal{I}_1, \ldots, \mathcal{I}_k\}$.

The usual composition theorem is the special case $k = 1$ and $R = \emptyset$. Camenisch et al. also provide a second composition theorem which guarantees the secure composition of an unbounded number of copies of the same protocol system.

Note that in all the above theorems, only minimal assumptions about the ideal and real protocols were made. It is not necessary to fix any internal structure of these protocols, a specific addressing mechanism, or corruption conventions.

2.5.4 The Constructive Cryptography (CC) Model with Random Systems

The model of constructive cryptography (CC) was introduced by Maurer [Mau10,Mau11b] and was based on an earlier work by Maurer and Renner [MR11]. Unlike the UC, GNUC, and IITM frameworks, whose computational model is based on interactive Turing machines and whose security definitions are thus necessarily complex, the Constructive Cryptography model uses a top-down approach where the definition of lower levels of abstractions (such as the machine model) is not required for proving theorems at higher levels. Results stated in CC are thus simpler than equivalent results in the other frameworks.

In this thesis, we consider CC instantiated with random systems [Mau02]. We provide a short overview of the parts that are needed for understanding the rest of this thesis.

2.5.4.1 Random Systems

Many cryptographic primitives and protocols can be described as random systems generating for each input x_k an output y_k. In full generality such a system can be described by a sequence of conditional probability distributions, however in this work we will chiefly describe such systems using algorithms.

Connecting random systems. Two random systems can be connected at the interface level, the result being another random system whose behaviour is defined via an interaction of the two sub-systems: a message input on a non-shared interface is processed by the corresponding sub-system, and a message that is output on an interface of the "shared" interface by one of the sub-systems, is then immediately processed as an input on a corresponding interface of the other sub-system.

2.5.4.2 Constructive Cryptography (CC)

CC considers three important types of random systems: *resources*, *converters*, and *distinguishers*, which we describe next.

Resources. Resources have a finite set of interfaces $\mathcal{I}$, where resources that can be accessed by multiple parties have one interface for each party. Two resources can be composed in parallel, the result being another resource whose interfaces are the disjoint union of the interfaces of the two systems; interfaces of such a system can also be merged into a single interface (it is assumed that there is an implicit "message routing" happening behind the scenes). An example of a resources is an authenticated communication channel.

In this chapter we chiefly consider *AEW-Resources* which have $\mathcal{I} = \{Alice, Eve, World\}$. Here the *Alice* interface corresponds to a party. The *Eve* interface corresponds to adversarial access to the resource. The *World* interface, introduced by Gaži et al. [GMT], models the influence of the environment on the resource, and is used to monitor the actions of the adversary or limit his capabilities; we will further comment on this interface in Section 2.5.4.2.

Converters. Converters model the local strategies employed by a party (or the adversary) on a resource. Converters have one *inner interface* and one *outer interface*. The inner interface of a converter may be connected to an interface $i \in \mathcal{I}$ of a resource, the result being another resource for which the outer interface of the converter serves as the new interface i. If the converter outputs a message on the inner interface, then that message is immediately input to the interface i of the resource, and vice-versa.

Converters attached to a party interface are also called a *protocol*.

Distinguishers. A distinguisher (also called an *environment*) has $n+1$ interfaces. One interface, which we call *Init* serves as trigger to start the computation, and outputs a single "result" bit (thus ending the computation). The other n interfaces can be connected to a resource (here an output on one of the interfaces is immediately used as an input to another interface). We write $\Pr[\mathsf{DR} = 1]$ to denote the probability that a distinguisher D connected to resource R, when activated (with an empty message) on *Init*, eventually outputs the bit 1 on *Init*. The goal of a distinguisher is to distinguish between two resources by outputting different bits when connected to one or the other.

The *advantage* of a distinguisher D for two resources R and S is defined as: $\epsilon = |\Pr[\mathsf{DR} = 1] - \Pr[\mathsf{DS} = 1]|$. We say that the two resources R and S are statistically equivalent within ϵ, denoted $\mathsf{R} \overset{\epsilon}{\equiv} \mathsf{S}$, if the advantage of every distinguisher is no more than ϵ. We also write $\mathsf{R} \equiv \mathsf{S}$ if the two resource are perfectly equivalent ($\epsilon = 0$).

Construction of Resources. An important goal in cryptography is to *construct* a resource S with some desired properties from a resource R that is assumed to be available by using a protocol π. In that respect, constructive cryptography follows the real-world/ideal-world approach of UC and its variants [Can00, Can01, PW00, HS11, KT13, CDPW07].

The "real world" consists of the protocol execution of one honest party and the adversary (and influence from the environment) and is modelled as the composition of the resource R with the protocol π.

In the "ideal world", the ideal resource S, specifying the desired functional and security goals, is composed with a *simulator* σ—a converter for the *Eve* interface. The purpose of σ is to adapt the *Eve*-interface of S so that it matches the one of $\pi\mathsf{R}$.

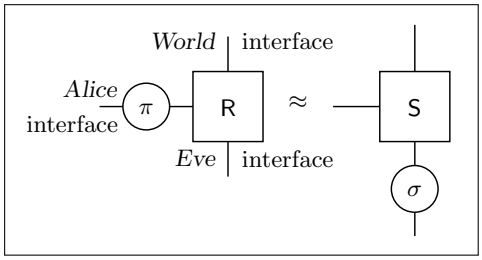


Fig. 2.2: The constructive statement in the context of an AEW-resource. Protocol/converter π constructs S from R if there exists a simulator σ such that: the resource R with the protocol π attached *Alice*-interface is indistinguishable from the resource S with σ attached to its *Eve*-interface.

Definition 2.24. *The protocol* π (securely) constructs S *from* R *within* ϵ*, denoted* $\mathsf{R} \xrightarrow{\pi}_{\epsilon} \mathsf{S}$*, if:* $\exists \sigma : \pi\mathsf{R} \overset{\epsilon}{\equiv} \mathsf{S}\sigma$*. Perfect constructions* ($\epsilon = 0$) *are denoted* $\mathsf{R} \xrightarrow{\pi} \mathsf{S}$*.*

Figure 2.2 shows this graphically in the context of AEW-Resources. In this chapter, we do not explicitly show the *availability* condition [MR11].

Composability. An important property of Definition 2.24 is its composability. That is if: $\mathsf{R} \xrightarrow{\pi_1}_{\epsilon_1} \mathsf{T}$ and $\mathsf{T} \xrightarrow{\pi_2}_{\epsilon_2} \mathsf{S}$, then $\mathsf{R} \xrightarrow{\pi_2 \circ \pi_1}_{(\epsilon_2+\epsilon_1)} \mathsf{S}$.

Computational security. In the context of computational security, one only considers protocols, resources, distinguishers, and simulators that can be modelled as probabilistic polynomial time (PPT) algorithms. In the sequel we assume that all of these random systems receive the security parameter 1^η as implicit input.

The overall runtime of a computation must also be polynomial in the security parameter. For the random systems considered in this chapter, this is always the case.

The definition of equivalent systems and secure constructions needs to be adapted to the computational setting: two (PPT) resources R and S are computationally indistinguishable, denoted $\mathsf{R} \approx \mathsf{S}$, if for all (PPT) distinguishers D, there is a negligible function negl such that for all security parameters η larger than a constant η_0:

$$\left| \Pr\left[\mathsf{DR} = 1\right] - \Pr\left[\mathsf{DS} = 1\right] \right| \leq \mathrm{negl}(\eta).$$

Definition 2.25. *In a computational setting, a (PPT) protocol* π (securely) constructs S *from* R*, denoted* $\mathsf{R} \xrightarrow{\pi}_{\mathsf{c}} \mathsf{S}$*, if there exists a (PPT) simulator* σ *such that* $\pi\mathsf{R} \approx \mathsf{S}\sigma$*.*

Adaptive Attacks and the *World* Interface. One of the most desirable security properties of cryptographic schemes is resistance against *adaptive* attacks. To that effect one wishes to model resources that have certain security guarantees while no attack has happened, and different (weaker) guarantees after an attack.

Resources that have only party interfaces and an *Eve*-interface cannot be used in such a setting, as in the distinguishing experiment there is no way of ensuring that both resources R and S are in the same attacked-or-not state, i.e., that we are comparing like-with-like: a simulator σ can often launch an attack on S without this being visible

to the distinguisher, and hence the distinguisher is tasked with distinguishing $\pi\mathsf{R}$ in a non-attacked state, with $\mathsf{S}\sigma$ in an attacked state—a meaningless experiment. It is thus necessary to enable the distinguisher to either check whether a resource has been attacked or not, or to actually enable such an attack without going through the simulator. To that effect Gaži et al. [GMT] have introduced the concept of a *World*-interface.[4]

Through the *World*-interface, the distinguisher can directly access the resource. This access is chiefly used to alter the state of the resource and make it "weak" in some form. This weakness does not have to be a binary state: one can envisage multiple levels of weakness, or, e.g., a resource with three "components" (each of which can be made separately weak by sending different messages on the *World*-interface) where at least two must be made weak before the resource itself becomes weak. Whether the resource ultimately allows a specific attack by the adversary on the *Eve*-interface may depend on not just weakness status, but also the rest of its internal state. The concept of *World*-interface is thus more general than the concept dynamic corruption of UC and its variants.

2.6 Some Basic Ideal Functionalities

This section describes the ideal functionalities we use as subroutines in our constructions. These are authenticated channels ($\mathcal{F}_{\text{ac}}$), one-sided–authenticated channels ($\mathcal{F}_{\text{osac}}$), zero-knowledge proofs of existence ($\mathcal{F}_{\text{gzk}}$). In the common reference string (CRS) or random oracle models, protocols may also use special ideal functionalities for CRS ($\mathcal{F}_{\text{crs}}^{D}$) or random oracles. We also briefly discuss those. Each composition framework requires a different instantiation of these ideal functionalities. Here we restrict ourselves to the instantiations in the UC and GNUC models.

2.6.1 Authenticated Channels

Let $\mathcal{F}_{\text{ac}}$ be a single-use authenticated channel [HS11]. Such a channel provides the guarantee to the recipient that the received message originated from the sender (and was not modified in transit). It does not guarantee confidentiality of the message. Appendix A.2 provides a formal definition of $\mathcal{F}_{\text{ac}}$ in the UC model; we refer to the GNUC paper [HS11] for the formal definition in the GNUC model.

2.6.2 One-sided–authenticated Channels

Let $\mathcal{F}_{\text{osac}}$ be a multi-use channel where only one party, the server, authenticates himself towards the other party, the client. The server has the guarantee that in a given session all messages come from the same client. Note that the first message from the client to the server is not authenticated and can be modified (*hijacked*) by the adversary—the original client will be excluded from the rest of the interaction. Appendix A.3 provides a formal definition of $\mathcal{F}_{\text{osac}}$ in the UC model. We also refer to the work of Barak et al. [BCL+05] for a formal treatment of communication without or with partial authentication.

[4] Note that the UC [Can00, Can01], GNUC [HS11], and IITM with conventions [CEK+16a] models all allow the environment to directly check (UC, IITM) or enable (GNUC) attacks by (ab)using the party interfaces.

2.6.3 Zero-knowledge Proofs of Knowledge and Existence

A zero-knowledge proof [GMR89] is a two-party protocol where, for a fixed binary relation R and a common input x, the prover can convince the verifier that he knows a witness w such that $R(x, w) = 1$, and the verifier learns nothing from this protocol (except for the fact that such a w exists).

Camenisch et al. [CKS11] have extended the standard functionality for zero-knowledge $\mathcal{F}_{\text{zk}}$ of Canetti et al. [Can00, CLOS02] to also support proofs of existence ($\mathcal{F}_{\text{gzk}}$). In a realization, these proofs of existence are cheaper than the corresponding proofs of knowledge, but they impose limitations on the simulator $\mathcal{S}$ in the security proof. Following Hofheinz and Shoup [HS11], in a realization of $\mathcal{F}_{\text{gzk}}$, the prover reveals the statement to be proven only in the *last* message. This is crucial for our construction, as this allows the prover to *erase* ($\not\Diamond$) witnesses and other data before disclosing the statement to be proven. Appendix A.4 provides a formal definition of $\mathcal{F}_{\text{gzk}}$ in both the UC and GNUC models.

Notation. When specifying the predicate to be proven, we use a combination of the Camenisch-Stadler notation [CS97] and the notation introduced by Camenisch, Krenn, and Shoup [CKS11]; for example: $\mathcal{F}_{\text{gzk}}\{(\rangle\!\alpha, \beta\ ;\ \exists\gamma) : y = g^{\gamma} \wedge z = g^{\alpha}k^{\beta}h^{\gamma}\}$ is used for proving the existence of the discrete logarithm to the base g, and of a representation of z to the bases g, k, and h such that the h-part of this representation is equal to the discrete logarithm of y to the base g. Furthermore, knowledge of the g-part and the k-part of the representation is proven. Variables quantified by $\rangle\!$ (knowledge) can be extracted by the simulator $\mathcal{S}$ in the security proof, while variables quantified by $\exists$ (existence) cannot.

By writing a proof on an arrow: $\overset{\pi_0}{\dashrightarrow}$ we denote the performance of such an interactive zero-knowledge proof protocol secure against adaptive corruptions with erasures. If additional public or secret data is written on the arrow, or data to be erased besides the arrow, then this data is transmitted with, or erased before, respectively, the last message of the proof protocol. The predicate of the proof may depend on that data.

Proofs with two verifiers. Let $\mathcal{F}_{\text{gzk}}^{\text{2v}}$ be the three-party ideal functionality to denote the parallel execution of two independent zero-knowledge proofs with the same prover and same specification, but two different verifiers. The prover waits for a reply from both verifiers before sending out the last message of each proof. This gives the prover the opportunity to erase the same witnesses in both proofs. Appendix A.5 provides a formal definition of $\mathcal{F}_{\text{gzk}}^{\text{2v}}$ in the UC model. The proof that the special composition theorem by Camenisch, Krenn, and Shoup [CKS11] holds also for $\mathcal{F}_{\text{gzk}}^{\text{2v}}$ is very similar to the proof that it holds for $\mathcal{F}_{\text{gzk}}$ and is omitted.

2.6.4 Ideal Functionalities For the CRS and Random Oracle Models

Canetti and Fischlin [CF01] showed that certain ideal functionalities, such as a commitment functionality, cannot be realized in the plain model. Commitments and other functionalities can however be realized in the *common reference string* model or the *random oracle* model. Special ideal functionalities for CRS or random oracles can be used by protocols in the respective model. In this thesis we will chiefly work in the CRS model.

2.6.4.1 Common Reference String

Let $\mathcal{F}_{\text{crs}}^{D}$ be a common reference string functionality, which provides to all parties a common string distributed according to some distribution D. We make use of several distributions in this thesis: $\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}$ provides a uniform CRS over $\mathbb{G}^3$ for some group $\mathbb{G}$ and $\mathcal{F}_{\text{crs}}^{\text{gzk}}$ provides a CRS as required by Camenisch et al.'s protocol π, the intended realization of $\mathcal{F}_{\text{gzk}}$ [CKS11]. Appendix A.1 provides the formal definition of $\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}$ in the UC model; we refer to the GNUC paper [HS11] for the formal definition in the GNUC model.

2.6.4.2 Random Oracles

A random oracle provides a common random function to all protocol participants [BR93]. Although random oracles have been used to prove the security of many schemes, Canetti et al. [CGH98] have shown that the random oracle model is unsound: there exist protocols that are secure in the random oracle model, but where any implementation of the random oracle results in an insecure protocol.

3

Practical Two-Party Computation of Arithmetic Circuits

In this chapter we are interested in practically useful UC-secure building block protocols that provide interfaces so that parties in higher-level protocols can prove to each other that their inputs to one building block protocol correspond to the outputs of another building block protocol. More precisely, we provide a set of two-party protocols for evaluating an arithmetic circuit over $\mathbb{Z}_n$, where n is a safe semi-prime, with reactive inputs and outputs. The protocols accept as (additional) inputs and provide as (additional) outputs tailored commitment values which, in conjunction with UC zero-knowledge proofs, make them a useful building block for higher-level protocols.

We demonstrate the usefulness of our protocols by providing as example application an oblivious pseudorandom function (OPRF) evaluation (see Section 3.7) and point out that our protocols can be used to implement the subprotocols required by Camenisch et al.'s credential authenticated identification and key-exchange protocols [CCGS10] (see Section 6.3 of their paper).

Apart from being the only protocols that allow for their use as building blocks, ours are also more efficient than existing UC-secure two-party reactive circuit evaluation protocols [DN03, IPS09, DO10a, BDOZ11] which were designed to be used as standalone protocols. The complexity of our protocols can be summarized as follows: if the circuits involved have t gates, the communication complexity is $O(t)$ elements of $\mathbb{Z}_{n^2}$ (and groups of similar or smaller order) and the computational complexity is $O(t)$ exponentiations in $\mathbb{Z}_{n^2}$ (and groups of similar or smaller order). We report on an experimental comparison of our protocols with relevant prior work in Section 3.6.1. We show that our protocols are practical, and that small circuits can be run in a few seconds—for examplethe OPRF computation above (for a 1248-bit modulus) would run in 0.84 seconds on the authors' laptop computers.

Roadmap. In Section 3.1 we present our new mixed trapdoor commitment scheme. We describe our ideal functionality $\mathcal{F}_{\text{abb}}$ for circuit evaluation in Section 3.2, and construct a concrete protocol in Section 3.3. In Section 3.4 we add additional features to our functionality. We prove our protocol secure in Section 3.5. In Section 3.6 we disucuss related work, and compare the efficiency of our protocols with relevant related work. In Section 3.7 we show how one can easily construct an OPRF using our protocol.

3.1 A New Dual-Mode Homomorphic "Mixed" Trapdoor (HMT) Commitment Scheme

We now construct a commitment scheme which we will use instead of traditional UC commitment schemes [CF01] in our circuit evaluation protocol. Our commitment scheme works well with proofs of *existence* using $\mathcal{F}_{\text{gzk}}$, resulting in an efficiency gain in the overall protocol.[1] To the best of our knowledge, this is a novel scheme.

We define a mixed trapdoor commitment scheme to be a commitment scheme that is either: perfectly hiding and equivocable; or statistically binding, depending on the distribution of the CRS. Mixed trapdoor commitments are similar to UC commitments [CF01] in that *1)* the simulator can equivocate commitments in the security proof without being caught, even if he has to provide all randomness used to generate the commitment to the adversary; and *2)* the simulator can use an adversary who equivocates commitments[2] to solve a hard cryptographic problem. However unlike UC commitments, in mixed trapdoor commitments *3)* the simulator *does not need to extract the openings or the committed values* from $\mathcal{F}_{\text{gzk}}$.

Definition 3.1 (Homomorphic Mixed-trapdoor (HMT) commitment). *A secure homomorphic mixed-trapdoor (HMT) commitment scheme for a given group generator* ggen *is a tuple of PPT algorithms* $(\mathsf{cgen}_0, \mathsf{cgen}'_0, \mathsf{cgen}_1, \mathsf{com}, \mathsf{cvfy}, \mathsf{ctrd}, \mathsf{cadd}, \mathsf{cmul}, \mathsf{cpc})$ *such that:*

- $(\mathsf{cgen}_0, \mathsf{com}, \mathsf{cvfy}, \mathsf{cadd}, \mathsf{cmul})$ *is a homomorphic* perfectly-hiding *secure commitment scheme for* ggen *that is compatible with zero-knowledge proofs,*
- $(\mathsf{cgen}_1, \mathsf{com}, \mathsf{cvfy}, \mathsf{cadd}, \mathsf{cmul})$ *is a homomorphic* statistically-binding *commitment scheme for* ggen *that is compatible with zero-knowledge proofs,*
- *the ensembles* $\{pc\}_{1^\eta}$ *where* $pc \xleftarrow{\$} \mathsf{cgen}_0(\mathsf{ggen}(1^\eta))$ *or* $pc \xleftarrow{\$} \mathsf{cgen}_1(\mathsf{ggen}(1^\eta))$ *or* $(pc, tc) \xleftarrow{\$} \mathsf{cgen}'_0(\mathsf{ggen}(1^\eta))$ *are pairwise computationally indistinguishable,*
- *given a trapdoor* tc *obtained through* $(pc, tc) \xleftarrow{\$} \mathsf{cgen}'_0(\mathsf{ggen}(1^\eta))$*, it is possible to* equivocate *commitments: given that* $1 = \mathsf{cvfy}(pc, ca, oa, a)$ *we have that for all* b*:* $1 = \mathsf{cvfy}(pc, ca, \mathsf{ctrd}(pc, tc, ca, oa, a, b), b)$*,*
- *and given a commitment* ca *it is possible to extract a Pedersen commitment [Ped92] to* a *with opening* oa *with* $\mathsf{cpc}(pc, ca)$*.*

3.1.1 A Scheme for Messages in $\mathbb{Z}_n$

We now provide the construction of a mixed trapdoor commitment scheme based on Elgamal encryption. Such a scheme is secure if the DDH assumption holds for the following group generator ggen, which generates a group of safe-semiprime order n.

[1] The efficiency gain due to using proofs of existence instead of proofs of knowledge outweighs the efficiency loss due to the more complex commitment scheme.

[2] As the commitment scheme is malleable, the protocol designer must take into account that the adversary might base his commitments on the simulator's commitments. Such problems can usually be avoided by requiring that for all *new* commitments, a proof of *knowledge* of the committed value is performed.

Group generator $\mathsf{ggen}(1^\eta)$**.** Let $n \xleftarrow{\$} \mathsf{rgen}(1^\eta)$ (in the sequel, rgen is the same algorithm as used for the CRH cryptosystem). Find the first prime p such that $p = k \cdot n + 1$ for some $k \in \mathbb{N}$. Find a generator g of a subgroup of $\mathbb{Z}_p$ of order n. Output (p, n, g). According to a heuristic[3] by Wagstaff [WJ79]: $p < n \cdot (\log n)^2$.

Algorithm $\mathsf{cgen}_0(p, n, g)$**.** $w, tc \xleftarrow{\$} \mathbb{Z}_n$;$m \leftarrow 0$; $h \leftarrow g^w$; $y \leftarrow g^m h^{tc}$; $u \leftarrow g^{tc}$; output (p, g, n, h, y, u). Here (y, u) is the Elgamal encryption of g^m with the public key (g, h).

Algorithm $\mathsf{cgen}'_0(n)$**.** Idem, except that tc is output as well.

Algorithm $\mathsf{cgen}_1(n)$**.** Idem to cgen_0 except that $m \xleftarrow{\$} \mathbb{Z}_n$. In practice, one can also choose h, y, and u at random from the subgroup generated by g.

Algorithm $\mathsf{com}(pc, a)$. $oa \xleftarrow{\$} \mathbb{Z}_n$; $ca_1 \leftarrow y^a h^{oa}$; $ca_2 \leftarrow u^a g^{oa}$; output $((ca_1, ca_2), oa)$. Notice that the commitment is a re-randomized ElGamal encryption of $g^{m \cdot a}$.

Algorithm $\mathsf{cvfy}((ca_1, ca_2), oa, a)$. Check that $ca_1 = y^a h^{oa}$ and $ca_2 = u^a g^{oa}$, and that ca_1 and ca_2 are in the subgroup generated by g.

Algorithm $\mathsf{ctrd}(pc, tc, (ca_1, ca_2), oa, a, b)$. Output $(a - b) \cdot tc + oa$.

Algorithm $\mathsf{cadd}(pc, (ca_{0,0}, ca_{0,1}), (ca_{1,0}, ca_{1,1}), \ldots)$**.** Output $(\prod_i ca_{i,0}, \prod_i ca_{i,1})$.

Algorithm $\mathsf{cmul}(pc, (ca_0, ca_1), b)$**.** Output (ca_0^b, ca_1^b).

Algorithm $\mathsf{cpc}(pc, (ca_0, ca_1))$**.** Output ca_0.

3.1.2 A Scheme over a Prime Order Group

We now provide a similar construction for group generators ggen that output a group description $\mathbb{G}$ for a group of prime order q with generator g where the DDH problem is hard.

Algorithm $\mathsf{cgen}_0(\mathbb{G}, q, g)$**.** This is similar to the construction for messages in $\mathbb{Z}_n$ above, except that $w, tc \xleftarrow{\$} \mathbb{Z}_q$ and that the output is $(\mathbb{G}, g, q, h, y, u)$.

Algorithm $\mathsf{cgen}_1(n)$**.** Idem to cgen_0 except that $m \xleftarrow{\$} \mathbb{Z}_q$.

Algorithm $\mathsf{com}(pc, a)$. This is similar to the construction for messages in $\mathbb{Z}_n$ above, except that $oa \xleftarrow{\$} \mathbb{Z}_q$.

Other algorithms. All other algorithms are the same as the construction for messages in $\mathbb{Z}_n$ above.

3.2 Our Ideal Functionality $\mathcal{F}_{\text{abb}}$

In this section, we will start by giving a short informal definition of the ideal functionality $\mathcal{F}_{\text{abb}}$ (arithmetic black box) for doing computation over $\mathbb{Z}_n$. We then provide the formal definition of $\mathcal{F}_{\text{abb}}$ using the notation of GNUC [HS11]. It is not necessary to read Subsection 3.2.2 to understand the construction of our scheme.

[3] We confirmed this experimentally for 250 randomly generated 1248-bit safe RSA moduli.

Arithmetic circuits. We denote our basic ideal functionality for verifiably evaluating arithmetic circuits modulo n by $\mathcal{F}_{\text{abb}}$ (our functionality is similar to Nielsen's arithmetic black box [Nie03], hence the name). Parties compute the circuit step-by-step in a reactive manner by sending identical instructions with identical common input to $\mathcal{F}_{\text{abb}}$. (For some instructions, one party must additionally provide private input to $\mathcal{F}_{\text{abb}}$.) We assume that a higher-level protocol orchestrates the steps the parties take.

$\mathcal{F}_{\text{abb}}$ processes instructions from the two parties of the following types: *Input:* a party inserts a value in $\mathbb{Z}_n$ into the circuit; *Linear Combination:* a linear combination of values in the circuit is computed; *Multiplication:* the product of two values in the circuit is computed; *Output:* a value in the circuit is output to a party; *Proof:* a party can prove an arbitrary statement to the other party in zero-knowledge involving values that she input in the circuit, values she got as an output, and values external to the circuit.

A party can use the *Proof* instruction to prove that the value inside a commitment used in the higher-level protocol is the same as a value in the circuit. This instruction thus makes it easy and practical to compose $\mathcal{F}_{\text{abb}}$ with a higher-level protocol. To input a committed value from a higher-level protocol into the circuit, $\mathcal{P}$ would first use the *Input* instruction to set the value in the circuit, and then use the *Proof* instruction to convince $\mathcal{Q}$ that the new value corresponds to what was in the commitment. Similarly to transfer a value from the circuit to the higher-level protocol, $\mathcal{P}$ would first get the value with the *Output* instruction, generate a commitment in the higher-level protocol, and then use the *Proof* instruction to convince $\mathcal{Q}$ that the commitment contains the value that was output by the circuit.

3.2.1 Informal Definition of $\mathcal{F}_{\text{abb}}$

The functionality $\mathcal{F}_{\text{abb}}$ reacts to a set of instructions. Per convention, both parties must agree on the instruction and the shared input before $\mathcal{F}_{\text{abb}}$ executes it. An instruction may require $\mathcal{P}$ and $\mathcal{Q}$ to send multiple messages to $\mathcal{F}_{\text{abb}}$ in a specific order, however $\mathcal{F}_{\text{abb}}$ may run other instructions concurrently while waiting for the next message. More precisely $\mathcal{P}$ and $\mathcal{Q}$ can: provide inputs to $\mathcal{F}_{\text{abb}}$; ask it do to a linear combination or multiplication of previous inputs or intermediate results; ask it to output a value to one of them; and do an arbitrary zero-knowledge proof involving inputs/outputs to/from the circuit and external witnesses. These instructions can be arbitrarily interleaved, intermediate results output and new inputs be provided. The input values provided by $\mathcal{P}$ and $\mathcal{Q}$ may depend on output values obtained. Following the GNUC formalism, each message sent to $\mathcal{F}_{\text{abb}}$ is prefixed with a label which contains, among others, the name of the instruction to execute, the current step in the instruction this message refers to, and the shared input φ; the private inputs are always part of the message body.

State. The ideal functionality $\mathcal{F}_{\text{abb}}$ is stateful. It maintains an associative array V, mapping identifiers (in $\Lambda^\star$) to integer values (in $\mathbb{Z}_n$).

Instructions. These are the instructions supported by $\mathcal{F}_{\text{abb}}$:

- *Input from $\mathcal{P}$:* $\mathcal{P}$'s private input is the value v. $\mathcal{F}_{\text{abb}}$ parses the shared input φ as $\langle k \rangle$, where k will be the identifier associated to the value v, and sets $V[k] \leftarrow v$.
- *Input from $\mathcal{Q}$:* $\mathcal{Q}$'s private input is v. $\mathcal{F}_{\text{abb}}$ parses φ as $\langle k \rangle$, and sets $V[k] \leftarrow v$.

- *Linear combination:* $\mathcal{F}_{\text{abb}}$ parses φ as $\langle m, k_0, v_0, \langle k_1, v_1\rangle, \ldots, \langle k_{m-1}, v_{m-1}\rangle\rangle$ and sets: $V[k_0] \leftarrow v_0 + \sum_{i=1}^{m-1} V[k_i] \cdot v_i$.
- *Multiplication:* $\mathcal{F}_{\text{abb}}$ parses φ as $\langle k_0, k_1, k_2\rangle$ and sets: $V[k_0] \leftarrow V[k_1] \cdot V[k_2]$.
- *Output to $\mathcal{P}$:* $\mathcal{F}_{\text{abb}}$ parses φ as $\langle k\rangle$, and sends $V[k]$ (as a *delayed output*) to $\mathcal{P}$.
- *Output to $\mathcal{Q}$:* $\mathcal{F}_{\text{abb}}$ parses φ as $\langle k\rangle$, and sends $V[k]$ (as a *delayed output*) to $\mathcal{Q}$.
- *Proof by $\mathcal{P}$:* This instruction can be used to prove a statement about values that were input/output to/from from the circuit ($\mathcal{F}_{\text{abb}}$) and witnesses from a higher-level protocol. $\mathcal{P}$'s private input is $\langle x, w_k, \langle\rangle\rangle$. $\mathcal{F}_{\text{abb}}$ parses φ as $\langle m, \langle k_0, \ldots, k_{m-1}\rangle, R\rangle$, where is R is a binary predicate that is compatible with $\mathcal{F}_{\text{gzk}}$ and which can involve *1)* values that were input by $\mathcal{P}$ to $\mathcal{F}_{\text{abb}}$, *2)* values that were output to $\mathcal{P}$ from $\mathcal{F}_{\text{abb}}$, and *3)* witnesses external to $\mathcal{F}_{\text{abb}}$; x is an instance for R; w_k is a list of witnesses that are external to the circuit whose knowledge are proven; and $k_0, \ldots, k_{m-1}$ are identifiers of values in the circuit that were input by $\mathcal{P}$ or output to $\mathcal{P}$. $\mathcal{F}_{\text{abb}}$ checks if the predicate holds, i.e., if $R\big(x,\ w_k \cup (V[k_0], \ldots, V[k_{m-1}])\big) = 1$; and sends $\langle x\rangle$ (as a *delayed output*) to $\mathcal{Q}$.
- *Proof by $\mathcal{Q}$:* Similar to *Proof by $\mathcal{P}$*, with the roles of $\mathcal{P}$ and $\mathcal{Q}$ reversed.
- *Dynamic corruption:* $\mathcal{F}_{\text{abb}}$ accepts a special corrupt message from $\mathcal{P}$ or $\mathcal{Q}$. From then on, all input and output of the corrupted party is redirected to the adversary $\mathcal{A}$, and $\mathcal{A}$ may recover all of the corrupted party's input (by asking $\mathcal{F}_{\text{abb}}$ for it).

Treatment of invalid input. In case $\mathcal{F}_{\text{abb}}$ receives a message it does not expect, a message that it cannot parse, or a message with a label it has seen previously from the same party, it simply ignores the message (we assume then environment is then activated with an empty message instead).

Comments. The value of n is not an input to $\mathcal{F}_{\text{abb}}$, nor is it modeled as a CRS. Rather, it is modeled in the GNUC framework as a *system parameter*. Roughly speaking, this is a special type of ideal functionality to which all parties, including the environment, have common access. The value of n is generated by a trusted party, and no other party learns its factorization. Furthermore, the modulus n can be re-used across different protocol instances. In the setting of credential-authenticated identification [CCGS10] this is completely natural, as one can use a modulus generated by the credential issuer. In a different context, we can also imagine using the modulus n of a well-known and respected certificate authority (e.g., the modulus in Verisign's root certificate).

Our ideal functionality $\mathcal{F}_{\text{abb}}$ shares some similarity with Nielsen's arithmetic black box (ABB) [Nie03], and Damgård and Orlandi's $\mathcal{F}_{\text{ampc}}$ [DO10a]. The major difference is that our $\mathcal{F}_{\text{abb}}$ includes the *Proof* instruction, allowing values from higher-level protocols to be input and output securely. This instruction is crucial as it allows meaningful composition with other protocols. Unlike $\mathcal{F}_{\text{ampc}}$, we do not support random number generation in the vanilla $\mathcal{F}_{\text{abb}}$ for simplicity; see Section 3.4.1 for an algorithm generating these that uses only our core set of instructions.

3.2.2 Formal Definition of $\mathcal{F}_{\text{abb}}$

We now formally define the $\mathcal{F}_{\text{abb}}$ functionality using the formalism of GNUC [HS11].

By $\langle \texttt{label}, \text{value1}, \text{value2}, \ldots \rangle$ we denote an ideal message with label `label` and payload value1, value2, ... By convention, if a party sends a message with the same label as a message it has sent previously, $\mathcal{F}_{\text{abb}}$ ignores the message.

System parameters. The safe RSA modulus n, which defines the ring $\mathbb{Z}_n$ in which all the arithmetic operations will be performed, and whose factorization is unknown to $\mathcal{P}$, $\mathcal{Q}$, and the adversary $\mathcal{A}$, is assumed to be part of the system parameters.

State. The ideal functionality $\mathcal{F}_{\text{abb}}$ is stateful, and maintains an associative array V, as well as the sets KPP, KQP, KPQ, KQQ, AP, AQ, RP, and RQ. The associative array V maps an identifier (in $\Lambda^\star$) to the corresponding value (in $\mathbb{Z}_n$) in the circuit. The set KPP contains the list of identifiers corresponding to values that were either input by $\mathcal{P}$ or output to $\mathcal{P}$, in $\mathcal{P}$'s view; these identifiers can be used in the *Proof by* $\mathcal{P}$ instruction. The sets KQP, KPQ, KQQ are similar, but for $\mathcal{Q}$'s values in $\mathcal{P}$'s view, $\mathcal{P}$'s values in $\mathcal{Q}$'s view, $\mathcal{Q}$'s values in $\mathcal{Q}$'s view, respectively. The set AP contains the list of identifiers which, in $\mathcal{P}$'s view, have already been used in the circuit; this set prevents parties from using the same identifier multiple times. The set AQ is similar, but for $\mathcal{Q}$'s view. The set RP contains the list of identifiers which, in $\mathcal{P}$'s view, are ready to be used in other instructions; this set prevents parties from using an identifier where the corresponding value has not been properly initialized yet. The set RQ is similar, but for $\mathcal{Q}$'s view.

Instructions. In what follows, we let $\varphi \in \Lambda^\star$ denote the command ID, a string which will be part of the label. The command ID φ will contain all the common input to an instruction.

The ideal functionality $\mathcal{F}_{\text{abb}}$ is composed of several instructions. By convention each step may be triggered only once (re-use of an instruction requires a different command ID φ). A logical expression in $[\ldots]$: is a guard that must be satisfied in order to trigger the step. Our ideal instructions are modelled closely after GNUC's zero-knowledge and secure function evaluation functionalities [HS11].

We take the convention that variables with an overbar, such as $\bar{v}$, are global variables associated with the command ID φ, whose scope is the instruction they are defined in. All other variables are local.

We also assume that the communication between the parties and $\mathcal{F}_{\text{abb}}$ cannot be delayed by $\mathcal{A}$, this makes sure that in the case one party re-uses an output label k in several instructions, it is clear which operation is to be ignored by $\mathcal{F}_{\text{abb}}$. The ideal functionality $\mathcal{F}_{\text{abb}}$ models message delays internally.

- **Input from $\mathcal{P}$:** In this instruction, parse the command ID φ as $\langle k \rangle$ where $k = \langle \varrho \rangle$ with $\varrho \in \Lambda^\star$.
  - `ip:send`:φ : Accept $\langle \texttt{ip:send}{:}\varphi, v \rangle$ from $\mathcal{P}$ where $v \in \mathbb{Z}_n \wedge k \notin AP$: $AP \leftarrow k$; $\bar{v} \leftarrow v$; send $\langle \texttt{ip:send}{:}\varphi \rangle$ to $\mathcal{A}$.
  - `ip:ready`:φ : Accept $\langle \texttt{ip:ready}{:}\varphi \rangle$ from $\mathcal{Q}$, where $k \notin AQ$: $AQ \leftarrow k$; send $\langle \texttt{ip:ready}{:}\varphi \rangle$ to $\mathcal{A}$.
  - `ip:lock`:φ [`ip:send`:$\varphi \wedge$ `ip:ready`:φ] : Accept $\langle \texttt{ip:lock}{:}\varphi \rangle$ from $\mathcal{A}$: $V[k] \leftarrow \bar{v}$; send $\langle\rangle$ to $\mathcal{A}$.
  - `ip:done`:φ [`ip:lock`:φ] : Accept $\langle \texttt{ip:done}{:}\varphi \rangle$ from $\mathcal{A}$: $RP \leftarrow k$; $KPP \leftarrow k$; send $\langle \texttt{ip:done}{:}\varphi \rangle$ to $\mathcal{P}$.

  - `ip:deliver`:φ [`ip:lock`:φ] : Accept $\langle$`ip:deliver`:$\varphi\rangle$ from $\mathcal{A}$: $RQ \leftarrow k$; $KPQ \leftarrow k$; send $\langle$`ip:deliver`:$\varphi\rangle$ to $\mathcal{Q}$.
  - `ip:reset`:φ [$\neg$`ip:lock`:$\varphi \wedge$ `corrupt:P`] : Accept $\langle$`ip:reset`:$\varphi, v\rangle$ from $\mathcal{A}$: $\bar{v} \leftarrow v$; send $\langle\rangle$ to $\mathcal{A}$.
  - `ip:expose`:φ [`ip:send`:$\varphi \wedge$ `corrupt:P`] : Accept $\langle$`ip:expose`:$\varphi\rangle$ from $\mathcal{A}$: send $\langle$`ip:expose`:$\varphi, \bar{v}\rangle$ to $\mathcal{A}$.
- **Input from $\mathcal{Q}$:** This is similar to the previous instruction, with the roles of $\mathcal{P}$ and $\mathcal{Q}$ reversed, and the label prefix is changed to `iq`. We do not formalize this instruction here.
- **Output to $\mathcal{P}$:** In this instruction, parse φ as $\langle k\rangle$ where $k \in \Lambda^\star$.
  - `op:p`:φ : Accept $\langle$`op:p`:$\varphi\rangle$ from $\mathcal{P}$, where $k \in RP$: send $\langle$`op:p`:$\varphi\rangle$ to $\mathcal{A}$.
  - `op:q`:φ : Accept $\langle$`op:q`:$\varphi\rangle$ from $\mathcal{Q}$, where $k \in RQ$: send $\langle$`op:q`:$\varphi\rangle$ to $\mathcal{A}$.
  - `op:lock`:φ [`op:p`:$\varphi \wedge$ `op:q`:φ] : Accept $\langle$`op:lock`:$\varphi\rangle$ from $\mathcal{A}$: send $\langle\rangle$ to $\mathcal{A}$.
  - `op:deliver`:φ [`op:lock`:φ] : Accept $\langle$`op:deliver`:$\varphi\rangle$ from $\mathcal{A}$: $KPP \leftarrow k$; send $\langle$`op:deliver`:$\varphi, V[k]\rangle$ to $\mathcal{P}$.
  - `op:done`:φ [`op:lock`:φ] : Accept $\langle$`op:done`:$\varphi\rangle$ from $\mathcal{A}$: $KPQ \leftarrow k$; send $\langle$`op:done`:$\varphi\rangle$ to $\mathcal{Q}$.
- **Output to $\mathcal{Q}$:** This is similar to the previous instruction, with the roles of $\mathcal{P}$ and $\mathcal{Q}$ reversed, and the label prefix is changed to `oq`. We do not formalize this instruction here.
- **Linear combination:** Parse φ as $\langle m, k_0, v_0, \langle k_1, v_1\rangle, \ldots, \langle k_{m-1}, v_{m-1}\rangle\rangle$ in this instruction, where $m \in \mathbb{N}^*$, $\forall i \in \mathbb{N}_m : k_i \in \Lambda^\star, v_i \in \mathbb{Z}_n$, and where $k_0 = \langle$`joint`$, \varrho\rangle$ with $\varrho \in \Lambda^\star$.
  - `l:p`:φ : Accept $\langle$`l:p`:$\varphi\rangle$ from $\mathcal{P}$ where $k_0 \notin AP \wedge (\forall i \in \mathbb{N}_m^* : k_i \in RP)$: $AP \leftarrow k_0$; send $\langle$`l:p`:$\varphi\rangle$ to $\mathcal{A}$.
  - `l:q`:φ : Accept $\langle$`l:q`:$\varphi\rangle$ from $\mathcal{Q}$ where $k_0 \notin AQ \wedge (\forall i \in \mathbb{N}_m^* : k_i \in RQ)$: $AQ \leftarrow k_0$; send $\langle$`l:q`:$\varphi\rangle$ to $\mathcal{A}$.
  - `l:lock`:φ [`l:p`:$\varphi \wedge$ `l:q`:φ] : Accept $\langle$`l:lock`:$\varphi\rangle$ from $\mathcal{A}$: $V[k_0] \leftarrow v_0 + \sum_{i=1}^{m-1} V[k_i] \cdot v_i$; send $\langle\rangle$ to $\mathcal{A}$.
  - `l:done:p`:φ [`l:lock`:φ] : Accept $\langle$`l:done:p`:$\varphi\rangle$ from $\mathcal{A}$: $RP \leftarrow k_0$; send $\langle$`l:done:p`:$\varphi\rangle$ to $\mathcal{P}$.
  - `l:done:q`:φ [`l:lock`:φ] : Accept $\langle$`l:done:q`:$\varphi\rangle$ from $\mathcal{A}$: $RQ \leftarrow k_0$; send $\langle$`l:done:q`:$\varphi\rangle$ to $\mathcal{Q}$.
- **Multiplication:** In this instruction, parse φ as $\langle k_0, k_1, k_2\rangle$ where $k_1, k_2 \in \Lambda^\star$ and $k_0 = \langle$`joint`$, \varrho\rangle$ with $\varrho \in \Lambda^\star$.
  - `m:p`:φ : Accept $\langle$`m:p`:$\varphi\rangle$ from $\mathcal{P}$ where $k_0 \notin AP \wedge (\forall i \in \mathbb{N}_3^* : k_i \in RP)$: $AP \leftarrow k_0$; send $\langle$`m:p`:$\varphi\rangle$ to $\mathcal{A}$.
  - `m:q`:φ : Accept $\langle$`m:q`:$\varphi\rangle$ from $\mathcal{Q}$ where $k_0 \notin AQ \wedge (\forall i \in \mathbb{N}_3^* : k_i \in RQ)$: $AQ \leftarrow k_0$; send $\langle$`m:q`:$\varphi\rangle$ to $\mathcal{A}$.
  - `m:lock`:φ [`m:p`:$\varphi \wedge$ `m:q`:φ] : Accept $\langle$`m:lock`:$\varphi\rangle$ from $\mathcal{A}$: $V[k_0] \leftarrow V[k_1] \cdot V[k_2]$; send $\langle\rangle$ to $\mathcal{A}$.
  - `m:done:p`:φ [`m:lock`:φ] : Accept $\langle$`m:done:p`:$\varphi\rangle$ from $\mathcal{A}$: $RP \leftarrow k_0$; send $\langle$`m:done:p`:$\varphi\rangle$ to $\mathcal{P}$.
  - `m:done:q`:φ [`m:lock`:φ] : Accept $\langle$`m:done:q`:$\varphi\rangle$ from $\mathcal{A}$: $RQ \leftarrow k_0$; send $\langle$`m:done:q`:$\varphi\rangle$ to $\mathcal{Q}$.

- **Proof by $\mathcal{P}$:** In this instruction, parse φ as $\langle m, \langle k_0, \ldots, k_{m-1}\rangle, R\rangle$, where $m \in \mathbb{N}$, $\forall i \in \mathbb{N}_m : k_i \in \Lambda^\star$, and where is R is a binary predicate that is compatible with $\mathcal{F}_{\text{gzk}}$.
  - `pp:send`:φ : Accept $\langle$`pp:send`:$\varphi, x, \tilde{m}, w_k, 0, \langle\rangle\rangle$ from $\mathcal{P}$ where x is an instance for R, $\tilde{m} \in \mathbb{N}$, $w_k = \langle w_{k,0}, \ldots, w_{k,\tilde{m}-1}\rangle$ is a list of external witnesses whose knowledge is proven, $R\big(x,\ (w_{k,0}, \ldots, w_{k,\tilde{m}-1}) \cup (V[k_0], \ldots, V[k_{m-1}])\big) = 1$, and $(\forall i \in \mathbb{N}_m : k_i \in KPP)$: $\bar{x} \leftarrow x$; $\bar{w}_k \leftarrow w_k$; send $\langle$`pp:send`:$\varphi, \ell(x, w_k)\rangle$ to $\mathcal{A}$.
  - `pp:ready`:φ : Accept $\langle$`pp:ready`:$\varphi\rangle$ from $\mathcal{Q}$ where $(\forall i \in \mathbb{N}_m : k_i \in KPQ)$; send $\langle$`pp:ready`:$\varphi\rangle$ to $\mathcal{A}$.
  - `pp:lock`:φ [`pp:send`:$\varphi \wedge$ `pp:ready`:φ] : Accept $\langle$`pp:lock`:$\varphi\rangle$ from $\mathcal{A}$: send $\langle\rangle$ to $\mathcal{A}$.
  - `pp:done`:φ [`pp:lock`:φ] : Accept $\langle$`pp:done`:$\varphi\rangle$ from $\mathcal{A}$: send $\langle$`pp:done`:$\varphi\rangle$ to $\mathcal{P}$.
  - `pp:deliver`:φ [`pp:lock`:φ] : Accept $\langle$`pp:deliver`:$\varphi, L\rangle$ from $\mathcal{A}$ where $L = \ell(\bar{x}, \bar{w}_k) \vee$ [`corrupt:Q`]: send $\langle$`pp:deliver`:$\varphi, \bar{x}\rangle$ to $\mathcal{Q}$.
  - `pp:reset`:φ [$\neg$`pp:lock`:$\varphi \wedge$ `corrupt:P`] : Accept $\langle$`pp:reset`:$\varphi, x, \tilde{m}, w_k, 0, \langle\rangle\rangle$ from $\mathcal{A}$ where x is an instance for R, $\tilde{m} \in \mathbb{N}$, $w_k = \langle w_{k,0}, \ldots, w_{k,\tilde{m}-1}\rangle$ is a list of witnesses, and $R\big(x,\ (w_{k,0}, \ldots, w_{k,\tilde{m}-1}) \cup (V[k_0], \ldots, V[k_{m-1}])\big) = 1$: $\bar{x} \leftarrow x$; $\bar{w}_k \leftarrow w_k$; send $\langle\rangle$ to $\mathcal{A}$.
  - `pp:expose`:φ [`pp:send`:$\varphi \wedge \neg$`pp:lock`:$\varphi \wedge$ `corrupt:P`] : Accept $\langle$`pp:expose`:$\varphi\rangle$ from $\mathcal{A}$: send $\langle$`pp:expose`:$\varphi, \bar{x}, \bar{w}_k\rangle$ to $\mathcal{A}$.
- **Proof by $\mathcal{Q}$:** This is similar to the previous instruction, with the roles of $\mathcal{P}$ and $\mathcal{Q}$ reversed, and the label prefix is changed to `pq`. We do not formalize this instruction here.
- **Dynamic corruption:**
  - `corrupt:P` : Accept a special $\langle$`corrupt`$\rangle$ message from $\mathcal{P}$: send $\langle$`corrupt:P`$\rangle$ to $\mathcal{A}$ together with an invitation for the messages $\langle$`ip:expose`:$\varphi\rangle$ (for all φ where the `ip:send`:φ step has been processed already) and $\langle$`pp:expose`:$\varphi\rangle$ (for all φ where the `pp:send`:φ step has been processed already).
  - `corrupt:Q` : Analogously, but for $\mathcal{Q}$.

Invalid input. In case $\mathcal{F}_{\text{abb}}$ receives a message it does not expect, a message that it cannot parse, or a message with a label it has seen previously from the same party, it simply ignores the message (thereafter the environment is activated with empty input).

3.3 Construction

We now show how to construct a protocol Π_{abb} for circuit evaluation modulo n. Our protocol uses two ideal functionalities: $\mathcal{F}_{\text{ac}}$ (authenticated channels) and $\mathcal{F}_{\text{gzk}}$ (zero-knowledge proofs). Additionally, we make use of a system parameter, the modulus n of unknown factorization; and a CRS, consisting of the output of $\texttt{cgen}_1$ (for the statistically-binding mode of the mixed trapdoor commitment).

High-level idea. The high-level idea of our construction is that $\mathcal{P}$ and $\mathcal{Q}$ generate additive shares of all the values (inputs and intermediate results) in the circuit. Identifiers are used to keep track of the values and the cryptographic objects associated with them.

Like for $\mathcal{F}_{\text{abb}}$, parties agree on the instruction to be performed by sending a message containing an identical instruction name and identical common input to the protocol Π_{abb}. The instructions of Π_{abb} are implemented as follows: *Input* is achieved by one party setting her share to the input, and generating a commitment to that share; the other party sets his share to zero. *Output* is achieved by one party sending her share to the other party. For the *Linear combination* instruction, each party does a linear combination of their shares locally. For the *Multiplication* instruction, we make use of two instances of a 2-party subroutine Π_{mul}: on $\mathcal{P}$'s input a, and $\mathcal{Q}$'s input b, Π_{mul} outputs u to $\mathcal{P}$ and v to $\mathcal{Q}$ such that $u + v = a \cdot b$. The *Proof* instruction can be done with the help of a zero-knowledge proof functionality $\mathcal{F}_{\text{gzk}}$. To ensure security against malicious adversaries, both parties update the commitments to the shares in each instruction, and prove in zero-knowledge that all their computations were done honestly.

The Π_{mul} subroutine makes use of a CRH encryption scheme ($\mathsf{rgen}, \mathsf{rkgen}, \mathsf{enc}, \mathsf{dec}, \mathsf{add}, \mathsf{mul}$)—here rgen outputs the ring $\mathbb{Z}_n$ used throughout the scheme—along with the statistically-biding commitment scheme ($\mathsf{cgen}, \mathsf{com}, \mathsf{cvfy}, \mathsf{cadd}, \mathsf{cmul}$) of the HMT commitment scheme for messages in $\mathbb{Z}_n$ defined in Section 3.1.1. To achieve security against adaptive corruptions, new encryption/decryption keys need to be generated for every multiplication. To do this in a practical way, we use the semantically secure version of Camenisch-Shoup encryption [CS03, DJ03, JS07] with a short private key and short randomness, as described in Section 2.4.1.2. One key feature of this scheme is that key generation is fast: just a single exponentiation modulo n^2. Another key feature is that many encryption/decryption keys can be used in conjunction with the same n, which is crucial. Our commitment scheme is also used extensively in the overall protocol. We use the construction presented in Section 3.1 and work in the group of integers modulo a prime of the form $k \cdot n + 1$. The homomorphic properties of the commitment scheme makes this choice of prime particularly useful and practical. Another tool we make heavy use of is UC zero-knowledge. Because of the proposed implementations of encryption and commitment schemes, these proof systems can all be implemented using the approach proposed by Camenisch et al. [CKS11]. Because the encryption and commitment schemes are both homomorphic modulo n, all of our cryptographic tools work very well together, and yield quite practical protocols. We also stress that our protocols are designed in a modular way: they only make use of these abstract primitives, and not of *ad hoc* algebraic constructions.

3.3.1 Realizing Π_{abb}

$\mathcal{P}$ and $\mathcal{Q}$ each maintain the following global state: several associative arrays mapping the identifier of a value in the circuit (in $\Lambda^\star$) to a variety of cryptographic objects: SP and SQ map to the shares of $\mathcal{P}$ and $\mathcal{Q}$ of the values in the circuit (in $\mathbb{Z}_n$), respectively; CP and CQ map to the commitment of the corresponding shares; XP (maintained by $\mathcal{P}$ only) and XQ ($\mathcal{Q}$ only) map to the opening of the commitments. For the *Proof* functionality, both parties maintain lists of identifiers corresponding to values that are known to $\mathcal{P}$ and $\mathcal{Q}$: KP and KQ, respectively. Additionally, to ensure "thread-safety", they also maintain: lists of assigned identifiers AP ($\mathcal{P}$ only) and AQ ($\mathcal{Q}$ only) to avoid assigning the same identifier to several variables; and lists of identifiers RP ($\mathcal{P}$ only) and

$\mathcal{P}$.**InputP**$(\langle\varphi, v\rangle)$ where $v \in \mathbb{Z}_n$:	$\mathcal{Q}$.**InputP**$(\langle\varphi\rangle)$:
Here $\varphi = \langle k\rangle$ and $k \in \Lambda^\star$.	
Abort if $k \in AP$.	Abort if $k \in AQ$.
$AP \leftarrow k$; $SP[k] \leftarrow v$; $SQ[k] \leftarrow 0$;	$AQ \leftarrow k$; $SQ[k] \leftarrow 0$;
$(CP[k], XP[k]) \xleftarrow{\$} \mathsf{com}(v)$.	$CQ[k] \leftarrow \mathsf{com}(0,0)$; $XQ[k] \leftarrow 0$.
	$CP[k], \pi_{\mathtt{ip}}$ ⟶
$CQ[k] \leftarrow \mathsf{com}(0,0)$.	
$RP \leftarrow k$; $KP \leftarrow k$.	$RQ \leftarrow k$; $KP \leftarrow k$.

Instantiation of zero-knowledge proofs:

$$\pi_{\mathtt{ip}} := \mathcal{F}_{\mathrm{gzk}}[\langle \mathtt{ip}{:}\varphi\rangle]\big\{\big(\nu v\ \exists XP[k]\big) : \mathsf{cvfy}(CP[k], XP[k], v)\big\}.$$

Fig. 3.1: Input from $\mathcal{P}$.

$\mathcal{P}$.**OutputQ**$(\langle\varphi\rangle)$ where $\varphi = \langle k\rangle \wedge k \in \Lambda^\star$:	$\mathcal{Q}$.**OutputQ**$(\langle\varphi\rangle)$ where $\varphi = \langle k\rangle \wedge k \in \Lambda^\star$:
Abort unless $k \in RP$.	Abort unless $k \in RQ$.
	$SP[k], \pi_{\mathtt{oq}}$ ⟶
$KQ \leftarrow k$.	$KQ \leftarrow k$; output $(SP[k] + SQ[k])$.

Instantiation of zero-knowledge proofs:

$$\pi_{\mathtt{oq}} := \mathcal{F}_{\mathrm{gzk}}[\langle \mathtt{oq}{:}\varphi\rangle]\big\{\big(\exists XP[k]\big) : \mathsf{cvfy}(CP[k], XP[k], SP[k])\big\}.$$

Fig. 3.2: Output to $\mathcal{Q}$.

RQ ($\mathcal{Q}$ only) corresponding to values that are ready to be used in other instructions. The array that one would obtain by summing the entries of SP and SQ corresponding to values that are ready (i.e., $\{(k, v) | k \in RP\ \cap RQ \wedge v = SP[k] + SQ[k]\}$), corresponds to the array V of the ideal functionality, that maps identifiers to values in the circuit.

All other variables that we will introduce are local to one instance of a instruction or an instance of the Π_{mul} subroutine. Several instructions may be active at the same time, however we assume (following the GNUC model) that all operations performed during an activation (the time interval between starting to process a new input message and sending a message to another functionality) happen atomically.

Input from $\mathcal{P}$. In this instruction, $\mathcal{P}$ inputs a value v into the circuit and associates it with the identifier k: $\mathcal{P}$ sets her own share to v, and $\mathcal{Q}$ sets his share to 0. Then $\mathcal{P}$ generates a commitment to her share, which she sends (along with proof) to $\mathcal{Q}$. See Figure 3.1 for the construction.

Input from $\mathcal{Q}$. Similar to the previous instruction, with the roles of $\mathcal{P}$ and $\mathcal{Q}$ reversed.

Output to $\mathcal{Q}$. In this instruction, $\mathcal{Q}$ retrieves the value identified by k from the circuit: $\mathcal{P}$ sends her share to $\mathcal{Q}$ together with a proof of correctness. See Figure 3.2.

Output to $\mathcal{P}$. Similar to the previous instruction, with the roles of $\mathcal{P}$ and $\mathcal{Q}$ reversed.

Linear combination. In this instruction, a linear combination of values in the circuit (plus an optional constant) is computed: $V[k_0] \leftarrow v_0 + \sum_{i=1}^{m-1} V[k_i] \cdot v_i$. Concretely, both parties perform local operations on their shares. Additionally, $\mathcal{P}$ sends φ to $\mathcal{Q}$ to ensure that both parties agree on the shared input φ. See Figure 3.3.

$\mathcal{P}$.**LinearCombination**($\langle\varphi\rangle$):	$\mathcal{Q}$.**LinearCombination**($\langle\varphi\rangle$):
Here $\varphi = \langle m, k_0, v_0, \langle k_1, v_1\rangle, \ldots, \langle k_{m-1}, v_{m-1}\rangle\rangle$ with $m \in \mathbb{N}^*$, $(\forall i \in \mathbb{N}_m : k_i \in \Lambda^\star)$ and $(\forall i \in \mathbb{N}_m : v_i \in \mathbb{Z}_n)$.	
Abort if $k_0 \in AP$.	Abort if $k_0 \in AQ$.
Abort unless $\forall i \in \mathbb{N}_m : k_i \in RP$.	Abort unless $\forall i \in \mathbb{N}_m : k_i \in RQ$.
$AP \leftarrow k_0$;	$AQ \leftarrow k_0$;
$SP[k_0] \leftarrow v_0 + \sum_{i=1}^{m-1} SP[k_i] \cdot v_i$;	$SQ[k_0] \leftarrow \sum_{i=1}^{m-1} SQ[k_i] \cdot v_i$;
$CP[k_0] \leftarrow \mathsf{cadd}(\mathsf{com}(v_0, 0),$ $\mathsf{cmul}(CP[k_1], v_1), \mathsf{cmul}(CP[k_2], v_2), \ldots)$;	$CQ[k_0] \leftarrow \mathsf{cadd}($ $\mathsf{cmul}(CQ[k_1], v_1), \mathsf{cmul}(CQ[k_2], v_2), \ldots)$;
$XP[k_0] \leftarrow \sum_{i=1}^{m-1} XP[k_i] \cdot v_i$;	$XQ[k_0] \leftarrow \sum_{i=1}^{m-1} XQ[k_i] \cdot v_i$;
$CQ[k_0] \leftarrow \mathsf{cadd}($ $\mathsf{cmul}(CQ[k_1], v_1), \mathsf{cmul}(CQ[k_2], v_2), \ldots)$.	$CP[k_0] \leftarrow \mathsf{cadd}(\mathsf{com}(v_0, 0),$ $\mathsf{cmul}(CP[k_1], v_1), \mathsf{cmul}(CP[k_2], v_2), \ldots)$.
φ (to ensure that they agree on φ) ⇢	
$RP \leftarrow k_0$.	$RQ \leftarrow k_0$.

Fig. 3.3: Linear combination.

Multiplication. In this instruction, the product of two values in the circuit is computed: $V[k_0] \leftarrow V[k_1] \cdot V[k_2]$. We can rewrite this as:

$$SP[k_0] + SQ[k_0] \leftarrow \underbrace{SP[k_1] \cdot SP[k_2]}_{\hat{p}} + \underbrace{SP[k_1] \cdot SQ[k_2]}_{(\tilde{u}+\tilde{v})} + \underbrace{SQ[k_1] \cdot SP[k_2]}_{(u+v)} + \underbrace{SQ[k_1] \cdot SQ[k_2]}_{\hat{q}}$$

where we introduce $\hat{p}, \hat{q}, \tilde{u}, \tilde{v}, u, v$ to simplify the discussion. The idea of this protocol is for $\mathcal{P}$ and $\mathcal{Q}$ to compute $\hat{p}$ and $\hat{q}$, respectively, using their private shares. They then jointly compute $\tilde{u}$ and $\tilde{v}$ using the Π_{mul} subroutine, which we introduce for clarity and which we describe in Section 3.3.2. Afterwards, u and v are computed using a second instantiation of Π_{mul}. Finally, $\mathcal{P}$ sets $SP[k_0] \leftarrow \hat{p} + \tilde{u} + u$ and $\mathcal{Q}$ sets $SQ[k_0] \leftarrow \hat{q} + \tilde{v} + v$. See Figure 3.4 for the construction.

One can optimize the protocol in Figure 3.4 by using the same homomorphic encryption key for both instances of Π_{mul} and merging the proofs inside and outside of Π_{mul} whenever possible.[4] We can thus save one proof of correctess for the encryption key, and save on some overhead in $\mathcal{F}_{\text{gzk}}$.

Proof by $\mathcal{P}$. In this instruction, $\mathcal{P}$ proves to $\mathcal{Q}$ in zero-knowledge some statement involving *1)* witnesses outside of the circuit, *2)* values that $\mathcal{P}$ input into the circuit, and *3)* values that $\mathcal{P}$ got as an output from the circuit. See Figure 3.5 for the construction.

Proof by $\mathcal{Q}$. Similar to the previous instruction, with the roles of $\mathcal{P}$ and $\mathcal{Q}$ reversed.

3.3.2 The Π_{mul} Subroutine for Multiplication of Committed Inputs

We now give the construction of the 2-party $\mathcal{F}_{\text{gzk}}$-hybrid protocol Π_{mul} for multiplication of committed inputs, which we use as a subroutine in Π_{abb} in the multiplication

[4] Concretely, one would merge the proofs with the following labels: *1)* $\langle$`m5:`$\varphi\rangle$, $\langle$`cm1:`$\langle$`m7:`$\varphi\rangle\rangle$ and $\langle$`cm1:`$\langle$`m8:`$\varphi\rangle\rangle$; *2)* $\langle$`m6:`$\varphi\rangle$, $\langle$`cm2:`$\langle$`m7:`$\varphi\rangle\rangle$, and $\langle$`cm2:`$\langle$`m8:`$\varphi\rangle\rangle$; *3)* $\langle$`cm3:`$\langle$`m7:`$\varphi\rangle\rangle$ and $\langle$`cm3:`$\langle$`m8:`$\varphi\rangle\rangle$; *4)* $\langle$`cm4:`$\langle$`m7:`$\varphi\rangle\rangle$ and $\langle$`cm4:`$\langle$`m8:`$\varphi\rangle\rangle$.

$\mathcal{P}$.**Multiply**($\langle\varphi\rangle$):	$\mathcal{Q}$.**Multiply**($\langle\varphi\rangle$):
Here $\varphi = \langle k_0, k_1, k_2\rangle$ with $k_0, k_1, k_2 \in \Lambda^\star$.	
Abort if $k_0 \in AP$.	Abort if $k_0 \in AQ$.
Abort unless $k_1, k_2 \in RP$.	Abort unless $k_1, k_2 \in RQ$.
$AP \leftarrow k_0$;	$AQ \leftarrow k_0$.
$\hat{p} \leftarrow SP[k_1] \cdot SP[k_2]$; $(c\hat{p}, o\hat{p}) \xleftarrow{\$} \mathsf{com}(\hat{p})$.	$\hat{q} \leftarrow SQ[k_1] \cdot SQ[k_2]$; $(c\hat{q}, o\hat{q}) \xleftarrow{\$} \mathsf{com}(\hat{q})$.
$c\hat{p}, \pi_{\mathtt{m5}}$ ⟶	
⟵ $c\hat{q}, \pi_{\mathtt{m6}}$	
$(\tilde{u}, c\tilde{u}, o\tilde{u}, c\tilde{v}) \xleftarrow{\$} \mathcal{P}.\Pi_{\mathrm{mul}}($ $SP[k_1], CP[k_1], XP[k_1], CQ[k_2], \langle\mathtt{m7}:\varphi\rangle)$;	$(\tilde{v}, c\tilde{v}, o\tilde{v}, c\tilde{u}) \xleftarrow{\$} \mathcal{Q}.\Pi_{\mathrm{mul}}($ $SQ[k_2], CQ[k_2], XQ[k_2], CP[k_1], \langle\mathtt{m7}:\varphi\rangle)$;
$(u, cu, ou, cv) \xleftarrow{\$} \mathcal{P}.\Pi_{\mathrm{mul}}($ $SP[k_2], CP[k_2], XP[k_2], CQ[k_1], \langle\mathtt{m8}:\varphi\rangle)$;	$(v, cv, ov, cu) \xleftarrow{\$} \mathcal{Q}.\Pi_{\mathrm{mul}}($ $SQ[k_1], CQ[k_1], XQ[k_1], CP[k_2], \langle\mathtt{m8}:\varphi\rangle)$;
$SP[k_0] \leftarrow \hat{p} + \tilde{u} + u$; $XP[k_0] \leftarrow o\hat{p} + o\tilde{u} + ou$;	$SQ[k_0] \leftarrow \hat{q} + \tilde{v} + v$; $XQ[k_0] \leftarrow o\hat{q} + o\tilde{v} + ov$;
$CP[k_0] \leftarrow \mathsf{cadd}(c\hat{p}, c\tilde{u}, cu)$;	$CQ[k_0] \leftarrow \mathsf{cadd}(c\hat{q}, c\tilde{v}, cv)$;
$CQ[k_0] \leftarrow \mathsf{cadd}(c\hat{q}, c\tilde{v}, cv)$; $RP \leftarrow k_0$.	$CP[k_0] \leftarrow \mathsf{cadd}(c\hat{p}, c\tilde{u}, cu)$; $RQ \leftarrow k_0$.

Instantiation of zero-knowledge proofs:

$$\pi_{\mathtt{m5}} := \mathcal{F}_{\mathrm{gzk}}[\langle\mathtt{m5}:\varphi\rangle]\big\{\big(\exists o\hat{p}, SP[k_1], SP[k_2], XP[k_1], XP[k_2]\big) :$$
$$\mathsf{cvfy}(c\hat{p}, o\hat{p}, SP[k_1] \cdot SP[k_2]) \wedge \mathsf{cvfy}(CP[k_1], XP[k_1], SP[k_1]) \wedge \mathsf{cvfy}(CP[k_2], XP[k_2], SP[k_2])\big\}.$$

$$\pi_{\mathtt{m6}} := \mathcal{F}_{\mathrm{gzk}}[\langle\mathtt{m6}:\varphi\rangle]\big\{\big(\exists o\hat{q}, SQ[k_1], SQ[k_2], XQ[k_1], XQ[k_2]\big) :$$
$$\mathsf{cvfy}(c\hat{q}, o\hat{q}, SQ[k_1] \cdot SQ[k_2]) \wedge \mathsf{cvfy}(CQ[k_1], XQ[k_1], SQ[k_1]) \wedge \mathsf{cvfy}(CQ[k_2], XQ[k_2], SQ[k_2])\big\}.$$

Fig. 3.4: Multiplication. The subroutine Π_{mul} is defined in Section 3.3.2 and Figure 3.6.

$\mathcal{P}$.**ProofP**($\langle\varphi, x, w\rangle$):	$\mathcal{Q}$.**ProofP**($\langle\varphi\rangle$):
Here $\varphi = \langle m, \langle k_0, \ldots, k_{m-1}\rangle, R\rangle$; R is a relation that is compatible with $\mathcal{F}_{\mathrm{gzk}}$; $m \in \mathbb{N}$; $(\forall i \in \mathbb{N}_m : k_i \in \Lambda^\star)$; $x \in R[0]$; and $w = \langle w_0, \ldots, w_{m-1}\rangle$.	
Abort unless $\forall i \in \mathbb{N}_m : k_i \in KP$.	Abort unless $\forall i \in \mathbb{N}_m : k_i \in KP$.
$x, \pi_{\mathtt{pp}}$ ⟶	
	Output x.

Instantiation of zero-knowledge proofs:

$$\pi_{\mathtt{pp}} := \mathcal{F}_{\mathrm{gzk}}[\langle\mathtt{pp}:\varphi\rangle]\big\{\big(\nexists w_0, \ldots, w_{m-1}\ \exists V[k_0], \ldots, V[k_{m-1}], XP[k_0], \ldots, XP[k_{m-1}]\big) :$$
$$\bigwedge_{i=0}^{m-1} \mathsf{cvfy}(CP[k_i], XP[k_i], V[k_i] - SQ[k_i]) \wedge R\big(x, (w_0, \ldots, w_{m-1}) \cup (V[k_i], \ldots, V[k_{m-1}])\big) = 1\big\}.$$

Fig. 3.5: Proof by $\mathcal{P}$.

instruction. In a nutshell: on $\mathcal{P}$'s private input a and $\mathcal{Q}$'s private input b, Π_{mul} outputs shares to the product: u to $\mathcal{P}$ and v to $\mathcal{Q}$, such that $u + v = a \cdot b$.

The protocol draws on ideas from Ishai et al's $\tilde{\pi}^{\mathsf{OT}}$ protocol—defined in Appendix A.2 of the full version of their paper [IPS08]—and uses a similar approach as many two-party computation protocols (e.g., Damgård and Orlandi's π_{mul} protocol [DO10b]). We fleshed out the details of Ishai et al.'s protocol to make it secure against *active* adversaries, improve its efficiency, and integrate it into our overall protocol.

$\mathcal{P}.\Pi_{\text{mul}}(a, ca, oa, cb, \lambda)$:		$\mathcal{Q}.\Pi_{\text{mul}}(b, cb, ob, ca, \lambda)$:
$(pk, sk) \xleftarrow{\$} \mathsf{rkgen}(n)$; $w \xleftarrow{\$} \mathbb{Z}_n$;		$s \xleftarrow{\$} \mathbb{Z}_n$; $t \xleftarrow{\$} \mathbb{Z}_n$;
$(ew, rw) \xleftarrow{\$} \mathsf{enc}(pk, w)$.		$(cs, os) \xleftarrow{\$} \mathsf{com}(s)$; $(ct, ot) \xleftarrow{\$} \mathsf{com}(t)$.
$(\text{⛀} : rw)$	$ew, pk, \pi_{\texttt{cm1}}$ ⟶	
$\sigma \leftarrow a - w$.		$(et, rt) \xleftarrow{\$} \mathsf{enc}(pk, t)$; $ey \leftarrow \mathsf{add}(\mathsf{mul}(ew, s), et)$.
	⟵ $cs, ct, ey, \pi_{\texttt{cm2}}$	$(\text{⛀} : rt)$
$y \leftarrow \mathsf{dec}(ey, sk)$; $(cy, oy) \xleftarrow{\$} \mathsf{com}(y)$.		$\delta \leftarrow b - s$; $o\delta \leftarrow ob - os$.
$(\text{⛀} : sk)$	$cy, \sigma, \pi_{\texttt{cm3}}$ ⟶	
	⟵ $\delta, \pi_{\texttt{cm4}}$	
$u \leftarrow \delta \cdot a + y$; $ou \leftarrow oa \cdot \delta + oy$;		$v \leftarrow \sigma \cdot s - t$; $ov \leftarrow os \cdot \sigma - ot$;
$cu \leftarrow \mathsf{cadd}(\mathsf{cmul}(ca, \delta), cy)$;		$cv \leftarrow \mathsf{cadd}(\mathsf{cmul}(cs, \sigma), \mathsf{cmul}(ct, -1))$.
$cv \leftarrow \mathsf{cadd}(\mathsf{cmul}(cs, \sigma), \mathsf{cmul}(ct, -1))$.		$cu \leftarrow \mathsf{cadd}(\mathsf{cmul}(ca, \delta), cy)$.
Return (u, cu, ou, cv).		Return (v, cv, ov, cu).

Instantiation of zero-knowledge proofs:

$$\pi_{\texttt{cm1}} := \mathcal{F}_{\text{gzk}}[\langle \texttt{cm1}\!:\!\varphi\rangle]\big\{\big(\gtrdot w\ \exists sk\big) : (pk, sk) \in \mathsf{rkgen}(n) \wedge w = \mathsf{dec}(ew, sk)\big\}.$$

$$\pi_{\texttt{cm2}} := \mathcal{F}_{\text{gzk}}[\langle \texttt{cm2}\!:\!\varphi\rangle]\big\{\big(\gtrdot s\ \exists t, os, ot, rt\big) : \mathsf{cvfy}(cs, os, s) \wedge \mathsf{cvfy}(ct, ot, t) \wedge ey = \mathsf{add}(\mathsf{mul}(ew, s), \mathsf{enc}(pk, t, rt))\big\}.$$

$$\pi_{\texttt{cm3}} := \mathcal{F}_{\text{gzk}}[\langle \texttt{cm3}\!:\!\varphi\rangle]\big\{\big(\exists y, w, oy, oa, sk\big) : (pk, sk) \in \mathsf{rkgen}(n) \wedge y = \mathsf{dec}(ey, sk) \wedge w = \mathsf{dec}(ew, sk) \wedge \mathsf{cvfy}(cy, oy, y) \wedge \mathsf{cvfy}(ca, oa, w + \sigma)\big\}.$$

$$\pi_{\texttt{cm4}} := \mathcal{F}_{\text{gzk}}[\langle \texttt{cm4}\!:\!\varphi\rangle]\big\{\big(\exists o\delta\big) : \mathsf{cvfy}(\mathsf{cadd}(cb, \mathsf{cmul}(cs, -1)), o\delta, \delta)\big\}.$$

Fig. 3.6: The Π_{mul} sub-protocol.

The basic idea of the protocol is for $\mathcal{P}$ and $\mathcal{Q}$ to first obtain shares y and $(-t)$ on the product of two *random* values w and s, respectively: $y - t = w \cdot s$; second to erase all intermediate state used in the previous step; third to exchange the values $\sigma = (a - w)$ and $\delta = (b - s)$; and finally to obtain shares on the product of the actual input values a and b by outputting $u = \delta \cdot a + y$ and $v = \sigma \cdot s - t$, respectively. Commitments and relevant proofs are used during all steps. We refer to Figure 3.6 for the construction.

The erasure in Step 2 is needed to ensure security against *adaptive* adversaries: since the encryption scheme used in our protocol is not receiver–non-committing [CHK05a], the simulator cannot produce a convincing view of the first step for any other value of w. In fact, there are no known practical receiver–non-committing schemes that satisfy our requirements. By erasing state in Step 2, the simulator is dispensed with producing that view in Step 3.

3.3.3 Efficiency Considerations for the Zero-Knowledge Proofs in Π_{abb}

Careful design enables us to achieve a very efficient and practical construction. In particular, we minimize the amount of computation required inside the realization π

of the zero-knowledge proof functionality $\mathcal{F}_{\text{gzk}}$, which accounts for the majority of the runtime of our protocol, as follows.

1) Instead of using the Paillier encryption scheme as in Camenisch et al. [CKS11] to verifiably encrypt the witnesses whose *knowledge* is proven in π, we use the Camenisch-Shoup encryption scheme with short keys, short randomness, and with modulus n^2. Paillier encryption implies the use of a different modulus, since the simulator needs to know its factorization to extract the witnesses.

2) We use homomorphic commitment and encryption schemes that work with groups of the same order n. Most of the witnesses used in $\mathcal{F}_{\text{gzk}}$ therefore live in a group of known order n, and most operations inside π stay inside groups of order n. We therefore do not need to encrypt values larger than n in π, and can avoid expensive integer commitments in π [CKS11].

3) We use the cheaper proofs of *existence* [CKS11] instead of proofs of *knowledge* wherever possible. This reduces the number of verifiable encryptions needed inside π.

4) Finally, we use an encryption scheme in Π_{mul} where the proof of correctness of key generation is cheap. (For Camenisch-Shoup encryption with full key length and Paillier encryption, this proof is very expensive.)

3.4 Additional Instructions for $\mathcal{F}_{\text{abb}}$

We will start this section by showing how one can create a $\mathcal{F}_{\text{abb}}$-hybrid protocol that includes additional instructions for generating random numbers, random bits, inverting, and doing several other useful operations. Certain useful instructions however require that the $\mathcal{F}_{\text{abb}}$ functionality itself is modified and not just used as a building block: we show here a new output instruction for $\mathcal{F}_{\text{abb}}$ that returns a value exponentiated by a certain group element g instead of revealing the value directly; we will use that instruction in Section 3.7 for constructing an oblivious pseudorandom function that is secure against *dynamic* corruptions in the UC model.

3.4.1 Instructions as Part of a Higher-Level Protocol

Random integers. A random value can be shared as follows: $\mathcal{P}$ and $\mathcal{Q}$ each choose a random number $a \xleftarrow{\$} \mathbb{Z}_n$ and $b \xleftarrow{\$} \mathbb{Z}_n$, respectively, input it into $\mathcal{F}_{\text{abb}}$, and finally sum their inputs $c \leftarrow a + b$ using the *Linear Combination* instruction. Provided that at least one of the two is honest, the value c is uniformly distributed in $\mathbb{Z}_n$.

Random bits. $\mathcal{P}$ and $\mathcal{Q}$ can share a random bit as follows: $\mathcal{P}$ and $\mathcal{Q}$ each choose a random number $a \xleftarrow{\$} \{-1, 1\}$ and $b \xleftarrow{\$} \{-1, 1\}$, respectively, and input it to $\mathcal{F}_{\text{abb}}$. They then compute a^2 and b^2 using the *Multiplication* instruction, and reveal the result to each other. The protocol aborts if $a^2 \neq 1$ or $b^2 \neq 1$. They then compute $c \leftarrow a \cdot b$. The value c is now uniformly distributed in $\{-1, 1\}$, provided that at least one of the two parties is honest and the factorization of n is unknown to both of them. To adjust the random value to $\mathbb{Z}_2$, they can compute $d \leftarrow c \cdot (2^{-1} \pmod n) + (2^{-1} \pmod n)$.

Inversion. This algorithm is based on a technique by Bar-Ilan and Beaver [BIB89]:

1. Let $V[k_1]$ denote the value to invert.
2. $\mathcal{P}$ and $\mathcal{Q}$ choose a random integer $V[k_2]$ as shown earlier in this section.
3. They multiply both values: $V[k_3] \leftarrow V[k_1] \cdot V[k_2]$.
4. The product is output first to $\mathcal{P}$, and then to $\mathcal{Q}$: $v_3 \leftarrow V[k_3]$.
5. They invert the value: $v_4 \leftarrow v_3^{-1} \pmod{n}$ (abort if v_3 is not invertible.);
6. and compute the result: $V[k_5] \leftarrow V[k_2] \cdot v_4 = V[k_2] \cdot (V[k_1] \cdot V[k_2])^{-1} = (V[k_1])^{-1}$.

As long as $V[k_1]$ is invertible mod n, this protocol aborts with negligible probability, is correct, and perfectly preserves the privacy of $V[k_1]$. If $V[k_1] = 0$, this fact will be revealed. As we assumed the factorization of n to be unknown, we can safely ignore the case where $V[k_1]$ is a multiple of a non-trivial factor of n.

Other useful operations. Our protocol is almost compatible with the algorithms by Damgård, Fitzi et al. [DFK+06] for performing comparisons (including inequalities), bit decompositions, modular reduction, modular exponentiation, etc. of the values in the circuit. Their setting assumed that the values in the circuit are in a prime order group, but in our scheme n is composite. Fortunately the only operation that they use in their paper that cannot be performed in a composite order group—finding a square root modulo n—is needed only for generating random bits; by replacing that algorithm by the version presented earlier, no more problems remain.

3.4.2 Modifying $\mathcal{F}_{\text{abb}}$ to Add New Instructions

Unfortunately there are some useful instructions that cannot be added "on top of" $\mathcal{F}_{\text{abb}}$ as described in the previous subsection, but have to be included "inside" $\mathcal{F}_{\text{abb}}$: the UC composition theorem can therefore not be applied, and the security proof has to be redone. We give here an example of such an instruction: it is a variant of the output instruction that outputs not $V[k]$ but $g^{V[k]}$, where $\langle g \rangle = \mathbb{G}$ is some abelian group (written multiplicatively) of order n.

Informal definition of the ideal functionality. The high-level description of the additional instructions is the following:

- *Exponentiated output to $\mathcal{P}$:* $\mathcal{F}_{\text{abb}}$ parses the common input φ as $\langle k, \mathbb{G}, g \rangle$ where $\mathbb{G}$ is the description of some group (written multiplicatively) of order n, and $g \in \mathbb{G}$ is a generator of $\mathbb{G}$. $\mathcal{F}_{\text{abb}}$ delivers $g^{V[k]}$ to $\mathcal{P}$.
- *Exponentiated output to $\mathcal{Q}$:* Idem, with the roles of $\mathcal{P}$ and $\mathcal{Q}$ reversed.

Formal definition of the ideal functionality. The formal definition of the additional instructions is the following:

- **Exponentiated output to $\mathcal{P}$:** In this instruction, parse φ as $\langle k, \mathbb{G}, g \rangle$ where $k \in \Lambda^\star$, $\mathbb{G}$ is the description of some group (written multiplicatively) of order n, and $g \in \mathbb{G}$ is a generator of $\mathbb{G}$.
  - `ep:p:`φ : Accept $\langle$`ep:p:`$\varphi\rangle$ from $\mathcal{P}$ where $k \in RP$: send $\langle$`ep:p:`$\varphi\rangle$ to $\mathcal{A}$.
  - `ep:q:`φ : Accept $\langle$`ep:q:`$\varphi\rangle$ from $\mathcal{Q}$ where $k \in RQ$: send $\langle$`ep:q:`$\varphi\rangle$ to $\mathcal{A}$.
  - `ep:lock:`φ [`ep:p:`$\varphi \wedge$ `ep:q:`φ] : Accept $\langle$`ep:lock:`$\varphi\rangle$ from $\mathcal{A}$: send $\langle\rangle$ to $\mathcal{A}$.

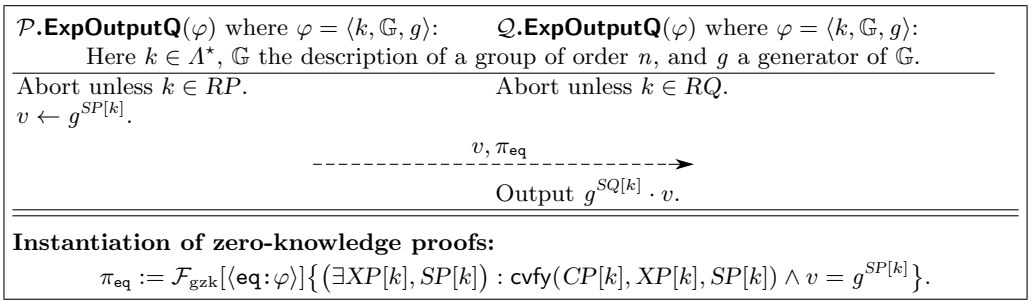


Fig. 3.7: Exponentiated output to $\mathcal{Q}$.

  - `ep:deliver`:φ [`ep:lock`:φ] : Accept $\langle$`ep:deliver`:$\varphi\rangle$ from $\mathcal{A}$: send $\langle$`ep:deliver`:$\varphi, g^{V[k]}\rangle$ to $\mathcal{P}$.
  - `ep:done`:φ [`ep:lock`:φ] : Accept $\langle$`ep:done`:$\varphi\rangle$ from $\mathcal{A}$: send $\langle$`ep:done`:$\varphi\rangle$ to $\mathcal{Q}$.
- **Exponentiated output to $\mathcal{Q}$:** This is similar to the previous instruction, with the roles of $\mathcal{P}$ and $\mathcal{Q}$ reversed, and the label prefix is changed to `eq`. We do not formalize this instruction here.

Construction of exponentiated output to $\mathcal{Q}$. $\mathcal{Q}$ retrieves an exponentiated value $g^{V[k]}$, where $\mathbb{G}$ and $g \in \mathbb{G}$ can be chosen freely. Concretely, $\mathcal{P}$ exponentiates her share and sends it to $\mathcal{Q}$ together with a proof of correctness. See Figure 3.7 for the construction.

3.5 Security Proof

We now show that the protocol of Section 3.3 with the extensions of Section 3.4.2 realizes $\mathcal{F}_{\mathrm{abb}}$. We start with a description of the main ideas of the security proof before presenting the full proof. The full proof proceeds in two steps: we first prove that our protocol is secure when run with *nice* environments. We then apply the special composition theorem of Camenisch et al. [CKS11] to prove that our protocol is secure against *all* environments.

3.5.1 Main Ideas

We use the standard approach for proving the security of protocols in the UC or GNUC models: we construct a straight-line simulator $\mathcal{S}$ such that for all polynomial-time–bounded environments $\mathcal{E}$ and all polynomial-time–bounded adversaries $\mathcal{A}$, the environment $\mathcal{E}$ cannot distinguish a protocol execution with $\mathcal{A}$ and Π_{abb} in the ($\mathcal{F}_{\mathrm{ac}}$, $\mathcal{F}_{\mathrm{gzk}}$)-hybrid "real" world from a protocol execution with $\mathcal{S}$ and $\mathcal{F}_{\mathrm{abb}}$ in the "ideal" world. We prove that $\mathcal{E}$ cannot distinguish these two worlds by defining a sequence of intermediate "hybrid" worlds (the first one being the real world and the last one the ideal world) and showing that $\mathcal{E}$ cannot distinguish between any two consecutive hybrid worlds in that sequence. We follow the formalism of the GNUC framework to deal with CRS's and system parameters (see Section 10 of the GNUC paper [HS11]).

The main difficulties in constructing the simulator $\mathcal{S}$ are as follows: *1)* $\mathcal{S}$ has to extract the inputs of all corrupted parties; *2)* $\mathcal{S}$ has to compute and send commitments and ciphertexts on behalf of the honest parties without knowing their inputs, i.e., $\mathcal{S}$ cannot commit and encrypt the right values; *3)* when an honest party gets corrupted mid-protocol, $\mathcal{S}$ has to provide to $\mathcal{A}$ the full *non-erased* intermediate state of the party, in particular the opening of the commitments and the randomness of the encryptions.

To address the first difficulty, recall that the parties are required to perform a proof of *knowledge* of all new inputs to the circuit. The simulator $\mathcal{S}$ can therefore recover the input of all corrupted parties with the help of $\mathcal{F}_{\text{gzk}}$. In the first few hybrid worlds, the statistically binding commitments ensure that the values in the circuit stay consistent with the inputs. In the subsequent hybrid worlds, the computational indistinguishability of the two types of CRS ensure that the adversary cannot equivocate commitments even when $\mathcal{S}$ uses the perfectly-hiding CRS with trapdoor.

We now address the second and third difficulty. Upon corruption of a party, $\mathcal{S}$ is allowed to recover the original input of that party from $\mathcal{F}_{\text{abb}}$. By using the perfectly-hiding CRS with trapdoor, $\mathcal{S}$ can equivocate all commitments it made so far to ensure that the committed values are consistent with the view of the adversary. By construction, $\mathcal{S}$ never needs to reveal the randomness used for an encryption for which it does not know the plaintext. Recall that in Π_{mul}, the parties first encrypt a random offset, then erase the decryption key and the randomness used to encrypt, and only then deliver the encryption of the offset plus party's input to the adversary (recall that $\mathcal{F}_{\text{gzk}}$ allows the erasure of witnesses *before* delivering the statement to be proven to the other party). The simulator $\mathcal{S}$ can adjust the offset so that the view delivered to the adversary is consistent. See also Appendix A.2 of Ishai et al.'s paper [IPS08].

The rest of the security proof is now straightforward.

3.5.2 Security Proof

Let $\Pi_{\text{abb}}^{\mathcal{F}_{\text{gzk}} \to \pi}$ be the $(\mathcal{F}_{\text{sch}}, \mathcal{F}_{\text{ac}})$-hybrid protocol in which every instance of $\mathcal{F}_{\text{gzk}}$ in Π_{abb} has been replaced by the zero-knowledge protocol π described in Camenisch et al.'s paper [CKS11]. To prove our scheme secure, we need to prove the following theorem:

Theorem 3.2. *Assuming the CRH encryption scheme is semantically secure and the DDH assumption holds for safe-semiprime order groups, the protocol* $\Pi_{\text{abb}}^{\mathcal{F}_{\text{gzk}} \to \pi}$ *realizes* $\mathcal{F}_{\text{abb}}$.

To prove the theorem, we first need the following two definitions and prove the following lemma.

Definition 3.3 (Nice envionrments [CKS11]). *A* nice *environment is an environment that never asks* $\mathcal{A}$ *to submit a false statement to* $\mathcal{F}_{\text{gzk}}$ *and* $\mathcal{F}_{\text{gzk}}^{2\text{v}}$.

Definition 3.4 ($\mathcal{F}_{\text{gzk}}$-friendly protocols [Kre12]). *A* $\mathcal{F}_{\text{gzk}}$-friendly *protocol is a protocol in which honest parties acting as provers only prove true statements with* $\mathcal{F}_{\text{gzk}}$ *and* $\mathcal{F}_{\text{gzk}}^{2\text{v}}$.

Lemma 3.5. *Assuming the CRH encryption scheme is semantically secure and the DDH assumption holds for safe-semiprime order groups: there exists a simulator* $\mathcal{S}$ *that*

does not extract the witnesses quantified by $\exists$ *in any* $\mathcal{F}_{\text{gzk}}$*, such that for all PPT* nice *environments* $\mathcal{E}$ *and the dummy adversary* $\mathcal{A}$*:* $\text{Exec}(\Pi_{\text{abb}}, \mathcal{A}, \mathcal{E})$ *and* $\text{Exec}(\mathcal{F}_{\text{abb}}, \mathcal{S}, \mathcal{E})$ *are computationally indistinguishable.*

In the above, the system parameter n *is chosen by* $\mathsf{rgen}(1^\eta)$ *before* $\mathcal{E}$ *starts executing.*

Proof of Lemma 3.5. In Section 3.5.3, we construct a simulator $\mathcal{S}$ and prove that is satisfies the requirements of the Lemma 3.5.

Proof of Theorem 3.2. Since the simulator we constructed in Section 3.5.3 satisfies the requirements of Lemma 3.5, and since, by construction, Π_{abb} is a $\mathcal{F}_{\text{gzk}}$-friendly protocol, it follows from the special composition theorem of Camenisch et al. [CKS11] that Theorem 3.2 holds.

3.5.3 Proof of Lemma 3.5

Notation. We adopt the convention that the ideal functionalities in the $(\mathcal{F}_{\text{gzk}}, \mathcal{F}_{\text{ac}})$-hybrid "real" world (and which are controlled by $\mathcal{S}$) are surrounded by quotes: "$\mathcal{F}_{\text{gzk}}$" and "$\mathcal{F}_{\text{ac}}$". Note that $\mathcal{S}$ does not have to run these ideal functionalities honestly, it just needs to ensure that the messages $\mathcal{S}$ sends on their behalf are indistinguishable from an honest execution. Furthermore, we denote the parties in the real world as "$\mathcal{P}$" and "$\mathcal{Q}$". When such a party is honest, it is controlled by $\mathcal{S}$; when that party is corrupted, it is controlled by the adversary $\mathcal{A}$.

The simulator $\mathcal{S}$ is a six-interface system. The simulator $\mathcal{S}$ communicates with $\mathcal{F}_{\text{abb}}$ through 3 interfaces: the $\mathcal{S}$-interface (where $\mathcal{F}_{\text{abb}}$ sends data to and receives data from the ideal adversary), the $\mathcal{P}$-interface (which is active only after $\mathcal{P}$ becomes corrupted, and where $\mathcal{F}_{\text{abb}}$ sends data to and receives data from the *corrupted* $\mathcal{P}$) and the $\mathcal{Q}$-interface (idem but for $\mathcal{Q}$). The simulator $\mathcal{S}$ runs one instance of the real-world adversary $\mathcal{A}$. It relays all messages between $\mathcal{E}$ and $\mathcal{A}$. The simulator $\mathcal{S}$ communicates with $\mathcal{A}$ through 3 interfaces: the "$\mathcal{A}$"-interface (connected to the adversary interfaces of all "$\mathcal{F}_{\text{gzk}}$" and "$\mathcal{F}_{\text{ac}}$" used in the protocol execution), the "$\mathcal{P}$"-interface (which is active only after $\mathcal{P}$ becomes corrupted, and which is connected to the $\mathcal{P}$-interface of all "$\mathcal{F}_{\text{gzk}}$" and "$\mathcal{F}_{\text{ac}}$" used in the protocol execution), and the "$\mathcal{Q}$" interface (idem for $\mathcal{Q}$). See Figures 3.8, 3.9, 3.10, and 3.11 for a schematic representation of the construction of $\mathcal{S}$ in the cases where no parties are corrupted, $\mathcal{P}$ is corrupted, $\mathcal{Q}$ is corrupted, and all parties are corrupted, respectively.

3.5.3.1 Initialization

Before running $\mathcal{A}$ for the first time, $\mathcal{S}$ programs the CRS using cgen'_0 so that commitments are perfectly hiding, and so that $\mathcal{S}$ knows the trapdoor tc which will enable it to equivocate all commitments it makes on behalf of "$\mathcal{P}$" and "$\mathcal{Q}$".

Recall that since n is part of the system parameters, $\mathcal{S}$ does not know its factorization.

3.5.3.2 $\mathcal{P}$ and $\mathcal{Q}$ Honest

When $\mathcal{P}$ and $\mathcal{Q}$ are both honest, $\mathcal{A}$ sees only status messages without any content. The construction of $\mathcal{S}$ is therefore straightforward. See Figure 3.8. For completeness, we will show the behaviour of $\mathcal{S}$ for the *Input from* $\mathcal{P}$ and *Multiplication* instructions. The behaviour of $\mathcal{S}$ for all other instructions is similar to its behaviour for *Input from* $\mathcal{P}$.

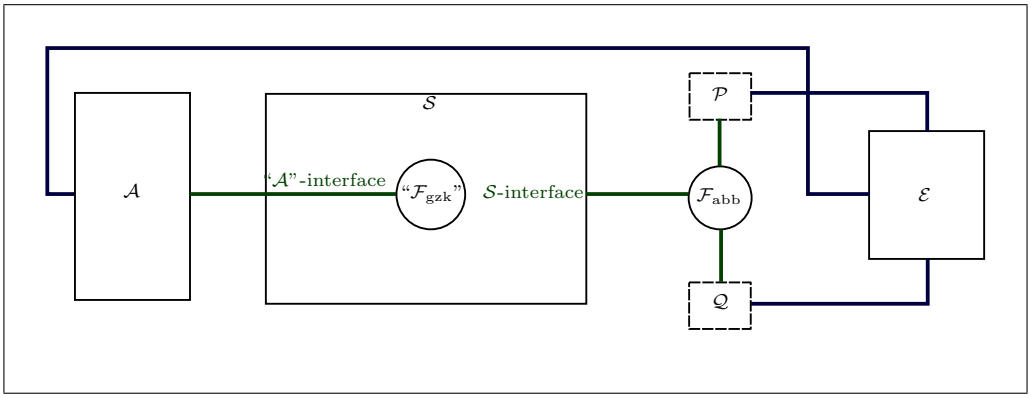


Fig. 3.8: Construction of $\mathcal{S}$ in case all parties are honest. For simplicity, we chose to represent only one ideal functionality in the construction of $\mathcal{S}$.

Input from $\mathcal{P}$. Upon receiving $\langle$`ip:send`:$\varphi\rangle$ from $\mathcal{F}_{\text{abb}}$ (through the $\mathcal{S}$-interface), send $\langle$`send`$, \ell(\ldots)\rangle$ to $\mathcal{A}$ (through the "$\mathcal{A}$"-interface). The length of the statement and witnesses is fixed, so $\mathcal{S}$ knows what value $\ell(\ldots)$ to send.

Upon receiving $\langle$`ip:ready`:$\varphi\rangle$ from $\mathcal{F}_{\text{abb}}$, send $\langle$`ready`$\rangle$ to $\mathcal{A}$.

Upon receiving $\langle$`lock`$\rangle$ from $\mathcal{A}$, send $\langle$`ip:lock`:$\varphi\rangle$ to $\mathcal{F}_{\text{abb}}$. Wait for $\langle\rangle$ from $\mathcal{F}_{\text{abb}}$, and send $\langle\rangle$ to $\mathcal{A}$.

Upon receiving $\langle$`deliver`$\rangle$ from $\mathcal{A}$, send $\langle$`ip:deliver`:$\varphi\rangle$ to $\mathcal{F}_{\text{abb}}$.

Upon receiving $\langle$`done`$\rangle$ from $\mathcal{A}$, send $\langle$`ip:done`:$\varphi\rangle$ to $\mathcal{F}_{\text{abb}}$.

Multiplication. This instruction is more complex than all others, since it contains several independant instances of "$\mathcal{F}_{\text{gzk}}$". We divide the "$\mathcal{A}$"-interface into m sub-interfaces (numbered from 1 to m), one for each instance of "$\mathcal{F}_{\text{gzk}}$".

Upon receiving $\langle$`m:p`:$\varphi\rangle$ from $\mathcal{F}_{\text{abb}}$, send $\langle$`send`$, \ell(\ldots)\rangle$ to $\mathcal{A}$ via the *first* sub-interface (of the "$\mathcal{A}$"-interface). The length of the statement and witness is fixed, so $\mathcal{S}$ knows what value $\ell(\ldots)$ to send.

Upon receiving $\langle$`m:q`:$\varphi\rangle$ from $\mathcal{F}_{\text{abb}}$, send $\langle$`ready`$\rangle$ to $\mathcal{A}$ via the *first* sub-interface.

Upon receiving $\langle$`lock`$\rangle$ from $\mathcal{A}$ via the ith sub-interface, where $i \neq m$, send $\langle\rangle$ to $\mathcal{A}$ via the ith sub-interface.

Upon receiving $\langle$`lock`$\rangle$ from $\mathcal{A}$ via the mth sub-interface, send $\langle$`m:lock`:$\varphi\rangle$ to $\mathcal{F}_{\text{abb}}$. Wait for $\langle\rangle$ from $\mathcal{F}_{\text{abb}}$, and send $\langle\rangle$ to $\mathcal{A}$ via the mth sub-interface.

Upon receiving $\langle$`done`$\rangle$ from $\mathcal{A}$ via the ith sub-interface, where $i \neq m$, send $\langle$`ready`$\rangle$ to $\mathcal{A}$ via the $(i+1)$st sub-interface.

Upon receiving $\langle$`deliver`$\rangle$ from $\mathcal{A}$ via the ith sub-interface, where $i \neq m$, send $\langle$`send`, $\ell(\ldots)\rangle$ to $\mathcal{A}$ via the $(i+1)$st sub-interface. The length $\ell(\ldots)$ is easy for $\mathcal{S}$ to determine.

Upon receiving $\langle$`done`$\rangle$ from $\mathcal{A}$ via the mth sub-interface, send $\langle$`m:done:p`:$\varphi\rangle$ to $\mathcal{F}_{\text{abb}}$.

Upon receiving $\langle$`deliver`$\rangle$ from $\mathcal{A}$ via the mth sub-interface, send $\langle$`m:done:q`:$\varphi\rangle$ to $\mathcal{F}_{\text{abb}}$.

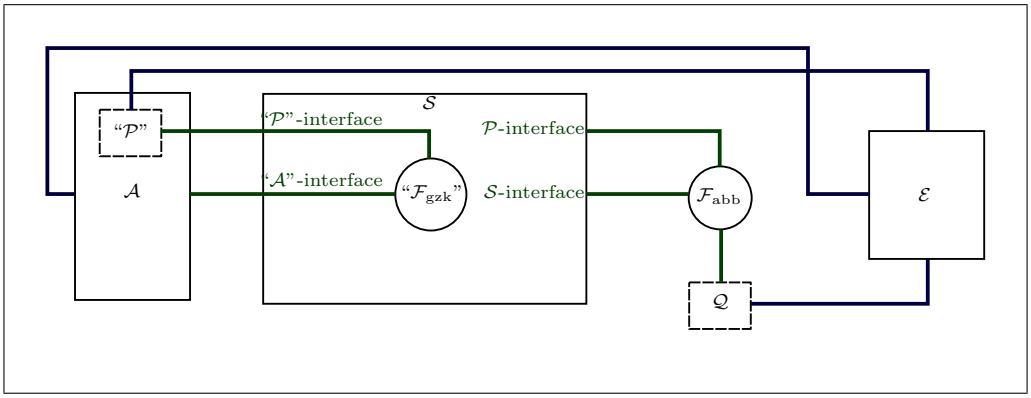


Fig. 3.9: Construction of $\mathcal{S}$ in case $\mathcal{P}$ is corrupted.

3.5.3.3 $\mathcal{P}$ Corrupted First

Without loss of generality, we may assume that whenever $\mathcal{P}$ gets corrupted, all of her subroutines are immediately corrupted as well. We only need to show how $\mathcal{S}$ operates in the case that $\mathcal{P}$ starts out corrupted: if $\mathcal{P}$ gets corrupted later, $\mathcal{S}$ starts by recovering $\mathcal{P}$'s input by sending one $\langle$`ip:expose`:$\varphi\rangle$ message to $\mathcal{F}_{\text{abb}}$ for each *Input by* $\mathcal{P}$ instruction that has already processed the $\langle$`ip:send`:$\varphi\rangle$ message. The simulator $\mathcal{S}$ also recovers $\mathcal{P}$'s external witnesses from each *Proof by* $\mathcal{P}$ instruction that has processed the $\langle$`pp:send`:$\varphi\rangle$ message but not the $\langle$`pp:lock`:$\varphi\rangle$ message by sending $\langle$`pp:expose`:$\varphi\rangle$ to $\mathcal{F}_{\text{abb}}$. Now, $\mathcal{S}$ internally restarts the simulation of "$\mathcal{P}$" from the beginning until the point where she was corrupted. For all instructions except the *Proof by* $\mathcal{P}$ that have already processed the $\langle$`pp:lock`:$\varphi\rangle$ message, $\mathcal{S}$ can perfectly re-create "$\mathcal{P}$"'s input. For the *Proof by* $\mathcal{P}$ instructions that have processed the $\langle$`pp:lock`:$\varphi\rangle$ message, $\mathcal{S}$ can use arbitrary input (of the correct length!) for "$\mathcal{P}$", since that input will be erased by "$\mathcal{P}$" and since it does not affect the remainder of the protocol. Finally, $\mathcal{S}$ hands over the internal state of "$\mathcal{P}$" to $\mathcal{A}$. This state is perfectly consistent with $\mathcal{A}$'s view so far. Of course, we will have to deal with the possibility that $\mathcal{Q}$ is corrupted later on, which we tackle later in this section.

Overview. Recall that when $\mathcal{P}$ is corrupted, $\mathcal{S}$ must play $\mathcal{P}$ for $\mathcal{F}_{\text{abb}}$ based on the actions of "$\mathcal{P}$" (assumed by $\mathcal{A}$), and must play "$\mathcal{Q}$" for $\mathcal{A}$ *without knowing the correct input of* $\mathcal{Q}$. See Figure 3.9.

In constructing the simulator, we maintain the invariant that $\mathcal{S}$ knows the value of "$\mathcal{P}$"'s shares $SP[k]$ when these are ready to be used, i.e., $k \in RP$.

Input from $\mathcal{P}$. For this instruction, $\mathcal{S}$ simply needs to extract the input of "$\mathcal{P}$" from the messages flowing on the "$\mathcal{P}$"-interface or the "$\mathcal{A}$"-interface. We show here the exact behaviour of $\mathcal{S}$ for completeness.

Upon receiving $\langle$`send`$, \langle CP[k]\rangle, \langle v\rangle, \ldots\rangle$ through the "$\mathcal{P}$"-interface, save v as $SP[k]$. Send $\langle$`send`$\rangle$ through the "$\mathcal{A}$"-interface.

Upon receiving $\langle$`reset`$, \langle CP[k]\rangle, \langle v\rangle, \ldots\rangle$ through the "$\mathcal{A}$"-interface, update $SP[k]$ with the new value of v. Send $\langle\rangle$ through the "$\mathcal{A}$"-interface.

Upon receiving $\langle$`expose`$\rangle$ through the "$\mathcal{A}$"-interface, send $\langle$`expose`$, \langle CP[k]\rangle, \langle SP[k]\rangle\rangle$ through the "$\mathcal{A}$"-interface.

Upon receiving $\langle$`ip:ready`:$\varphi\rangle$ through the $\mathcal{S}$-interface, send $\langle$`ready`$\rangle$ through the "$\mathcal{Q}$"-interface.

Upon receiving $\langle$`lock`$\rangle$ through the "$\mathcal{A}$"-interface, send $\langle$`ip:send`:$\varphi, SP[k]\rangle$ through the $\mathcal{P}$-interface. Wait for $\langle$`ip:send`:$\varphi\rangle$ through the $\mathcal{S}$-interface, send $\langle$`ip:lock`:$\varphi\rangle$ through the $\mathcal{S}$-interface. Wait for $\langle\rangle$ through the $\mathcal{S}$-interface, send $\langle\rangle$ through the "$\mathcal{A}$"-interface.

Upon receiving $\langle$`deliver`$, \ell\rangle$ through the "$\mathcal{A}$"-interface, send $\langle$`ip:deliver`:$\varphi\rangle$ through the $\mathcal{S}$-interface.

Upon receiving $\langle$`done`$\rangle$ through the "$\mathcal{A}$"-interface, send $\langle$`ip:done`:$\varphi\rangle$ through the $\mathcal{S}$-interface. Wait for $\langle$`ip:done`:$\varphi\rangle$ through the $\mathcal{P}$-interface, send $\langle$`done`$\rangle$ through the "$\mathcal{P}$"-interface.

Input from $\mathcal{Q}$. For this instruction, $\mathcal{S}$ needs to generate an equivocable commitment to $\mathcal{Q}$'s input, which $\mathcal{S}$ doesn't know.

The construction of $\mathcal{S}$ in response to the `send`, `ready`, `lock` and `done` messages is straightforward.

Upon receiving $\langle$`deliver`$, \ell\rangle$ through the "$\mathcal{A}$"-interface, commit to 0 using an equivocable commitment: $(SQ[k], XQ[k]) \xleftarrow{\$} \mathsf{com}(0)$, and send $\langle$`deliver`$, \langle SQ[k]\rangle\rangle$ through the "$\mathcal{P}$"-interface.

Output to $\mathcal{P}$. In this instruction, $\mathcal{S}$ recovers the value that is output from the circuit from $\mathcal{F}_{\text{abb}}$ just in time to be able to play "$\mathcal{Q}$" in a consistent manner.

The construction of $\mathcal{S}$ in response to the `q`, `ready`, and `done` messages is straightforward.

Upon receiving $\langle$`lock`$\rangle$ through the "$\mathcal{A}$"-interface, $\mathcal{S}$ sends $\langle$`op:lock`:$\varphi\rangle$ through the $\mathcal{S}$-interface and expects $\langle\rangle$ through the $\mathcal{S}$-interface. Then, $\mathcal{S}$ sends $\langle$`op:deliver`:$\varphi\rangle$ through the $\mathcal{S}$-interface, and expects $\langle$`op:deliver`:$\varphi, V[k]\rangle$ through the $\mathcal{P}$-interface, thereby recovering $V[k]$.

Upon receiving $\langle$`deliver`$, \ell\rangle$ through the "$\mathcal{A}$"-interface, $\mathcal{S}$ sends the message $\langle$`deliver`$, V[k] - SP[k]\rangle$ through the "$\mathcal{P}$" interface.

Exponentiated output to $\mathcal{P}$. For this instruction, $\mathcal{S}$ behaves similarily than for the *Output to $\mathcal{P}$* instruction. The difference is that it receives $\langle$`ep:deliver`:$\varphi, g^{V[k]}\rangle$ through the $\mathcal{P}$-interface, and sends $\langle$`deliver`$, g^{V[k]}/g^{SP[k]}\rangle$ through the "$\mathcal{P}$"-interface.

Output to $\mathcal{Q}$. The simulation of this instruction is straightforward.

Exponentiated output to $\mathcal{Q}$. The simulation of this instruction is straightforward.

Linear combination. The simulation of this instruction is straightforward. Additionally, $\mathcal{S}$ computes "$\mathcal{P}$"'s share $SP[k_0]$ based on the values of $SP[k_i]$ (which $\mathcal{S}$ knows).

Proof by $\mathcal{P}$. The simulation of this instruction is also relatively straightforward. Futhermore, we explain why $\mathcal{S}$ also works properly in the security proof of Π_{abb}, the realization of $\mathcal{F}_{\text{gabb}}$ which allows one to prove the *existence* of external witnesses.

$\mathcal{S}$'s reaction to the `ready`, `done`, and `deliver` messages is straightforward.

Upon receiving $\langle$`send`$, x, w, \ldots\rangle$ through the "$\mathcal{P}$"-interface, save x and w. Send $\langle$`send`$, \ell(x, w)\rangle$ through the "$\mathcal{A}$" interface.

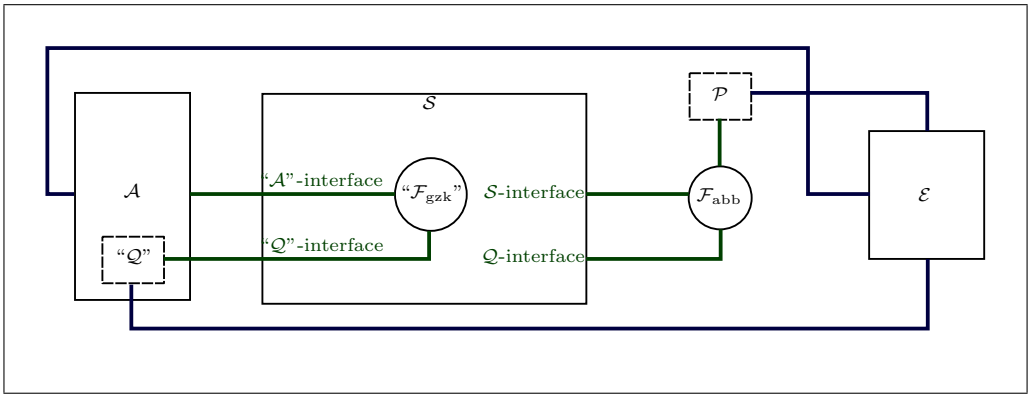


Fig. 3.10: Construction of $\mathcal{S}$ in case $\mathcal{Q}$ is corrupted.

Upon receiving $\langle \texttt{reset}, x, w, \ldots \rangle$ through the "$\mathcal{A}$"-interface, save the updated x and w. Send $\langle\rangle$ through the "$\mathcal{A}$" interface.

Upon receiving $\langle \texttt{expose} \rangle$ through the "$\mathcal{A}$"-interface, send $\langle \texttt{expose}, x, w \rangle$ through the "$\mathcal{A}$"-interface.

Upon receiving $\langle \texttt{lock} \rangle$ through the "$\mathcal{A}$" interface, send $\langle \texttt{pp:send}{:}\varphi, x, |w|, w, \texttt{0}, \langle\rangle \rangle$ through the $\mathcal{P}$-interface. Expect $\langle \texttt{pp:send}{:}\varphi, \ell \rangle$ through the $\mathcal{S}$-interface. Send $\langle \texttt{pp:lock}{:}\varphi \rangle$ through the $\mathcal{S}$-interface, expect $\langle\rangle$ through the $\mathcal{S}$-interface. Send $\langle\rangle$ through the "$\mathcal{A}$"-interface.

Proof by $\mathcal{Q}$. The simulation of this instruction is straightforward.

Multiplication. See the next paragraph for the behaviour of $\mathcal{S}$ inside Π_{mul}: $\mathcal{S}$ recovers "$\mathcal{P}$"'s private outputs $\tilde{u}$ and u. The simulation of the remainder of this instruction is straightforward. Additionally, $\mathcal{S}$ computes "$\mathcal{P}$"'s share $SP[k_0]$ based on the values of $SP[k_1]$, $SP[k_2]$, $\tilde{u}$ and u (which $\mathcal{S}$ knows).

Sub-protocol Π_{mul}. Recall that $\mathcal{S}$ knows "$\mathcal{P}$"'s input a from the *Multiplication* instruction. The construction of the simulator is straightforward, expect for three changes where $\mathcal{S}$ deviates from the honest execution.

First, when receiving $\langle \texttt{send}, \langle ew, pk \rangle, w, \ldots \rangle$ through the $\langle \texttt{cm1:}\lambda \rangle$ sub-interface of the "$\mathcal{P}$"-interface, $\mathcal{S}$ saves the value w instead of discarding it.

Second, instead of sending the correct $\langle \texttt{deliver}, \langle cs, ct, ey \rangle \rangle$ message through the $\langle \texttt{cm2:}\lambda \rangle$ sub-interface of the "$\mathcal{P}$"-interface, $\mathcal{S}$ chooses a random y, encrypts it $(ey, ry) \xleftarrow{\$} \texttt{enc}(y)$, and delivers the inconsistent ey. Note that $\mathcal{S}$ will never have to show ry, since "$\mathcal{Q}$" would have erased that value already.

Third, instead of sending the correct $\langle \texttt{deliver}, \delta \rangle$ message through the $\langle \texttt{cm4:}\lambda \rangle$ sub-interface of the "$\mathcal{P}$"-interface, $\mathcal{S}$ chooses a random δ, and delivers it.

Finally, $\mathcal{S}$ recovers "$\mathcal{P}$"'s output as follows: $u \leftarrow \delta \cdot a + y$ (with the values of y and δ that $\mathcal{S}$ chose). Notice that $\mathcal{S}$ did not use "$\mathcal{Q}$"'s input b.

3.5.3.4 $\mathcal{Q}$ Corrupted First

This case is similar to the case where $\mathcal{P}$ was corrupted first. For all instructions except Π_{mul}, $\mathcal{S}$'s behaviour can be inferred from its behaviour in the case where $\mathcal{P}$ was corrupted first. See Figure 3.10.

Sub-protocol Π_{mul}. Recall that $\mathcal{S}$ knows "$\mathcal{Q}$"'s input b from the *Multiplication* instruction. The construction of the simulator is straightforward, expect for two changes where $\mathcal{S}$ deviates from the honest execution.

First, when receiving $\langle \texttt{send}, \langle cs, ct, ey \rangle, s, \ldots \rangle$ through the $\langle \texttt{cm2:}\lambda \rangle$ sub-interface of the "$\mathcal{Q}$"-interface, $\mathcal{S}$ saves the value s instead of discarding it.

Second, instead of sending the correct $\langle \texttt{deliver}, \langle cy, \sigma \rangle \rangle$ message through the $\langle \texttt{cm3:}\lambda \rangle$ sub-interface of the "$\mathcal{Q}$"-interface, $\mathcal{S}$ chooses a random σ, and delivers that. $\mathcal{S}$ will never have to show sk, since "$\mathcal{P}$" would have erased that value already.

Finally $\mathcal{S}$ recovers "$\mathcal{Q}$"'s output v as follows: $s \leftarrow b - \delta; t \leftarrow y - w \cdot s; v \leftarrow \sigma \cdot s - t$ (using the value of σ that $\mathcal{S}$ chose). Notice that $\mathcal{S}$ did not use "$\mathcal{P}$"'s input a.

3.5.3.5 Adjusting "$\mathcal{Q}$"'s State When $\mathcal{Q}$ is Corrupted Second

When $\mathcal{Q}$ gets corrupted second, $\mathcal{S}$ needs to come up with a believable internal state for "$\mathcal{Q}$". In order to do so, $\mathcal{S}$ sets "$\mathcal{Q}$"'s internal state (SQ, XQ and local variables) in each instruction, in the order in which they were processed, as follows:

Input from $\mathcal{P}$. The adjustements to make are straightforward.

Input from $\mathcal{Q}$. If $\mathcal{S}$ accepted the $\langle \texttt{iq:send:}\varphi \rangle$ message, $\mathcal{S}$ recovers the orignal input of $\mathcal{Q}$: $\mathcal{S}$ sends $\langle \texttt{iq:expose:}\varphi \rangle$ through the $\mathcal{S}$-interface, and expects $\langle \texttt{iq:expose:}\varphi, SQ[k] \rangle$ through the $\mathcal{S}$-interface. $\mathcal{S}$ now *adjusts the opening* $XQ[k]$ *of the commitment* $CQ[k]$, i.e., $\mathcal{S}$ uses the Trap to find a new value of the opening $XQ[k]$ of the commitment $CQ[k]$ so that $\mathsf{cvfy}(CQ[k], XQ[k], SQ[k]) = \mathsf{true}$.

Output to $\mathcal{P}$. The adjustements to make are straightforward. Notice that the value $(V[k] - SP[k])$ that $\mathcal{S}$ delivered is equal to $SQ[k]$ as expected, unless $\mathcal{A}$ somehow managed to equivocate one of her commitments.

Output to $\mathcal{Q}$. The adjustements to make are straightforward.

Exponentiated output to $\mathcal{P}$. The adjustements to make are straightforward. Notice that the value $(g^{V[k]}/g^{SP[k]})$ that $\mathcal{S}$ delivered is equal to $g^{SQ[k]}$ as expected, unless $\mathcal{A}$ somehow managed to equivocate one of her commitments.

Exponentiated output to $\mathcal{Q}$. The adjustements to make are straightforward.

Proof by $\mathcal{P}$. The adjustements to make are straightforward.

Proof by $\mathcal{Q}$. If $\mathcal{S}$ accepted the $\langle \texttt{pq:send:}\varphi \rangle$ message, but did not yet deliver the $\langle \texttt{pq:lock:}\varphi \rangle$ message, $\mathcal{A}$ still has a chance to send $\langle \texttt{expose} \rangle$ to $\mathcal{S}$ through the "$\mathcal{A}$"-interface, and therefore $\mathcal{S}$ needs to recover $\mathcal{Q}$'s input, i.e., x and w. $\mathcal{S}$ sends $\langle \texttt{pq:expose:}\varphi \rangle$ through the $\mathcal{S}$-interface, and expects $\langle \texttt{pq:expose:}\varphi, x, w \rangle$. $\mathcal{S}$ saves x.

The remainder of the adjustements to make are straightforward.

Linear combination. $\mathcal{S}$ computes "$\mathcal{Q}$"'s share $SQ[k_0]$ based on the other shares $SQ[k_i]$, and adjusts the opening $XQ[k_0]$.

Multiplication. $\mathcal{S}$ performs the necessary adjustements inside the Π_{mul} subroutine. $\mathcal{S}$ will recover "$\mathcal{Q}$"'s output $\tilde{v}$ and v from Π_{mul}.

$\mathcal{S}$ computes "$\mathcal{Q}$"'s share $SQ[k_0]$ based on $SQ[k_1]$, $SQ[k_2]$, $\tilde{v}$, and v. $\mathcal{S}$ adjusts the opening $XQ[k_0]$.

The remainder of the adjustements to make are straightforward.

Sub-protocol Π_{mul}. If $\mathcal{Q}$ gets corrupted before the delivery of the $\langle \texttt{deliver}, \langle cs, ct, ey \rangle \rangle$ message through the $\langle \texttt{cm2:}\lambda \rangle$ sub-interface of the "$\mathcal{P}$"-interface, the adjustements to make are trivial.

If $\mathcal{Q}$ gets corrupted after the delivery of the $\langle \texttt{deliver}, \langle cs, ct, ey \rangle \rangle$ message through the $\langle \texttt{cm2:}\lambda \rangle$ sub-interface of the "$\mathcal{P}$"-interface, but before the delivery of the $\langle \texttt{deliver}, \delta \rangle$ message through the $\langle \texttt{cm4:}\lambda \rangle$ sub-interface of the "$\mathcal{P}$"-interface, $\mathcal{S}$ is committed to its "incorrect" ey. Since $\mathcal{A}$ can decrypt that value, $\mathcal{S}$ is therefore also committed to y. $\mathcal{S}$ needs to find s and t consistent with y: $\mathcal{S}$ sets s at random, and computes $t \leftarrow y - w \cdot s$. $\mathcal{S}$ then adjusts the opening os and ot using the trapdoor. $\mathcal{S}$ does not need to compute rt, as it can claim that "$\mathcal{Q}$" securely erased that value already. et can be re-computed from ey, ew and s.

If $\mathcal{Q}$ gets corrupted after the delivery of the $\langle \texttt{deliver}, \delta \rangle$ message through the $\langle \texttt{cm4:}\lambda \rangle$ sub-interface of the "$\mathcal{P}$"-interface, $\mathcal{S}$ is committed to ey (i.e., to y) and to δ. $\mathcal{S}$ sets $s \leftarrow b - \delta$, $t \leftarrow y - w \cdot s$, $v \leftarrow \sigma \cdot s - t$, and adjusts the openings ot, os, and ov using the trapdoor. $\mathcal{S}$ now knows the correct output of "$\mathcal{Q}$".

The remainder of the adjustements to make are straightforward.

3.5.3.6 Adjusting "$\mathcal{P}$"'s State when $\mathcal{P}$ is Corrupted Second

This case is similar to the case where $\mathcal{Q}$ was corrupted second. For all instructions except Π_{mul}, $\mathcal{S}$'s behaviour can be inferred from $\mathcal{S}$'s behaviour in the case where $\mathcal{Q}$ was corrupted second.

Sub-protocol Π_{mul}. If $\mathcal{P}$ gets corrupted before the delivery of the $\langle \texttt{deliver}, \langle cy, \sigma \rangle \rangle$ message through the $\langle \texttt{cm3:}\lambda \rangle$ sub-interface of the "$\mathcal{Q}$"-interface, the adjustements to make are trivial.

If $\mathcal{P}$ gets corrupted after the delivery of that message, $\mathcal{S}$ is bound to w (via ew and pk) and to σ (which was delivered to $\mathcal{A}$). However at this point, $\mathcal{S}$ can claim that "$\mathcal{P}$" already securely erased rw and sk, and so it can get away with revealing a value of w that is inconsistent with ew and pk (the semantic security of CRH cryptosystem hides that inconsistency, as proven more formally later). $\mathcal{S}$ now needs to adjust the values y and w in "$\mathcal{P}$"'s internal state: $\mathcal{S}$ computes $w' \leftarrow a - \sigma$ and $y' \leftarrow (w' - w) \cdot s + y$, and replaces y by y' and w by w' in "$\mathcal{P}$"'s internal state. Furthermore $\mathcal{S}$ adjusts oy using the trapdoor. Finally, $\mathcal{S}$ recomputes "$\mathcal{P}$"'s output u: $u \leftarrow \delta \cdot a + y'$ and adjusts the opening ou using the trapdoor.

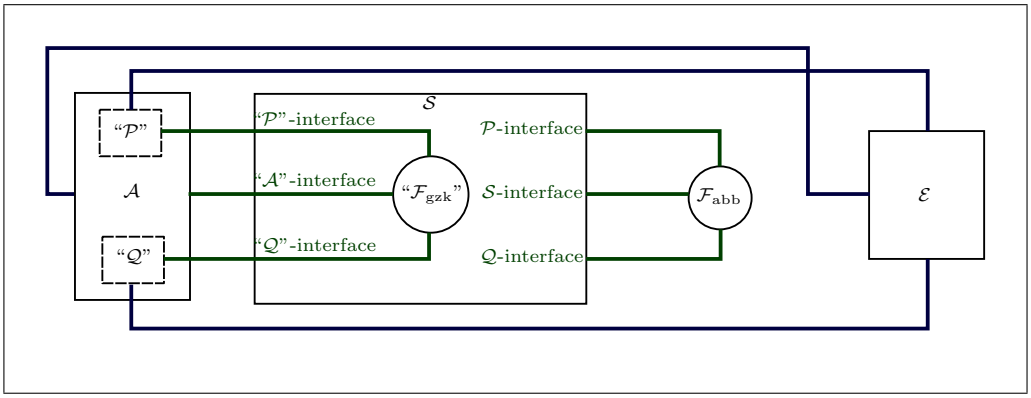


Fig. 3.11: Construction of $\mathcal{S}$ in case all parties are corrupted.

3.5.3.7 Both Parties Corrupted

Once $\mathcal{S}$ has handed over the complete *non-erased* internal state of the second corrupted party to $\mathcal{A}$, the simulation is trivial: $\mathcal{S}$ runs "$\mathcal{F}_{\text{gzk}}$" and "$\mathcal{F}_{\text{ac}}$" honestly, and does not send any messages to $\mathcal{F}_{\text{abb}}$. See Figure 3.11.

3.5.3.8 Proof of Indistinguishability

We are going to define a sequence of games Game_1 to $\mathsf{Game}_{N_{\text{games}}}$, as described by Shoup [Sho04]. In the first game, everything is distributed as in the protocol Π_{abb}, whereas in the last game everything is distributed as in the ideal world $\mathcal{F}_{\text{abb}}$. By the piling-up lemma, the advantage of $\mathcal{E}$ is less than the sum of the advantages in distinguishing between Game_i and Game_{i+1}. We are going to prove that $\mathcal{E}$ only has negligible advantage in distinguishing between two consecutive games, based either on a reduction to a hard cryptographic problem, or by "failure events" happening with negligible probability. As long as the number of games is polynomial w.r.t. the security parameter, the total advantage of $\mathcal{E}$ is negligible.

We stress that in all intermediate games, $\mathcal{E}$ and $\mathcal{A}$ interact with a machine that runs both $\mathcal{F}_{\text{abb}}$ and $\mathcal{S}_i$. Without loss of generality, we assume that $\mathcal{S}_i$ thus obtains the inputs and outputs of all honest parties from $\mathcal{F}_{\text{abb}}$. It is only in the last game, which is identical to the "ideal world" and where $\mathcal{S}$ is equal to $\mathcal{S}_i$, that $\mathcal{S}_i$ does not make use of these inputs and outputs.

Game_1. As observed in the previous paragraph, $\mathcal{S}_1$ receives the input of all honest parties. $\mathcal{S}_1$ simply runs the parties it controls honestly, and exposes their internal state to $\mathcal{A}$ when corrupted. $\mathcal{S}_1$ generates the CRS honestly using cgen_1. By construction, this setting is perfectly indistinguishable from the $(\mathcal{F}_{\text{gzk}}, \mathcal{F}_{\text{ac}})$-hybrid "real" world Π_{abb}.

Game_2. $\mathcal{S}_2$ runs like $\mathcal{S}_1$, except that it aborts if $\mathcal{A}$'s output is inconsistent with its inputs at any time during the protocol. The probability that $\mathcal{S}_2$ aborts is at most the probability that the commitment was not binding after all, which is negligible.

Game_3. $\mathcal{S}_3$ runs like $\mathcal{S}_2$, except that now it chooses the CRS with cgen'_0 instead of cgen_1. The commitment scheme is now perfectly hiding, and $\mathcal{S}_3$ can now efficiently equivocate commitments using the trapdoor information. The advantage that $\mathcal{E}$ has in distinguishing between Game_3 and Game_2 is equal to its advantage in breaking DDH in the group modulo p generated by g, which is negligible.

Game_4. $\mathcal{S}_4$ runs as $\mathcal{S}_3$, except that during the Π_{mul} subroutines, it behaves as described earlier in this section, i.e., it ignores the input of the parties it controls during the Π_{mul} protocol and reconstructs a plausible history upon corruption. It is easy to see that the only way $\mathcal{E}$ can get an advantage in distinguishing between Game_4 and Game_3 is, if upon corruption of "$\mathcal{P}$", it notices that the values w, ew and pk are inconsistent. We now argue that the advantage of $\mathcal{E}$ is at most the advantage $\mathcal{E}$ has in breaking the semantic security of the Camenisch-Shoup cryptosystem times the number of Π_{mul} sub-protocols that "$\mathcal{P}$" started, which is still negligible.

Let $N_{\Pi_{\text{mul}}}$ be the number of times "$\mathcal{P}$" calls Π_{mul} in Π_{abb}.

We now define a polynomial number of hybrid games $\mathsf{Game}_{3:0}$ to $\mathsf{Game}_{3:N_{\Pi_{\text{mul}}}}$, where in $\mathsf{Game}_{3:i}$ $\mathcal{S}$ behaves like $\mathcal{S}_4$ for the first i calls to Π_{mul}, and like $\mathcal{S}_3$ for the subsequent calls. Clearly $\mathsf{Game}_{3:0}$ is exactly Game_3 and $\mathsf{Game}_{3:N_{\Pi_{\text{mul}}}}$ is exactly Game_4.

If there exists $\mathcal{E}$ which has non-negligible advantage γ in distinguishing between Game_4 and Game_3, then there must exist another environment $\mathcal{E}'$ and a value $i \in \mathbb{N}^*_{N_{\Pi_{\text{mul}}}}$ such that $\mathcal{E}'$ has advantage $\gamma/N_{\Pi_{\text{mul}}}$ in distinguishing between $\mathsf{Game}_{3:i}$ and $\mathsf{Game}_{3:i-1}$, which is still a non-negligible advantage.

We now show how $\mathcal{S}$ can use such an $\mathcal{E}'$ to break the semantic security of the CRH cryptosystem function with advantage $\gamma/N_{\Pi_{\text{mul}}}$.

In the ith run of the Π_{mul} protocol on behalf of "$\mathcal{P}$", instead of computing ew honestly, $\mathcal{S}$ submits two plaintexts w and w' to the challenger of the semantic security game, yielding a challenge plaintext $e\bar{w}$ which is either equal to the consistent ew or the inconsistent ew', and a public key pk. Recall that $\mathcal{S}$ does not need to know the value of w in order to properly run the simulation; w is only needed upon corruption of $\mathcal{P}$. $\mathcal{S}$ now uses $e\bar{w}$ instead of ew.

If $\mathcal{P}$ becomes corrupted before the delivery of σ in the ith Π_{mul} protocol, then $\mathcal{S}$ aborts the simulation (it cannot produce a convincing value of sk) and returns a random guess to the challenger. Since in this run the view of $\mathcal{E}'$ would have been perfectly indistinguishable, $\mathcal{S}$ does not lose any advantage by aborting.

If however $\mathcal{P}$ become corrupted after the delivery of σ (or not at all), then $\mathcal{S}$ will produce an internal state for "$\mathcal{P}$" that contains, among others: w, $e\bar{w}$, and pk. $\mathcal{S}$ returns the same guess as $\mathcal{E}'$ to the challenger: if $e\bar{w} = ew$ then the view of $\mathcal{E}'$ is exactly that of $\mathsf{Game}_{3:i-1}$, and if $e\bar{w} = ew'$ then the view is exactly that of $\mathsf{Game}_{3:i}$.

The distinguishing advantage of $\mathcal{E}$ between Game_4 and Game_3 is therefore negligible.

Game_5. $\mathcal{S}_5$ runs as described earlier in this section for all instructions, and not just Π_{mul}. By construction, $\mathcal{S}_5$ does not need to know the input of the honest parties ($\mathcal{S}_5$ extracts it from $\mathcal{F}_{\text{abb}}$) and $\mathcal{S}_5$'s behaviour is perfectly indistinguishable from $\mathcal{S}$'s behaviour in the ideal world. The difference between Game_5 and Game_4 is zero, thanks to the perfectly hiding commitments.

Conclusion. This concludes the proof of Lemma 3.5, i.e., Π_{abb} securely implements $\mathcal{F}_{\text{abb}}$ for all *nice* environments.

3.6 Related Work and Comparison

There is an extensive literature on the subject of multi-party computation (MPC); however, most of these settings consider only the case of an honest majority, which is not helpful for the two-party case.

Canetti et al. [CLOS02] present the first MPC protocols for general functionalities that are secure with dishonest majority in the UC framework; however, these protocols are rather a proof of concept, i.e., they are not at all practical, as they rely on generic zero-knowledge proofs.

More efficient MPC protocols for evaluating *boolean* circuits, secure with dishonest majority, have been designed [LP07, LPS08, NO09, PSSW09]. Impressive results have been obtained in particular for the evaluation of the AES block cipher [PSSW09, DK10, DKL+12, KshS12, NNOB12]. While such protocols could be used to evaluate arithmetic circuits modulo n, a heavy price would have to be paid: each gate in the arithmetic circuit would "blow up" into many boolean gates, resulting in an impractical protocol.

The first practical protocols for evaluating arithmetic circuits modulo n were presented by Cramer et al. [CDN01] (CDN-protocol) and Damgård and Nielsen [DN03] (DN-protocol). While both protocols assume an honest majority, they can be shown to be secure in the two-party case (as noted by Ishai et al. [IPS09, IPS08]) if one relaxes the requirement for fair delivery of messages (fair delivery is impossible in the two-party case). Both protocols have stronger set-up assumptions than ours: they assume the existence of a trusted third party that distributes shares of the secret key to all parties. The CDN-protocol is only *statically* secure and is not UC-secure, and we therefore exclude it from our comparison. The DN-protocol is *adaptively* secure (with erasures) in the UC model (secure *without erasures* only in the honest majority case), and is slightly (about 30%) slower than ours.

Ishai et al. [IPS08,IPS09] present protocols for evaluating arithmetic circuits in several algebraic rings, including one for the ring $\mathbb{Z}_n$ for a composite n. These protocols achieve security with a dishonest majority, and are secure with respect to *adaptive* corruptions (assuming erasures), but only against *honest-but-curious* adversaries. They note that standard techniques can be used to make their protocols secure also for *malicious* adversaries, however it is not clear if the resulting construction will be practical. Our protocol draws on ideas from their construction, however we are able to achieve a significant speed-up compared to a naive implementation using "standard techniques" by ensuring that all commitments live in $\mathbb{Z}_n$ and by using the short-key variant of the homomorphic encryption scheme.

Damgård and Orlandi [DO10a] (DO-protocol), as well as Bendlin et al. [BDOZ11] (BDOZ-protocol), give protocols for evaluating arithmetic circuits modulo a prime p. Damgård et al. [DPSZ12] (SPDZ-protocol) later improved upon the BDOZ-protocol. These protocols divide the workload into a computationally intensive *pre-processing* phase and a much lighter *on-line* phase. The pre-processing phase is statically secure, however the on-line phase can be made adaptively secure (in the UC-model) [DO10a, BDOZ11, DPSZ12]. These papers optimize the runtime of the on-line phase (the BDOZ- and SPDZ-protocols make use of local additions and multiplications only). In the pre-processing phase of these protocols, it is necessary to prepare for many multiplications gates (about 80 in the BDOZ-, several hundred in the DO-, and tens of thousands

in the SPDZ-protocol) making these protocols impractical for small circuits. This pre-processing phase takes several minutes even for reasonable security parameters. Our protocol is better suited for small circuits.

Even for large circuits, the computational complexity of our protocol is about 3.3 times lower than that of the BDOZ- and DO-protocols. It must be noted that the BDOZ- and DO-protocols have slightly weaker setup assumptions than ours: they only require a *random string* as the CRS, while we also need an *RSA modulus with unknown factorization* as a system parameter. (This is not a huge drawback of our protocol, see Section 3.2.1.)

The SPDZ-protocol is about an order of magnitude faster than our protocol, however, unlike the BDOZ-, DO-, and our protocols, it cannot evaluate reactive circuits, severely limiting its applicability in the real world. It also requires a trusted key setup, which is a stronger setup assumption than ours. (Concurrently to our work, Damgård et al. [DKL+13] lifted the restriction on reactive circuits, but only in the *random oracle model*. They also lifted the restriction on the trusted key setup but only for *covert* security.)

None of the UC-secure protocols discussed have an equivalent to the *Proof* instruction in their ideal functionality. This makes it hard to compose them with other protocols because of the issue with non-committed inputs in a 2-party setting, as dicussed in the introduction, thus negating some of the advantages of working in the UC model.

3.6.1 Efficiency Comparison

Table 3.1 summarizes the amortized runtimes per multiplication gate of our protocol, the DN- (when run as a 2-party protocol), the DO-, and the BDOZ-protocols. We assume that the runtime of an exponentiation with a fixed modulus length scales linearly with the size of the exponent. Let $\exp.n$ denote the runtime per bit in the exponent of an exponentiation modulo n or modulo p,[5] and similarily $\exp.n^2$ for exponentiations modulo n^2. Let $\operatorname{lb} n$ be equal to $\log_2(n)$. Recall that η denotes security parameter. For each protocol, we counted the number of exponentiations with an exponent of at least η bits. Faster operations, in particular multiplications and divisions, are ignored. We also ignored the time needed for secure channel setup, did not consider multi-exponentiations, and ignored network delay. We provide an estimate of the runtime when run with the "smallest general purpose" security level of the Ecrypt-II recommendations [BCC+11] ($\eta = 80$, $\operatorname{lb} n = 1248$) on a standard laptop with a 64-bit operating system.[6]

For a fair comparison, we replace all Paillier encryptions [Pai99] in the protocols we compare with by Paillier encryptions *with short randomness*. The encryption function is thus changed as follows: $r \xleftarrow{\$} \mathbb{Z}_{\lfloor\sqrt{n}\rfloor}$, $c \leftarrow (1+n)^m g^r \pmod{n^2}$; output c. (Where $g = (g')^n$ is pre-computed and part of the public key.)

[5] In practice, exponentiations modulo p are only a few percent slower than modulo n.

[6] The computer used for the benchmarks had an Intel i7 Q820 processor clocked at 1.73 GHz. We used version 5.0.2 of the GNU Multiple Precision Arithmetic Library.

	Amortized runtime per multiplication gate	with $\eta=80$
This work	$(90 \cdot \eta + 200 \cdot \text{lb}\, n)\ \text{exp}.n + (\ 66 \cdot \eta + \ 40.5 \cdot \text{lb}\, n)\ \text{exp}.n^2$	**602 ms**
2-party DN-protocol [DN03]	$(216 \cdot \eta + 130 \ \cdot \text{lb}\, n)\ \text{exp}.n^2$	862 ms
DO-protocol [DO10a]	$(2004 \cdot \eta + 151 \cdot \eta^2)\ \text{exp}.n + (\ 84 \cdot \eta + \ 88 \ \cdot \text{lb}\, n)\ \text{exp}.n^2$	2025 ms
BDOZ-protocol [BDOZ11]	$(256 \cdot \eta + 368 \ \cdot \text{lb}\, n)\ \text{exp}.n^2$	2303 ms

Table 3.1: Estimated amortized runtime per multiplication in various protocols. The numbers in the last column are for $\eta = 80$, $\text{lb}\, n = 1248$, $\text{exp}.n = 1.3\,\mu\text{s}$, and $\text{exp}.n^2 = 4.8\,\mu\text{s}$. Results for our work use the *optimized* variant of our *Multiplication* instruction. Results for the DO-protocol and the BDOZ-protocol are for circuits having a multiple of $4.8 \cdot \eta$ and η multiplication gates, respectively; the performance of these protocols degrades dramatically for smaller circuits. For the DO-protocol we used parameters $\lambda = 0.25$ and $B = 3.6 \cdot \eta$.

3.6.2 Comments about the Efficiency of Related Work

Here we comment on the performance of the DO-protocol and the BDOZ-protocol, both of which use a very different approach than our protocol and the DN-protocol.

DO-protocol. In the DO-protocol, the computational load is split between a pre-processing and an on-line phase [DO10a]. Their protocol optimizes the cost of the on-line phase, at the expense of the pre-processing phase. The crux of the pre-processing phase is to generate so-called triplets of commitments to random values (a, b, c), where $c = a \cdot b$. One triplet is required per multiplication gate. Instead of using traditional zero-knowledge proofs, they uses a technique called "cut-and-choose", where $\mathcal{P}$ generates a number of triplets without proof, and then selectively reveals a fraction of these to $\mathcal{Q}$. Afterwards, $\mathcal{P}$ and $\mathcal{Q}$ "distill" the remaining triplets—which involves interpolation with Lagrange polynomials—to obtain UC-secure triplets.

Their approach however suffers from two drawbacks: *1)* a large number of triplets have to be generated no matter what, and then used up during "distillation" to ensure security, and *2)* due to the Lagrange interpolation, the runtime of the pre-processing phase is *quadratic* in the number of triplets generated in each batch.

In our analysis in Table 3.1, we used the parameters $\lambda = 0.25$ and $B = 3.6 \cdot \eta$ as recommended. We computed the *amortized* runtime (offline + on-line) per muliplication gate when doing exactly $4.8 \cdot \eta$ multiplications (the value $4.8 \cdot \eta$ was chosen because it comes to within 1% of the minimum runtime per gate for security levels $\eta = 80$, $\eta = 96$, and $\eta = 112$). When more multiplications gates are required, the pre-processing phase should be done in batches of $4.8 \cdot \eta$. Note that no matter how many triplets are generated, the pre-processing phase is very slow—at least four minutes for $\eta = 80$.

BDOZ-protocol. Similarily to the DO-protocol, in the BDOZ-protocol the computational load is split between a heavy pre-processing phase and a very fast on-line phase (with essentially no cryptographic operations) [BDOZ11]. Like in the DO-protocol, a number of triplets are generated during the pre-processing phase. The technique used to generate them is somewhat different, and as observed by the authors is slightly slower: triplets are generated in batches of η, and a Σ-protocol (with binary challenge run on η instances simultaneously) ensures correctness.

In our analysis in Table 3.1, we determined the *amortized* runtime per multiplication gate in the pre-processing phase. The online phase was not considered, since it only consists of modular additions and multiplications.

3.7 Example of a Useful Protocol Constructed with $\mathcal{F}_{\text{abb}}$

In this section we give an example of how to use our framework to construct a UC-secure variant of the Dodis-Yampolskiy oblivious pseudorandom function [DY05] in a group of order n as originally proposed by Jarecki and Liu [JL09]. Jarecki and Liu proposed a two-party protocol for computing the following oblivious pseudorandom function (OPRF) [JL09], inspired by a similar construct by Dodis and Yampolskiy [DY05]:

$$f_y(x) = \begin{cases} g^{1/(y+x)} & \text{if } \gcd(y+x, n) = 1 \\ 1 & \text{otherwise} \end{cases}$$

where $\mathcal{P}$'s private input is x, $\mathcal{Q}$'s private input is y, $\mathcal{P}$'s output is $f_y(x)$ and $\mathcal{Q}$'s receives a bit b where $b = 0$ iff $\gcd(y + x, n) = 1$.

Their protocol is only secure against *static* corruptions and has not been proven to be UC secure, which is unfortunate since many of the applications they proposed in their paper would benefit from being able to treat the OPRF generation protocol as a black box. We remedy to this situation here by leveraging $\mathcal{F}_{\text{abb}}$ and the extensions presented in Section 3.4. The price to pay is that our construction is about 3.2 times slower (see Table 3.2).

3.7.1 Ideal Functionality

For completeness we explicitly show here the ideal functionality $\mathcal{F}_{\text{oprf}}$ which is parametrized by an abelian group $\mathbb{G}$ of order n (written multiplicatively), and a genrator g of $\mathbb{G}$.

- Preparing f_y (needs to be done once only):
  - `input:y` : Accept $\langle$`input:y`$, y\rangle$ from $\mathcal{Q}$ where $y \in \mathbb{Z}_n$: $\bar{y} \leftarrow y$; send $\langle$`input:y`$\rangle$ to $\mathcal{A}$.
  - `ready:y` : Accept $\langle$`ready:y`$\rangle$ from $\mathcal{P}$: send $\langle$`ready:y`$\rangle$ to $\mathcal{A}$.
  - `commit:y` [`input:y` $\wedge$ `ready:y`] : Accept $\langle$`commit:y`$\rangle$ from $\mathcal{A}$: send $\langle$`commit:y`$\rangle$ to $\mathcal{A}$.
  - `done:y` [`commit:y`] : Accept $\langle$`done:y`$\rangle$ from $\mathcal{A}$: send $\langle$`done:y`$\rangle$ to $\mathcal{Q}$.
  - `deliver:y` [`commit:y`] : Accept $\langle$`deliver:y`$\rangle$ from $\mathcal{A}$: send $\langle$`deliver:y`$, g^{\bar{y}}\rangle$ to $\mathcal{P}$.
- Computing $f_y(x_i)$ (can be repeated many times by using a different $\varphi \in \Lambda^\star$):
  - `input:x:`φ [`deliver:y`] : Accept $\langle$`input:x:`$\varphi, x_\varphi\rangle$ from $\mathcal{Q}$ where $x_\varphi \in \mathbb{Z}_n$: $\bar{x}_\varphi \leftarrow x_\varphi$; send $\langle$`input:x:`$\varphi\rangle$ to $\mathcal{A}$.
  - `ready:x:`φ [`done:y`] : Accept $\langle$`ready:x:`$\varphi\rangle$ from $\mathcal{P}$: send $\langle$`ready:x:`$\varphi\rangle$ to $\mathcal{A}$.
  - `lock:x:`φ [`input:x:`$\varphi \wedge$ `ready:x:`φ] : Accept $\langle$`lock:x:`$\varphi\rangle$ from $\mathcal{A}$: if $\gcd(n, \bar{y} + \bar{x}_\varphi) = 1$ then $\bar{b}_\varphi \leftarrow 0$, else $\bar{b}_\varphi \leftarrow 1$; send $\langle$`lock:x:`$\varphi\rangle$ to $\mathcal{A}$.

  - `done:x:`φ [`lock:x:`φ] : Accept $\langle$`done:x:`$\varphi\rangle$ from $\mathcal{A}$; send $\langle$`done:x:`$\varphi, \bar{b}_\varphi\rangle$ to $\mathcal{Q}$.
  - `deliver:x:`φ [`lock:x:`φ] : Accept $\langle$`deliver:x:`$\varphi\rangle$ from $\mathcal{A}$: send $\langle$`deliver:x:`φ, $f_{\bar{y}}(\bar{x}_\varphi)\rangle$ to $\mathcal{P}$.
- Dealing with corruptions:
  - `corrupt:P` : Accept special $\langle$`corrupt`$\rangle$ message from $\mathcal{P}$: send $\langle$`corrupt:P`$\rangle$ to $\mathcal{A}$.
  - `corrupt:Q` : Accept special $\langle$`corrupt`$\rangle$ message from $\mathcal{Q}$: send $\langle$`corrupt:Q`$\rangle$ to $\mathcal{A}$.
  - `expose:y` [`input:y`$\wedge$`corrupt:Q`] : Accept $\langle$`expose:y`$\rangle$ from $\mathcal{A}$: send $\langle$`expose:y`, $\bar{y}\rangle$ to $\mathcal{A}$.
  - `reset:y` [$\neg$`commit:y`$\wedge$ `corrupt:Q`] : Accept $\langle$`reset:y`$, y\rangle$ from $\mathcal{A}$: $\bar{y} \leftarrow y$; send $\langle$`reset:y`$\rangle$ to $\mathcal{A}$.
  - `expose:x:`φ [`input:x:`$\varphi \wedge$ `corrupt:P`] : Accept $\langle$`expose:x`$\rangle$ from $\mathcal{A}$: send $\langle$`expose:x`$, \bar{x}_\varphi\rangle$ to $\mathcal{A}$.
  - `reset:x:`φ [$\neg$`lock:x:`$\varphi \wedge$ `corrupt:Q`] : Accept $\langle$`reset:x:`$\varphi, x_\varphi\rangle$ from $\mathcal{A}$: $\bar{x}_\varphi \leftarrow x_\varphi$; send $\langle$`reset:x:`$\varphi\rangle$ to $\mathcal{A}$.

3.7.2 Construction

Preparing f_y (needs to be done once only):

1. $\mathcal{Q}$ inputs value y to $\mathcal{F}_{\text{abb}}$ with identifier k_0 using the *Input* instruction.
2. $\mathcal{Q}$ outputs the "public key" g^y using the *Exponentiated Output* instruction. (This step can be omitted if the value g^y is not needed. Indeed $\mathcal{Q}$ is committed to y through $\mathcal{F}_{\text{abb}}$ anyway.)

Computing $f_y(x_i)$ (can be repeated many times):

1. $\mathcal{P}$ inputs value x_i to $\mathcal{F}_{\text{abb}}$ with identifier k_{3i+1} using the *Input* instruction.
2. They compute $y + x_i$: $V[k_{3i+2}] \leftarrow V[k_0] + V[k_{3i+1}]$.
3. They invert the previous result using the protocol shown in Section 3.4.1: $V[k_{3i+3}] \leftarrow (V[k_{3i+2}])^{-1}$. (If the inversion fails, then $\mathcal{P}$ and $\mathcal{Q}$ output 1 and skip the next step—this is similar to how Jarecki et al. proceed [JL09]).
4. $\mathcal{P}$ retrieves $g^{V[k_{3i+3}]} = g^{1/(y+x_i)} = f_y(x_i)$ using *Exponentiated Output*.
5. $\mathcal{Q}$ returns 0.

3.7.3 Security

Correctness and privacy of the input follow directly from the construction of the extended $\mathcal{F}_{\text{abb}}$.

	Runtime for OPRF setup and one OPRF compute	with $\eta = 80$
This work	$(219 \cdot \eta + 290 \cdot \mathrm{lb}\, n) \cdot \mathrm{exp}.n + (74 \cdot \eta + 52.5 \cdot \mathrm{lb}\, n) \cdot \mathrm{exp}.n^2$	836 ms
Jarecki-Liu [JL09]	$(\ 45 \cdot \eta + \ \ 8 \cdot \mathrm{lb}\, n) \cdot \mathrm{exp}.n + (14 \cdot \eta + 40 \ \ \cdot \mathrm{lb}\, n) \cdot \mathrm{exp}.n^2$	263 ms

Table 3.2: Estimated runtime per OPRF computation including preparation, using the same notation as Table 3.1. Note that Jarecki and Liu's protocol [JL09] is not UC-secure, and only secure against *static* corruptions.

4

Practical 2-Server Password-Authenticated Secret Sharing

Properly protecting our digital assets still is a major challenge today. Because of their convenience, we protect access to our data almost exclusively by passwords, despite their inherent weaknesses. Indeed, not a month goes by without the announcement of another major password breach in the press. In 2013, hundreds of millions of passwords were stolen through server compromises, including massive breaches at Adobe, Evernote, LivingSocial, and Cupid Media. In August 2014, more than one billion passwords from more than 400,000 websites were reported stolen by a single crime ring. Barring some technical blunders on the part of Adobe, most of these passwords were properly salted and hashed. But even the theft of password hashes is detrimental to the security of a system. Indeed, the combination of weak human-memorizable passwords (NIST estimates sixteen-character passwords to contain only 30 bits of entropy [BDN+11]) and the blazing efficiency of brute-force dictionary attacks (currently testing up to 350 billion guesses per second on a rig of 25 GPUs [Gos12]) mean that any password of which a hash was leaked should be considered cracked.

Stronger password hash functions [PM99] only give a linear security improvement, in the sense that the required effort from the attacker increases at most with the same factor as the honest server is willing to spend on password verification. Since computing password hashes is the attacker's core business, but only a marginal activity to a respectable web server, the former probably has the better hardware and software for the job.

A much better approach to password-based authentication, first suggested by Ford and Kaliski [FK00], is to distribute the capability to test passwords over multiple servers. The idea is that no single server by itself stores enough information to allow it to test whether a password is correct and therefore to allow an attacker to mount an offline dictionary attack after having stolen the information. Rather, each server stores an information-theoretic share of the password and engages in a cryptographic protocol with the user and the other servers to test password correctness. As long as less than a certain threshold of servers are compromised, the password and the stored data remain secure.

Building on this approach, several threshold password-authenticated key exchange (TPAKE) protocols have since appeared in the literature [FK00, Jab01, MSJ02, BJKS03, DG03, SK05, KMTG05, JKK14], where, if the password is correct, the user shares a

different secret key with each of the servers after the protocol. Finally addressing the problem of protecting user data, threshold password-authenticated secret sharing (TPASS) protocols [BJSL11, CLN12, CLLN14, JKK14] combine data protection and user authentication into a single protocol. They enable the password-authenticated user to reconstruct a strong secret, which can then be used for further cryptographic purposes, e.g., decrypting encrypted data stored in the cloud. An implementation of the protocol by Brainard et al. [BJKS03] is commercially available as EMC's *RSA Distributed Credential Protection* (DCP) [EMC].

Unfortunately, none of the protocols proposed to date provide a satisfying level of security. Indeed, for protocols that are meant to resist server compromise, the research papers are surprisingly silent about what needs to be done when a server actually gets corrupted and how to recover from such an event. The work by Di Raimondo and Gennaro [DG03] is the only one to mention the possibility to extend their protocol to provide proactive security by refreshing the shares between time periods; unfortunately, no details are provided. The RSA DCP product description [EMC] mentions a re-randomization feature that "can happen proactively on an automatic schedule or reactively, making information taken from one server useless in the event of a detected breach." This feature is however not described in any of the underlying research papers [BJKS03, SK05], and neither is a security proof known. Taking only protocols with provable security guarantees into account, the existing ones can protect against servers that are malicious from the beginning, but do not offer any guarantees against adaptive corruptions. The latter is a much more realistic setting, modelling for instance servers getting compromised by malicious hackers. This state of affairs is rather troubling, given that the main threats to password security today, and arguably, the whole *raison d'être* of TPAKE/TPASS schemes, come from the latter type of attacks.

One would hope to be able to strengthen existing protocols with ideas from proactive secret sharing [HJKY95] to obtain security against adaptive corruptions, but this task is not straightforward and so far neither the resulting protocol details nor the envisaged security properties have ever been spelled out. Indeed, designing cryptographic protocols secure against adaptive corruptions is much more difficult than against static corruptions. One difficulty thereby is that in the security proof the simulator must generate network traffic for honest parties *without* knowing their inputs, but, once the party is corrupted, must be able to produce realistic state information that is consistent with the now revealed actual inputs as well as the previously simulated network traffic. Generic multiparty computation protocols secure against adaptive corruption can be applied, but these are too inefficient. In fact, evaluating a single multiplication gate in the most efficient two-party computation protocol secure against adaptive corruptions [CES13] (see also Chapter 3) is more than three times slower than a full execution of the dedicated protocol we present here. We note that our protocol is well within reach of a practical implementation: users and servers have to perform a few hundred exponentiations each, which translates to an overall computation time of less than 0.1 seconds per party.

We prove our protocol secure in the universal composability (UC) framework [Can00, Can01]. The very relevant advantages of composable security notions for the particular case of password-based protocols have been argued before [CHK+05b, CLN12]; we briefly summarize them here. In composable notions, the passwords for honest users, as well as their password attempts, are provided by the environment. Passwords and password

attempts can therefore be distributed arbitrarily and even dependently, reflecting real users who may choose the same or similar passwords for different accounts. It also correctly models typos made by honest users when entering their passwords: all property-based notions in the literature limit the adversary to seeing transcripts of honest users authenticating with their correct password, so in principle security breaks down as soon as a user mistypes the password. Finally, composable definitions absorb the inherent polynomial success probability of the adversary into the functionality. Thus, security is retained when the protocol is composed with other protocols, in particular, protocols that use the stored secret as a key. In contrast, composition of property-based notions with non-negligible success probabilities is problematic because the adversary's advantage may be inflated. Also, strictly speaking, the security provided by property-based notions is guaranteed only if a protocol is used in isolation.

Roadmap. In Section 4.1 we recall some additional concepts related to adaptive and transient corruptions in the UC framework. In Section 4.2 we describe our ideal functionality $\mathcal{F}_{\text{2pass}}$ for 2-server password authenticated secret sharing. In Section 4.3 we construct a concrete protocol Π_{2pass} realizing $\mathcal{F}_{\text{2pass}}$, and compare it with related work. Finally, in Section 4.4 we prove that our protocol is secure.

4.1 Corruption in the UC Model

In the following we discuss the modelling of transient corruptions [Can00] in the UC framework, how one can use ideal functionalities designed for adaptive corruptions in a protocol designed for transient corruptions, and finally we discuss a particular problem that appears in protocols secure against adaptive or transient corruptions: the selective decommitment problem.

Modelling transient corruptions in real/hybrid protocols. We now recall how corruption and recovery is modelled in real/hybrid protocols.

Corruption of a party. See Section 2.5.1.2.

Recovery from corruption. $\mathcal{A}$ may cede control from a corrupted party. When doing that, $\mathcal{A}$ may specify a new internal state for the party. We then say that the party formally recovered. In real life, a party might know it recovered if it detected a breach and has restored from backup.

In most protocols however, formal recovery is not enough: the adversary still knows parts of the internal state of the formally recovered party. To allow the party to effectively recover its security, it must take additional steps, e.g., notify its subroutines (and stop using the subroutines that cannot handle recovery) and run a protocol-specific *Refresh* instruction. The party might thereby drop all currently running queries.

A party initiates a Refresh query to modify its internal state so that firstly it is synchronized with the other protocol participants, and so that secondly $\mathcal{A}$'s knowledge of the old state does not interfere with security of the new state. Parties should initiate a Refresh query when they formally recover from corruption. (If parties cannot detect formal recovery, they should run Refresh periodically.) The Refresh query might fail if the state of the party is inconsistent with that of the others. The party might also not

necessarily recover its security even after succesful completion of the query, e.g., because all other participants are corrupted. Note that the security of a party is fully restored (if at all) only after Refresh completes: in the grey zone bewteen formal recovery and completion of Refresh, the party must not run any queries other than Refresh.

Using ideal functionalities designed for the adaptive type in a transient-secure hybrid protocol. Protocols secure against transient corruptions may use ideal functionalities as subroutines that were designed to handle adaptive corruptions, e.g., $\mathcal{F}_{\text{ac}}$, $\mathcal{F}_{\text{osac}}$, $\mathcal{F}_{\text{gzk}}$, and $\mathcal{F}_{\text{gzk}}^{2\text{v}}$: upon formal recovery, the party must stop using all instances of these ideal functionalities. Thereby, it has to abort all currently running queries. Thereafter, it has to use fresh instances of these ideal functionalities for running the Refresh query, and all subsequent queries.

The selective decommitment problem. Hofheinz demonstrated that it is impossible to prove any protocol secure against adaptive corruptions (and thus, against transient corruptions) that uses perfectly binding commitments or (binding) encryptions to commit to or to encrypt the parties' input, respectively [Hof11]. Let us expand on this. For example, assume that in a protocol a user $\mathcal{U}$ with an input i must send out a binding commitment c or an encryption e depending on i, e.g., $(c, o) = \mathsf{com}(i)$ or $(e, er) = \mathsf{enc}(pk, i, l)$. The simulator $\mathcal{S}$ in the security proof must be able to simulate the honest $\mathcal{U}$ without knowing her input i, i.e., $\mathcal{S}$ must send c or e to the adversary $\mathcal{A}$, containing some value that is most likely different from i. If $\mathcal{U}$ then gets corrupted, $\mathcal{S}$ must produce an internal state for $\mathcal{U}$, namely the opening o or the randomness er used to encrypt and—if applicable—the secret key sk, that is consistent with both her real input i and the values c or e already sent out to the adversary. However, due to the binding nature of the commitment and encryption, and unless it could predict i, $\mathcal{S}$ cannot find an internal state for $\mathcal{U}$ consistent with these values and therefore the security proof will not go through.

4.2 Our Ideal Functionality $\mathcal{F}_{\text{2pass}}$

We now describe our ideal functionality $\mathcal{F}_{\text{2pass}}$ for two-server password-authenticated secret sharing, secure against transient corruptions. We start with a high level description, and then provide the formal definition of $\mathcal{F}_{\text{2pass}}$ in the UC framework [Can00]. It is not necessary to read Section 4.2.2 to understand the construction of our scheme. $\mathcal{F}_{\text{2pass}}$ is reminiscent of similar functionalities by Camenisch et al. [CLN12, CLLN14], the main differences being our modifications to handle transient corruptions. We compare the ideal functionalities in Section 4.3.5.1.

4.2.1 Informal Definition of $\mathcal{F}_{\text{2pass}}$

The functionality $\mathcal{F}_{\text{2pass}}$ involves two servers, $\mathcal{P}$ and $\mathcal{Q}$, and a plurality of users. We chose to define $\mathcal{F}_{\text{2pass}}$ for a single user account, specified by the session ID sid. Multiple accounts can be realized by multiple instances of $\mathcal{F}_{\text{2pass}}$ or with a multi-session realization of $\mathcal{F}_{\text{2pass}}$. The session identifier sid consists of $(pid_{\mathcal{P}}, pid_{\mathcal{Q}}, (\mathbb{G}, q, g), uacc, ssid)$, i.e., the identity of the two servers, the description of a group of prime order q with generator

$\mathcal{F}_{\text{2pass}}$ processes the instructions as follows. $\mathcal{F}_{\text{2pass}}$ accepts inputs and messages only for a specific *sid*. It further checks that the *sid* has the correct format. Whenever $\mathcal{F}_{\text{2pass}}$ receives an input from a party it will eventually send a message to $\mathcal{A}$ containing the identity of the party, the type of input, *sid*, *qid*, and—if applicable—sends out delayed messages. [a]

Setup: The user inputs $\langle \text{Setup}, sid, qid = \text{“Setup”}, p, k\rangle$ to $\mathcal{F}_{\text{2pass}}$ and the two servers each input $\langle \text{ReadySetup}, sid, qid = \text{“Setup”}\rangle$ to $\mathcal{F}_{\text{2pass}}$. $\mathcal{F}_{\text{2pass}}$ then sends a public delayed message $\langle \text{Done}, sid, qid\rangle$ to the user and each of the two servers.

Retrieve: To start, the user inputs $\langle \text{Retrieve}, sid, qid, a\rangle$ to $\mathcal{F}_{\text{2pass}}$, and the two servers each input $\langle \text{ReadyRetrieve}, sid, qid\rangle$ to $\mathcal{F}_{\text{2pass}}$. $\mathcal{F}_{\text{2pass}}$ waits for a message $\langle \text{Lock}, sid, qid\rangle$ from $\mathcal{A}$, and then replies whether the user's password attempt was correct by sending $\langle \text{Lock}, sid, qid, b\rangle$ to $\mathcal{A}$—where $b = 1$ if $a = p$ and $b = 0$ otherwise. $\mathcal{F}_{\text{2pass}}$ then sends a public delayed message $\langle \text{Delivered}, sid, qid, b\rangle$ to the two servers, and a private delayed message $\langle \text{Deliver}, sid, qid, k'\rangle$ to the user, where $k' = k$ if $a = p$, and $k' = \varepsilon$ otherwise.

Corrupt: $\mathcal{A}$ can corrupt a party $\mathcal{R}$ by sending $\langle \text{Corrupt}, sid, \mathcal{R}\rangle$ to $\mathcal{F}_{\text{2pass}}$. Recall that $\mathcal{A}$ thereafter obtains control of the corrupted party's input to and output from $\mathcal{F}_{\text{2pass}}$. $\mathcal{A}$ may prevent a subsequent Refresh query from succeeding in case the server later recovers from corruption—in a real protocol, $\mathcal{A}$ may tamper with the server's internal state. If both servers are corrupted at the same time (or corrupted in sequence with no Refresh query in between), $\mathcal{F}_{\text{2pass}}$ will send (k, p) to $\mathcal{A}$ and allow $\mathcal{A}$ to provide arbitrary replacement values. That is, $\mathcal{A}$ can force $\mathcal{F}_{\text{2pass}}$ to return arbitrary values to the user if the latter interacts with two corrupted servers in a Retrieve query.

Recover: A party $\mathcal{R}$ recovers from corruption if $\mathcal{A}$ sends $\langle \text{Recover}, sid, \mathcal{R}\rangle$ to $\mathcal{F}_{\text{2pass}}$. $\mathcal{F}_{\text{2pass}}$ then stops accepting input and messages for all currently running Setup and Retrieve queries, and will not accept any further Setup and Retrieve queries until a Refresh query suceeds.

Refresh: To start a Refresh query, each server inputs $\langle \text{Refresh}, sid, qid\rangle$ to $\mathcal{F}_{\text{2pass}}$. While this query is in progress, no further Setup, Retrieve, and Refresh queries are accepted, and currently running queries are dropped. Once it has received a message from both servers, $\mathcal{F}_{\text{2pass}}$ sends $\langle \text{RefreshDone}, sid, qid\rangle$ as public delayed messages to the two servers. $\mathcal{F}_{\text{2pass}}$ then resumes accepting new queries. Note that while a server was corrupted, $\mathcal{A}$ might have prevented it from completing this Refresh query.

Hijack: Just after a user provided its first input to $\mathcal{F}_{\text{2pass}}$ in a Setup or Retrieve query and before $\mathcal{A}$ sends anything to $\mathcal{F}_{\text{2pass}}$ for the same query, $\mathcal{A}$ has the option of stealing the ID of the query by sending a $\langle \text{HijackSetup}, sid, qid, p, k\rangle$ or $\langle \text{HijackRetrieve}, sid, qid, a\rangle$ message, respectively, to $\mathcal{F}_{\text{2pass}}$. In that case, $\mathcal{F}_{\text{2pass}}$ ignores the user's first message and runs the query with $\mathcal{A}$ instead of the user, with the *qid* chosen by the user but input—(p, k) or a—provided by $\mathcal{A}$.

[a] Messages from an ideal functionality to a party are direct outputs, unless they are specified to be delayed outputs. In the latter case, $\mathcal{F}_{\text{2pass}}$ notifies $\mathcal{A}$ it wishes to send the message and waits for a confirmation by $\mathcal{A}$ before actually sending out the message. A public delayed output means that $\mathcal{A}$ learns the message; a private message means that $\mathcal{A}$ will learn only the type of the message and the recipient.

Fig. 4.1: High-level definition of $\mathcal{F}_{\text{2pass}}$. See the text for explanations, and see Section 4.2.2 for the full formalization.

g, the name of the user account *uacc* (any string), and an arbitrary suffix *ssid*. Only the parties with identities $pid_{\mathcal{P}}$ and $pid_{\mathcal{Q}}$ can provide input in the role of $\mathcal{P}$ and $\mathcal{Q}$, respectively, to $\mathcal{F}_{\text{2pass}}$. When starting a fresh query, any party can provide input in the role of a user to $\mathcal{F}_{\text{2pass}}$; for subsequent inputs in that query, $\mathcal{F}_{\text{2pass}}$ ensures it comes from the same party; additionally, $\mathcal{F}_{\text{2pass}}$ does not disclose the identity of the user to the servers.

$\mathcal{F}_{\text{2pass}}[sid]$ reacts to a set of instructions, each requiring the parties to send multiple inputs to $\mathcal{F}_{\text{2pass}}$ in a specific order. The main instructions are Setup, Retrieve, and Refresh. Additionally $\mathcal{F}_{\text{2pass}}$ reacts to instructions modelling dishonest behavior, namely Corrupt, Recover, and Hijack. $\mathcal{F}_{\text{2pass}}$ may process multiple queries concurrently. A query identifier *qid* is used to distinguish between separate executions of the main instructions. We now provide a summary of the instructions. We refer to Figure 4.1 for a high-level definition of $\mathcal{F}_{\text{2pass}}$ and to Section 4.2.2 for the full formalization.

With the *Setup* instruction, a user sets up the user account by submitting a key k and a password p to $\mathcal{F}_{\text{2pass}}$ for storage, protected under the password. This instruction can be run only once, which we enforce by fixing *qid* to "Setup". With the *Retrieve* instruction, any user can then retrieve that k provided her submitted password attempt a is correct, i.e., $a = p$, and the servers are willing to participate in this query. Giving the server the choice to refuse to participate in a query is important to counter online password guessing attacks. $\mathcal{F}_{\text{2pass}}$ allows for the adaptive corruption of users and servers with the *Corrupt* instruction, and for recovery from corruption of servers at any time with the *Recover* instruction. Servers should run the *Refresh* instruction whenever they recover from corruption or at regular intervals; in the real protocol, the two servers re-randomize their state in this instruction and thereby clear the residual knowledge $\mathcal{A}$ might have. If both servers are corrupted at the same time or sequentially with no Refresh in between, the adversary $\mathcal{A}$ will learn the current key and password (k, p) and is allowed to set them to different values. Finally, recall that in our realization of $\mathcal{F}_{\text{2pass}}$, the first message from the user to the servers is not authenticated. $\mathcal{A}$ can therefore learn the *qid* from that message, drop the message, and send his own message to the servers with that *qid*. We model this attack in $\mathcal{F}_{\text{2pass}}$ with the *Hijack* instruction. Servers will not notice this attack, but the user will conclude his query failed.

Our $\mathcal{F}_{\text{2pass}}$ functionality gives the following security guarantees: k and p are protected from $\mathcal{A}$ as long as at least one server is honest and no corrupt user is able to correctly guess the password. Furthermore, if at least one server is honest, no offline password guessing attacks are possible. Honest servers can limit online guessing attacks by limiting Retrieve queries after too many failed attempts. Finally, an honest user's password attempt a remains hidden even if a Retrieve query is directed at two corrupt servers.

4.2.2 Formal Definition of $\mathcal{F}_{\text{2pass}}$

We now provide a full definition of our $\mathcal{F}_{\text{2pass}}$ ideal functionality. As we model the single-session variant of $\mathcal{F}_{\text{2pass}}$, the session ID $\underline{sid}$ is fixed. Recall that *sid* comprises the identity of the two servers $pid_{\mathcal{P}}$ and $pid_{\mathcal{Q}}$, the description of a group $(\mathbb{G}, q, g)$ of prime order q, the name of the user account *uacc*, and a suffix *ssid*. Each instance of $\mathcal{F}_{\text{2pass}}$ checks that $\underline{sid}$ is of the correct format when first invoked. Also, only messages with the correct $sid = \underline{sid}$ are accepted.

Interfaces. $\mathcal{F}_{\text{2pass}}$ is a four-interface system:

- The network interface, connected to the ideal adversary/simulator.
- The $\mathcal{U}$-interface, connected to the ideal peers of the users. This interface is multiplexed; we assume that headers are added to all messages to enable proper routing.
- The $\mathcal{P}$-interface, connected to the ideal peer of the first server $\mathcal{P}$.
- The $\mathcal{Q}$-interface, connected to the ideal peer of the second server $\mathcal{Q}$.

State. The ideal functionality is stateful and maintains the following data structures. We underline these datastructures to distinguish them from local variables.

- $\underline{Seen}$: associative array between $\{0,1\}^*$ and a subset of $\{0,1\}^*$. Keeps track of which messages were accepted per qid.
- $\underline{Corrupted}$: a set of party ids (servers and users). Keeps track of who is currently corrupted.
- $\underline{SetupDone}$: subset of $\{pid_{\mathcal{P}}, pid_{\mathcal{Q}}\}$. Keeps track of which server successfully completed the Setup query.
- $\underline{JustRecovered}$: a subset of $\{pid_{\mathcal{P}}, pid_{\mathcal{Q}}\}$. Keeps track of which servers have formally recovered from corruption, but not yet completed the subsequent Refresh query.
- $\underline{Refreshing}$: a subset of $\{pid_{\mathcal{P}}, pid_{\mathcal{Q}}\}$. Keeps track of which servers have started doing a Refresh query.
- $\underline{RefreshPeriod}^{\mathcal{P}}$ and $\underline{RefreshPeriod}^{\mathcal{Q}}$: integers. Keep track of the refresh period (the time between two Refresh queries) of each server.
- $\underline{QidPeriod}^{\mathcal{P}}$ and $\underline{QidPeriod}^{\mathcal{Q}}$: associative arrays between $\{0,1\}^*$ and an integer. Keep track of which qids belong to which refresh period.
- $\underline{CorruptedIn}$: associative array between an integer and a subset of $\{pid_{\mathcal{P}}, pid_{\mathcal{Q}}\}$. Keeps track of which servers where corrupted in a given refresh period.
- $\underline{p}^{\mathcal{P}}$ and $\underline{p}^{\mathcal{Q}}$: elements of $\mathbb{Z}_q \cup \{\perp, \triangle\}$. The password stored by the user, in the view of the given server. (Normally both values are equal. The values are unequal in case a server recovered from corruption in an altered state. All queries including the Refresh query will be blocked if the values are unequal. The symbol $\triangle$ is represents the fact that the simulator does not yet know the value of the stored password, however it must provide a concrete value before the end of the Refresh query.)
- $\underline{k}^{\mathcal{P}}$ and $\underline{k}^{\mathcal{Q}}$: elements of $\mathbb{Z}_q \cup \{\perp, \triangle\}$. The key stored by the user, in the view of the given server. (Same comments as for $\underline{p}^{\mathcal{P}}$ and $\underline{p}^{\mathcal{Q}}$.)
- $\underline{a}$: associative array between $\{0,1\}^*$ and an element of $\mathbb{Z}_q \cup \{\perp\}$. The password attempts per qid.
- $\underline{k}'$: associative array between $\{0,1\}^*$ and an element of $\mathbb{Z}_q \cup \{\perp, \varepsilon\}$. The key to be returned to the user per qid, where ε means the user input an incorrect password attempt.
- $\underline{U}$: associative array between $\{0,1\}^*$ and a user in the system. Keeps track of which user initiated a query. If $\underline{U}[qid] = \mathcal{A}$, $\mathcal{F}_{\text{osac}}$ sends/receives messages on the network interface instead of the $\mathcal{U}$-interface as written for the query qid.

The default value of these are as follows: associative arrays are initially empty. If no value is associated to a given key in an array, then $\perp$ is returned. Sets are initially empty. Integers are initially 0. All other values are initially $\perp$.

Aborting all queries during a refresh. In $\mathcal{F}_{\text{2pass}}$, we enforce that once an honest server $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ starts a Refresh query, all other queries are aborted. To simplify the exposition, we define the following predicates: let AcceptNewQid($qid, \mathcal{R}$) denote the predicate

$$pid_{\mathcal{R}} \in \underline{Corrupted} \vee (pid_{\mathcal{R}} \notin \underline{Refreshing} \wedge pid_{\mathcal{R}} \notin \underline{JustRecovered} \wedge \underline{QidPeriod}^{\mathcal{R}}[qid] = \perp)$$

and let AcceptQid($qid, \mathcal{R}$) denote the predicate

$$pid_{\mathcal{R}} \in \underline{Corrupted} \vee (pid_{\mathcal{R}} \notin \underline{Refreshing} \wedge pid_{\mathcal{R}} \notin \underline{JustRecovered} \wedge \underline{QidPeriod}^{\mathcal{R}}[qid] = \underline{RefreshPeriod}^{\mathcal{R}}).$$

The first predicate checks that qid has not yet been seen by a server $\mathcal{R}$. The second predicate checks that qid has already been seen, and that there was no Refresh query between now and the moment qid was first seen. Both predicates also check that $\mathcal{R}$ is not currently blocking new queries due to recently formally recovering from corruption or due to currently running a Refresh query. Both predicates can be overriden by a corrupt $\mathcal{R}$.

Reacting to Messages. Our $\mathcal{F}_{\text{2pass}}$ reacts to messages as follows.

Setup instruction. Recall that the Setup instruction allows a user to store her password and key in the ideal functionality. The user starts by privately inputing her key and password to $\mathcal{F}_{\text{2pass}}$ (Message 1). The servers have to state that they are ready to execute a Setup query by notifying $\mathcal{F}_{\text{2pass}}$ (Message 2). (Messages 1 and 2 can happen in any order.) After these two steps, $\mathcal{F}_{\text{2pass}}$ sends a public delayed acknowledgement to the servers (Message 3), and finally sends a public delayed acknowledgement to the user (Message 4).

1. Receive $\langle \text{Setup}, \underline{sid}, qid, p, k \rangle$ on $\mathcal{U}$ (from a user $pid_{\mathcal{U}}$),
   where $qid =$ "Setup", $p \in \mathbb{Z}_q$, and $k \in \mathbb{Z}_q$,
   such that $\{$"Setup", "Retrieve"$\} \cap \underline{Seen}[qid] = \emptyset$:

   Insert "Setup" into $\underline{Seen}[qid]$.
   Record the user: $\underline{U}[qid] \leftarrow pid_{\mathcal{U}}$. Save the user's input: $\underline{p}^{\mathcal{P}} \leftarrow p$, $\underline{p}^{\mathcal{Q}} \leftarrow p$, $\underline{k}^{\mathcal{P}} \leftarrow k$, and $\underline{k}^{\mathcal{Q}} \leftarrow k$.
   Send $\langle \text{Setup}, \underline{sid}, qid, \mathcal{U} \rangle$ on network.

2. Receive $\langle \text{ReadySetup:}\mathcal{R}, \underline{sid}, qid \rangle$ on $\mathcal{R}$,
   where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$, $qid =$ "Setup", and AcceptNewQid($qid, \mathcal{R}$) ,
   such that $\{$"ReadySetup:$\mathcal{R}$", "ReadyRetrieve:$\mathcal{R}$", "Refresh:$\mathcal{R}$"$\} \cap \underline{Seen}[qid] = \emptyset$:

Insert "ReadySetup:$\mathcal{R}$" into $\underline{Seen}[qid]$.
Set $\underline{QidPeriod}^{\mathcal{R}}[qid] \leftarrow \underline{RefreshPeriod}^{\mathcal{R}}$.
Send $\langle$ReadySetup:$\mathcal{R}$, $\underline{sid}$, $qid\rangle$ on network.

3. Receive $\langle$Done:$\mathcal{R}$, $\underline{sid}$, $qid\rangle$ on network,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ and AcceptQid$(qid, \mathcal{R})$,
such that $\{$"Done:$\mathcal{R}$"$\} \cap \underline{Seen}[qid] = \emptyset$,
and $\{$"Setup", "ReadySetup:$\mathcal{P}$", "ReadySetup:$\mathcal{Q}$"$\} \subset \underline{Seen}[qid]$:

Insert "Done:$\mathcal{R}$" into $\underline{Seen}[qid]$.
Insert $pid_{\mathcal{R}}$into $\underline{SetupDone}$.
Send $\langle$Done:$\mathcal{R}$, $\underline{sid}$, $qid\rangle$ on $\mathcal{R}$.

4. Receive $\langle$Done, $\underline{sid}$, $qid\rangle$ on network,
such that $\{$"Done"$\} \cap \underline{Seen}[qid] = \emptyset$
and $\{$"Done:$\mathcal{P}$", "Done:$\mathcal{Q}$"$\} \subset \underline{Seen}[qid]$:

Insert "Done" into $\underline{Seen}[qid]$.
Send $\langle$Done, $\underline{sid}$, $qid\rangle$ on $\mathcal{U}$ (to user $\underline{U}[qid]$).

Retrieve instruction. Recall that the Retrieve instruction allows a user to recover the key stored in $\mathcal{F}_{\text{2pass}}$ provided she knows the correct password. The user starts by privately inputing her password attempt to $\mathcal{F}_{\text{2pass}}$ (Message 5). The servers have to state that they are ready to execute a Retrieve query and willing to service the user's query by notifying $\mathcal{F}_{\text{2pass}}$ (Message 6). (Messages 5 and 6 can happen in any order.) The adversary is the first to learn of the result of the password check: by sending a lock message to $\mathcal{F}_{\text{2pass}}$, the latter tells the former the result of that check (Message 7). After these two steps, $\mathcal{F}_{\text{2pass}}$ sends a public delayed message to the servers with the result of the check (Message 8), and finally sends a public delayed message to the user containing the key or an empty message in case the password was wrong (Message 9).

5. Receive $\langle$Retrieve, $\underline{sid}$, $qid, a\rangle$ on $\mathcal{U}$ (from user $pid_{\mathcal{U}}$),
where $a \in \mathbb{Z}_q$,
such that $\{$"Retrieve", "Setup"$\} \cap \underline{Seen}[qid] = \emptyset$:

Insert "Retrieve" into $\underline{Seen}[qid]$.
Record the user: $\underline{U}[qid] \leftarrow \mathcal{U}$. Save the user's input: $\underline{a}[qid] \leftarrow a$.
Send $\langle$Retrieve, $\underline{sid}$, qid, $\mathcal{U}\rangle$ on network.

6. Receive $\langle$ReadyRetrieve:$\mathcal{R}$, $\underline{sid}$, $qid\rangle$ on $\mathcal{R}$,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$, AcceptNewQid$(qid, \mathcal{R})$, and $pid_{\mathcal{R}} \in (\underline{SetupDone} \cup \underline{Corrupted})$,
such that $\{$"ReadyRetrieve:$\mathcal{R}$", "ReadySetup:$\mathcal{R}$", "Refresh:$\mathcal{R}$"$\} \cap \underline{Seen}[qid] = \emptyset$:

Insert "ReadyRetrieve:$\mathcal{R}$" into $\underline{Seen}[qid]$.
Set $\underline{QidPeriod}^{\mathcal{R}}[qid] \leftarrow \underline{RefreshPeriod}^{\mathcal{R}}$.
Send $\langle$ReadyRetrieve:$\mathcal{R}$, $\underline{sid}$, $qid\rangle$ on network.

7. Receive $\langle \text{Lock}, \underline{sid}, qid \rangle$ on network,
where $\underline{p}^{\mathcal{P}} = \underline{p}^{\mathcal{Q}}$, $\underline{k}^{\mathcal{P}} = \underline{k}^{\mathcal{Q}}$, $\underline{p}^{\mathcal{P}} \neq \triangle$, and $\underline{k}^{\mathcal{P}} \neq \triangle$,
such that $\{\text{“Lock”}\} \cap \underline{Seen}[qid] = \emptyset$,
and $\{\text{“Retrieve”}, \text{“ReadyRetrieve:}\mathcal{P}\text{”}, \text{“ReadyRetrieve:}\mathcal{Q}\text{”}\} \subset \underline{Seen}[qid]$:

   Insert “Lock” into $\underline{Seen}[qid]$.
   If $\underline{a}[qid] = \underline{p}^{\mathcal{P}}$, then set $b \leftarrow 1$ and $\underline{k}'[qid] \leftarrow \underline{k}^{\mathcal{P}}$; else set $b \leftarrow 0$ and $\underline{k}'[qid] \leftarrow \varepsilon$.
   Send $\langle \text{Lock}, \underline{sid}, qid, b \rangle$ on network.

8. Receive $\langle \text{Delivered:}\mathcal{R}, \underline{sid}, qid \rangle$ on network,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ and $\text{AcceptQid}(qid, \mathcal{R})$,
such that $\{\text{“Delivered:}\mathcal{R}\text{”}\} \cap \underline{Seen}[qid] = \emptyset$,
and $\{\text{“Lock”}\} \subset \underline{Seen}[qid]$:

   Insert “Delivered:$\mathcal{R}$” into $\underline{Seen}[qid]$.
   If $\underline{k}'[qid] \neq \varepsilon$, then set $b \leftarrow 1$; else set $b \leftarrow 0$.
   Send $\langle \text{Delivered:}\mathcal{R}, \underline{sid}, qid, b \rangle$ on $\mathcal{R}$.

9. Receive $\langle \text{Deliver}, \underline{sid}, qid \rangle$ on network,
such that $\{\text{“Deliver”}\} \cap \underline{Seen}[qid] = \emptyset$
and $\{\text{“Delivered:}\mathcal{P}\text{”}, \text{“Delivered:}\mathcal{Q}\text{”}\} \subset \underline{Seen}[qid]$:

   Insert “Deliver” into $\underline{Seen}[qid]$.
   Send $\langle \text{Deliver}, \underline{sid}, qid, \underline{k}'[qid] \rangle$ on $\mathcal{U}$ (to user $\underline{U}[qid]$).

Refresh instruction. Recall that the Refresh instruction allows the servers to jointly re-randomize their internal states and thereby clear the residual knowledge that $\mathcal{A}$ might have. The servers have to state that they are ready to execute a Refresh query by publicly notifying $\mathcal{F}_{\text{2pass}}$ (Message 10). Afterwards, $\mathcal{F}_{\text{2pass}}$ sends a public delayed acknowledgement to the servers (Message 11). We note that while Refresh is active, $\mathcal{F}_{\text{2pass}}$ accepts no other queries and drops all incomplete queries.

10. Receive $\langle \text{Refresh:}\mathcal{R}, \underline{sid}, qid \rangle$ on $\mathcal{R}$,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$, $pid_{\mathcal{R}} \in (\underline{SetupDone} \cup \underline{Corrupted})$, and $pid_{\mathcal{R}} \notin \underline{Refreshing}$,
such that $\{\text{“Refresh:}\mathcal{R}\text{”}, \text{“ReadySetup:}\mathcal{R}\text{”}, \text{“ReadyRetrieve:}\mathcal{R}\text{”}\} \cap \underline{Seen}[qid] = \emptyset$:

   Insert “Refresh:$\mathcal{R}$” into $\underline{Seen}[qid]$.
   Insert $pid_{\mathcal{R}}$into $\underline{Refreshing}$.
   If $\underline{Corrupted} \neq \emptyset$, then
      $\underline{CorruptedIn}[\underline{RefreshPeriod}^{\mathcal{R}} + 1] \leftarrow \underline{CorruptedIn}[\underline{RefreshPeriod}^{\mathcal{R}}]$.
   Send $\langle \text{Refresh:}\mathcal{R}, \underline{sid}, qid \rangle$ on network.

11. Receive $\langle \text{RefreshDone:}\mathcal{R}, \underline{sid}, qid \rangle$ on network,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$, $\underline{p}^{\mathcal{P}} = \underline{p}^{\mathcal{Q}}$, $\underline{k}^{\mathcal{P}} = \underline{k}^{\mathcal{Q}}$, $\underline{p}^{\mathcal{P}} \neq \triangle$, and $\underline{k}^{\mathcal{Q}} \neq \triangle$,
such that $\{\text{“RefreshDone:}\mathcal{R}\text{”}\} \cap \underline{Seen}[qid] = \emptyset$,
and $\{\text{“Refresh:}\mathcal{P}\text{”}, \text{“Refresh:}\mathcal{Q}\text{”}\} \subset \underline{Seen}[qid]$:

Insert "RefreshDone:$\mathcal{R}$" into $\underline{Seen}[qid]$.
Increment $\underline{RefreshPeriod}^{\mathcal{R}}$ by 1.
Remove $pid_{\mathcal{R}}$ from $\underline{Refreshing}$ and $\underline{JustRecovered}$.
Send $\langle$RefreshDone:$\mathcal{R}, \underline{sid}, qid\rangle$ on $\mathcal{R}$.

Corruption. We now present all instructions that have to do with corruption, hijacking, and recovery from corruption. Servers (Message 12) and users (Message 13) can be corrupted if $\mathcal{F}_{\text{2pass}}$ receives a special corrupt message from the adversary. In our protocol, the first message of the user can be hijacked by $\mathcal{A}$; in $\mathcal{F}_{\text{2pass}}$ this is modelled by allowing $\mathcal{A}$ to take over the query and the user doesn't continue with the query (Messages 14 and 15). $\mathcal{F}_{\text{2pass}}$ allows $\mathcal{A}$ the following behaviour. If the user is corrupt, $\mathcal{A}$ can recover her input (Messages 16 and 17). If both servers were corrupt in the same refresh period (between Refresh queries), then the user's password and key are exposed (Message 18). If both servers are corrupt at the same time, then $\mathcal{A}$ may make $\mathcal{F}_{\text{2pass}}$ return whatever it wants during Retrieve (Message 19). If a server is corrupt, $\mathcal{A}$ can modify its internal state (Message 20)—note that unless that state is consistent across both servers, none of the queries will work—, here we note that $\mathcal{A}$ may set the state to a special symbol $\triangle$ instead of providing a value directly. If a server's state is $\triangle$, $\mathcal{A}$ may set the real state at a later point in time (Message 21). This models the fact that $\mathcal{S}$ may not know the value of the saved password or key if $\mathcal{A}$ sets the servers to an inconsistent state. Note that $\mathcal{F}_{\text{2pass}}$ will not complete any queries while in that state. Finally, the environment may uncorrupt servers by sending a special Recover message (Message 22). We note that $\mathcal{A}$ has had the chance to specify the internal state of the recovered server before Recover is called with the previous messages.

12. Receive $\langle$Corrupt$, \underline{sid}, \mathcal{R}\rangle$ on network,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ and $pid_{\mathcal{R}} \notin \underline{Corrupted}$:

Insert $pid_{\mathcal{R}}$into $\underline{Corrupted}$ and into $\underline{CorruptedIn}[\underline{RefreshPeriod}^{\mathcal{R}}]$.
If $pid_{\mathcal{R}} \in \underline{Refreshing}$, then
$\underline{CorruptedIn}[\underline{RefreshPeriod}^{\mathcal{R}} + 1] \leftarrow \underline{CorruptedIn}[\underline{RefreshPeriod}^{\mathcal{R}}]$.
Send $\langle$Corrupt$, \underline{sid}\rangle$ on $\mathcal{R}$.

13. Receive $\langle$CorruptUser$, \underline{sid}, pid_{\mathcal{U}}\rangle$ on network,
where $\mathcal{U} \notin \underline{Corrupted}$:

Insert $pid_{\mathcal{U}}$into $\underline{Corrupted}$.
Send $\langle$CorruptUser$, \underline{sid}\rangle$ on $\mathcal{U}$ (to user $pid_{\mathcal{U}}$).

To simplify the presentation, we do not allow users to recover from corruption.

14. Receive $\langle$HijackSetup$, \underline{sid}, qid, p, k\rangle$ on network,
such that {"HijackSetup", "Done:$\mathcal{P}$", "Done:$\mathcal{Q}$"} $\cap$ $\underline{Seen}[qid] = \emptyset$
and {"Setup"} $\subset$ $\underline{Seen}[qid]$:

Insert "HijackSetup" into $\underline{Seen}[qid]$.
Change $\underline{U}[qid] \leftarrow \mathcal{A}$. Change the input: $\underline{p}^{\mathcal{P}} \leftarrow p$, $\underline{p}^{\mathcal{Q}} \leftarrow p$, $\underline{k}^{\mathcal{P}} \leftarrow k$, and $\underline{k}^{\mathcal{Q}} \leftarrow k$.
Send $\langle$HijackSetup$, \underline{sid}, qid\rangle$ on network.

15. Receive $\langle \text{HijackRetrieve}, \underline{sid}, qid, a \rangle$ on network,
    such that $\{\text{“HijackRetrieve”}, \text{“Lock”}\} \cap \underline{Seen}[qid] = \emptyset$
    and $\{\text{“Retrieve”}\} \subset \underline{Seen}[qid]$:

    Insert “HijackRetrieve” into $\underline{Seen}[qid]$.
    Change $\underline{U}[qid] \leftarrow \mathcal{A}$. Change the input: $\underline{a}[qid] \leftarrow a$.
    Send $\langle \text{HijackRetrieve}, \underline{sid}, qid \rangle$ on network.

16. Receive $\langle \text{ExposeUserSetup}, \underline{sid}, qid \rangle$ on network,
    where $\underline{U}[qid] \in \underline{Corrupted}$,
    such that $\{\text{“ExposeUserSetup”}, \text{“Done:}\mathcal{P}\text{”}, \text{“Done:}\mathcal{Q}\text{”}\} \cap \underline{Seen}[qid] = \emptyset$,
    and $\{\text{“Setup”}\} \subset \underline{Seen}[qid]$:

    Insert “ExposeUserSetup” into $\underline{Seen}[qid]$.
    Send $\langle \text{ExposeUserSetup}, \underline{sid}, qid, \underline{p}^{\mathcal{P}}, \underline{k}^{\mathcal{P}} \rangle$ on network.

17. Receive $\langle \text{ExposeUserRetrieve}, \underline{sid}, qid \rangle$ on network,
    where $\underline{U}[qid] \in \underline{Corrupted}$,
    such that $\{\text{“ExposeUserRetrieve”}, \text{“Lock”}\} \cap \underline{Seen}[qid] = \emptyset$,
    and $\{\text{“Retrieve”}\} \subset \underline{Seen}[qid]$:

    Insert “ExposeUserRetrieve” into $\underline{Seen}[qid]$.
    Send $\langle \text{ExposeUserRetrieve}, \underline{sid}, qid, \underline{a}[qid] \rangle$ on network.

18. Receive $\langle \text{ExposeSetup}, \underline{sid}, qid \rangle$ on network,
    where $\underline{RefreshPeriod}^{\mathcal{P}} = \underline{RefreshPeriod}^{\mathcal{Q}}$,
    $\underline{CorruptedIn}[\underline{RefreshPeriod}^{\mathcal{P}}] = \{pid_{\mathcal{P}}, pid_{\mathcal{Q}}\}$, $\underline{p}^{\mathcal{P}} = \underline{p}^{\mathcal{Q}}$, and $\underline{k}^{\mathcal{P}} = \underline{k}^{\mathcal{Q}}$,
    such that $\{\text{“ExposeSetup”}\} \cap \underline{Seen}[qid] = \emptyset$,
    and $\{\text{“Setup”}\} \subset \underline{Seen}[qid]$:

    Insert “ExposeSetup” into $\underline{Seen}[qid]$.
    Send $\langle \text{ExposeSetup}, \underline{sid}, qid, \underline{p}^{\mathcal{P}}, \underline{k}^{\mathcal{P}} \rangle$ on network.

19. Receive $\langle \text{ModifyRetrieveResponse}, \underline{sid}, qid, k \rangle$ on network,
    where $\underline{Corrupted} = \{pid_{\mathcal{P}}, pid_{\mathcal{Q}}\}$ and $k \in \mathbb{Z}_q \cup \{\varepsilon\}$:

    Replace: $\underline{k}'[qid] \leftarrow k$.
    Send $\langle \text{ModifyRetrieveResponse}, \underline{sid}, qid \rangle$ on network.

20. Receive $\langle \text{ModifySetup:}\mathcal{R}, \underline{sid}, p, k, b \rangle$ on network,
    where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$, $pid_{\mathcal{R}} \in \underline{Corrupted}$, $p \in \mathbb{Z}_q \cup \{\perp, \triangle\}$, $k \in \mathbb{Z}_q \cup \{\perp, \triangle\}$, and $b \in \mathbb{Z}_2$:

    Replace: $\underline{p}^{\mathcal{R}} \leftarrow p$, $\underline{k}^{\mathcal{R}} \leftarrow k$. If $b = 1$ then add $pid_{\mathcal{R}}$to $\underline{SetupDone}$; else remove $pid_{\mathcal{R}}$from $\underline{SetupDone}$.
    Send $\langle \text{ModifySetup:}\mathcal{R}, \underline{sid} \rangle$ on network.

21. Receive $\langle \text{FalseMemory:}\mathcal{R}, \underline{sid}, qid, p, k\rangle$ on network,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$, $\underline{p}^{\mathcal{R}} = \triangle$, $\underline{k}^{\mathcal{R}} = \triangle$, $p \in \mathbb{Z}_q$, and $k \in \mathbb{Z}_q$:

    Replace: $\underline{p}^{\mathcal{R}} \leftarrow p$, $\underline{k}^{\mathcal{R}} \leftarrow k$.
    Send $\langle \text{FalseMemory:}\mathcal{R}, \underline{sid}\rangle$ on network.

22. Receive $\langle \text{Recover}, \underline{sid}, \mathcal{R}\rangle$ on network,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ and $pid_{\mathcal{R}} \in \underline{Corrupted}$:

    If $\underline{Corrupted} = \{pid_{\mathcal{P}}, pid_{\mathcal{Q}}\}$, then let
    $b \leftarrow \max(\underline{RefreshPeriod}^{\mathcal{P}}, \underline{RefreshPeriod}^{\mathcal{Q}}) + 1$,
    $\underline{RefreshPeriod}^{\mathcal{P}} \leftarrow b$, $\underline{RefreshPeriod}^{\mathcal{Q}} \leftarrow b$, and $\underline{CorruptedIn}[b] \leftarrow \{pid_{\mathcal{P}}, pid_{\mathcal{Q}}\}$.
    Remove $pid_{\mathcal{R}}$ from $\underline{Corrupted}$ and from $\underline{Refreshing}$.
    If $pid_{\mathcal{R}} \in \underline{SetupDone}$, then insert $pid_{\mathcal{R}}$ into $\underline{JustRecovered}$;
    else remove $pid_{\mathcal{R}}$ from $\underline{JustRecovered}$.
    Send $\langle \text{Recover}, \underline{sid}\rangle$ on $\mathcal{R}$.

To simplify the presentation, we chose not to model the fact that the recovered server might accept *qid*s he has seen already or reject *qid*s that he has not yet seen. Fixing this issue is tedious but not difficult. In practice where *qid*s are chosen at random by a protocol preceeding ours, this issue is moot.

4.3 Our Construction of TPASS Secure Against Transient Corruptions

In this section we present our realization Π_{2pass} of the $\mathcal{F}_{\text{2pass}}$ ideal functionality in the $(\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}, \mathcal{F}_{\text{osac}}, \mathcal{F}_{\text{ac}}, \mathcal{F}_{\text{gzk}}, \mathcal{F}_{\text{gzk}}^{\text{2v}})$-hybrid setting. Our Π_{2pass} protocol further uses a CCA2-secure cryptosystem and an HMT commitment scheme. As for $\mathcal{F}_{\text{2pass}}$, we describe Π_{2pass} for a single user account only, i.e., each instance of Π_{2pass} uses a fixed *sid*.

We start this section by discussing the high-level ideas of our construction. We then elaborate on the novel core ideas in our construction, before providing the detailed construction. We provide an estimate of the computational and communication complexity of Π_{2pass}. Finally, we compare our protocol with related work.

4.3.1 High Level Approach of our TPASS Protocol

Our protocol Π_{2pass} implements the Setup, Retrieve, and Refresh instructions of $\mathcal{F}_{\text{2pass}}$. An adversary can hijack a Setup or Retrieve query through the $\mathcal{F}_{\text{osac}}$ subroutine. The other instructions of $\mathcal{F}_{\text{2pass}}$ are purely conceptual for the security proof. At a high level, the realizations of the Setup and Retrieve instructions of Π_{2pass} are reminiscent of the schemes by Camenisch et al. [CLN12, CLLN14] and Brainard et al. [BJKS03]: during Setup, the user generates shares of his key and password and sends them to the servers (together with some commitments that will later be used in Retrieve). During Retrieve, the servers run a subprotocol with the user to verify the latter's password attempt using the commitments and shares obtained in Setup. If the verification succeeds, the

servers send the shares of the key back to the user, who can then reconstruct the key. Furthermore, the correctness of all values exchanged is enforced by zero-knowledge proofs. To deal with transient corruptions, our Π_{2pass} needs to implement the Refresh instruction, which allows the servers to re-randomize their shares of the key and password and thereby to re-secure their states when one of them is recovering from corruption. Naturally, prior schemes do not have a Refresh instruction as they do not provide security against transient corruptions.

The novelties of our construction arise from how we turn this basic approach into a scheme that is secure against adaptive and transient corruptions and at the same time efficient enough to be considered for practical deployment.

4.3.2 Key Ideas of our TPASS Protocol

We now present the key ideas that make it possible for our TPASS protocol to be secure against transient corruptions. These ideas are novel and of independent interest.

4.3.2.1 Three-party computation for determining equality to zero

The core subprotocol ChkPwd is depicted in Figures 4.2–4.3. To check if the password attempt a input by the user during a Retrieve query matches the stored password $p = p_{\mathcal{P}} + p_{\mathcal{Q}}$, the user and the two servers engage in a three-party computation to check if $\delta := p_{\mathcal{P}} + p_{\mathcal{Q}} - a \stackrel{?}{=} 0$, where $p_{\mathcal{P}}$ and $p_{\mathcal{Q}}$ are the shares stored by the respective servers. For efficiency reasons, it does not make sense to base that protocol on a generic multiparty computation protocol. Indeed, running one Retrieve query in our protocol is more than 3.7 times faster than evaluating a single multiplication gate in the best generic two-party computation protocol that is secure against adaptive corruptions [CES13] (see Section 4.3.5.3).

The first observation is that a commitment in the HMT scheme we use essentially consists of a *pair* of Pedersen commitments. Thus, while all components need to be considered to prove that a commitment is formed correctly, it is often sufficient to consider just one component later when doing computations with them. Now, based on this, a first idea for the desired subprotocol would be as follows. The servers' commitments $cp_{\mathcal{P}}$ and $cp_{\mathcal{Q}}$ to the shares of the password are distributed to all the parties, who then generate a commitment on the sum of the two shares using the homomorphic property of HMT commitments, and extract the first component thereof to obtain a value

$$C := \mathsf{PedC}(\mathsf{cadd}(cp_{\mathcal{P}}, cp_{\mathcal{Q}})) = y^{p_{\mathcal{P}}+p_{\mathcal{Q}}} h^{op_{\mathcal{P}}+op_{\mathcal{Q}}} ,$$

where y and h are part of the CRS. That value is an equivocable Pedersen commitment to $p := p_{\mathcal{P}} + p_{\mathcal{Q}}$ with equivocation trapdoor $\log_y h$. Given C, the user subtracts his password attempt a from that commitment:

$$B := Cy^{-a} = y^{\delta} h^{op_{\mathcal{P}}+op_{\mathcal{Q}}}.$$

We now consider the Elgamal "ciphertext" $\big(A := h^{-1}, B\big)$, which is an encryption of y^{δ} under the shared secret key $(-op_{\mathcal{P}} - op_{\mathcal{Q}})$ with fixed randomness -1. This ciphertext is then passed from $\mathcal{U}$ to $\mathcal{P}$, from $\mathcal{P}$ to $\mathcal{Q}$, and then from $\mathcal{Q}$ back to $\mathcal{P}$, where at each step, the sender exponentiates that ciphertext by a non-zero random number $r_{\mathcal{U}}$, $r_{\mathcal{P}}$, and $r_{\mathcal{Q}}$, respectively, thereby multiplying the plaintext by that random number. Also, if possible, the sender will partially decrypt the ciphertext by removing $op_{\mathcal{P}}$ or $op_{\mathcal{Q}}$: $\mathcal{U}$ computes

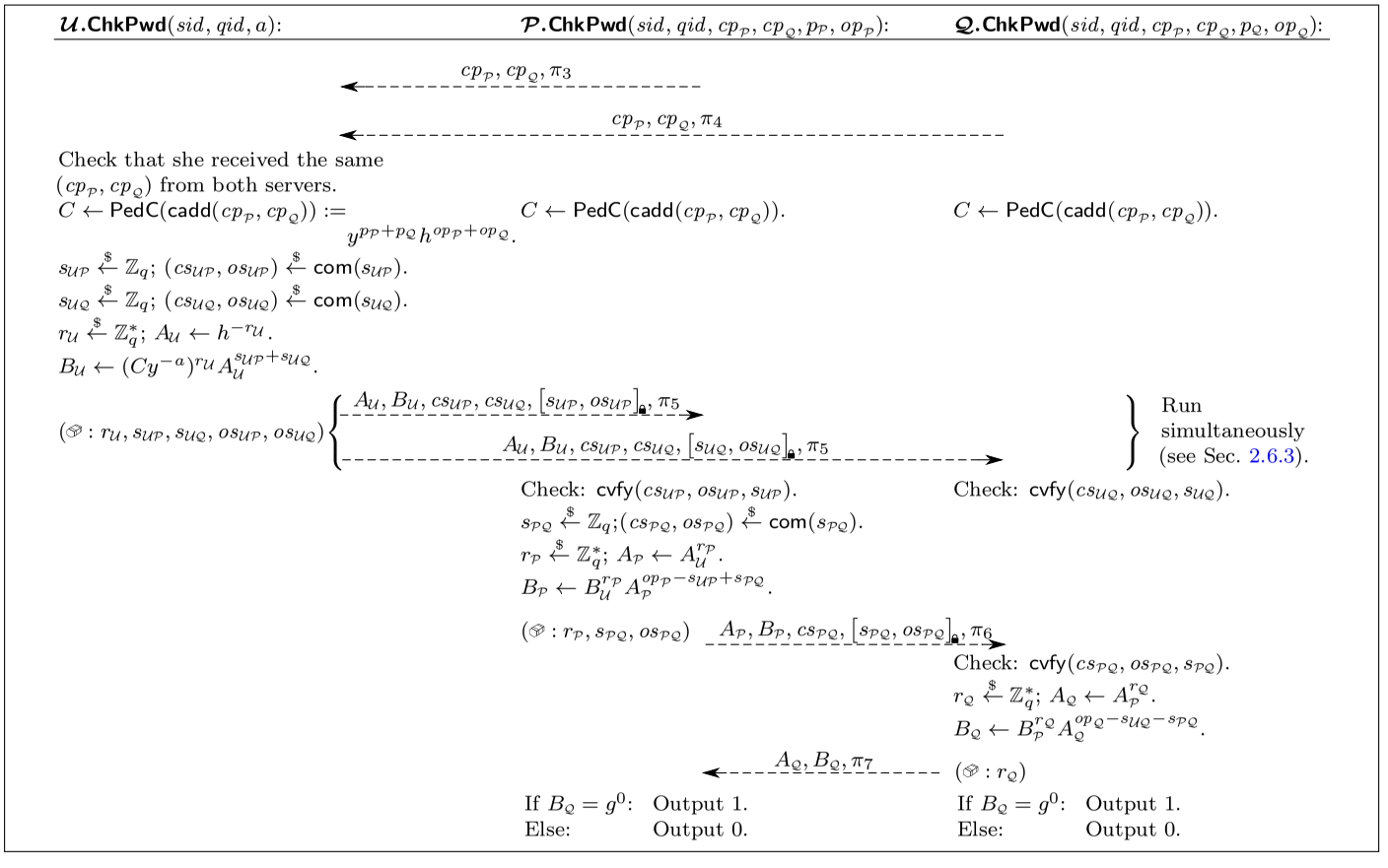


Fig. 4.2: Subroutine ChkPwd: the servers check if $\mathcal{U}$'s password attempt a is equal to the password $p_\mathcal{P} + p_\mathcal{Q}$. See Figure 4.3 for the instantiation of the zero-knowledge proofs.

$\pi_3 := \mathcal{F}_{\text{gzk}}[sid, qid, 3]\{(\nearrow p_{\mathcal{P}}, op_{\mathcal{P}}) : \mathsf{cvfy}(cp_{\mathcal{P}}, op_{\mathcal{P}}, p_{\mathcal{P}})\}.$

$\pi_4 := \mathcal{F}_{\text{gzk}}[sid, qid, 4]\{(\nearrow p_{\mathcal{Q}}, op_{\mathcal{Q}}) : \mathsf{cvfy}(cp_{\mathcal{Q}}, op_{\mathcal{Q}}, p_{\mathcal{Q}})\}.$

$\pi_5 := \mathcal{F}^{2\text{v}}_{\text{gzk}}[sid, qid, cp_{\mathcal{P}}, cp_{\mathcal{Q}}, 5]\{(\nearrow a, \sigma\ ;\ \exists \rho, \beta) :$
$h = A_{\mathcal{U}}^{\rho} \wedge C = (B_{\mathcal{U}}^{-1})^{\rho} y^{a} h^{\sigma} \wedge \mathsf{cvfy}(\mathsf{cadd}(cs_{\mathcal{UP}}, cs_{\mathcal{UQ}}), \beta, \sigma)$
$\}$, where $\sigma := s_{\mathcal{UP}} + s_{\mathcal{UQ}}$, $\rho := -1/r_{\mathcal{U}}$, and $\beta := os_{\mathcal{UP}} + os_{\mathcal{UQ}}$.

$\mathcal{U}$ runs two proofs, one with $\mathcal{P}$ and one with $\mathcal{Q}$, in parallel: she performs the erasures and sends out the last message of both proofs only *after* she received the second message of the proof from both servers (see *Proofs with two verifiers* in Section 2.6.3).

$\pi_6 := \mathcal{F}_{\text{gzk}}[sid, qid, cp_{\mathcal{P}}, cp_{\mathcal{Q}}, cs_{\mathcal{UP}}, cs_{\mathcal{UQ}}, A_{\mathcal{U}}, B_{\mathcal{U}}, 6]\{(\exists p_{\mathcal{P}}, op_{\mathcal{P}}, r_{\mathcal{P}}, \sigma, \beta) :$
$A_{\mathcal{P}} = A_{\mathcal{U}}^{r_{\mathcal{P}}} \wedge A_{\mathcal{P}} \neq g^0 \wedge B_{\mathcal{P}} = B_{\mathcal{U}}^{r_{\mathcal{P}}} A_{\mathcal{P}}^{op_{\mathcal{P}}+\sigma} \wedge$
$\mathsf{cvfy}(cp_{\mathcal{P}}, op_{\mathcal{P}}, p_{\mathcal{P}}) \wedge \mathsf{cvfy}(\mathsf{cadd}(cs_{\mathcal{PQ}}, cs_{\mathcal{UP}}^{-1}), \beta, \sigma)$
$\}$, where $\sigma := s_{\mathcal{PQ}} - s_{\mathcal{UP}}$ and $\beta := os_{\mathcal{PQ}} - os_{\mathcal{UP}}$.

$\pi_7 := \mathcal{F}_{\text{gzk}}[sid, qid, cs_{\mathcal{PQ}}, A_{\mathcal{P}}, B_{\mathcal{P}}, 7]\{(\exists p_{\mathcal{Q}}, op_{\mathcal{Q}}, r_{\mathcal{Q}}, \sigma, \beta) :$
$A_{\mathcal{Q}} = A_{\mathcal{P}}^{r_{\mathcal{Q}}} \wedge A_{\mathcal{Q}} \neq g^0 \wedge B_{\mathcal{Q}} = B_{\mathcal{P}}^{r_{\mathcal{Q}}} A_{\mathcal{Q}}^{op_{\mathcal{Q}}-\sigma} \wedge$
$\mathsf{cvfy}(cp_{\mathcal{Q}}, op_{\mathcal{Q}}, p_{\mathcal{Q}}) \wedge \mathsf{cvfy}(\mathsf{cadd}(cs_{\mathcal{UQ}}, cs_{\mathcal{PQ}}), \beta, \sigma)$
$\}$, where $\sigma := s_{\mathcal{UQ}} + s_{\mathcal{PQ}}$ and $\beta := os_{\mathcal{UQ}} + os_{\mathcal{PQ}}$.

Fig. 4.3: Instantiation of zero-knowledge proofs for ChkPwd.

$$\left(A_{\mathcal{U}}, D_{\mathcal{U}}\right) := \left(A^{r_{\mathcal{U}}}, B^{r_{\mathcal{U}}}\right) = \left(h^{-r_{\mathcal{U}}}, y^{\delta \cdot r_{\mathcal{U}}} h^{(op_{\mathcal{P}}+op_{\mathcal{Q}}) \cdot r_{\mathcal{U}}}\right)$$

and sends it to $\mathcal{P}$, $\mathcal{P}$ computes

$$\left(A_{\mathcal{P}}, D_{\mathcal{P}}\right) := \left(A_{\mathcal{U}}^{r_{\mathcal{P}}}, D_{\mathcal{U}}^{r_{\mathcal{P}}} A_{\mathcal{P}}^{op_{\mathcal{P}}}\right) = \left(h^{-r_{\mathcal{U}} \cdot r_{\mathcal{P}}}, y^{\delta \cdot r_{\mathcal{U}} \cdot r_{\mathcal{P}}} h^{op_{\mathcal{Q}} \cdot r_{\mathcal{U}} \cdot r_{\mathcal{P}}}\right)$$

and sends it to $\mathcal{Q}$, and $\mathcal{Q}$ computes

$$\left(A_{\mathcal{Q}}, B_{\mathcal{Q}}\right) := \left(A_{\mathcal{P}}^{r_{\mathcal{Q}}}, D_{\mathcal{P}}^{r_{\mathcal{Q}}} A_{\mathcal{Q}}^{op_{\mathcal{Q}}}\right) = \left(h^{-r_{\mathcal{U}} \cdot r_{\mathcal{P}} \cdot r_{\mathcal{Q}}}, y^{\delta \cdot r_{\mathcal{U}} \cdot r_{\mathcal{P}} \cdot r_{\mathcal{Q}}}\right)$$

and sends it to $\mathcal{P}$. If in the end the result $B_{\mathcal{Q}}$ is the neutral element, then $\delta = 0$, and the password was correct.

Unfortunately, this first idea doesn't quite work: if $\delta = 0$, $D_{\mathcal{U}}$ fixes a value for $(op_{\mathcal{P}} + op_{\mathcal{Q}})$ and $D_{\mathcal{P}}$ fixes a value for $op_{\mathcal{Q}}$. Thus $cp_{\mathcal{P}}$ and $cp_{\mathcal{Q}}$, together with $D_{\mathcal{U}}$ and $D_{\mathcal{P}}$ form unequivocable statistically binding commitments to $p_{\mathcal{P}}$ and $p_{\mathcal{Q}}$. This causes a selective decommitment problem. Our solution is to blind the values $D_{\mathcal{U}}$ and $D_{\mathcal{P}}$ with non-committing random shifts $s_{\mathcal{UP}}$, $s_{\mathcal{UQ}}$, and $s_{\mathcal{PQ}}$ as follows, thereby circumventing the problem. $\mathcal{U}$ chooses $s_{\mathcal{UP}}$ and $s_{\mathcal{UQ}}$, and sends them to $\mathcal{P}$ and $\mathcal{Q}$, respectively, in a non-committing manner. $\mathcal{U}$ then generates $B_{\mathcal{U}}$ by multiplying $D_{\mathcal{U}}$ with the blinding factor $A_{\mathcal{U}}^{s_{\mathcal{UP}}+s_{\mathcal{UQ}}}$, i.e.,

$$\left(A_{\mathcal{U}}, B_{\mathcal{U}}\right) := \left(A^{r_{\mathcal{U}}}, B^{r_{\mathcal{U}}} A_{\mathcal{U}}^{s_{\mathcal{UP}}+s_{\mathcal{UQ}}}\right) = \left(h^{-r_{\mathcal{U}}}, y^{\delta \cdot r_{\mathcal{U}}} h^{(op_{\mathcal{P}}+op_{\mathcal{Q}}-s_{\mathcal{UP}}-s_{\mathcal{UQ}}) \cdot r_{\mathcal{U}}}\right)$$

and sends $B_{\mathcal{U}}$ instead of $D_{\mathcal{U}}$ to $\mathcal{P}$. The ciphertext $(A_{\mathcal{U}}, B_{\mathcal{U}})$ is now encrypted under the shared key $(s_{\mathcal{UP}} + s_{\mathcal{UQ}} - op_{\mathcal{P}} - op_{\mathcal{Q}})$. Similarily, $\mathcal{P}$ chooses $s_{\mathcal{PQ}}$ and sends it to $\mathcal{Q}$. $\mathcal{P}$ generates $B_{\mathcal{P}}$ like $D_{\mathcal{P}}$ but uses $B_{\mathcal{U}}$ instead of $D_{\mathcal{U}}$ in the formula and multiplies the result by $A_{\mathcal{P}}^{-s_{\mathcal{UP}}+s_{\mathcal{PQ}}}$, i.e.,

$$\left(A_{\mathcal{P}}, B_{\mathcal{P}}\right) := \left(A_{\mathcal{U}}^{r_{\mathcal{P}}}, B_{\mathcal{U}}^{r_{\mathcal{P}}} A_{\mathcal{P}}^{op_{\mathcal{P}}-s_{\mathcal{UP}}+s_{\mathcal{PQ}}}\right) = \left(h^{-r_{\mathcal{U}} \cdot r_{\mathcal{P}}}, y^{\delta \cdot r_{\mathcal{U}} \cdot r_{\mathcal{P}}} h^{(op_{\mathcal{Q}}-s_{\mathcal{UQ}}-s_{\mathcal{PQ}}) \cdot r_{\mathcal{U}} \cdot r_{\mathcal{P}}}\right)$$

and sends $B_{\mathcal{P}}$ to $\mathcal{Q}$ instead of $D_{\mathcal{P}}$, i.e., the ciphertext $(A_{\mathcal{P}}, B_{\mathcal{P}})$ is now encrypted under the shared key $(s_{\mathcal{UQ}} + s_{\mathcal{PQ}} - op_{\mathcal{Q}})$. Finally $\mathcal{Q}$ computes $B_{\mathcal{Q}}$ differently by replacing $D_{\mathcal{P}}$ by

$B_{\mathcal{P}}$ in the formula and multiplying the result by $A_{\mathcal{Q}}^{-s_{\mathcal{UQ}}-s_{\mathcal{PQ}}}$, i.e.,

$$\left(A_{\mathcal{Q}}, B_{\mathcal{Q}}\right) := \left(A_{\mathcal{P}}^{r_{\mathcal{Q}}}, B_{\mathcal{P}}^{r_{\mathcal{Q}}} A_{\mathcal{Q}}^{op_{\mathcal{Q}}-s_{\mathcal{UQ}}-s_{\mathcal{PQ}}}\right) = \left(h^{-r_{\mathcal{U}} r_{\mathcal{P}} \cdot r_{\mathcal{Q}}}, y^{\delta \cdot r_{\mathcal{U}} \cdot r_{\mathcal{P}} \cdot r_{\mathcal{Q}}}\right).$$

At the end of each step, the parties prove to each other in zero-knowledge that they computed their values correctly; whereby the parties use the trick explained in the next paragraph to refer to $s_{\mathcal{UP}}$, $s_{\mathcal{UQ}}$, and $s_{\mathcal{PQ}}$ in the proofs. These proofs also allow the simulator to extract a, $p_{\mathcal{P}}$, $p_{\mathcal{Q}}$, $op_{\mathcal{P}}$, $op_{\mathcal{Q}}$, and $(s_{\mathcal{UP}} + s_{\mathcal{PQ}})$ in the security proof.

4.3.2.2 Transmission of secrets for later use in proofs

In the protocol just described, $\mathcal{U}$ must send the value $s_{\mathcal{UP}}$ to $\mathcal{P}$ in a non-committing manner and all parties must be able to prove knowledge of that same value in subsequent zero-knowledge proofs. Simply having $\mathcal{U}$ encrypt $s_{\mathcal{UP}}$ is not sufficient, because $\mathcal{P}$ can later not prove knowledge of the encrypted $s_{\mathcal{UP}}$ in proofs. A similar situation also arises in other parts of our protocol, for example in the Setup instruction when $\mathcal{U}$ must send a share $p_{\mathcal{P}}$ to the password to $\mathcal{P}$ in a non-committing manner.

In a setting that considers only static corruptions, such problems are often solved by requiring $\mathcal{U}$ to send a Pedersen commitment $cs_{\mathcal{UP}}$ to $s_{\mathcal{UP}}$ to all parties, and to send $s_{\mathcal{UP}}$ and the opening $os_{\mathcal{UP}}$ to the commitment to $\mathcal{P}$, encrypted under $\mathcal{P}$'s public key. Thus, with $cs_{\mathcal{UP}}$, $\mathcal{P}$ can later prove that it correctly used $s_{\mathcal{UP}}$ in its computations.

When dealing with adaptive or transient corruptions, this does not work: the encryption of $s_{\mathcal{UP}}$ causes a selective decommitment problem. Instead, we have $\mathcal{U}$ generate an equivocable commitment $cs_{\mathcal{UP}}$ to $s_{\mathcal{UP}}$ with opening $os_{\mathcal{UP}}$, then establish a one-time pad (OTP) with $\mathcal{P}$, and then encrypt both $s_{\mathcal{UP}}$ and $os_{\mathcal{UP}}$ with the OTP. $\mathcal{U}$ then sends the resulting ciphertext to $\mathcal{P}$ in any convenient manner (in this specific example, $\mathcal{U}$ sends it as part of proof protocol π_5 in Figures 4.2–4.3 that actually uses the values $s_{\mathcal{UP}}$, $os_{\mathcal{UP}}$, and $cs_{\mathcal{UP}}$ in some indirect form; in the Setup instruction where she needs to send $p_{\mathcal{P}}$ to $\mathcal{P}$ in a non-committing manner, $\mathcal{U}$ sends the ciphertext to $\mathcal{P}$ directly). Afterwards, $\mathcal{P}$ can refer to $s_{\mathcal{UP}}$ in zero-knowledge proofs by means of $cs_{\mathcal{UP}}$, e.g., $\mathcal{F}_{\text{gzk}}[sid]\{(\exists s_{\mathcal{UP}}, os_{\mathcal{UP}}) : cs_{\mathcal{UP}} = \mathsf{com}(s_{\mathcal{UP}}, os_{\mathcal{UP}})\}$. This approach will allow $\mathcal{S}$ to equivocate $s_{\mathcal{UP}}$, provided that no extra dependencies on the opening $os_{\mathcal{UP}}$ are introduced in other protocol steps (the first idea of the three-party protocol above describes the problems when such an extra dependency is introduced on $op_{\mathcal{P}}$).

4.3.3 Detailed Construction of $\Pi_{\mathbf{2pass}}$

We now give the full details of the instructions of our protocol and their respective subprotocols. Let us start with a few remarks. First, our protocol is parametrized by an encryption scheme $(\mathsf{kgen}, \mathsf{enc}, \mathsf{dec})$ with labels (looking ahead, we require this scheme to be CCA-2 secure). Second, our protocol is further parametrized by a statistically binding commitment scheme $(\mathsf{cgen}, \mathsf{com}, \mathsf{cvfy}, \mathsf{cadd}, \mathsf{cmul})$ that also implements cpc of the HMT commitment scheme over a prime-order group defined in Section 3.1.2. Using Pedersen commitments instead would require expensive zero-knowledge proofs of *knowledge* in the protocol, thereby massively increasing the computational complexity. We assume that all protocol participants agree on the group $(\mathbb{G}, q, g) \xleftarrow{\$} \mathsf{ggen}(1^\eta)$ the scheme operates in ahead of time (i.e., it is a system parameter), and that the parameters

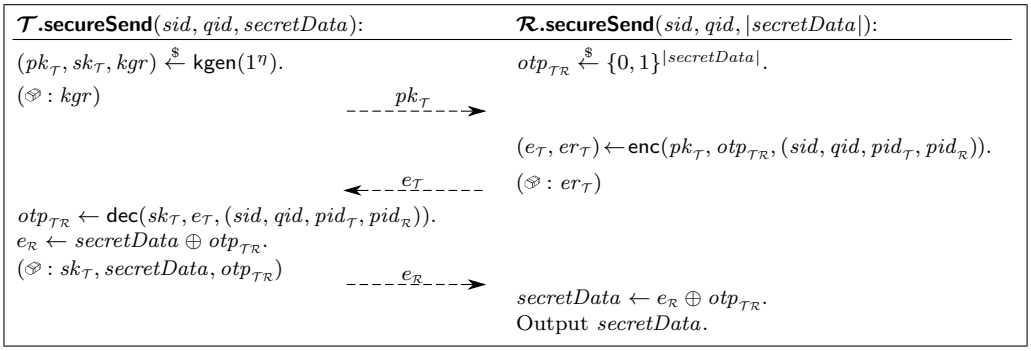


Fig. 4.4: Subroutine secureSend, the realization of $\overset{[secretData]_{\text{🔒}}}{\dashrightarrow}$: a party $\mathcal{T}$ (user or server) sends $secretData$ to $\mathcal{R}$ (user or server) in a non-committing encrypted form.

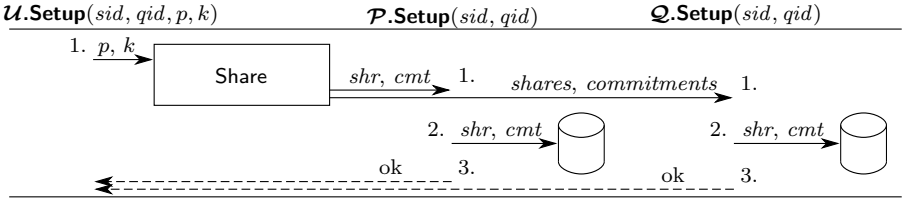
$(\cdot,\cdot,\cdot,h,y,u) \xleftarrow{\$} \mathsf{cgen}(\mathbb{G},q,g)$ of the scheme are distributed through the CRS $\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}$. Third, we assume that for each query the user establishes a single instance of a one-side-authenticated channel $\mathcal{F}_{\text{osac}}[(sid,qid),\mathcal{P}]$ and $\mathcal{F}_{\text{osac}}[(sid,qid),\mathcal{Q}]$ with each respective server; all communication denoted by arrows: $\dashrightarrow$, and all communication inside the zero-knowledge functionalities $\mathcal{F}_{\text{gzk}}$ and $\mathcal{F}_{\text{gzk}}^{2\text{v}}$ happen through that instance.[1] The two servers communicate with each other through regular authenticated channels $\mathcal{F}_{\text{ac}}[(sid, qid),\mathcal{P},\mathcal{Q},ssid]$. Fourth, parties can send data in a non-committing and confidential manner, i.e., secure against adaptive corruptions, by using the secureSend subroutine depicted in Figure 4.4. We denote such communication by: $\overset{[secretData]_{\text{🔒}}}{\dashrightarrow}$ (cf. Section 2.1). The parties establish a one-time pad (OTP) with each other, encrypt the data with that OTP, and erase the OTP before sending the ciphertext [BH93]. Fifth, we implicitly assume that a party aborts a query without output if any check fails.

4.3.3.1 The Setup instruction

Recall that the goal of the Setup instruction is for a user to set up an account $uacc$ with the two servers $\mathcal{P}$ and $\mathcal{Q}$ and store a key $k \in \mathbb{Z}_q$ protected under a password $p \in \mathbb{Z}_q$ therein. The servers will silenty abort a Setup query if the user account has already been established.

When a user $\mathcal{U}$ receives an input $\langle \text{Setup}, sid = (pid_\mathcal{P}, pid_\mathcal{Q}, (\mathbb{G},q,g), uacc, ssid), qid = \text{“Setup”}, p, k\rangle$ from the environment $\mathcal{E}$, she starts a Setup query. Each of the servers starts a Setup query when he receives an input $\langle \text{ReadySetup}, sid, qid\rangle$ from $\mathcal{E}$. As the first step of the Setup query, $\mathcal{U}$ distributes shares of k and p to both servers using the Share subprotocol. In that subprotocol, the user establishes an OTP with each server and encrypts the shares with the respective OTPs in order to circumvent the *selective decommitment problem* [Hof11]. Finally, the servers store their shares as their internal state and send an acknowledgement back to the user. See Figure 4.5. At the end of the Setup query, each of the three parties outputs $\langle \text{Done}, sid, qid\rangle$ to $\mathcal{E}$.

[1] Refer to Barak et al. [BCL+05] for details about modelling communication with partial authentication in the UC model.



1. $\mathcal{U}$ generates shares and commitments to her password and key and sends them to the servers: $(\varepsilon; (cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}, p_{\mathcal{P}}, k_{\mathcal{P}}, op_{\mathcal{P}}, ok_{\mathcal{P}}); (cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}, p_{\mathcal{Q}}, k_{\mathcal{Q}}, op_{\mathcal{Q}}, ok_{\mathcal{Q}})) \xleftarrow{\$} \langle \mathcal{U}.\mathsf{Share}(p, k), \mathcal{P}.\mathsf{Share}(), \mathcal{Q}.\mathsf{Share}() \rangle (sid, qid)$.
2. Each server $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ stores $(cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}, p_{\mathcal{R}}, k_{\mathcal{R}}, op_{\mathcal{R}}, ok_{\mathcal{R}})$ into his long-term storage.
3. The servers send an acknowledgement to $\mathcal{U}$.

Fig. 4.5: Setup instruction: $\mathcal{U}$ distributedly stores a key k protected under a password p on two servers $\mathcal{P}$ and $\mathcal{Q}$.

The Share subprotocol Setup uses is depicted in Figure 4.6. In that subprotocol $\mathcal{U}$ splits her inputs p and k into random additive shares $p_{\mathcal{P}} + p_{\mathcal{Q}} := p$ and $k_{\mathcal{P}} + k_{\mathcal{Q}} := k$, and sends $(p_{\mathcal{P}}, k_{\mathcal{P}})$ to $\mathcal{P}$ and sends $(p_{\mathcal{Q}}, k_{\mathcal{Q}})$ to $\mathcal{Q}$. She commits to all shares and sends all commitments to both servers; additionally she sends the openings for a server's shares to the respective server; thus enabling the servers to later perform zero-knowledge proofs about their shares and the commitments to them. The servers then ensure they got the same commitments and prove to each other that they know their shares. In π_2, $\mathcal{Q}$ also proves to $\mathcal{P}$ that he knows the opening $op_{\mathcal{Q}}$ corresponding to his share of the password: this is needed so that $\mathcal{S}$ can properly simulate $B_{\mathcal{P}} = (A_{\mathcal{P}})^{s_{\mathcal{U}\mathcal{Q}} + s_{\mathcal{P}\mathcal{Q}} - op_{\mathcal{Q}}}$ in ChkPwd (we note that $\mathcal{S}$ does not need to know the value $op_{\mathcal{P}}$ from π_1 at this point).

4.3.3.2 The Retrieve instruction

Recall that the goal of the Retrieve instruction is for a user (not necessarily the same as during Setup) to retrieve the key k, contingent upon her holding a correct password attempt $a \in \mathbb{Z}_q$.

When a user $\mathcal{U}$ receives an input $\langle \text{Retrieve}, sid, qid, a \rangle$ with the same sid as during Setup from $\mathcal{E}$, she starts a Retrieve query. Each of the servers starts a Retrieve query when he receives an input $\langle \text{ReadyRetrieve}, sid, qid \rangle$ from $\mathcal{E}$. The servers may refuse to service the query if they for instance suspect that an online password guessing attack is in progress, e.g., if they have processed too many failed Retrieve queries for that user account already. As many policies for throttling down can be envisaged, we decided not to include the policy in our model but rather to let $\mathcal{E}$ decide: if the server should refuse service, $\mathcal{E}$ does not provide the initial input $\langle \text{ReadyRetrieve}, sid, qid \rangle$. The Retrieve instruction runs as follows and is depicted in Figure 4.7. The servers start a Retrieve query by retrieving their internal state. The user and the servers then engage in a three-party computation to determine whether $\delta := p_{\mathcal{P}} + p_{\mathcal{Q}} - a \stackrel{?}{=} 0$, i.e., whether the password attempt is correct, using the ChkPwd subprotocol. If the password is correct, the servers send their shares of the key back to the user using the Reconstr subprotocol;

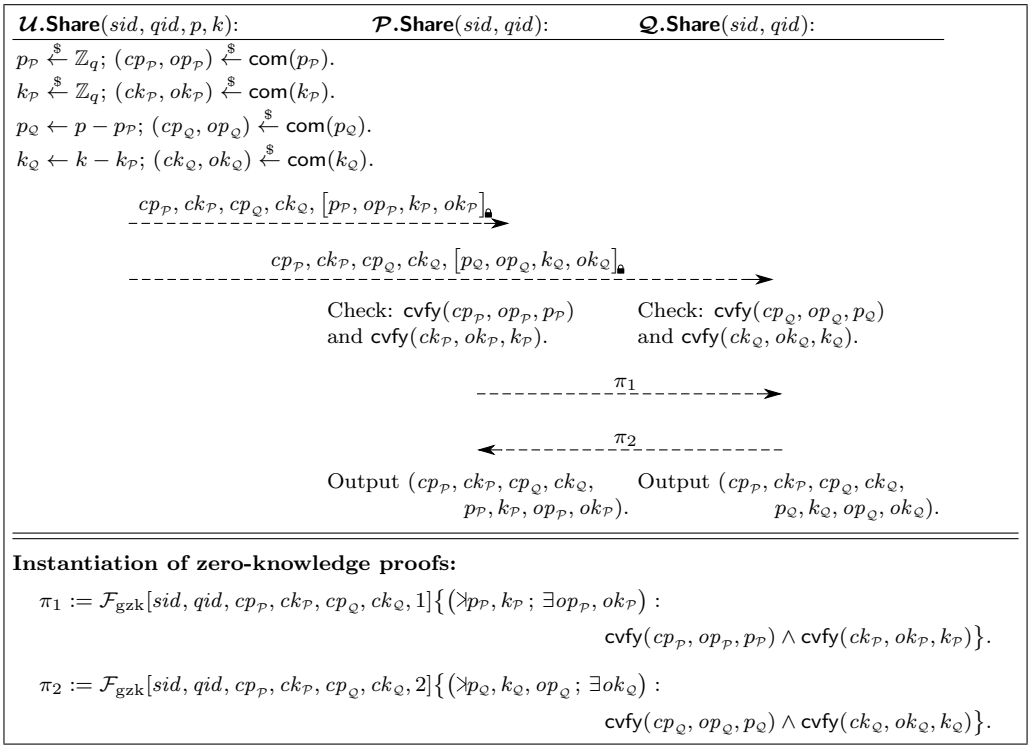
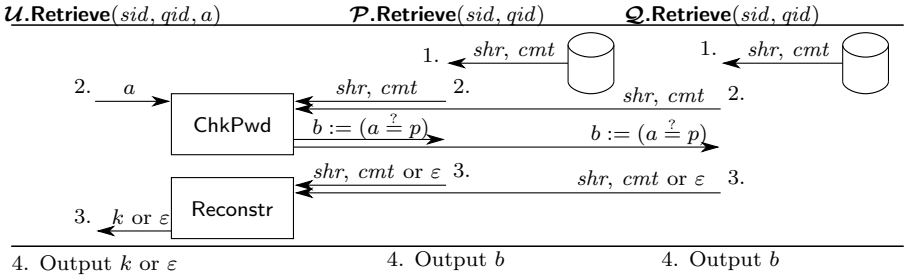


Fig. 4.6: Subroutine Share: $\mathcal{U}$ generates shares to her password p and key k, and sends them to the servers.

if wrong, they send back ε. At the end of the Retrieve query, $\mathcal{U}$ outputs $\langle \text{Deliver}, sid, qid, k' \rangle$ to $\mathcal{E}$, and each server outputs $\langle \text{Delivered}, sid, qid, b \rangle$ to $\mathcal{E}$—where $k' = k$ and $b = 1$ if the password attempt was correct, else $k' = \varepsilon$ and $b = 0$.

We now describe the two subprotocols that the Retrieve instruction uses. ChkPwd was already explained in Section 4.3.2.1 and was depicted in Figures 4.2–4.3. Reconstr is depicted in Figure 4.8. In this subprotocol, each server sends his share of the key ($k_\mathcal{P}$ or $k_\mathcal{Q}$) and the corresponding opening to $\mathcal{U}$. Both servers also send her the two commitments to the shares of the key. The user checks that she received the same commitments from both servers, that the shares and openings are correct, and reconstructs the key $k := k_\mathcal{P} + k_\mathcal{Q}$. The servers may send ε instead to denote a failed password attempt; in that case $\mathcal{U}$ outputs ε.

In both the ChkPwd and the Reconstr subprotocols, $\mathcal{U}$ needs to send data in a non-committing and confidential manner to $\mathcal{P}$. Instead of generating the OTPs for each subprotocol separately, the two parties could generate a single OTP of double the length in one operation and use the first half of the OTP during ChkPwd and the second half during Reconstr. This optimization would save one key generation (for the CCA2-secure cryptosystem), one encryption, and one decryption. The same optimization can be applied between $\mathcal{U}$ and $\mathcal{Q}$.



1. Each server $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ retrieves $(cp_{\mathcal{P}}, ck_{\mathcal{P}}, cp_{\mathcal{Q}}, ck_{\mathcal{Q}}, p_{\mathcal{R}}, k_{\mathcal{R}}, op_{\mathcal{R}}, ok_{\mathcal{R}})$ from his long-term storage.
2. $\mathcal{U}$, $\mathcal{P}$, and $\mathcal{Q}$ run a three-party protocol to determine if the password attempt is correct:
   $(\varepsilon, b, b) \xleftarrow{\$} \langle \mathcal{U}.\mathsf{ChkPwd}(a), \mathcal{P}.\mathsf{ChkPwd}(cp_{\mathcal{P}}, cp_{\mathcal{Q}}, p_{\mathcal{P}}, op_{\mathcal{P}}), \mathcal{Q}.\mathsf{ChkPwd}(cp_{\mathcal{P}}, cp_{\mathcal{Q}}, p_{\mathcal{Q}}, op_{\mathcal{Q}}) \rangle (sid, qid)$.
3. • If $b = 1$ (i.e., $a = p$: the password attempt was correct), the servers send the key $k' = k$ to $\mathcal{U}$:
   $(k', \varepsilon, \varepsilon) \xleftarrow{\$} \langle \mathcal{U}.\mathsf{Reconstr}(), \mathcal{P}.\mathsf{Reconstr}(ck_{\mathcal{P}}, ck_{\mathcal{Q}}, k_{\mathcal{P}}, ok_{\mathcal{P}}), \mathcal{Q}.\mathsf{Reconstr}(ck_{\mathcal{P}}, ck_{\mathcal{Q}}, k_{\mathcal{Q}}, ok_{\mathcal{Q}}) \rangle (sid, qid)$.
   • Else if $b = 0$, the servers send a empty value $k' = \varepsilon$ instead:
   $(k'; \varepsilon; \varepsilon) \xleftarrow{\$} \langle \mathcal{U}.\mathsf{Reconstr}(), \mathcal{P}.\mathsf{Reconstr}(\varepsilon, \varepsilon, \varepsilon, \varepsilon), \mathcal{Q}.\mathsf{Reconstr}(\varepsilon, \varepsilon, \varepsilon, \varepsilon) \rangle (sid, qid)$.

Fig. 4.7: Retrieve instruction: $\mathcal{U}$ retrieves the key k if she provides the correct password.

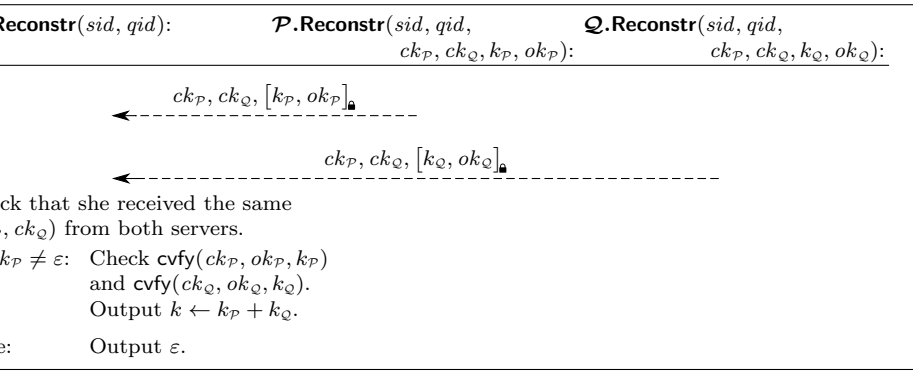
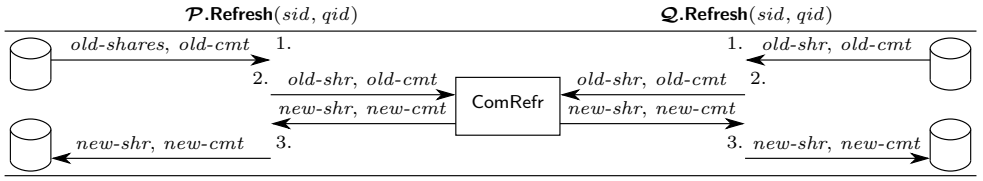


Fig. 4.8: Subroutine Reconstr: the servers send their commitments and shares of the key to $\mathcal{U}$ so that she may reconstruct her key k.

4.3.3.3 The Refresh instruction

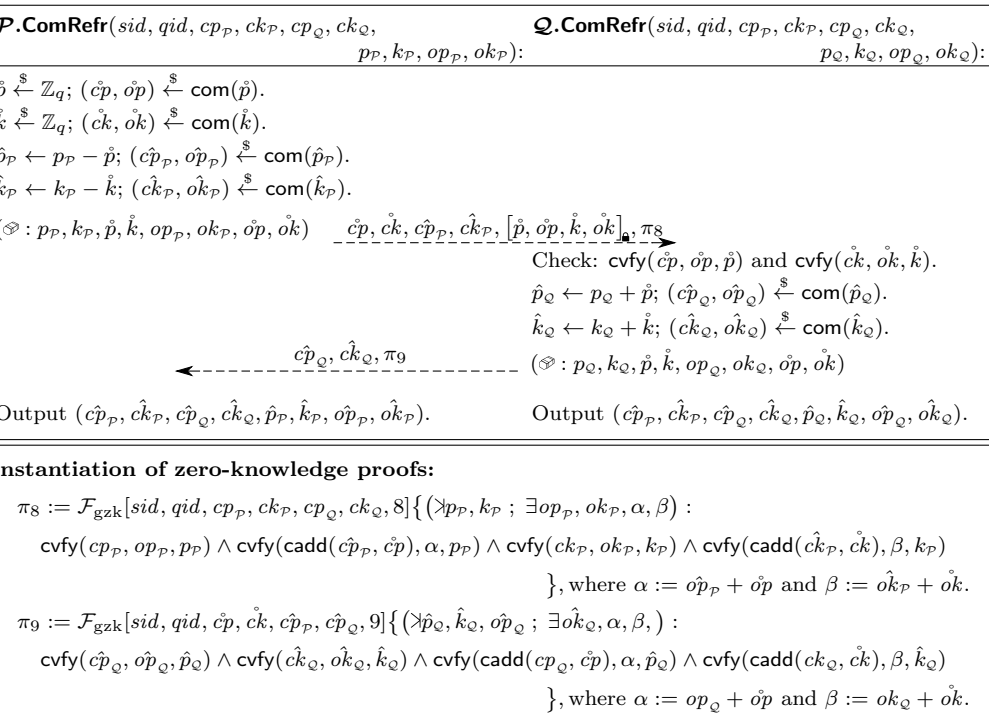
In the Refresh instruction, the servers re-randomize their shares and generate new commitments to them. This ensures that $\mathcal{A}$ no longer has any knowledge about the internal state of a party who recovered from corruption. Servers execute a Refresh query immediately after they formally recover from corruption (see Section 4.1). Upon starting a Refresh query, the servers abort all running Setup and Retrieve queries and stop accepting new ones. Upon completion of the Refresh query, they resume acceptance of new Setup and Retrieve queries.

When a server receives an input $\langle \text{Refresh}, sid, qid \rangle$ with the same sid as during Setup from $\mathcal{E}$, he starts the Refresh instruction. The Refresh protocol runs as follows and is depicted in Figure 4.9. The servers start by recovering their internal state. The servers



1. Each server $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ retrieves $(cp_\mathcal{P}, ck_\mathcal{P}, cp_\mathcal{Q}, ck_\mathcal{Q}, p_\mathcal{R}, k_\mathcal{R}, op_\mathcal{R}, ok_\mathcal{R})$ from his long-term storage.
2. The servers re-randomize their shares:
$((\hat{cp}_\mathcal{P}, \hat{ck}_\mathcal{P}, \hat{cp}_\mathcal{Q}, \hat{ck}_\mathcal{Q}, \hat{p}_\mathcal{P}, \hat{k}_\mathcal{P}, \hat{op}_\mathcal{P}, \hat{ok}_\mathcal{P}); (\hat{cp}_\mathcal{P}, \hat{ck}_\mathcal{P}, \hat{cp}_\mathcal{Q}, \hat{ck}_\mathcal{Q}, \hat{p}_\mathcal{Q}, \hat{k}_\mathcal{Q}, \hat{op}_\mathcal{Q}, \hat{ok}_\mathcal{Q})) \xleftarrow{\$}$
$\langle \mathcal{P}.\mathsf{ComRefr}(p_\mathcal{P}, k_\mathcal{P}, op_\mathcal{P}, ok_\mathcal{P}), \mathcal{Q}.\mathsf{ComRefr}(p_\mathcal{Q}, k_\mathcal{Q}, op_\mathcal{Q}, ok_\mathcal{Q})\rangle(sid, qid, cp_\mathcal{P}, ck_\mathcal{P}, cp_\mathcal{Q}, ck_\mathcal{Q})$.
3. Each server $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$ stores $(\hat{cp}_\mathcal{P}, \hat{ck}_\mathcal{P}, \hat{cp}_\mathcal{Q}, \hat{ck}_\mathcal{Q}, \hat{p}_\mathcal{R}, \hat{k}_\mathcal{R}, \hat{op}_\mathcal{R}, \hat{ok}_\mathcal{R})$ into his long-term storage.

Fig. 4.9: Refresh instruction: the servers re-randomize their internal state.



Instantiation of zero-knowledge proofs:

$\pi_8 := \mathcal{F}_{\text{gzk}}[sid, qid, cp_\mathcal{P}, ck_\mathcal{P}, cp_\mathcal{Q}, ck_\mathcal{Q}, 8]\{(\not\exists p_\mathcal{P}, k_\mathcal{P} \; ; \; \exists op_\mathcal{P}, ok_\mathcal{P}, \alpha, \beta) :$
$\mathsf{cvfy}(cp_\mathcal{P}, op_\mathcal{P}, p_\mathcal{P}) \wedge \mathsf{cvfy}(\mathsf{cadd}(\hat{cp}_\mathcal{P}, \mathring{cp}), \alpha, p_\mathcal{P}) \wedge \mathsf{cvfy}(ck_\mathcal{P}, ok_\mathcal{P}, k_\mathcal{P}) \wedge \mathsf{cvfy}(\mathsf{cadd}(\hat{ck}_\mathcal{P}, \mathring{ck}), \beta, k_\mathcal{P})$
$\}$, where $\alpha := \hat{op}_\mathcal{P} + \mathring{op}$ and $\beta := \hat{ok}_\mathcal{P} + \mathring{ok}$.

$\pi_9 := \mathcal{F}_{\text{gzk}}[sid, qid, \mathring{cp}, \mathring{ck}, \hat{cp}_\mathcal{P}, \hat{cp}_\mathcal{Q}, 9]\{(\not\exists \hat{p}_\mathcal{Q}, \hat{k}_\mathcal{Q}, \hat{op}_\mathcal{Q} \; ; \; \exists \hat{ok}_\mathcal{Q}, \alpha, \beta,) :$
$\mathsf{cvfy}(\hat{cp}_\mathcal{Q}, \hat{op}_\mathcal{Q}, \hat{p}_\mathcal{Q}) \wedge \mathsf{cvfy}(\hat{ck}_\mathcal{Q}, \hat{ok}_\mathcal{Q}, \hat{k}_\mathcal{Q}) \wedge \mathsf{cvfy}(\mathsf{cadd}(cp_\mathcal{Q}, \mathring{cp}), \alpha, \hat{p}_\mathcal{Q}) \wedge \mathsf{cvfy}(\mathsf{cadd}(ck_\mathcal{Q}, \mathring{ck}), \beta, \hat{k}_\mathcal{Q})$
$\}$, where $\alpha := op_\mathcal{Q} + \mathring{op}$ and $\beta := ok_\mathcal{Q} + \mathring{ok}$.

Fig. 4.10: Subroutine ComRefr: the servers generate new commitments and shares of the password and key based on the old ones.

then re-randomize their shares of the password and key using the ComRefr subprotocol. Finally both servers store their new internal state. At the end of the protocol, each server outputs $\langle \text{RefreshDone}, sid, qid \rangle$ to $\mathcal{E}$.

The Refresh instruction uses the ComRefr subprotocol, depicted in Figure 4.10, the goal of which is for both servers $\mathcal{P}$ and $\mathcal{Q}$ to re-randomize their respective shares $(p_\mathcal{P}, k_\mathcal{P})$

and $(p_{\mathcal{Q}}, k_{\mathcal{Q}})$. $\mathcal{P}$ randomly selects two offsets $\mathring{p}$ and $\mathring{k}$ and subtracts them from his shares. $\mathcal{P}$ then commits to the offsets and his new shares. $\mathcal{P}$ proves to $\mathcal{Q}$ that all operations were done correctly. As part of the proof, $\mathcal{P}$ sends all the commitments and a ciphertext that contains the offsets and the corresponding openings encrypted under an OTP to $\mathcal{Q}$. $\mathcal{Q}$ likewise updates his shares and generates new commitments to them. $\mathcal{Q}$ proves to $\mathcal{P}$ that all operations were done honestly and that he knows the opening $o\hat{p}_{\mathcal{Q}}$ corresponding to his new share of the password (for the same reason as in Share: $\mathcal{S}$ needs $o\hat{p}_{\mathcal{Q}}$ when simulating $B_{\mathcal{P}}$ in ChkPwd). As part of the proof, $\mathcal{Q}$ sends the new commitments to $\mathcal{P}$.

4.3.4 Computational and Communication Complexity

The sum of the computation time of all parties for Setup, Retrieve, and Refresh queries is less than 0.08, 0.16, and 0.09 seconds for 80/1248-bit security[2] on modern computers,[3] and the communication complexity is 5, 7, and 3 round trips (when combining messages wherever possible), respectively. For the Setup instruction, 43 elements of $\mathbb{Z}_q$, 56 elements of $\mathbb{G}$, 12 elements of $\mathbb{Z}_n$, and 4 elements of $\mathbb{Z}_{n^2}$ are transmitted over plain channels in our preferred embodiment, corresponding to roughly 5.2 kilobytes for 80/1248-bit security when $\mathbb{G}$ is an elliptic curve. For the Retrieve instruction, 73.5, 99, 16, and 6 elements of $\mathbb{Z}_q$,$\mathbb{G}$,$\mathbb{Z}_n$, and $\mathbb{Z}_{n^2}$ are transmitted respectively (8 kB). For the Refresh instruction, 34, 46, 10, and 4 elements of $\mathbb{Z}_q$,$\mathbb{G}$,$\mathbb{Z}_n$, and $\mathbb{Z}_{n^2}$ are transmitted respectively (4.5 kB). Due to the fact that our protocol is secure against adaptive corruptions, it is computationally more expensive than a standard-model instantiation of the CLN protocol [CLN12] (i.e., with *interactive* zero-knowledge proofs): our Retrieve queries are about 10 and 2.6 times slower for users and servers, respectively; and more data is transferred; however the number of round trips is identical. See Section 4.3.5.3 for a detailed analysis.

4.3.5 Comparison with Related Work

In this section, we compare the ideal functionalities and the protocol constructions of our protocol against those by Camenisch, Lysyanskaya, and Neven (CLN) [CLN12] and by Camenisch, Lehmann, Lysyanskaya, and Neven (CLLN) [CLLN14]. We also provide a more detailed runtime comparison between our protocol and the CLN-protocol.

4.3.5.1 Comparison of Ideal Functionalities

In the following, we compare our ideal functionality $\mathcal{F}_{\text{2pass}}$ with those of the CLN and CLLN protocols. On a high level, our $\mathcal{F}_{\text{2pass}}$ is similar to both functionalities, in that we also model Setup and Retrieve instructions and let $\mathcal{A}$ hijack queries. We note the following differences: *1)* Our $\mathcal{F}_{\text{2pass}}$ allows for adaptive corruptions and recovery from corruption, not just static corruptions. *2)* Our $\mathcal{F}_{\text{2pass}}$ on one hand and the CLN- and CLLN-$\mathcal{F}_{\text{2pass}}$ on the other all give the servers the option to refuse to service a Retrieve request, but model this option differently. In CLN- and CLLN-$\mathcal{F}_{\text{2pass}}$, the servers are

[2] The subgroup size $|q|$ is $2 \cdot 80$ bits and the RSA modulus size $|n|$ is 1248 bits.

[3] When using the GNU MP (GMP) bignum library on 64-bit Linux on a computer with an Intel Core i7 Q720 1.60GHz CPU.

activated by the ideal functionality whenever a user wants to perform a retrieval; the servers then contact the environment and ask for permission to continue. In our $\mathcal{F}_{\text{2pass}}$, the ideal functionality does not activate the servers directly: instead, the servers activate $\mathcal{F}_{\text{2pass}}$. In order to allow the environment to decide whether the servers should service the request, the servers in our $\mathcal{F}_{\text{2pass}}$ don't need to explicity send a message to the environment to ask for permission to continue: the environment can simply refuse to provide the input that activates the server. *3)* Finally, similarly to the CLLN model but unlike the CLN one, the user's password attempt is protected during Retrieve in our $\mathcal{F}_{\text{2pass}}$ no matter the corruption status of the servers. Thus, if the user by mistake talks to the wrong servers for Retrieve, she is still safe, while the CLN functionality in this case hands the password over the adversary.

4.3.5.2 Comparison of our Construction

Comparison with the CLN-protocol. As we already pointed out in Section 4.2, the Setup and Retrieve instructions in both follow a similar structure, and we will thus mainly focus on the differences between the two.

In the CLN-protocol, the user's password and key are group elements instead of integers. This allows for an efficient way to enable the simulator $\mathcal{S}$ to extract the user's input from commitments sent by the user. Unlike the CLN protocol, our protocol must perform more expensive zero-knowledge proofs that allow the simulator $\mathcal{S}$ to extract that input.

In the CLN-protocol, the user never performs any zero-knowledge proofs. This means that unlike our protocol (and the CLLN-protocol), the user's password attempt is not protected in case he contacts two corrupted servers.

The CLN-protocol [CLN12] is secure against *static* corruptions only, while our protocol allows for *adaptive* corruptions. Due to the selective decommitment problem, which affects only protocols secure against adaptive corruptions, our protocol and the CLN-protocol further differ on three counts: *1)* in our protocol, parties need to establish OTPs among themselves, which is not needed in the CLN-protocol. We need this extra step to be able to encrypt in a non-committing way. *2)* In the CLN-protocol, the user and the servers communicate using perfectly-binding commitments. The servers don't need to prove knowledge of their shares to each other like we do at the end of Share, as $\mathcal{S}$ can extract the password and key from the perfectly-binding commitments. *3)* Our ChkPwd subroutine is different than the corresponding protocol in CLN, since we cannot allow the servers to send committing ciphertexts to each other.

Comparison with the CLLN-protocol. The CLLN-protocol [CLLN14] is also secure against only static corruptions, and thus the points in the previous paragraph also apply. Their protocol uses non-interactive zero-knowledge proofs extensively, and it is not clear how to instantiate their protocol in the standard model without resorting to impractical generic non-interactive zero-knowledge proofs.

4.3.5.3 Comparison of Computational Complexity for the Standard Model Constructions

In Table 4.1 we provide an estimate of the computational complexity of our protocol and compare it with the complexity of the CLN-protocol (adapted to the standard model) [CLN12]. We re-did the estimation for CLN using a different way of counting multi-exponentiations, so our numbers are slightly different from those provided in the original paper.

We also provide an estimate of the computation time when run with the "smallest general purpose" security level of the Ecrypt-II recommendations [BCC+11] ($\eta = 80$, $\log_2 q = 2\eta = 160$, $\log_2 n = 1248$, where $n = p' \cdot p''$ is a safe semi-prime) on a standard laptop with a 64-bit operating system using the GMP Multiple Precision Library.

We used the following runtime estimates for the basic building blocks, and assume the runtime of exponentiations scales linearly depending on the bitlength of the exponent:

- Let $\mathsf{exp}.\mathbb{G}$ be the runtime of exponentiation in $\mathbb{G}$ per bit of the exponent. For $\eta = 80$ and for a subgroup of the integers modulo a large prime, we use the estimate $\mathsf{exp}.\mathbb{G} = 1.42$ µs (i.e., a full exponentiation takes $2\eta \cdot \mathsf{exp}.\mathbb{G} = 227$ µs).
- Let $\mathsf{exp.n}$ be the runtime per bit of exponentiation modulo n. For $\eta = 80$ we use the estimate $\mathsf{exp.n} = 1.42$ µs (i.e., a full exponentiation takes $\log_2 n \cdot \mathsf{exp.n} = 1820$ µs).
- Let $\mathsf{exp.p}$ be the runtime per bit of exponentiation modulo p' or p''. For $\eta = 80$ we use the estimate $\mathsf{exp.p} = 0.42$ µs (i.e., an RSA decryption with Chinese remainder theorem takes $2((\log_2 n)/2 \cdot \mathsf{exp.p}) = 538$ µs).
- Let $\mathsf{exp.n}^2$ be the runtime per bit of exponentiation modulo n^2. For $\eta = 80$ we use the estimate $\mathsf{exp.n}^2 = 5.14$ µs (i.e., a Paillier encryption $(n+1)^x r^n \pmod{n^2}$ takes time $\log_2 n \cdot \mathsf{exp.n}^2 = 6580$ µs).
- Let $\mathsf{tprime}.2\eta$ be the average runtime to generate a prime of size 2η. For $\eta = 80$ we use the estimate $\mathsf{tprime}.2\eta = 329$ µs.

We did not consider the optimizations that are possible when running multibase exponentiations, or using precomputations for fixed bases. We chose to ignore the runtime of "fast" operations: additions, multiplications, inversions, and symmetric cryptographic operations. We also ignored the network delay: our Setup protocol requires one additional roundtrip than the CLN's, and both our and the CLN's Retrieve protocol have the same number of roundtrips. We also do not consider the various setup costs, such as generating a fresh RSA modulus for the signature, which need to be done once only and can be done ahead of time.

We chose the following standard-model implementations of primitives:

- For CCA-2 secure encryption in both our and the CLN-protocol, we use the Cramer-Shoup cryptosystem [CS98].
- For the signature scheme in the CLN-protocol, we use the Cramer-Shoup signature scheme [CS99].

We treated all zero-knowledge proofs in the CLN-protocols as proofs of existence, since $\mathcal{S}$ can extract the key and password from the El-Gamal ciphertexts. For the zero-knowledge proofs of knowledge we use in our protocol, we fit up to three witnesses into the plaintext of the verifiable encryption: Given witnesses $0 \leq a, b, c < q$, we encrypt

$(a + q^2b + q^4c)$; by adding a range proofs that $-q^2/2 < a, b, c < q^2/2$ to the statement of the zero-knowledge proof and by recalling that $q^6 < n$, one is ensured that $\mathcal{S}$ can always recover a, b, and c; this range proof is essentially for free since the verifier simply needs to check the bit length of the responses in the zero-knowledge proof; this trick allows us to save on the very expensive operations modulo n^2, which are by far the dominant cost in the proofs.

Finally, we note that for efficiency reasons it does not make sense to use generic multi-party computations protocols to implement Π_{2pass}: the computational complexity of just a single multiplication in the best two-party computation protocol UC-secure against adaptive corruptions is $(90\eta + 200\log_2 n) \cdot \mathsf{exp.n} + (66\eta + 40.5\log_2 n) \cdot \mathsf{exp.n}^2$ [CES13], corresponding to a computation time of 660 ms at $\eta = 80$, i.e., more than 3.7 times slower than our entire Retrieve protocol.

4.4 Security Proof

We now show that the protocol Π_{2pass} of Section 4.3 realizes $\mathcal{F}_{\text{2pass}}$. We start with a description of the main ideas of the security proof before presenting the full proof. The full proof proceeds in two steps: we first prove that our protocol is secure when run with *nice* environments (see Definition 3.3). We then apply the special composition theorem of Camenisch et al. [CKS11] to prove that our protocol is secure against *all* environments.

4.4.1 Main Ideas

We use the standard approach for proving the security of UC protocols: we construct a straight-line simulator $\mathcal{S}$ such that for all polynomial-time bounded environments and all polynomial-time bounded adversaries $\mathcal{A}$ it holds that the environment $\mathcal{E}$ cannot distinguish its interaction with $\mathcal{A}$ and Π_{2pass} in the $(\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}, \mathcal{F}_{\text{osac}}, \mathcal{F}_{\text{ac}}, \mathcal{F}_{\text{gzk}}, \mathcal{F}_{\text{gzk}}^{\text{2v}})$-hybrid *real* world from its interaction with $\mathcal{S}$ and $\mathcal{F}_{\text{2pass}}$ in the *ideal* world. We prove this statement by defining a sequence of intermediate *hybrid* worlds (the first one being the real world and the last one the ideal world) and showing that $\mathcal{E}$ cannot distinguish between any two consecutive hybrid worlds.

The main difficulties in constructing $\mathcal{S}$ (and accordingly in designing our protocol to allow us to address those difficulties) are as follows: *1)* $\mathcal{S}$ has to extract the inputs of all corrupted parties from the interaction with them; *2)* $\mathcal{S}$ has to compute and send commitments and ciphertexts to the corrupted parties on behalf of the honest parties without knowing the latter's inputs, i.e., $\mathcal{S}$ needs to commit and encrypt dummy values; *3)* but when an honest party gets corrupted mid-protocol, $\mathcal{S}$ has to provide $\mathcal{A}$ with the full *non-erased* intermediate state of that party, in particular the opening of commitments that were sent out and the randomness used to compute encryptions that were sent out (if these value need to be retained by a party).

To address the first difficulty, recall that parties are required to perform proofs of *knowledge* of their shares upon their first use in the protocol. $\mathcal{S}$ can therefore recover the inputs of all corrupted parties with the help of $\mathcal{F}_{\text{gzk}}$ and $\mathcal{F}_{\text{gzk}}^{\text{2v}}$. The commitments

		CLN-protocol [CLN12]		Our protocol	
Adaptive corruptions		no		yes, and servers can recover	
Computation time of Setup	user	$0_{\log_2 n \cdot \mathsf{exp.p}} + 32_{\eta \cdot \mathsf{exp.n}} + 40_{\eta \cdot \mathsf{exp.}\mathbb{G}} + 0_{\mathsf{tprime.2}\eta}$	8 ms	$(16\ _{\eta} + 0\ _{\log_2 n}) \cdot_{\mathsf{exp.n}} + 92_{\eta \cdot \mathsf{exp.}\mathbb{G}} + (0_{\eta} + 0_{\log_2 n}) \cdot_{\mathsf{exp.n}^2}$	12 ms
	$\mathcal{P}$	$2_{\log_2 n \cdot \mathsf{exp.p}} + 20_{\eta \cdot \mathsf{exp.n}} + 18_{\eta \cdot \mathsf{exp.}\mathbb{G}} + 2_{\mathsf{tprime.2}\eta}$	6 ms	$(30\ _{\eta} + 3\ _{\log_2 n}) \cdot_{\mathsf{exp.n}} + 90_{\eta \cdot \mathsf{exp.}\mathbb{G}} + (1_{\eta} + 3_{\log_2 n}) \cdot_{\mathsf{exp.n}^2}$	39 ms
	$\mathcal{Q}$	$1_{\log_2 n \cdot \mathsf{exp.p}} + 14_{\eta \cdot \mathsf{exp.n}} + 18_{\eta \cdot \mathsf{exp.}\mathbb{G}} + 1_{\mathsf{tprime.2}\eta}$	5 ms	$(32\ _{\eta} + 3\ _{\log_2 n}) \cdot_{\mathsf{exp.n}} + 90_{\eta \cdot \mathsf{exp.}\mathbb{G}} + (1_{\eta} + 3_{\log_2 n}) \cdot_{\mathsf{exp.n}^2}$	39 ms
Computation time of Retrieve	user	$0_{\log_2 n \cdot \mathsf{exp.p}} + 32_{\eta \cdot \mathsf{exp.n}} + 36_{\eta \cdot \mathsf{exp.}\mathbb{G}} + 0_{\mathsf{tprime.2}\eta}$	8 ms	$(68\ _{\eta} + 6\ _{\log_2 n}) \cdot_{\mathsf{exp.n}} + 188_{\eta \cdot \mathsf{exp.}\mathbb{G}} + (2_{\eta} + 6_{\log_2 n}) \cdot_{\mathsf{exp.n}^2}$	79 ms
	$\mathcal{P}$	$2_{\log_2 n \cdot \mathsf{exp.p}} + 20_{\eta \cdot \mathsf{exp.n}} + 125_{\eta \cdot \mathsf{exp.}\mathbb{G}} + 2_{\mathsf{tprime.2}\eta}$	18 ms	$(26\ _{\eta} + 3\ _{\log_2 n}) \cdot_{\mathsf{exp.n}} + 175_{\eta \cdot \mathsf{exp.}\mathbb{G}} + (1_{\eta} + 3_{\log_2 n}) \cdot_{\mathsf{exp.n}^2}$	48 ms
	$\mathcal{Q}$	$2_{\log_2 n \cdot \mathsf{exp.p}} + 20_{\eta \cdot \mathsf{exp.n}} + 125_{\eta \cdot \mathsf{exp.}\mathbb{G}} + 2_{\mathsf{tprime.2}\eta}$	18 ms	$(26\ _{\eta} + 3\ _{\log_2 n}) \cdot_{\mathsf{exp.n}} + 166_{\eta \cdot \mathsf{exp.}\mathbb{G}} + (1_{\eta} + 3_{\log_2 n}) \cdot_{\mathsf{exp.n}^2}$	47 ms
Comp. time of Refresh	$\mathcal{P}$	—		$(36\ _{\eta} + 4\ _{\log_2 n}) \cdot_{\mathsf{exp.n}} + 136_{\eta \cdot \mathsf{exp.}\mathbb{G}} + (0_{\eta} + 4_{\log_2 n}) \cdot_{\mathsf{exp.n}^2}$	52 ms
	$\mathcal{Q}$	—		$(26\ _{\eta} + 2\ _{\log_2 n}) \cdot_{\mathsf{exp.n}} + 138_{\eta \cdot \mathsf{exp.}\mathbb{G}} + (2_{\eta} + 2_{\log_2 n}) \cdot_{\mathsf{exp.n}^2}$	36 ms

Table 4.1: Estimate of the computation time per party of our protocol and the CLN-protocol. The computation time in milliseconds is an estimate for $\eta = 80$ on a standard laptop.

and proofs of *existence* with $\mathcal{F}_{\text{gzk}}$ and $\mathcal{F}_{\text{gzk}}^{2\text{v}}$ ensure that the corrupted parties are unable to alter their inputs mid-protocol.

The second and third difficulty we address as follows. In general, $\mathcal{S}$ runs honest parties with random input and adjusts their internal state as follows when it learns the correct values. When $\mathcal{S}$ is told by $\mathcal{F}_{\text{2pass}}$ whether the password attempt was correct in a Retrieve query, it can generate credible values $B_{\mathcal{U}}$, $B_{\mathcal{P}}$, and $B_{\mathcal{Q}}$ in the ChkPwd subroutine because $\mathcal{S}$ can recover the opening values *op* from dishonest servers through $\mathcal{F}_{\text{gzk}}$ and $\mathcal{F}_{\text{gzk}}^{2\text{v}}$. When a user gets corrupted during Setup, or both servers get corrupted, $\mathcal{S}$ can recover the actual password and key associated with the user account from $\mathcal{F}_{\text{2pass}}$ and then needs to equivocate all relevant commitments and encryptions sent earlier to the corrupted parties. This is also the case when a user gets corrupted during Retrieve, where $\mathcal{S}$ is also allowed to recover the actual password attempt. $\mathcal{S}$ can equivocate such commitments, with the help of the trapdoor, and equivocate the ciphertexts containing the openings of commitments it sent between two honest parties by altering the one-time pads. By the time a one-time pad is used, the decryption keys and randomness used to establish it have been erased and so they can be changed to equivocate. Additionally, $\mathcal{S}$ never needs to reveal the randomness used inside the ChkPwd subroutine, in particular because $\mathcal{F}_{\text{gzk}}$ and $\mathcal{F}_{\text{gzk}}^{2\text{v}}$ allow for the erasure of witnesses *before* delivering the statement to be proven to the other party. The rest of the security proof is rather straightforward.

4.4.2 Security Proof

In this section we prove that our protocol Π_{2pass} securely realizes the ideal functionality $\mathcal{F}_{\text{2pass}}$. We proceed as follows: we start by stating the main theorem and a number of lemmas, and then prove the main theorem. We then proceed to prove the main lemma in two steps: first, we describe the construction of a simulator $\mathcal{S}$, and then prove that $\mathcal{S}$ meets the requirements of the main lemma. Finally, we comment on multi-session realizations of $\mathcal{F}_{\text{2pass}}$ that use a constant-size CRS.

Recall that Π_{2pass} is a $(\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}, \mathcal{F}_{\text{gzk}}, \mathcal{F}_{\text{gzk}}^{2\text{v}}, \mathcal{F}_{\text{osac}}, \mathcal{F}_{\text{ac}})$-hybrid protocol. Let $\Pi_{\text{2pass}}^{\mathcal{F}_{\text{gzk}} \to \pi}$ be the $(\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}, \mathcal{F}_{\text{crs}}^{\text{gzk}}, \mathcal{F}_{\text{osac}}, \mathcal{F}_{\text{ac}})$-hybrid protocol in which every instance of $\mathcal{F}_{\text{gzk}}$ and $\mathcal{F}_{\text{gzk}}^{2\text{v}}$ in Π_{2pass} has been replaced by the zero-knowledge protocol π described in Camenisch, Krenn, and Shoup's paper [CKS11]. To prove our scheme secure, we need to prove the following theorem:

Theorem 4.1. *Assuming the encryption scheme is CCA-2 secure and the DDH assumption holds for* ggen*, the protocol* $\Pi_{\text{2pass}}^{\mathcal{F}_{\text{gzk}} \to \pi}$ *realizes* $\mathcal{F}_{\text{2pass}}$*.*

To prove the theorem, we need to prove the following lemma:

Lemma 4.2. *Assuming the encryption scheme is CCA-2 secure and the DDH assumption holds for* ggen*: there exists a simulator* $\mathcal{S}$ *that does not extract the witnesses quantified by* $\exists$ *in any* $\mathcal{F}_{\text{gzk}}$ *and in any* $\mathcal{F}_{\text{gzk}}^{2\text{v}}$*, such that for all PPT* nice *environments* $\mathcal{E}$ *and the dummy adversary* $\mathcal{A}$*:* $\text{Exec}(\Pi_{\text{2pass}}, \mathcal{A}, \mathcal{E})$ *and* $\text{Exec}(\mathcal{F}_{\text{2pass}}, \mathcal{S}, \mathcal{E})$ *are computationally indistinguishable.*

In the above, we assume that $(\mathbb{G}, p, g)$ *are chosen by* $\text{ggen}(1^\eta)$ *before* $\mathcal{E}$ *starts executing.*

Proof of Lemma 4.2. In Section 4.4.2.1 we construct a simulator $\mathcal{S}$, and in Section 4.4.2.2 we prove that is satisfies the requirements of Lemma 4.2.

Proof of Theorem 4.1. Since we prove in Section 4.4.2.2 that the simulator $\mathcal{S}$ we construct in Section 4.4.2.1 satisfies the requirements of Lemma 4.2 and because Π_{2pass} is a $\mathcal{F}_{\text{gzk}}$-friendly protocol (see Definition 3.4), it follows from the special composition theorem of Camenisch et al. [CKS11] that Theorem 4.1 holds. (Note: this composition theorem was not proven for $\mathcal{F}_{\text{gzk}}^{\text{2v}}$, but it is easy to adapt their proof to handle that case.)

4.4.2.1 Construction of the Simulator

We now construct the simulator $\mathcal{S}$ needed for Lemma 4.2.

Notation and Modelling. We adopt the convention that the ideal functionalities in the $(\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}, \mathcal{F}_{\text{osac}}, \mathcal{F}_{\text{ac}}, \mathcal{F}_{\text{gzk}}, \mathcal{F}_{\text{gzk}}^{\text{2v}})$-hybrid "real" world (and which are controlled by $\mathcal{S}$) are surrounded by quotes: "$\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}$", "$\mathcal{F}_{\text{osac}}$", "$\mathcal{F}_{\text{ac}}$", "$\mathcal{F}_{\text{gzk}}$", "$\mathcal{F}_{\text{gzk}}^{\text{2v}}$". Note that $\mathcal{S}$ does not have to run these ideal functionalities honestly, it just needs to ensure that the messages it sends on their behalf are indistinguishable from an honest execution.

Simulator. The simulator $\mathcal{S}$ is an ten-interface system, with five external and five internal interfaces. We also use quotes to designate internal interfaces of $\mathcal{S}$. These interfaces are the $\mathcal{E}$-, $\mathcal{A}$-, $\mathcal{P}$-, $\mathcal{Q}$-, and $\mathcal{U}$-interfaces on one hand, and the "$\mathcal{E}$"-, "$\mathcal{A}$"-, "$\mathcal{P}$"-, "$\mathcal{Q}$"-, and "$\mathcal{U}$"-interfaces on the other hand. See Figure 4.11.

The simulator $\mathcal{S}$ runs one instance of the adversary $\mathcal{A}$ internally. $\mathcal{S}$ connects to the environment through its external $\mathcal{E}$-interface. It communicates with $\mathcal{F}_{\text{2pass}}$ through four external interfaces: the $\mathcal{A}$-, the $\mathcal{P}$-, the $\mathcal{Q}$-, and the (multiplexed) $\mathcal{U}$-interfaces. The $\mathcal{A}$-interface is connected to the network interface of $\mathcal{F}_{\text{2pass}}$, it is through this interface that $\mathcal{S}$ sends the messages in the role of the ideal adversary to $\mathcal{F}_{\text{2pass}}$ and expects to receive the messages destined to the ideal adversary. The latter three interfaces are connected to the ideal peer of the respective party; such an interface becomes active only when the corresponding party is corrupted. The simulator interacts with $\mathcal{A}$ through its five internal interfaces: the "$\mathcal{E}$"-, the "$\mathcal{A}$"-, the "$\mathcal{P}$"-, the "$\mathcal{Q}$"-, and the (multiplexed) "$\mathcal{U}$"-interfaces. Through the "$\mathcal{E}$" interface, $\mathcal{S}$ must simulate the messages from the environment. Through the "$\mathcal{A}$" interface, the simulator must simulate all traffic between $\mathcal{A}$ and the network interface of "$\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}$", "$\mathcal{F}_{\text{osac}}$", "$\mathcal{F}_{\text{ac}}$", "$\mathcal{F}_{\text{gzk}}$", and "$\mathcal{F}_{\text{gzk}}^{\text{2v}}$". Through the latter three interfaces, the simulator must simulate the ideal peers of the respective parties; similarly to above, such an interface becomes active only when the corresponding party is corrupted.

Ideal peers. Each ideal peer is a three-interface system. The IO-interface of the ideal peer is connected to the environment in the ideal world. $\mathcal{S}$ also simulates ideal peers for each of the ideal subroutines of Π_{2pass} for the sake of $\mathcal{A}$, the IO-interface of these ideal peers is then connected to Π_{2pass}. The subroutine-interface is connected to an ideal functionality. The network interface is connected to the adversary or the simulator. When the party corresponding to the ideal peer is honest, the ideal peer fowards all messages between the IO-interface to the subroutine-interface in both directions, i.e., the environment/protocol communicates directly with the ideal functionality. When the party is corrupted, the ideal peer forwards all messages from the IO-interface and

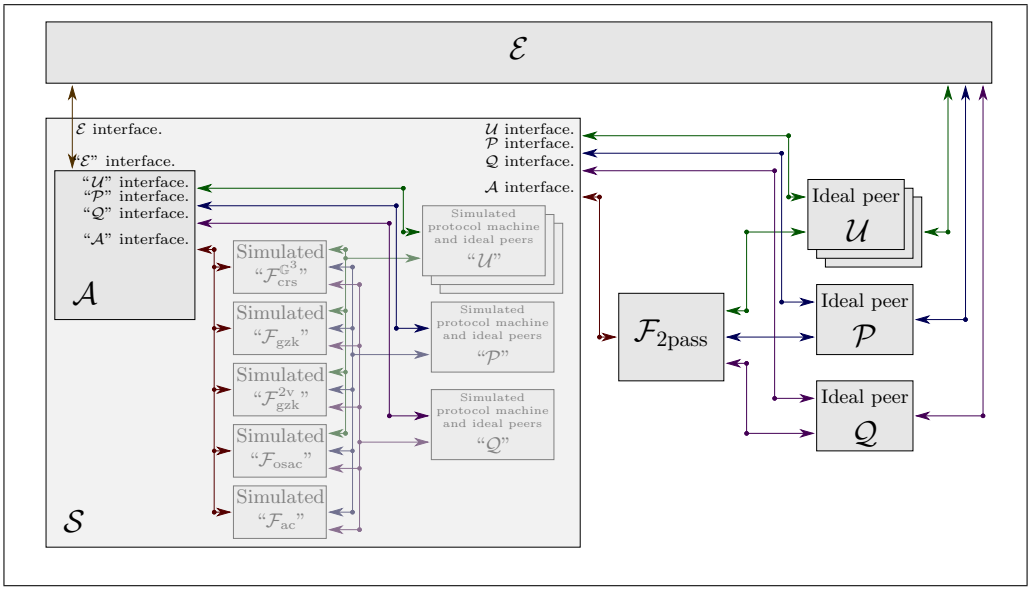


Fig. 4.11: The interfaces of the simulator $\mathcal{S}$.

the subroutine-interface to the network interface, and forwards all messages from the network interface to either the IO-interface or the subroutine-interface (we assume that there is some sort of header that indicates where the message must be routed to); i.e., the adversary/simulator has direct access to the ideal functionality, learns the input of the ideal peer, and provides the output.

When an ideal peer receives a special $\langle \text{Corrupt}, \ldots \rangle$ message from the subroutine-interface, it forwards this message on the IO-interface and considers itself corrupted. When a corrupted ideal peer receives a special $\langle \text{Recover}, \ldots \rangle$ message from the subroutine-interface, it forwards this message on the IO-interface and considers itself formally recovered.

Protocol machines. A protocol machine is a multi-interface system. The IO-interface of the protocol machine is connected to the environment or another protocol machine. Each of zero or more subroutine-interfaces is connected to an ideal peer or a protocol machine. The network interface is connected to the adversary. Protocol machines execute the code of honest parties. When they receive a special corrupt message from the network-interface they send a message to the IO-interface (and permit the adversary to query their internal state) ; thereafter they act as forwarders between the network interface and the IO-interface on the one hand, and between the network interface and the subroutine-interfaces on the other hand as for the ideal peers. For simplicity, we assume that the adversary corrupts all subroutines of a corrupted machine. The adversary may request to change the internal state of corrupted protocol machines through the network interface in case machines recover from corruption. When they receive a special recovery message from the network interface, they stop forwarding traffic for the adversary and resume normal operation; however the protocol machines

must use a new set of ideal peers and ideal functionalities in case the latter donot support recovery from corruption.

We now describe how to construct $\mathcal{S}$.

Environment interface. $\mathcal{S}$ forwards all messages between its $\mathcal{E}$-interface and its "$\mathcal{E}$"-interface in both directions, i.e., it relays all messages between $\mathcal{E}$ and $\mathcal{A}$.

Party interfaces. When a party is honest, no messages are sent through the party interfaces. When a party becomes corrupted, and after $\mathcal{S}$ has handed the (simulated) internal state of that party to $\mathcal{E}$, $\mathcal{S}$ relays all messages coming from the external interface (e.g., the $\mathcal{P}$-interface) to the internal corresponding interface (e.g., the "$\mathcal{P}$"-interface), and relays all messages from the internal interface destined for the environment to the external interface; this means that $\mathcal{A}$ receives the party's input and provides the party's output directly to $\mathcal{E}$.

Common reference string. Upon the first query to "$\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}$", $\mathcal{S}$ chooses a common reference (h, y, u) string with cgen'_0, so that $\mathcal{S}$ knows the trapdoor tc which will enable it to equivocate all commitments it makes on behalf of "$\mathcal{U}$", "$\mathcal{P}$" and "$\mathcal{Q}$".

General behavior of $\mathcal{S}$. In general, $\mathcal{S}$ simulates the ideal functionalities "$\mathcal{F}_{\text{ac}}$", "$\mathcal{F}_{\text{osac}}$", "$\mathcal{F}_{\text{gzk}}$", and "$\mathcal{F}_{\text{gzk}}^{2\text{v}}$" honestly, and simulates the ideal peers and protocol machines "$\mathcal{U}$", "$\mathcal{P}$", and "$\mathcal{Q}$" honestly. In fact, when $\mathcal{S}$ knows the correct input of the parties (which we can get either though the ideal peers in the ideal world or by extracting information from "$\mathcal{F}_{\text{gzk}}$"/"$\mathcal{F}_{\text{gzk}}^{2\text{v}}$"), it is easy to see how $\mathcal{S}$ proceeds. We emphasize that $\mathcal{S}$ never needs to send any input to "$\mathcal{F}_{\text{gzk}}$"/"$\mathcal{F}_{\text{gzk}}^{2\text{v}}$" on behalf of parties it controls, i.e., $\mathcal{S}$ can make proofs of false statements with "$\mathcal{F}_{\text{gzk}}$"/"$\mathcal{F}_{\text{gzk}}^{2\text{v}}$" or do proofs of knowledge even though it doesn't know the correct witnesses. In the remainder of this subsection, we will describe what $\mathcal{S}$ does when it doesn't know the correct inputs of the parties and is forced to lie.

Adjustments when $\mathcal{S}$ doesn't know the parties' input. When $\mathcal{S}$ does not know the input of some parties and must nevertheless produce output that depends on said input, $\mathcal{S}$ performs the following adjustments:

Setup.

- *Everybody honest:* $\mathcal{S}$ proceeds as if the user's input was random.
- *one server $\mathcal{R}$ corrupt ($\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$), others honest:* $\mathcal{S}$ proceeds as if the user's input was random.
- ($\mathcal{P}$ *and* $\mathcal{Q}$ *corrupt,* $\mathcal{U}$ *honest:* $\mathcal{S}$ learns the user's input by sending $\langle$ExposeSetup, $\cdots\rangle$ to $\mathcal{F}_{\text{2pass}}$.)
- ($\mathcal{U}$ *corrupt, others honest:* $\mathcal{S}$ runs Setup queries honestly. $\mathcal{S}$ recovers $\mathcal{U}$'s input during Share.)
- ($\mathcal{U}$ *and one server $\mathcal{R}$ corrupt ($\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$), other server honest:* $\mathcal{S}$ runs Setup queries honestly. In Share, $\mathcal{S}$ recovers one of $\mathcal{U}$'s shares directly, the other through "$\mathcal{F}_{\text{gzk}}$".)
- ($\mathcal{U}$, $\mathcal{P}$ *and* $\mathcal{Q}$ *corrupt:* The simulation is internal to the adversary.)

Retrieve. Upon receiving $\langle \text{Lock}, \cdots \rangle$ from $\mathcal{F}_{\text{2pass}}$, $\mathcal{S}$ knows whether $\delta = 0$ or not.

- *Everybody honest:* In general, $\mathcal{S}$ proceeds as if the user's input was random. $\mathcal{S}$ sends out random values $A_{\mathcal{U}}$, $B_{\mathcal{U}}$, $A_{\mathcal{P}}$, $B_{\mathcal{P}}$, and $A_{\mathcal{Q}}$. If $\delta = 0$, $\mathcal{S}$ sends out $B_{\mathcal{Q}} = g^0$, otherwise $\mathcal{S}$ sends out a random $B_{\mathcal{Q}}$.
- $\mathcal{P}$ *corrupt, others honest:* In general, $\mathcal{S}$ proceeds as if the user's input was random. $\mathcal{S}$ sends out random values $A_{\mathcal{U}}$, $B_{\mathcal{U}}$, and $A_{\mathcal{Q}}$. If $\delta = 0$, $\mathcal{S}$ sends out $B_{\mathcal{Q}} = g^0$, otherwise $\mathcal{S}$ sends out a random $B_{\mathcal{Q}}$.
- $\mathcal{Q}$ *corrupt, others honest:* In general, $\mathcal{S}$ proceeds as if the user's input was random. $\mathcal{S}$ sends out random values $A_{\mathcal{U}}$, $B_{\mathcal{U}}$, $A_{\mathcal{P}}$, and $A_{\mathcal{Q}}$. If $\delta = 0$, $\mathcal{S}$ sends out $B_{\mathcal{P}} = (A_{\mathcal{P}})^{s_{\mathcal{U}\mathcal{Q}} + s_{\mathcal{P}\mathcal{Q}} - op_{\mathcal{Q}}}$ on behalf of $\mathcal{P}$ (at this point, $\mathcal{S}$ knows the value of $op_{\mathcal{Q}}$ through "$\mathcal{F}_{\text{gzk}}[\cdots, 2]$" from Share in the Setup query or through "$\mathcal{F}_{\text{gzk}}[\cdots, 9]$" from ComRefr in the Refresh query), otherwise $\mathcal{S}$ sends out a random $B_{\mathcal{P}}$.
- $\mathcal{P}$ *and* $\mathcal{Q}$ *corrupt,* $\mathcal{U}$ *honest:* Upon receiving $\langle \text{Lock}, \cdots \rangle$ from $\mathcal{F}_{\text{2pass}}$, $\mathcal{S}$ knows whether $\delta = 0$ or not. $\mathcal{S}$ sends out a random value $A_{\mathcal{U}}$. If $\delta = 0$, $\mathcal{S}$ sends out $B_{\mathcal{U}} = (A_{\mathcal{U}})^{s_{\mathcal{U}\mathcal{P}} + s_{\mathcal{U}\mathcal{Q}} - op_{\mathcal{P}} - op_{\mathcal{Q}}}$ on behalf of $\mathcal{U}$ (at this point, $\mathcal{S}$ knows the value of $op_{\mathcal{P}}$ and $op_{\mathcal{Q}}$ because of the earlier "$\mathcal{F}_{\text{gzk}}[\cdots, 3]$" and "$\mathcal{F}_{\text{gzk}}[\cdots, 4]$" in ChkPwd), otherwise $\mathcal{S}$ sends out a random $B_{\mathcal{U}}$.
- $\mathcal{U}$ *corrupt, others honest:* $\mathcal{S}$ sends out random values $A_{\mathcal{P}}$, $B_{\mathcal{P}}$, and $A_{\mathcal{Q}}$. If $\delta = 0$, $\mathcal{S}$ sends out $B_{\mathcal{Q}} = g^0$, otherwise $\mathcal{S}$ sends out a random $B_{\mathcal{Q}}$.
- $\mathcal{U}$ *and* $\mathcal{P}$ *corrupt,* $\mathcal{Q}$ *honest:* $\mathcal{S}$ sends out a random value $A_{\mathcal{Q}}$. If $\delta = 0$, $\mathcal{S}$ sends out $B_{\mathcal{Q}} = g^0$, otherwise $\mathcal{S}$ sends out a random $B_{\mathcal{Q}}$.
- $\mathcal{U}$ *and* $\mathcal{Q}$ *corrupt,* $\mathcal{P}$ *honest:* $\mathcal{S}$ sends out a random value $A_{\mathcal{P}}$. If $\delta = 0$, $\mathcal{S}$ sends out $B_{\mathcal{P}} = (A_{\mathcal{P}})^{s_{\mathcal{U}\mathcal{Q}} + s_{\mathcal{P}\mathcal{Q}} - op_{\mathcal{Q}}}$ (at this point, $\mathcal{S}$ knows the value of $op_{\mathcal{Q}}$ through "$\mathcal{F}_{\text{gzk}}[\cdots, 2]$" from Share in the Setup query or through "$\mathcal{F}_{\text{gzk}}[\cdots, 9]$" from ComRefr in the Refresh query), otherwise $\mathcal{S}$ sends out a random $B_{\mathcal{P}}$.
- ($\mathcal{U}$, $\mathcal{P}$ *and* $\mathcal{Q}$ *corrupt:* The simulation is internal to the adversary.)

Refresh. $\mathcal{S}$ can run Refresh queries honestly, even if it doesn't know the correct value of the shares of the servers.

Adjustements upon corruption of $\mathcal{U}$. When a user $\mathcal{U}$ gets corrupted, $\mathcal{S}$ needs to perform the following adjustments, depending at which point in the simulation the corruption happened.

Share *after 1st Send of "*$\mathcal{F}_{\text{osac}}$*".* No adjustments needed, since most of the state was erased.

ChkPwd *after Lock of "*$\mathcal{F}^{\text{2v}}_{\text{gzk}}[\cdots, 5]$*".* No adjustments needed, since most of the state was erased.

Reconstr *after 1st Deliver of "*$\mathcal{F}_{\text{osac}}$*".* If not done already, immediately send $\langle$Deliver, $\cdots\rangle$ on the $\mathcal{U}$ interface to recover output.

- $\mathcal{P}$ *and* $\mathcal{Q}$ *honest:* adjust $k_{\mathcal{Q}}$ to match output of $\mathcal{U}$. $\mathcal{S}$ will need to adjust $ok_{\mathcal{Q}}$ (with help of trapdoor), and the OTP used to transmit it.
- $\mathcal{P}$ *corrupt,* $\mathcal{Q}$ *honest:* adjust $k_{\mathcal{Q}}$ to match output of $\mathcal{U}$. $\mathcal{S}$ will need to adjust $ok_{\mathcal{Q}}$ (with trapdoor), and the OTP used to transmit it.
- $\mathcal{P}$ *honest,* $\mathcal{Q}$ *corrupt:* adjust $k_{\mathcal{P}}$ to match output of $\mathcal{U}$. $\mathcal{S}$ will need to adjust $ok_{\mathcal{P}}$ (with trapdoor), and the OTP used to transmit it.

- $\mathcal{P}$ *and* $\mathcal{Q}$ *corrupt:* there is nothing to adjust.

We note that the values $k_{\mathcal{P}}$, $k_{\mathcal{Q}}$, $ok_{\mathcal{P}}$, and $ok_{\mathcal{Q}}$ are fixed after the corruption of the first user in the Retrieve query that succeeded in retrieving the key. When re-adjusting these values after a subsequent corruption of a user in the Retrieve query, the values will not change again.

Adjustments upon corruption of $\mathcal{P}$. When $\mathcal{P}$ gets corrupted, $\mathcal{S}$ needs to perform the following adjustments, depending at which point in the simulation the corruption happened.

Share after Deliver of "$\mathcal{F}_{\text{osac}}$*".*

- $\mathcal{U}$ *was honest during setup,* $\mathcal{U}$ *was always honest during retrieve,* $\mathcal{Q}$ *honest:* No adjustments needed.
- $\mathcal{U}$ *was honest during setup,* $\mathcal{U}$ *was always honest during retrieve,* $\mathcal{Q}$ *corrupt:* Adjust $p_{\mathcal{P}}$ and $k_{\mathcal{P}}$ to match $\mathcal{U}$'s input. $\mathcal{S}$ also needs to adjust $op_{\mathcal{P}}$ and $ok_{\mathcal{P}}$ (with help of trapdoor) and the OTPs used to transmit those.
- $\mathcal{U}$ *was honest during setup,* $\mathcal{U}$ *was corrupted at least once during retrieve,* $\mathcal{Q}$ *honest:* $\mathcal{S}$ needs to adjust the OTP used to transmit the shares of the key. The values $k_{\mathcal{P}}$ and $ok_{\mathcal{P}}$ were already adjusted when $\mathcal{U}$ was corrupted during Retrieve.
- $\mathcal{U}$ *was honest during setup,* $\mathcal{U}$ *was corrupted at least once during retrieve,* $\mathcal{Q}$ *corrupt:* Adjust $p_{\mathcal{P}}$ to match $\mathcal{U}$'s input. The values $k_{\mathcal{P}}$ and $ok_{\mathcal{P}}$ were already adjusted when $\mathcal{U}$ was corrupted during Retrieve. $\mathcal{S}$ also needs to adjust $op_{\mathcal{P}}$ (with help of trapdoor) and the OTPs used to transmit messages between $\mathcal{U}$ and $\mathcal{P}$.
- $\mathcal{U}$ *was corrupt during setup:* No adjustments needed.

ChkPwd after Deliver of "$\mathcal{F}_{\text{gzk}}^{2\text{v}}[\cdots, 5]$*".*

- $\mathcal{U}$ *and* $\mathcal{Q}$ *honest:* Nothing to adjust.
- $\mathcal{U}$ *corrupted:* Nothing to adjust.
- $\mathcal{U}$ *honest,* $\mathcal{Q}$ *corrupted:* If $\delta = 0$: adjust $s_{\mathcal{U}\mathcal{P}}$ so that $B_{\mathcal{U}} = (A_{\mathcal{U}})^{s_{\mathcal{U}\mathcal{P}} + s_{\mathcal{U}\mathcal{Q}} - op_{\mathcal{P}} - op_{\mathcal{Q}}}$. $\mathcal{S}$ also needs to adjust $os_{\mathcal{U}\mathcal{P}}$ (with trapdoor) and the OTP used to transmit it.

ChkPwd after Deliver of "$\mathcal{F}_{\text{gzk}}[\cdots, 6]$*".*

- $\mathcal{Q}$ *honest:* Nothing to adjust, since parts of the state were erased.
- $\mathcal{Q}$ *corrupted:* Nothing to adjust.

Reconstr after Send of "$\mathcal{F}_{\text{osac}}$*".*

- $\mathcal{U}$ *and* $\mathcal{Q}$ *honest:* Nothing to adjust.
- $\mathcal{U}$ *corrupted:* Nothing to adjust.
- $\mathcal{U}$ *honest,* $\mathcal{Q}$ *corrupted:* Adjust $k_{\mathcal{P}}$ to match correct output of $\mathcal{U}$. $\mathcal{S}$ will need to adjust $ok_{\mathcal{P}}$ (with trapdoor) and the OTP used to transmit it.

Adjustments upon corruption of $\mathcal{Q}$. When $\mathcal{Q}$ gets corrupted, $\mathcal{S}$ needs to perform the following adjustments, depending at which point in the simulation the corruption happened.

Share after Deliver of "$\mathcal{F}_{\text{osac}}$*".* Similar as for $\mathcal{P}$.

ChkPwd after Deliver of "$\mathcal{F}_{\text{gzk}}^{\text{2v}}[\cdots, 5]$".

- $\mathcal{U}$ *honest,* $\mathcal{P}$ *corrupted:* If $\delta = 0$: adjust $s_{\mathcal{UQ}}$ so that $B_{\mathcal{U}} = (A_{\mathcal{U}})^{s_{\mathcal{UP}}+s_{\mathcal{UQ}}-op_{\mathcal{P}}-op_{\mathcal{Q}}}$. $\mathcal{S}$ also needs to adjust $os_{\mathcal{UQ}}$ (with trapdoor) and the OTP used to transmit it.
- *Other cases:* Nothing to adjust.

ChkPwd after Deliver of "$\mathcal{F}_{\text{gzk}}[\cdots, 6]$".

- $\mathcal{P}$ *honest:* Nothing to adjust.
- $\mathcal{P}$ *corrupted:* If $\delta = 0$: adjust $s_{\mathcal{PQ}}$ so that $B_{\mathcal{P}} = (A_{\mathcal{P}})^{s_{\mathcal{UQ}}+s_{\mathcal{PQ}}-op_{\mathcal{Q}}}$. $\mathcal{S}$ also needs to adjust $os_{\mathcal{PQ}}$ (with trapdoor) and the OTP used to transmit it.

Reconstr after Send of "$\mathcal{F}_{\text{osac}}$". Similar as for $\mathcal{P}$.

4.4.2.2 Proof of Indistinguishability

We are going to define a sequence of games Game_1 to $\mathsf{Game}_{N_{\text{games}}}$, as described by Shoup [Sho04]. In the first game, everything is distributed as in the protocol Π_{2pass}, whereas in the last game everything is distributed as in the ideal world $\mathcal{F}_{\text{2pass}}$. By the piling-up lemma, the advantage of $\mathcal{E}$ is less than the sum of the advantages in distinguishing between Game_i and Game_{i+1}. We are going to prove that $\mathcal{E}$ only has negligible advantage in distinguishing between two consecutive games, based either on a reduction to a hard cryptographic problem, or by "failure events" happening with negligible probability. As long as the number of games is polynomial w.r.t. the security parameter, the total advantage of $\mathcal{E}$ is negligible.

We stress that in all intermediate games, $\mathcal{E}$ and $\mathcal{A}$ interact with a machine that runs both $\mathcal{F}_{\text{2pass}}$ and $\mathcal{S}_i$. Without loss of generality, we assume that $\mathcal{S}_i$ thus obtains the inputs and outputs of all honest parties from $\mathcal{F}_{\text{2pass}}$. It is only in the last game, which is identical to the "ideal world" and where $\mathcal{S}$ is equal to $\mathcal{S}_i$, that $\mathcal{S}_i$ does not make use of these inputs and outputs.

Game_1. As observed in the previous paragraph, $\mathcal{S}_1$ receives the input of all honest parties. $\mathcal{S}_1$ runs all parties honestly, and runs "$\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}$", "$\mathcal{F}_{\text{osac}}$", "$\mathcal{F}_{\text{ac}}$", "$\mathcal{F}_{\text{gzk}}$", and "$\mathcal{F}_{\text{gzk}}^{\text{2v}}$" honestly. By construction, this setting is perfectly indistinguishable from the $(\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}, \mathcal{F}_{\text{osac}}, \mathcal{F}_{\text{ac}}, \mathcal{F}_{\text{gzk}}, \mathcal{F}_{\text{gzk}}^{\text{2v}})$-hybrid *real* world Π_{2pass}.

Game_2. $\mathcal{S}_2$ runs like $\mathcal{S}_1$, except that it aborts if the adversary manages to open a commitment to two different values. $\mathcal{S}_2$ detects that case when:

- A corrupt $\mathcal{P}$ uses a different share $p_{\mathcal{P}}$ or $k_{\mathcal{P}}$ in any "$\mathcal{F}_{\text{gzk}}$" with a honest party, than the value it received from honest $\mathcal{U}$.
- A corrupt $\mathcal{P}$ uses a different values of shares $p_{\mathcal{P}}$ or $k_{\mathcal{P}}$ in subsequent runs of "$\mathcal{F}_{\text{gzk}}$".
- A corrupt $\mathcal{Q}$ uses a different share $p_{\mathcal{Q}}$ or $k_{\mathcal{Q}}$ in any "$\mathcal{F}_{\text{gzk}}$" with a honest party, than the value it received from honest $\mathcal{U}$.
- A corrupt $\mathcal{Q}$ uses a different values of shares $p_{\mathcal{Q}}$ or $k_{\mathcal{Q}}$ in subsequent runs of "$\mathcal{F}_{\text{gzk}}$".
- A corrupt $\mathcal{U}$ sends a value $B_{\mathcal{U}}$ in "$\mathcal{F}_{\text{gzk}}^{\text{2v}}[\cdots, 5]$" to an honest $\mathcal{P}$ or $\mathcal{Q}$ that is incompatible with $s_{\mathcal{UP}}$ or $s_{\mathcal{UQ}}$, respectively. (Here $\mathcal{S}_2$ doesn't need to extract those values, it can simply decrypt $B_{\mathcal{U}}$ and see if δ is equal or not equal to zero as expected.)

- A corrupt $\mathcal{P}$ sends a value $B_{\mathcal{P}}$ to honest $\mathcal{Q}$ in "$\mathcal{F}_{\text{gzk}}[\cdots, 6]$" that is incompatible with $s_{\mathcal{UP}}$ or $s_{\mathcal{PQ}}$ or $p_{\mathcal{P}}$. (Here $\mathcal{S}_2$ doesn't need to extract those values, it can simply decrypt $B_{\mathcal{P}}$ and see if δ is equal or not equal to zero as expected.)
- A corrupt $\mathcal{Q}$ sends a value $B_{\mathcal{Q}}$ to an honest $\mathcal{P}$ in "$\mathcal{F}_{\text{gzk}}[\cdots, 7]$" that is incompatible with $s_{\mathcal{UQ}}$ or $s_{\mathcal{PQ}}$ or $p_{\mathcal{Q}}$. (Here $\mathcal{S}_2$ doesn't need to extract those values, it can simply see if δ is equal or not equal to zero as expected.)

The probability that $\mathcal{S}_2$ aborts is at most the probability that the commitment was not binding after all (recall that the ideal functionalities $\mathcal{F}_{\text{gzk}}$ and $\mathcal{F}_{\text{gzk}}^{2\text{v}}$ provide perfect soundness), which is negligible.

Game_3. $\mathcal{S}_3$ runs like $\mathcal{S}_2$, except that when $\mathsf{secureSend}$ is run between two honest parties, the parties use a different one-time-pad than the one that was encrypted in $e_{\mathcal{T}}$. Recall that both honest parties have deleted the randomness and decryption key for that ciphertext by the time the one-time-pad is first put into use.

The advantage of $\mathcal{E}$ in distinguishing between Game_3 and Game_2 is at most the advantage of a polynomial-time environment in the CCA-2 security game of the cryptosystem, which is negligible.

Game_4. $\mathcal{S}_4$ runs like $\mathcal{S}_3$, except that when ChkPwd is run by an honest $\mathcal{U}$ and whenever $\delta \neq 0$, $\mathcal{S}_4$ chooses $A_{\mathcal{U}}$ and $B_{\mathcal{U}}$ at random from $\mathbb{G}$. $\mathcal{S}_4$ will make proofs of false statements with $\mathcal{F}_{\text{gzk}}^{2\text{v}}$.

We now argue that the advantage that $\mathcal{E}$ has in distinguishing between Game_4 and Game_3 is negligible under the DDH assumption. We do this by running a hybrid arguments over all retrieve queries.

In the hybrid j, the simulator $\mathcal{S}_{3,j}$ behaves like $\mathcal{S}_4$ for the first j queries, and like $\mathcal{S}_3$ for the following queries.

We now construct a distinguisher $\mathcal{S}$ that operates with an environment which tries to distinguish between hybrid $j-1$ and j: the simulator $\mathcal{S}$ gets a DDH challenge $(\mathbb{G}, p, g, g', g'', g''')$ and chooses $w \xleftarrow{\$} \mathbb{Z}_p$ and sets $h \leftarrow g^w, y \leftarrow (g')^w, Y \leftarrow (g'')^w, Z \leftarrow (g''')^w$. I.e., either

- (h, y, Y, Z) are independent random elements of $\mathbb{G}$ (*left* setting); or
- (h, y, Y, Z) form a DDH tuple (*right* setting), i.e. $Y = h^{r_{\mathcal{U}}}$ and $Z = y^{r_{\mathcal{U}}}$ for some $r_{\mathcal{U}}$.

$\mathcal{S}$ chooses u at random, and sets the CRS to (h, y, u). In the first $j-1$ Retrieve queries, $\mathcal{S}$ behaves like $\mathcal{S}_4$. In queries $j+1$ and following, $\mathcal{S}$ behaves like $\mathcal{S}_3$. In jth query, whenever $\delta \neq 0$ and whenever $\mathcal{U}$ is honest, $\mathcal{S}$ recovers $op_{\mathcal{P}}$ from "$\mathcal{F}_{\text{gzk}}[\cdots, 3]$" if needed, recovers $op_{\mathcal{Q}}$ from "$\mathcal{F}_{\text{gzk}}[\cdots, 4]$" if needed, sets $A_{\mathcal{U}} \leftarrow Y^{-1}$, sets $B_{\mathcal{U}} \leftarrow Z^{\delta}(A_{\mathcal{U}})^{s_{\mathcal{UP}}+s_{\mathcal{UQ}}-op_{\mathcal{P}}-op_{\mathcal{Q}}}$, and makes a proof of a false statement in the subsequent $\mathcal{F}_{\text{gzk}}$. $\mathcal{S}$ then outputs whatever the environment outputs. Notice how in the *left* setting, the distribution of all values is like in hybrid j, and in the *right* setting, the distribution of all values is like in the hybrid $j-1$. Also note that since $r_{\mathcal{U}}$ is erased before the last message of "$\mathcal{F}_{\text{gzk}}^{2\text{v}}[\cdots, 5]$", $\mathcal{S}_4$ will not get into trouble if $\mathcal{U}$ is corrupted.

The advantage of $\mathcal{S}$ in the DDH-game is therefore equal or better than the advantage of the environment in distinguishing between the two hybrids. The former being negligible by assumption, the latter must also be negligible. Since the number of queries

is polynomial, the overall advantage of the environment in distinguishing between Game_4 and Game_3 is negligible.

Game_5. $\mathcal{S}_5$ runs like $\mathcal{S}_4$, except that when ChkPwd is run by an honest $\mathcal{P}$ and whenever $\delta \neq 0$, $\mathcal{S}_5$ chooses $A_\mathcal{P}$ and $B_\mathcal{P}$ at random from $\mathbb{G}$. $\mathcal{S}_5$ will make proofs of false statements with $\mathcal{F}_{\text{gzk}}$.

We now argue that the advantage that $\mathcal{E}$ has in distinguishing between Game_5 and Game_4 is negligible under the DDH assumption. We do this by running a hybrid arguments over all retrieve queries.

In the hybrid j, the simulator $\mathcal{S}_{4,j}$ behaves like $\mathcal{S}_5$ for the first j queries, and like $\mathcal{S}_4$ for the following queries.

We now construct a distinguisher $\mathcal{S}$ that operates with an environment which tries to distinguish between hybrid $j-1$ and j: the simulator $\mathcal{S}$ gets a DDH challenge $(\mathbb{G}, p, g, g', g'', g''')$ and chooses $w \xleftarrow{\$} \mathbb{Z}_p$ and sets $h \leftarrow g^w, y \leftarrow (g')^w, Y \leftarrow (g'')^w, Z \leftarrow (g''')^w$. I.e., either

- (h, y, Y, Z) are independent random elements of $\mathbb{G}$ (*left* setting); or
- (h, y, Y, Z) form a DDH tuple (*right* setting), i.e. $Y = h^{r_\mathcal{U} \cdot r_\mathcal{P}}$ and $Z = y^{r_\mathcal{U} \cdot r_\mathcal{P}}$ for some $(r_\mathcal{U} \cdot r_\mathcal{P})$.

$\mathcal{S}$ chooses u at random, and sets the CRS to (h, y, u). In the first $j-1$ Retrieve queries, $\mathcal{S}$ behaves like $\mathcal{S}_5$. In queries $j+1$ and following, $\mathcal{S}$ behaves like $\mathcal{S}_4$. In jth query, whenever $\delta \neq 0$ and whenever $\mathcal{P}$ is honest, $\mathcal{S}$ recovers $s_{\mathcal{U}\mathcal{Q}}$ from "$\mathcal{F}^{2\text{v}}_{\text{gzk}}[\cdots, 5]$" if needed, ($op_\mathcal{Q}$ was recovered in "$\mathcal{F}_{\text{gzk}}[\cdots, 2]$" in Share during Setup or in "$\mathcal{F}_{\text{gzk}}[\cdots, 9]$" in ComRefr during Refresh) sets $A_\mathcal{P} \leftarrow Y^{-1}$, sets $B_\mathcal{P} \leftarrow Z^\delta (A_\mathcal{P})^{s_{\mathcal{P}\mathcal{Q}} + s_{\mathcal{U}\mathcal{Q}} - op_\mathcal{Q}}$, and makes a proof of a false statement in the subsequent $\mathcal{F}_{\text{gzk}}$. $\mathcal{S}$ then outputs whatever the environment outputs. Notice how in the *left* setting, the distribution of all values is like in hybrid j, and in the *right* setting, the distribution of all values is like in the hybrid $j-1$. Also note that since $r_\mathcal{P}$ is erased before the last message of "$\mathcal{F}_{\text{gzk}}[\cdots, 6]$", $\mathcal{S}_5$ will not get into trouble if $\mathcal{P}$ is corrupted.

The advantage of $\mathcal{S}$ in the DDH-game is therefore equal or better than the advantage of the environment in distinguishing between the two hybrids. The former being negligible by assumption, the latter must also be negligible. Since the number of queries is polynomial, the overall advantage of the environment in distinguishing between Game_5 and Game_4 is negligible.

Game_6. $\mathcal{S}_6$ runs like $\mathcal{S}_5$, except that when ChkPwd is run by an honest $\mathcal{Q}$ and whenever $\delta \neq 0$, $\mathcal{S}_6$ chooses $A_\mathcal{Q}$ and $B_\mathcal{Q}$ at random from $\mathbb{G}$. $\mathcal{S}_6$ will make proofs of false statements with $\mathcal{F}_{\text{gzk}}$.

We now argue that the advantage that $\mathcal{E}$ has in distinguishing between Game_6 and Game_5 is negligible under the DDH assumption. We do this by running a hybrid arguments over all retrieve queries.

In the hybrid j, the simulator $\mathcal{S}_{5,j}$ behaves like $\mathcal{S}_6$ for the first j queries, and like $\mathcal{S}_5$ for the following queries.

We now construct a distinguisher $\mathcal{S}$ that operates with an environment which tries to distinguish between hybrid $j-1$ and j: the simulator $\mathcal{S}$ gets a DDH challenge $(\mathbb{G}, p, g, g', g'', g''')$ and chooses $w \xleftarrow{\$} \mathbb{Z}_p$ and sets $h \leftarrow g^w, y \leftarrow (g')^w, Y \leftarrow (g'')^w, Z \leftarrow (g''')^w$. I.e., either

- (h, y, Y, Z) are independent random elements of $\mathbb{G}$ (*left* setting); or
- (h, y, Y, Z) form a DDH tuple (*right* setting), i.e. $Y = h^{r_{\mathcal{U}} \cdot r_{\mathcal{P}} \cdot r_{\mathcal{Q}}}$ and $Z = y^{r_{\mathcal{U}} \cdot r_{\mathcal{P}} \cdot r_{\mathcal{Q}}}$ for some $(r_{\mathcal{U}} \cdot r_{\mathcal{P}} \cdot r_{\mathcal{Q}})$.

$\mathcal{S}$ chooses u at random, and sets the CRS to (h, y, u). In the first $j-1$ Retrieve queries, $\mathcal{S}$ behaves like $\mathcal{S}_6$. In queries $j+1$ and following, $\mathcal{S}$ behaves like $\mathcal{S}_5$. In jth query, whenever $\delta \neq 0$ and whenever $\mathcal{Q}$ is honest, $\mathcal{S}$ sets $A_{\mathcal{Q}} \leftarrow Y^{-1}$, sets $B_{\mathcal{Q}} \leftarrow Z^{\delta}$, and makes a proof of a false statement in the subsequent $\mathcal{F}_{\text{gzk}}$. $\mathcal{S}$ then outputs whatever the environment outputs. Notice how in the *left* setting, the distribution of all values is like in hybrid j, and in the *right* setting, the distribution of all values is like in the hybrid $j-1$. Also note that since $r_{\mathcal{Q}}$ is erased before the last message of "$\mathcal{F}_{\text{gzk}}[\cdots, 7]$", $\mathcal{S}_6$ will not get into trouble if $\mathcal{Q}$ is corrupted.

The advantage of $\mathcal{S}$ in the DDH-game is therefore equal or better than the advantage of the environment in distinguishing between the two hybrids. The former being negligible by assumption, the latter must also be negligible. Since the number of queries is polynomial, the overall advantage of the environment in distinguishing between Game_6 and Game_5 is negligible.

Game_7. $\mathcal{S}_7$ runs like $\mathcal{S}_6$, except that now it chooses the CRS with CRSGen'_0 instead of CRSGen_1 upon the first query to "$\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}$". The commitment scheme is now perfectly hiding, and $\mathcal{S}_7$ can now efficiently equivocate commitments using the trapdoor information.

The advantage that $\mathcal{E}$ has in distinguishing between Game_7 and Game_6 is equal to its advantage in breaking the security of the HMT commitment scheme, which is negligible as DDH assumption holds for ggen by assumption.

Game_8. $\mathcal{S}_8$ runs like $\mathcal{S}$ described in Section 4.4.2.1.

This is a purely conceptual change, so $\mathcal{E}$ has no advantage in distinguishing between Game_8 and Game_7 .

5

Memory Erasability Amplification

Persistent and erasable memory is a crucial ingredient of many practical cryptographic protocols that are secure against adaptive adversaries. However, for storage devices such as solid state disks, hard disks, and tapes it is rather difficult to truly erase information written on them. Therefore, constructions have been proposed that use a small amount of memory that is easier to erase (or at least harder for an attacker to tap into), such as smart cards and processor registers, to store a cryptographic key, and then to encrypt the data to be stored so that it no longer matters whether or not the ciphertext can be erased [Gut96, JL00, RCB12, RRBC13, RBC13, Yee94, YT95]. This approach is sometimes referred to as crypto paging. Surprisingly, no formal model of erasable memory has been proposed to date, despite of the importance of erasable memory for cryptographic protocol design and the cryptographic constructions for it.

Contributions. In this chapter we rectify this and first model erasable memory as a general resource in the constructive cryptography framework [MR11, Mau11a]. Our memory resource defines how a user, an adversary, and the environment can interact with the resource and to what extent stored data can be erased. In particular, different memory resources are characterized by what information about the stored data an adversary will be able to obtain when the environment allows it access to the memory resource. As we discuss, this allows one to model many different types of memory such as hard disks, solid state drives, RAM, and smart cards. Next, we study different constructions of erasable memory from one with weaker erasability properties or, in other words, constructions that amplify erasability. These constructions also show how memory resources can be used in protocol design and analysis. We then study the approach of crypto paging in our setting, i.e., constructions of a large erasable memory from a small one and a non-erasable memory. As it turns out, achieving the strongest possible type of erasable memory with this approach requires non-committing encryption and hence is only possible in the random oracle model. We also show what kind of erasable memory can be achieved with this approach in the standard model.

One of our memory constructions employs All-or-Nothing Transforms (AoNT) [CDH+00] to obtain a perfectly erasable memory from one that leaks a constant fraction of the erased data. Motivated by this protocol, we study AoNTs and propose several new transforms that enjoy better parameters than previously known ones, a result that

may be of independent interest. For example, we improve the standard construction of a perfectly-secure AoNT from a Linear Block Code (LBC), by observing that an LBC with a large minimum distance does not yield an AoNT with optimal privacy threshold. We propose the metric of *ramp minimum distance* and show that LBCs optimized for this metric yield perfectly secure AoNTs with better parameters than what can be achieved with the standard construction. We further propose a computationally secure AoNT that operates over a large alphabet (large enough for one symbol to encode a cryptographic key) and that is optimal: the encoded data is just one symbol longer than the original data, and the transform is secure even if all but one of the symbols of the encoded data leak. We show that such an AoNT can be realized from a pseudo-random generator (PRG) with some specific properties.

Related Work. In most security frameworks, unlimited and perfectly erasable memory is available to protocols as part of the framework, with the exception of protocols that are proven to be adaptively secure in the non-erasure model, where no erasable memory is available. However, as mentioned already, no security framework explicitly models memory and consequently security proofs treat the adversary's access to the memory of a compromised party informally only. The only exception to this is the work by Canetti et al. and Lim [CEGL08a, Lim08], who model memory as special tapes of the parties' Turing machines and define how an adversary can access these special tapes. This very specific modelling therefore changes the machine model underlying the UC framework.

Hazay et al. [HLP15] follow a different approach. They introduce the concept of *adaptive security with partial erasures*, where security holds if at least one party of a given protocol can successfully erase. Their model requires a special protocol design and has some restrictions regarding composition.

Both these approaches are rather limited. Indeed, if one wanted to consider different types of memory, one would have to change the modelling framework each time and potentially have to prove all composition theorems all over again. Moreover, these approaches do not allow one to analyse protocols that construct one type of memory from another type of memory, as we do in this chapter. Indeed, one cannot analyse the security of protocols such as Yee's crypto-paging technique [Yee94, YT95] and the constructions of Di Crescenzo et al. [DCFIJ99]. In contrast, we model memory as a resource (or ideal functionality) *within* the security framework (the constructive cryptography framework in our case) and thus do not suffer from these limitations.

5.1 Modelling Imperfectly Erasable Memory

We now present our erasable memory resource. Recall that we aim to model memory that is used for persistent storage (such as hard disks, solid state drives, RAM, and smart cards), and not processor registers that store temporary values during computations. To this end, we define how the resource behaves upon inputs on the user, the adversary, and the world interfaces. It allows a user *Alice* to store a single data item *once*, retrieve it (many times), and erase it. The adversary can get access to the data only if such access is enabled on the *World*-interface. That is, the data stored is not initially available to her. Then, once access is enabled via a *weaken* input on the *World*-interface, the adversary can either *read* the data item stored (if the user has not yet deleted it) or

The resource $\mathsf{M}\langle\Sigma, \psi, \rho, \kappa\rangle$:

Internal state and initial values: DATA $= \bot$, LDAT $= \bot$, HIST $= ()$.

Behavior:

- *Alice*(`store`, $\mu \in \Sigma$): if DATA $= \bot$: DATA $\leftarrow \mu$; LDAT $\xleftarrow{\$} \psi(\mu)$; *Alice* $\leftarrow ()$.
- *Alice*(`retrieve`): if "`e`" $\notin$ HIST: *Alice* $\leftarrow$ DATA.
- *Alice*(`erase`): if "`e`" $\notin$ HIST $\wedge$ DATA $\neq \bot$: HIST $\leftarrow$ (HIST, "`e`"); *Alice* $\leftarrow ()$.
- *Eve*(`gethist`): *Eve* $\leftarrow$ HIST.
- *Eve*(`read`): if ρ(HIST): *Eve* $\leftarrow$ DATA.
- *Eve*(`leak`, ξ): if κ(HIST, ξ): HIST $\leftarrow$ (HIST, "`l`"$||\xi$); *Eve* $\leftarrow \xi$(LDAT).
- *World*(`weaken`, w): if ("`w`"$||w) \notin$ HIST: HIST $\leftarrow$ (HIST, "`w`"$||w$); *World* $\leftarrow ()$.

Fig. 5.1: The general imperfectly erasable memory resource $\mathsf{M}\langle\cdot\rangle$.

leak the data, meaning that she will obtain as answer a function of the once stored data. This function determines the information that is still *leaked* although the data has been deleted. The adversary can influence the leakage by providing an additional input to the function (e.g., specify some bits that are leaked).

In reality, there might be many different reason why an adversary gains access to the contents of a memory. This might be because the memory device is lost, the adversary controlling some malware on the computer that uses the memory, or the adversary running a cache-timing attack [Ber05] on the computer, etc. Offering a *World*-interface via which it is determined what access is given to the adversary by the memory resource, models any such event. The UC and GNUC frameworks use a similar mechanism for corrupting parties, except that they (ab)use the party interfaces to do so. In UC, it is the adversary who corrupts and the environment is informed of the corruption through the party interfaces. In GNUC, the environment corrupts parties and the adversary is informed thereof.

There seem to be two natural extensions to our erasable memory resource which for simplicity we chose not to consider. First, we assume that inputs at the *World*-interface do not impact the user's ability to access the data, which might often not be the case. Although this would not be hard to model, it is not important for the scope of this chapter. Second, the user cannot change the stored data or store many different data items, Again, while it would not be hard to extend the resource to allow for that, we choose not to do that for simplicity. Also, this is not a serious restrictions as such requirements can also be addressed by using several instances of our memory resources.

5.1.1 Specification of the General Imperfectly Erasable Memory Resource

We now present our formal specification of the general resource for imperfectly erasable memory $\mathsf{M}\langle\Sigma, \psi, \rho, \kappa\rangle$ that is given in Figure 5.1 and then discuss in the next subsection a few instantiations of this general resource that match different types of memory. The resource maintains three variables DATA, LDAT, and HIST. The first one stores the data provided by the user, the second the data that can potentially be leaked to the adversary, and the third one logs the history of events, namely the erasure event, the parameter of each call on the *World* interface, and the input arguments of each successful leakage query. The resource is parametrized by an alphabet Σ, a conditional probability

distribution ψ, and two predicates ρ and κ. The alphabet Σ is the set of possible values that can be stored. The conditional distribution ψ operates on the data and determines what information could potentially leak to the adversary by outputting LDAT. This models the extent to which the resource is able to erase the data. The predicate ρ takes as input the history of the resource and determines whether or not the adversary is allowed to read the memory. Finally, the predicate κ takes as input the history of the resource and the deterministic function ξ submitted by the adversary and determines whether or not the adversary obtains the leakage $\xi(\text{LDAT})$.

Most of the commands that can be submitted to the resource and its behaviour should now be clear from Figure 5.1, however, a few details merit explanation. First, the data that is potentially leaked, LDAT, is determined using ψ already when the data is stored in the resource. This is without loss of generalty but is here useful because, depending on the predicate κ, the adversary may query the resource multiple times with the `leak` command and the answers to these commands need to be consistent. Second, when the adversary queries the resource with a leak command, she can input a parameter ξ that may influence the leakage she obtains. This models the process of an adversary reading the erased data from a memory device, e.g., an adversary might try to read the data bit by bit, each time influencing the remaining bits in the memory. Third, the adversary is allowed to obtain the history from the resource at any time. This is necessary so that a simulator has enough information to properly simulate a construction. Finally, the *World*-interface accepts any value w for an external event, because these depend on the particular resource that is modelled and possibly on how it is constructed. This will become clear later when we discuss constructions of one type of memory from other types in Section 5.2.

5.1.2 Instantiations of $\mathsf{M}\langle\Sigma, \psi, \rho, \kappa\rangle$

We now describe special cases of the $\mathsf{M}\langle\Sigma, \psi, \rho, \kappa\rangle$ resource that correspond to memory devices appearing in the real world. We start by describing non-erasable memory, i.e., memory that becomes readable by the adversary once access is enabled by the *World*-interface. This models what happens in a typical file system: files that are unlinked are not actually erased and can often be completely recovered with specialized tools (at least until the blocks are re-used). We then describe perfectly erasable memory. Such a memory could be implemented by specialized hardware, such as smartcards, but often will have only limited capacity. Large perfectly erasable memories are often not directly available in reality. We are thus interested in the construction of such memories from resources with lesser guarantees. Each of the latter can be influenced through *World*-events separately, hence we will describe both a variant of the perfectly erasable memory that accepts a single type of *World*-event (easier to describe) and a variant that accepts an arbitrary number of events. Finally we describe imperfectly erasable memories, i.e., memories with security guarantees between the two extremes just discussed. Such memories leak partial information if the adversary is granted access by *World after* an erasure. In reality, often not all the data is actually removed during an erasure: on magnetic storage, overwritten data can still be partially recovered with specialized equipment [Gut96]. Similarly, often parts of the data were copied to a different medium (swap space, backup, file system journal, etc.) before the erasure

and the copies were not fully erased themselves. One can thus easily imagine that the adversary can deduce a constant number of bits that were stored, or obtains a noisy version of the data that was stored. For simplicity, we consider imperfectly erasable memories which ignore the parameter of `weaken` (only a single *World*-event can be modelled), and only leak once (no adaptive leakage).

Non-erasable Memory. To model non-erasable memory, we let ρ return true if `weaken` was called irrespective of `erase`. (In fact, the `erase` command could be dropped entirely.) The memory does not leak, hence κ always returns false and ψ is irrelevant. The only relevant parameter is the alphabet Σ and thus we denote this resource by $\mathsf{NM}\langle\Sigma\rangle$.

Perfectly Erasable Memory. To model perfectly erasable memory, we let ρ return true only if `weaken` was called (perhaps multiple times with specific parameters) before `erase` was called.[1] This memory does not leak, hence κ always returns false and ψ is irrelevant. We describe two versions of the resource: $\mathsf{PM}\langle\Sigma\rangle$ fixes ρ to return true if `weaken` appears in the history earlier than or without `erase`, hence only a single *World*-event can be modelled. $\mathsf{PMM}\langle\Sigma,\rho\rangle$ lets one specify a custom ρ, allowing the modelling of many *World*-events. Figure 5.5 in the next section shows examples of ρ in the case where there are two relevant *World*-events.

Imperfectly Erasable Memory. To model imperfectly erasable memory, we fix ρ and split κ into two predicates: a fixed predicate that checks only the history and a freely specifiable predicate Ξ that checks only the adversary's choice ξ. The other parameters Σ and ψ can be freely specified. We denote this resource by $\mathsf{IM}\langle\Sigma,\psi,\Xi\rangle$. The predicate ρ returns true only if the first recorded event in the history is a `weaken` command (as opposed to an `erase` command). The fixed predicate returns true if the first two recorded events in the history are an `erase` command followed by a `weaken` command (if `weaken` was called first, the adversary should call `read` and not `leak`), and no `leak` query succeeded previously. The predicate κ returns true if the fixed predicate does so and Ξ accepts ξ. In the next section, when we discuss erasability amplification, we further specialize this resource.

5.2 Constructing Better Memory Resources

In this section we consider constructions of memory resources with stronger security properties from memory resources with weaker ones. We start by showing how to use our memory resources in protocol constructions and then explain the issues that arise when doing so. Thereafter, we describe several specializations of the imperfectly erasable memory resource $\mathsf{IM}\langle\cdot\rangle$ presented in the previous section and then show how to construct memory resources with stronger properties from ones with weaker properties.

[1] In this chapter, we chose to consider monotone ρ's. We chose to model the memory resource in such a way that it only responds on the same interface it was activated, hence it is not possible for the adversary to be notified of an event that causes the memory to become readable. To simplify the modelling of simulators, we consider the adversary to be eager and trying to read the memory as soon as possible and then placing the resulting data in an "intermediate buffer" that can then be collected though the *Eve*-interface at a later point.

For example, we show how to construct perfectly erasable memories from memories that leak a certain number of bits. Finally, we consider the construction of a large perfectly erasable memory from a small one plus a large non-erasable memory.

5.2.1 Admissible Converters for Constructions using Erasable Memory

As stated previously, one of our reasons to model memory is to be able to analyse cryptographic protocols where the adversary at some point obtains access to the memory. This means that one needs to restrict converters to use only our memory resources for storage. Assuming that an adversary in a real environment may typically not be able to get access to processor registers, we still allow a converter to store temporary values locally and use a memory resource only for persistent storage. Let us now formalize the distinction between persistent and temporary storage and the restrictions we put on converters.

The computation done by a converter is divided in *computation phases*. A phase starts when a converter is activated outside of a computation phase. Informally, a phase ends as soon as the converter responds to that activation or makes a request that is not guaranteed to be answered immediately, i.e., where there is a chance that the adversary is activated before the request completes. For example, a computation phase ends if the converter makes a request that goes over an unreliable communication network, but does not end if the converter asks to store or retrieve data from a memory resource.

In this chapter, all our resources always respond on the same interface they were activated. It is then easy to define a computation phase of a converter: the phase starts as soon as the converter's outer interface is activated, and stops as soon as the converter writes on its outer interface. That is, activations of the inner interface do not interrupt the phase. However, in a more general setting, resources may respond on a different interface than the one they were activated on, and thereby activate a different party or the adversary. The definition of computation phase of converters must therefore be adjusted to take this into account.

State that is discarded at the end of a computation phase is *temporary*. State that must persist between two or more computation phases is *persistent*. (Converters must keep all persistent state in memory resources.) This distinction ensures that whenever the adversary has control, the entire internal state of a protocol is inside memory resources, and thus subject to attack.

Discussion. Other models, notably Canetti et al. and Lim [CEGL08a, Lim08], also make a distinction between storage needed during computation and persistent storage. However they do it in a way that does not cleanly separate the various layers of abstraction: they assume the existence of a constant number of "processor registers" that are perfectly erasable and place no restriction on the amount of time that data can remain in such a register. For example, their model therefore does not exclude reserving a part of the CPU registers to permanently store a cryptographic key, and use a crypto paging technique [Yee94, YT95] to have as much (computationally secure) perfectly erasable memory as required. Thus, to ensure a meaningful analysis, a similar restriction would have to be used in their approach.

5.2.2 Memory Erasability Amplification

We now describe several variants of imperfectly erasable memory that are relevant for practice, namely memory that leaks a constant number of bits, memory that leaks bits with a certain probability, memory that leaks a noisy version of the data, and memory that leaks the output of a length-shrinking function of the data. We then show how to construct memories which leak less information from each of these variants, in other words, we show how to amplify the erasability of each variant.

5.2.2.1 Amplifying Memory Leaking Exactly d Symbols

On many file systems, unlinked files are not necessarily immediately erased in their entirety. For instance, on most SSDs, deleted data persists until the flash translation layer flashes the corresponding erase block. Furthermore, data may survive erasure if it was copied to another medium, such as a cache, the swap space or backups. An adversary could therefore potentially recover parts of data that were believed to be erased. In full generality, the adversary may not obtain the entire data but still have an influence on which parts of the data she obtains in an attack, e.g., because she can steal just one backup tape, because of the cost of the attack or time constrains forcing her to choose the most juicy parts of the data, or because the adversary could influence the system beforehand to some degree and ensure that the parts of the data she is interested in were backed-up/swapped/cached.

To model such a scenario, we define the memory resource $\mathsf{IMR}\langle\Phi, n, d\rangle$ storing n symbols of an alphabet Φ, and where the adversary can obtain exactly d symbols of his choice when the memory leaks. This resource is a specialization of $\mathsf{IM}\langle\Sigma, \psi, \Xi\rangle$, where $\Sigma = \Phi^n$, ψ is the identity function, and Ξ accepts any function that reads at most d symbols from LDAT.

In a real setting, and depending on the nature of the attack, the adversary may obtain less than d symbols or might not have full control over which symbols she obtains. A memory resource in such a setting can be perfectly constructed from $\mathsf{IMR}\langle\Phi, n, d\rangle$ with the identity converter. (A memory resource where the adversary can obtain more than d symbols with a small probability ϵ can also be constructed from $\mathsf{IMR}\langle\Phi, n, d\rangle$, albeit with an error probability equal to ϵ; see, e.g., Section 5.2.2.2.)

The converter $\mathsf{I2P}$ shown in Figure 5.2 constructs $\mathsf{PM}\langle\Phi^k\rangle$ from $\mathsf{IMR}\langle\Phi, n, d\rangle$. This converter is parametrized by an AoNT (cf. Section 2.4.9). In a nutshell, $\mathsf{I2P}$ just applies the AoNT encoding algorithm $\mathsf{aenc}(\cdot)$ to the incoming data before storing it in $\mathsf{IMR}\langle\cdot\rangle$; and decodes the encoded data stored in $\mathsf{IMR}\langle\cdot\rangle$ using $\mathsf{adec}(\cdot)$ before outputting it. The erasure command is transmitted to $\mathsf{IMR}\langle\cdot\rangle$ directly. The privacy property of the AoNT guarantees that if the adversary obtains d symbols of the encoded data, she obtains no meaningful information about the original data. Thus, we obtain the following theorem.

Theorem 5.1. *If* $(\mathsf{aenc}, \mathsf{adec})$ *is an* ϵ*-secure* (Φ, n, d, k)*-AoNT, then*

$$\mathsf{IMR}\langle\Phi, n, d\rangle \xrightarrow{\mathsf{I2P}\langle\Phi,k,\mathsf{aenc},\mathsf{adec}\rangle}_{\epsilon} \mathsf{PM}\langle\Phi^k\rangle.$$

A similar theorem can be stated for the computational case.

The converter $\mathsf{I2P}\langle\Phi, k, \mathsf{aenc}, \mathsf{adec}\rangle$:

Behavior:

- *Outer*(`store`, $\mu \in \Phi^k$): *Inner* $\xleftarrow{\$}$ (`store`, $\mathsf{aenc}(\mu)$). *Inner* $\rightarrow$ (). *Outer* $\leftarrow$ ().
- *Outer*(`retrieve`): *Inner* $\leftarrow$ (`retrieve`). *Inner* $\rightarrow \phi$.
  If $\phi \neq ()$: *Outer* $\leftarrow \mathsf{adec}(\phi)$. Else: *Outer* $\leftarrow$ ().
- *Outer*(`erase`): *Inner* $\leftarrow$ (`erase`). *Inner* $\rightarrow$ (). *Outer* $\leftarrow$ ().

Fig. 5.2: The converter I2P constructing $\mathsf{PM}\langle\Phi^k\rangle$ from $\mathsf{IMR}\langle\Phi, n, d\rangle$. The converter is parametrized by a (Φ, n, d, k)-AoNT $(\mathsf{aenc}, \mathsf{adec})$.

The simulator $\mathsf{SI2P}\langle\Phi, n, d, k, \mathsf{aenc}\rangle$:

Internal state and initial values: LEAKED $= 0$.

Behavior:

- *Outer*(`gethist`): *Inner* $\leftarrow$ (`gethist`). *Inner* $\rightarrow \lambda$. *Outer* $\leftarrow \lambda$.
- *Outer*(`read`): upon error in the following, abort with *Outer* $\leftarrow$ ().
  Inner $\leftarrow$ (`read`). *Inner* $\rightarrow \mu \in \Phi^k$. *Outer* $\xleftarrow{\$}$ $\mathsf{aenc}(\mu)$.
- *Outer*(`leak`, $\xi \in (x \mapsto [x]_L$ with $L \in 2^{\{1,\ldots,n\}} \cap \mathbb{N}^d)$):
  upon error in the following, abort with *Outer* $\leftarrow$ ().
  If LEAKED $= 0$:
  Inner $\leftarrow$ (`gethist`); *Inner* $\rightarrow$ (“`e`”, “`W`”); LEAKED $\leftarrow 1$; *Outer* $\xleftarrow{\$}$ $\xi(\mathsf{aenc}(0^k))$.

Fig. 5.3: The simulator SI2P in the proof of the construction of $\mathsf{PM}\langle\Phi^k\rangle$ from $\mathsf{IMR}\langle\Phi, n, d\rangle$ using a (Φ, n, d, k)-AoNT $(\mathsf{aenc}, \mathsf{adec})$.

Proof. Figure 5.3 shows the simulator SI2P. We now prove that all distinguishers have at most advantage ϵ in distinguishing $\mathsf{I2P}\langle\Phi, n, \mathsf{aenc}, \mathsf{adec}\rangle\mathsf{IMR}\langle\Phi, n, d\rangle$ (the “real world”) from $\mathsf{PM}\langle\Phi^k\rangle\mathsf{SI2P}\langle\Phi, k, d, n, \mathsf{aenc}\rangle$ (the “ideal world”).

Consider a system W that interacts with any distinguisher D and that behaves like the ideal world except that instead of outputting $\xi(\mathsf{aenc}(0^k))$ during a leak, W plays the AoNT distinguishing game with ξ and 0^k and the message μ that was stored in the memory, obtains $(0^k, \mu, \omega)$ from that game, and outputs ω on *leak*. W outputs the same value as D. If D never triggers a leak, W outputs a random bit.

It is easy to see that if leakage output by the AoNT privacy game corresponds to the message 0^k, the behavior of W is exactly the same as the ideal world for D; and the distribution output by the AoNT privacy game corresponds to μ, the behavior of W is exactly the same as the real world for D. W's advantage in the AoNT distinguishing game is thus not less than the distinguishing advantage of D.

□

Multi-part leakage. It is sometimes the case that the memory is segmented into multiple independent parts, e.g., over two different file systems on different partitions of the same physical disc and that each part reacts differently to an attack.

We define a multi-part memory resource $\mathsf{IMRP}\langle\Phi, s_1, s_2, d\rangle$ storing data in $\Phi^{s_1+s_2}$. The memory is divided in two parts, the first part consisting of the first s_1 symbols and the second of the other s_2 symbols. The first part of the memory leaks similarly to $\mathsf{IMR}\langle\Phi, s_1, d\rangle$, while the second one leaks the entire data. When attacking the memory, the adversary must submit the choice of leakage for the first part *before* obtaining the leakage of the second part. We get the following theorem.

Theorem 5.2. *If* $(\mathsf{aenc}, \mathsf{adec})$ *is an* ϵ*-secure* $(\Phi, n+\nu, d, k)$*-AoNT with public part* ν*, then*

$$\mathsf{IMRP}\langle \Phi, n, \nu, d\rangle \xrightarrow{\mathsf{I2P}\langle \Phi, k, \mathsf{aenc}, \mathsf{adec}\rangle}_{\epsilon} \mathsf{PM}\langle \Phi^k\rangle.$$

Proof. The proof is similar to the proof of Theorem 5.1 and is omitted.

A similar theorem can be stated for the computational case.

Choice of alphabet. The most suitable choice of Φ depends on the application. Possible values are GF(2) when bits can be leaked independently, e.g., because the adversary must read them one by one from the surface of a disc; $\mathrm{GF}(2^{512\cdot 8})$ to $\mathrm{GF}(2^{4096\cdot 8})$ when the smallest leakable unit is a file system block; or even $\mathrm{GF}(2^{128\cdot 1024\cdot 8})$ to $\mathrm{GF}(2^{8192\cdot 1024\cdot 8})$ when the smallest leakable unit is an erase blocks of an SSD. In the latter two cases, it is also possible to design the system in such a way that only parts of a block are written to before proceeding with the next one, thereby reducing the alphabet size and limiting the amount of exposure per leaked block.

5.2.2.2 Amplifying Memory Leaking Symbols with Probability p

Above, we modelled an adversary who chooses which symbols leak from the imperfect memory. In practice, the adversary may not have this much power: for example, some parts of a deleted file might still be present in the journal, but the adversary has no control over which ones. To model this, let us now consider an adversary who obtains each symbol of the data uniformly and independently at random with a certain probability p during a leakage. We denote a memory with such a behaviour by $\mathsf{IME}\langle \Phi, n, d\rangle$. This resource is a specialization of $\mathsf{IM}\langle \Sigma, \psi, \Xi\rangle$ where $\Sigma = \Phi^n$, ψ acts like an erasure channel with erasure probability $(1-p)$ (i.e., each symbol of the data is transmitted correctly with probability p and otherwise is replaced with "$\perp$"), and Ξ accepts only the identity function.

One can treat $\mathsf{IME}\langle\cdot\rangle$ similarly to $\mathsf{IMR}\langle\cdot\rangle$ in constructions with just a small statistical error, as the following observation shows. Constructing $\mathsf{PM}\langle \Phi^k\rangle$ from $\mathsf{IME}\langle \Phi, n, p\rangle$ directly without first constructing $\mathsf{IMR}\langle \Phi, n, d\rangle$ might be more efficient (better parameters, less statistical error), but such a direct construction is out of the scope of this chapter.

Observation 5.3 *For all* $(n, d) \in \mathbb{N}^2$*,* $p \in [0,1]$*, fields* Φ*:* $\mathsf{IME}\langle \Phi, n, p\rangle \xrightarrow{\mathsf{id}}_{\epsilon} \mathsf{IMR}\langle \Phi, n, d\rangle$*. Here* id *is the identity converter, and* $\epsilon = (1 - \mathrm{BinomialCDF}(d; n, p)) = \sum_{i=d+1}^{n} \binom{n}{i} p^i \cdot (1-p)^{n-i}$*.*

Proof. In Figure 5.4 we show a simulator $\mathsf{SE2R}$, such that $\mathsf{IMR}\langle \Phi, n, d\rangle\mathsf{SE2R}\langle n, d, p\rangle$ and $\mathsf{IME}\langle \Phi, n, p\rangle$ can be distinguished with advantage at most $(1 - \mathrm{BinomialCDF}(d; n, p))$.

It is easy to see that if the simulator does not abort, then the simulation is perfect. Since the number of bits of b set to 1 follows a binomial distribution with parameters n and p, the probability of an abort is $(1 - \mathrm{BinomialCDF}(d; n, p))$. Hence the maximal advantage of any distinguisher is $(1 - \mathrm{BinomialCDF}(d; n, p))$.

□

The simulator $\mathsf{SE2R}\langle n, d, p\rangle$:

Internal state and initial values: LEAKED $= 0$.

Behavior:

- *Outer*(`gethist`): *Inner* $\leftarrow$ (`gethist`). *Inner* $\rightarrow \lambda$. *Outer* $\leftarrow \lambda$.
- *Outer*(`read`): *Inner* $\leftarrow$ (`read`). *Inner* $\rightarrow \mu$. *Outer* $\leftarrow \mu$.
- *Outer*(`leak`): *Inner* $\leftarrow$ (`gethist`). *Inner* $\rightarrow \lambda$.

  if LEAKED $= 0 \wedge \lambda =$ (“`e`”, “`W`”):

  LEAKED $\leftarrow 1$; $b \leftarrow 0^n$; $\omega \leftarrow (\bot, \ldots, \bot)$ with $|\omega| = n$;

  for $i \in \{1, \ldots, n\}$, set $b_i \leftarrow 1$ with probability p;

  if $\sum_i b_i > d$, then abort with *Outer* $\leftarrow ()$;

  $\xi \leftarrow \{i \mid b_i = 1\}$; add additional indices to ξ until $\xi \in (2^{\{1,\ldots,n\}} \cap \mathbb{N}^d)$;

  Inner $\leftarrow$ (`leak`, ξ); *Inner* $\rightarrow \delta$;

  for all i where $b_i = 1$, set ω_i to the corresponding value in δ; *Outer* $\leftarrow \omega$.

Fig. 5.4: The simulator $\mathsf{SE2R}$ in the proof of the construction of $\mathsf{IMR}\langle \Phi, n, d\rangle$ from $\mathsf{IME}\langle \Phi, n, p\rangle$.

5.2.2.3 Amplifying Memory with Noisy Leakage with Crossover Probability $(1-p)/|\Phi|$

Another possible setting is that the data is written to and erased from magnetic storage, and the adversary, who has physical access to the storage medium, must make an educated guess for each bit of the data [Gut96]. One can model this as if the data was transmitted through a noisy binary symmetric channel. We denote such a memory by $\mathsf{IMN}\langle \Phi, n, d\rangle$. This resource is a specialization of $\mathsf{IM}\langle \Sigma, \psi, \Xi\rangle$ where $\Sigma = \Phi^n$, ψ acts like a noisy $|\Phi|$-ary channel with crossover probability $(1-p)/|\Phi|$ (i.e., each symbol of the data is transmitted correctly with probability p and otherwise is replaced with a symbol drawn uniformly at random from Φ), and Ξ accepts only the identity function.

Observation 5.4 *For all* $(n, d) \in \mathbb{N}^2$, $p \in [0, 1]$, *fields* Φ: $\mathsf{IMN}\langle \Phi, n, p\rangle \xrightarrow{id} \mathsf{IME}\langle \Phi, n, p\rangle$.

Proof. The proof for this observation is very similar to the proof of Observation 5.3 and is omitted. The difference is that the simulator instead initializes ω to a random element of Φ^n.

□

5.2.2.4 Amplifying Memory with Limited Leakage Output Volume

Another possible setting, is that the adversary does not obtain individual symbols of the data, but rather a function of the data. For example, with a cache-timing attack [Ber05], she might deduce some information about the data without recovering it completely. In general, one can consider an adversary that obtains any *length-shrinking* function of the contents of the memory. We denote such a memory by $\mathsf{IML}\langle \Sigma, v\rangle$. This resource is a specialization of $\mathsf{IM}\langle \Sigma, \psi, \Xi\rangle$, where ψ is the identity function and Ξ accepts only functions that have at most v different output values.

For any non-trivial parameters, it is not possible to construct a perfectly erasable memory from $\mathsf{IML}\langle \cdot\rangle$, because the adversary can submit a leakage function $\xi \in \Xi$ that

runs the decoding logic of the converter. The reason for this is as follows. Let $v \geq 2$, $|\Sigma'| \geq 2$, $|\Sigma| \geq 2$, and let π be a converter that constructs $\mathsf{PM}\langle\Sigma'\rangle$ from $\mathsf{IML}\langle\Sigma, v\rangle$. We now show that this construction has a statistical error of at least $\frac{1}{2}$. The distinguisher chooses two distinct messages $a_0, a_1 \in \Sigma'$, flips a coin $b \xleftarrow{\$} \{0,1\}$, and stores a_b. He then makes the memory weak by setting the relevant flags on the *World*-interface and submits a leakage function ξ that returns 0 iff a_0 was encoded in $\mathsf{IML}\langle\cdot\rangle$ by using the decoding logic of π—recall that the distinguisher may depend on π. The distinguisher then outputs 1 iff ξ outputs b. No simulator will be able to properly simulate that scenario with probability more than $\frac{1}{2}$ as it does not know if the distinguisher stored a_0 or a_1.

However, one can obtain a meaningful construction by starting from a memory resource with multi-part leakage. Let $\mathsf{IMLP}\langle\Phi, s_1, s_2, v\rangle$ be analogous to $\mathsf{IMRP}\langle\Phi, s_1, s_2, d\rangle$ defined previously, except that the first part leaks similarly to $\mathsf{IML}\langle\Phi^{s_1}, v\rangle$. Here it is crucial to note that the function ξ submitted by the adversary can read only the first part of the memory. In particular, given a universal hash function $\mathsf{h} : \Phi^a \times \Phi^n \mapsto \Phi^k$, one can construct the resource $\mathsf{PM}\langle\Phi^k\rangle$ from $\mathsf{IMLP}\langle\Phi, n, a+k, v\rangle$, by using the AoNT defined in Section 2.4.9.3 with I2P. The construction is $(2v2^{(k-n)/2})$-secure [CDH+00, CEGL08b]. This construction is essentially the one proposed by Canetti et al. [CEGL08a] and Lim [Lim08].

5.2.3 Constructing a Large Perfectly Erasable Memory from a Small One

We now discuss how a small perfectly erasable memory can be used together with a large, possibly non-erasable memory to construct a large perfectly erasable memory. The basic idea underlying this construction is that of Yee et al.'s crypto paging [Yee94, YT95]: one stores a cryptographic key in the small perfectly erasable memory, encrypts the data with that key, and stores the resulting ciphertext in the large, possibly non-erasable memory. The resulting resource $\mathsf{PMA}\langle\mathrm{GF}(2^{\ell(\eta)})\rangle$ will allow the adversary to read the stored data if the resource is weakened by the environment before the user erases the key. The specification of this resource is given in Figure 5.5a and the protocol XPM for the construction is provided in Figure 5.6.

The resource just constructed allows the adversary to read the stored data if either the small erasable *or* the large non-erasable memory become weak before the user erases the key. Thus, this resource is weaker than what one would expect, i.e., it should be the case that the adversary can only read the data if *both* underlying resources become weak before the user erases the key. The corresponding resource $\mathsf{PMB}\langle\mathrm{GF}(2^{\ell(\eta)})\rangle$ is depicted in Figure 5.5b. Unfortunately, the realization of this resource would require a non-committing encryption scheme, which can only be constructed in the random oracle model but not in the standard model.

However, it is possible to construct the somewhat better resource $\mathsf{PMC}\langle\mathrm{GF}(2^{\ell(\eta)})\rangle$, shown in Figure 5.5c. Here the adversary can read the stored data if the memory storing the ciphertext becomes weak before the user calls delete. It is not hard to see that $\mathsf{PMC}\langle\mathrm{GF}(2^{\ell(\eta)})\rangle$ implies $\mathsf{PMA}\langle\mathrm{GF}(2^{\ell(\eta)})\rangle$, essentially the simulator attached to the Eve interface of $\mathsf{PMA}\langle\mathrm{GF}(2^{\ell(\eta)})\rangle$ has to hold back the leaked data until the non-erasable memory becomes leakable. In summary, we get the following theorem.

(a) The resource $\mathsf{PMA}\langle\Sigma\rangle$ (b) The resource $\mathsf{PMB}\langle\Sigma\rangle$ (c) The resource $\mathsf{PMC}\langle\Sigma\rangle$

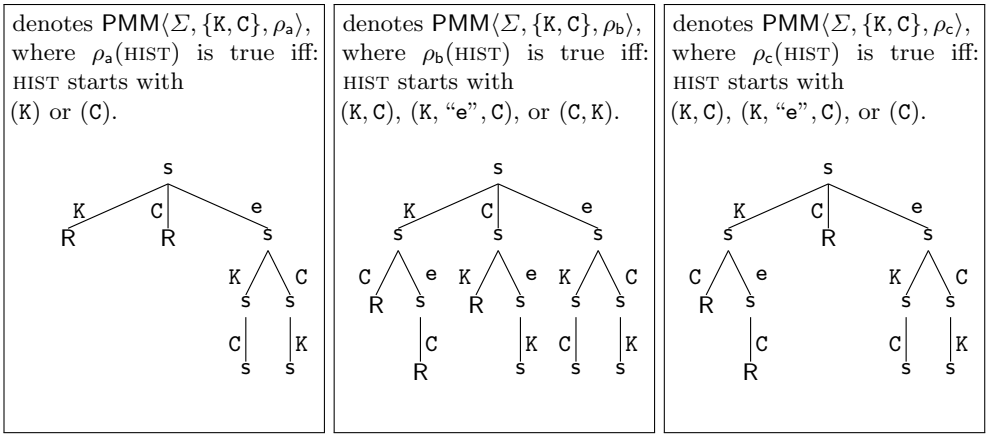


Fig. 5.5: Several variants of a perfectly erasable memory resource with two *World*-flags. The prefix decision trees visualize whether the adversary has read access to the memory depending on the event history HIST. A branch labelled "e" represents an erasure event, and branches labelled "K" (key) or "C" (ciphertext) represent the setting of the corresponding flags on the *World*-interface. An "R" node means that the memory is readable (and allows the adversary to collect the data at any time from then on), and an "s" (secure) node means that it does not.

The converter $\mathsf{XPM}\langle\ell, \mathsf{prg}\rangle$:

Behavior:

- *Outer*(`store`, $\mu \in \mathrm{GF}(2^{\ell(\eta)})$): $sk \xleftarrow{\$} \mathrm{GF}(2^{\eta})$. $\delta \leftarrow \mathsf{prg}(sk) + \mu$. *Inner* $\leftarrow$ (`MEMP`, `store`, sk). *Inner* $\rightarrow$ (). *Inner* $\leftarrow$ (`MEMNE`, `store`, δ). *Inner* $\rightarrow$ (). *Outer* $\leftarrow$ ().
- *Outer*(`retrieve`): upon error in the following, abort with *Outer* $\leftarrow$ ().
  Inner $\leftarrow$ (`MEMP`, `retrieve`). *Inner* $\rightarrow sk \in \mathrm{GF}(2^{\eta})$.
  Inner $\leftarrow$ (`MEMNE`, `retrieve`). *Inner* $\rightarrow \delta$. $\mu \leftarrow \delta - \mathsf{prg}(sk)$. *Outer* $\leftarrow \mu$.
- *Outer*(`erase`): *Inner* $\leftarrow$ (`MEMP`, `erase`); *Inner* $\rightarrow$ (). *Outer* $\leftarrow$ ().

Fig. 5.6: The converter XPM constructing a large perfectly erasable memory $\mathsf{PMA}\langle\mathrm{GF}(2^{\ell(\eta)})\rangle$ or $\mathsf{PMC}\langle\mathrm{GF}(2^{\ell(\eta)})\rangle$ using a small perfectly erasable memory $\mathsf{PM}\langle\mathrm{GF}(2^{\eta})\rangle$ and a large non-erasable memory $\mathsf{NM}\langle\mathrm{GF}(2^{\ell(\eta)})\rangle$. The converter is parametrized by an ℓ-PRG prg, and the implicit security parameter η.

Theorem 5.5. *If* prg *is a secure ℓ-PRG, then*

$$\left[\mathsf{PM}\langle\mathrm{GF}(2^{\eta})\rangle, \mathsf{NM}\langle\mathrm{GF}(2^{\ell(\eta)})\rangle\right] \xrightarrow{\mathsf{XPM}\langle\ell,\mathsf{prg}\rangle}_{\mathsf{c}} \mathsf{PMC}\langle\mathrm{GF}(2^{\ell(\eta)})\rangle.$$

Proof. Figure 5.7 shows the simulator SXPM.

For the sake of contradiction, let us assume the existence of an efficient distinguisher D that has non-negligible advantage in distinguishing between the "real world" $\mathsf{XPM}\langle\ell, \mathsf{prg}\rangle[\mathsf{PM}\langle\mathrm{GF}(2^{\eta})\rangle, \mathsf{NM}\langle\mathrm{GF}(2^{\ell(\eta)})\rangle]$ and $\mathsf{PMC}\langle\mathrm{GF}(2^{\ell(\eta)})\rangle\mathsf{SXPM}\langle\ell, \mathsf{prg}, \rho_{\mathsf{a}}\rangle$, the "ideal world". (We assume that the difference on the *World*-interface is implicitly taken

The simulator $\mathsf{SXPM}\langle \ell, \mathsf{prg}, \rho \rangle$:

Internal state and initial values: $\text{CT} = \bot, \text{SK} = \bot$.

Behavior:

- *Outer*(`MEMP`, `gethist`): *Inner* $\leftarrow$ (`gethist`). *Inner* $\rightarrow \lambda$.
  Remove any "`C`" from λ. *Outer* $\leftarrow \lambda$.
- *Outer*(`MEMNE`, `gethist`): *Inner* $\leftarrow$ (`gethist`). *Inner* $\rightarrow \lambda$.
  Remove any "`K`" and "`e`" from λ. *Outer* $\leftarrow \lambda$.
- *Outer*(`MEMP`, `read`): *Inner* $\leftarrow$ (`gethist`). *Inner* $\rightarrow \lambda$.
  If λ does not start with ("`K`") nor ("`C`", "`K`"): abort with *Outer* $\leftarrow ()$.
  If $\text{SK} = \bot$: $\text{SK} \xleftarrow{\$} \text{GF}(2^\eta)$.
  Outer $\leftarrow$ SK.
- *Outer*(`MEMNE`, `read`): *Inner* $\leftarrow$ (`gethist`). *Inner* $\rightarrow \lambda$.
  If "`C`" $\notin \lambda$: abort with *Outer* $\leftarrow ()$.
  If $\text{SK} = \bot$: $\text{SK} \xleftarrow{\$} \text{GF}(2^\eta)$.
  If $\text{CT} = \bot \wedge \rho(\lambda)$: *Inner* $\leftarrow$ (`read`); *Inner* $\rightarrow \mu$; $\text{CT} \leftarrow \mu + \mathsf{prg}(\text{SK})$.
  If $\text{CT} = \bot$: $\text{CT} \xleftarrow{\$} \text{GF}(2^{\ell(\eta)})$.
  Outer $\leftarrow$ CT.

Fig. 5.7: The simulator SXPM in the proof of the construction of $\mathsf{PMC}\langle \text{GF}(2^{\ell(\eta)}) \rangle$ from $\mathsf{PM}\langle \text{GF}(2^\eta) \rangle$ and $\mathsf{NM}\langle \text{GF}(2^{\ell(\eta)}) \rangle$ using the converter $\mathsf{XPM}\langle \ell, \mathsf{prg} \rangle$ and with $\rho := \rho_{\mathsf{c}}$. The same simulator with $\rho := \rho_{\mathsf{a}}$ can be used in the construction of $\mathsf{PMA}\langle \text{GF}(2^{\ell(\eta)}) \rangle$.

care of.) We show how to construct an efficient distinguisher W with non-negligible advantage for the PRG distinction game.

W behaves like the ideal world, except that, when asked to leak the non-erasable memory when the "`C`" *World*-flag was set after the data was erased but before the "`K`" *World*-flag was set (in the sequel, we call this the ("`e`", "`C`")-event), W obtains a challenge c from the PRG distinction game and outputs $(c + \mu)$. W then outputs the same thing as D. If D terminates without having provoked the ("`e`", "`C`")-event, then W outputs a random bit.

Notice that W is constructed in such a way that:

- If the ("`e`", "`C`")-event does not happen, then the real and ideal worlds are perfectly indistinguishable.
- If the PRG distinction game outputs $c = \mathsf{prg}(sk)$, then W behaves exactly like the real world to D.
- If the PRG distinction game outputs a random c, then W behaves exactly like the ideal world to D.

Hence WD has the same non-negligible advantage in the PRG distinction game than D has in distinguishing the real and ideal world.

□

As stated above, XPM also constructs $\mathsf{PMA}\langle \text{GF}(2^{\ell(\eta)}) \rangle$ from the same resources. Furthermore, in the random oracle model, a protocol that is identical to XPM except that calls to prg are replaced by calls to the random oracle, constructs $\mathsf{PMB}\langle \text{GF}(2^{\ell(\eta)}) \rangle$ from the same resources.

Let us discuss our the memory resources just discussed in light of some secure memory constructions in the literature. As mentioned, Yee et al. introduce crypto paging [Yee94, YT95] to let a secure co-processor encrypt its virtual memory before paging it out to its host's physical memory or hard disk. Translated to our setting, this means that the non-erasable memory is weak from the beginning. Therefore, to get meaningful guarantees, only the resource $\mathsf{PMB}\langle \mathrm{GF}(2^{\ell(\eta)})\rangle$ can be used in their setting, the other two would allow the adversary to always read the data. Thus, to realize their system, Yee et al. require a non-committing encryption scheme (and hence random oracles).

Di Crescenzo et al. [DCFIJ99] consider a memory resource that allows one to update the stored data such that when the resource becomes weak the adversary can only read the data stored last. They then provide a construction for a large such resource from a small one and a large non-erasable memory. Again they assume that for both resources the data can be updated and that the non-erasable one leaks all data ever stored in it. None of our resources does allow for such updates but, as already discussed, resources that allow this can be constructed by using several of our respective resources in parallel. Thus, their security definition and construction can be indeed modelled and analysed with the memory resources we define, however, doing this is out of scope of this chapter.

5.3 New Realizations of All-or-Nothing Transforms

In Section 5.2 we saw the importance of AoNTs for constructing perfectly erasable memory from certain types of imperfectly erasable ones. In this section we present several novel AoNTs. We start by showing the dual of the $\mathsf{I2P}$ protocol: any protocol that constructs $\mathsf{PM}\langle \Phi^k\rangle$ from $\mathsf{IMR}\langle \Phi, n, d\rangle$ can be used to realize a (Φ, n, k, d)-AoNT. We then present a perfect AoNT with better parameters than what is found in the literature, based on the novel concept of *ramp minimum distance* of a matrix. We then show that one can combine several AoNTs to achieve an AoNT over a small field but with a large message space and a good privacy threshold d. Finally, we provide a computationally-secure AoNT over a large field that has a very large privacy threshold.

5.3.1 AoNT from a Protocol that Constructs $\mathsf{PM}\langle \Phi^k\rangle$ from $\mathsf{IMR}\langle \Phi, n, d\rangle$

Section 5.2.2.1 described the protocol $\mathsf{I2P}$, parametrized by an AoNT, that constructs a perfectly erasable memory $\mathsf{PM}\langle \Phi^k\rangle$ from an imperfectly erasable one $\mathsf{IMR}\langle \Phi, n, d\rangle$. As the following theorem states, any protocol π (not necessarily one based on an AoNT) that constructs $\mathsf{PM}\langle \Phi^k\rangle$ from $\mathsf{IMR}\langle \Phi, n, d\rangle$ can be used to construct an AoNT using the algorithm $\mathsf{C2A}$, albeit one that where adec is a probabilistic algorithm and where decoding might fail with a small probability.

Theorem 5.6. *If π is a converter such that $\mathsf{IMR}\langle \Phi, n, d\rangle \xrightarrow{\pi}_{\epsilon} \mathsf{PM}\langle \Phi^k\rangle$, then the algorithm $\mathsf{C2A}\langle \Phi, n, k, \pi\rangle$ is a 6ϵ-secure (Φ, n, d, k)-AoNT with a probabilistic adec and where decoding may fail with probability less than 2ϵ.*

The algorithm $\mathsf{C2A}\langle \Phi, n, k, \pi \rangle$:

Behavior:

- $\mathsf{aenc}(\mu \in \Phi^k)$: $\pi.\mathit{Outer} \leftarrow (\mathtt{store}, \mu)$.
  While true:
  If $\pi.\mathit{Inner} \rightarrow (\mathtt{store}, \phi \in \Phi^n)$: return ϕ.
  Else if anything is sent by $\pi.\mathit{Inner}$: $\pi.\mathit{Inner} \leftarrow ()$.
  Else: abort by returning $\perp$.
- $\mathsf{adec}(\phi \in \Phi^n)$: $\pi.\mathit{Outer} \leftarrow (\mathtt{retrieve})$.
  While true:
  If $\pi.\mathit{Inner} \rightarrow (\mathtt{retrieve})$: $\pi.\mathit{Inner} \leftarrow \phi$.
  Else if $\pi.\mathit{Outer} \rightarrow \mu \in \Phi^k$: return μ.
  Else if $\pi.\mathit{Inner} \rightarrow (\mathtt{erase})$: abort by returning $\perp$.
  Else if anything is sent by $\pi.\mathit{Inner}$: $\pi.\mathit{Inner} \leftarrow ()$.
  Else: abort by returning $\perp$.

Fig. 5.8: The algorithm $\mathsf{C2A}$ that realizes a (Φ, n, d, k)-AoNT from a converter π, where π constructs $\mathsf{PM}\langle \Phi^k \rangle$ from $\mathsf{IMR}\langle \Phi, n, d \rangle$.

The distinguisher $\mathsf{D}\langle \mu, x \rangle$:

Upon error in any of the following, abort and return 1.
Do x times: $\mathsf{Y}.\mathit{Alice} \leftarrow (\mathtt{retrieve})$; $\mathsf{Y}.\mathit{Alice} \rightarrow ()$.
$\mathsf{Y}.\mathit{Alice} \leftarrow (\mathtt{store}, \mu)$. $\mathsf{Y}.\mathit{Alice} \rightarrow ()$.
Do $x + 1$ times: $\mathsf{Y}.\mathit{Alice} \leftarrow (\mathtt{retrieve})$; $\mathsf{Y}.\mathit{Alice} \rightarrow \mu_b$.
$\mathsf{Y}.\mathit{Alice} \leftarrow (\mathtt{erase})$. $\mathsf{Y}.\mathit{Alice} \rightarrow ()$.
Do x times: $\mathsf{Y}.\mathit{Alice} \leftarrow (\mathtt{retrieve})$; $\mathsf{Y}.\mathit{Alice} \rightarrow ()$.
Return 0.

Fig. 5.9: The distinguisher for the correctness condition of Theorem 5.6.

Proof of correctness. We first show that if there exists a message μ such that encoding and decoding fails with probability at least 2ϵ, then the distinguisher D shown in Figure 5.9 distinguishes between the "real world" $\pi\mathsf{IMR}\langle \Phi, n, d \rangle$ and the "ideal world" $\mathsf{PM}\langle \Phi^k \rangle \sigma$ with advantage at least ϵ for any σ, which would be a contradiction. Let Y denote the system D is interacting with. Let x be an integer such that $(\frac{1}{4})^x < \epsilon$.

Notice that σ is never activated, hence D works for all simulators.

It is clear that if D interacts with the ideal world, D always outputs 0. When interacting with the real world, intuitively, D outputs 1 if encoding or decoding failed, i.e., with probability 2ϵ. However, one has to take possible "malicious behavior" of π into account: it is possible that $\mathsf{C2A}\langle \Phi, n, k, \pi \rangle$ fails because π issued an `erase` command during retrieval or because π never stored anything in the memory, but D outputs 0 anyway.

Recall that π cannot keep state between phases, hence if π issued an `erase` command during the first retrieval in the second loop, π cannot distinguish between any of the retrieval phases in the second and third loop. Let y denote the probability that π returns μ if faced with an empty memory. The probability that no error happens in D if the memory was erased during the first retrieve by π, is thus at most $y^x \cdot (1-y)^x < (\frac{1}{4})^x < \epsilon$. A similar argument shows that if π never stored anything in the memory during the

The distinguisher $\mathsf{D}\langle\mu_0, \mu_1, \xi, x, \mathsf{W}\rangle$:
Upon error in the following "first part", abort and return 1.
Do x times: $\mathsf{Y}.\mathit{Alice} \leftarrow (\texttt{retrieve})$; $\mathsf{Y}.\mathit{Alice} \rightarrow ()$.
$b \xleftarrow{\$} \{0,1\}$. $\mathsf{Y}.\mathit{Alice} \leftarrow (\texttt{store}, \mu_b)$. $\mathsf{Y}.\mathit{Alice} \rightarrow ()$.
Do x times: $\mathsf{Y}.\mathit{Alice} \leftarrow (\texttt{retrieve})$; $\mathsf{Y}.\mathit{Alice} \rightarrow \mu_b$.
$\mathsf{Y}.\mathit{Alice} \leftarrow (\texttt{erase})$. $\mathsf{Y}.\mathit{Alice} \rightarrow ()$.
Do x times: $\mathsf{Y}.\mathit{Alice} \leftarrow (\texttt{retrieve})$; $\mathsf{Y}.\mathit{Alice} \rightarrow ()$.
$\mathsf{Y}.\mathit{World} \leftarrow (\texttt{weaken})$. $\mathsf{Y}.\mathit{World} \rightarrow ()$.
Upon error in the following "second part", abort and return 0.
$\mathsf{Y}.\mathit{Eve} \leftarrow (\texttt{leak}, \xi)$. $\mathsf{Y}.\mathit{Eve} \rightarrow \lambda \in \Phi^d$.
$\mathsf{W} \leftarrow (\mu_0, \mu_1, \lambda)$. $\mathsf{W} \rightarrow g$. If $b = g$: return 1. Else: return 0.

Fig. 5.10: The distinguisher for the privacy condition of Theorem 5.6.

`store` command, then π cannot distinguish any of the retrieval phases in the first and second loop. The probability that no error happens is thus also smaller than ϵ.

Hence the distinguishing advantage is at least $2\epsilon - \epsilon = \epsilon$.

Proof of privacy. We now show that if there exists a distinguisher W for the AoNT distinguishing game with advantage at least 6ϵ on a set ξ with messages μ_0 and μ_1, then the distinguisher D shown in Figure 5.10 that distinguishes between the "real world" $\pi\mathsf{IMR}\langle\Phi, n, d\rangle$ and the "ideal world" $\mathsf{PM}\langle\Phi^k\rangle\sigma$ has distinguishing advantage at least ϵ for any σ, which would be a contradiction. Let Y denote the system D is interacting with. Let x be an integer such that $(\frac{1}{4})^x \leq \epsilon$.

It is clear that D never aborts in the first part if it interacts with the ideal world (recall that σ was never activated up to now).

It is also clear that if D interacts with the real world and *if something was stored in the memory and the memory has been erased* D never aborts in the second part (recall that D doesn't interact with π in the second part).

Let us calculate the probability that π did not erase the memory and that no error happened during the first part. Recall that π does not keep state between phases. Since π does not erase the memory in any of the $2x$ retrieve queries following the `store` command, the memory behaves identically in all queries, and thus π cannot determine which query number it is currently servicing. Hence for a given μ_b, there must be a constant probability y that π returns μ_b in a retrieve phase. Also, π must return μ_b as expected in all x queries of the second loop (probability $= y^x$) and return $()$ as expected in all x queries of the third loop (probability at most $(1-y)^x$). The probability is thus $(y \cdot (1-y))^x \leq (\frac{1}{4})^x$.

If no error occurred in D, and D interacted with the ideal world, it is clear that the leakage λ is independent of the chosen bit b, hence W cannot have any advantage in the AoNT distinction game.

If D interacted with the real world, and *if the memory contains the value that would have been returned by* $\mathsf{C2A.aenc}$, W will output the correct guess with probability at least $\frac{1}{2} + 3\epsilon$. Similar to the correctness condition, it is possible that π did not store anything in the memory during the `store` command: this happens with probability at most $(\frac{1}{4})^x$. Thus if D interacted with the real world, and no error occurred, then the probability that W outputs the correct guess will be at least $\frac{1}{2} + 3\epsilon - (\frac{1}{4})^x$.

To conclude, the distinguishing advantage of D is at least $3\epsilon - 2 \cdot (\frac{1}{4})^x \geq \epsilon$, as required. □

One can make an analogous statement in the computational case.

5.3.2 Perfectly Secure AoNT Based on Matrices with Ramp Minimum Distance

This subsection shows how one can improve the standard realization of AoNTs based on linear block codes of Canetti et al. [CDH+00] by using our novel concept of *ramp minimum distance*.

The standard realization. Let $\mathbf{G}$ be the $k \times n$ generator matrix with elements in $\mathrm{GF}(q)$ of a linear block code with minimum distance d. The encoding function of the perfectly secure $(\mathrm{GF}(q), (n+k), d, k)$-AoNT is as follows:

$$\mathsf{aenc}(\mathbf{a} \in \mathrm{GF}(q)^k) : \mathbf{b} \xleftarrow{\$} \mathrm{GF}(q)^n; \mathbf{y} \leftarrow \begin{bmatrix} \mathbf{I}_n & \mathbf{0} \\ \mathbf{G} & \mathbf{I}_k \end{bmatrix} \begin{bmatrix} \mathbf{b} \\ \mathbf{a} \end{bmatrix}; \text{ return } \mathbf{y}.$$

Further details are given in Section 2.4.9.2.

Let us now show how to use the concept of ramp minimal distance to construct better AoNTs.

Definition 5.7. *A $k \times n$ matrix $\mathbf{G}$ with elements in $\mathrm{GF}(q)$ has* ramp minimum distance *d if for every $r \in \{1, \ldots, k\}$, every $r \times (n - (d - r))$ submatrix of $\mathbf{G}$ has rank r.*

Note that the concept of (regular) minimum distance comes from coding theory, and requires that all $k \times (n - (d-1))$ sub-matrices of G have rank k (which is equivalent to saying that for every $r \in \{1, \ldots, k\}$, all $r \times (n - (d-1))$ sub-matrices of $\mathbf{G}$ have rank r), where $\mathbf{G}$ is the generator matrix of a linear block code. A matrix with minimum distance d also has a *ramp* minimum distance d (the converse is obviously not true).

Now for the generator matrix with ramp minimum distance, we can construct an AoNT and thus obtain the following theorem.

Theorem 5.8. *The standard realization of a AoNT (sketched above and detailed in Section 2.4.9.2), parametrized by a $k \times n$ matrix $\mathbf{G}$ with elements in $\mathrm{GF}(q)$ with* ramp *(instead of regular)* minimum distance *d, is a perfectly secure $(\mathrm{GF}(q), (n+k), d, k)$-AoNT.*

Proof. We now show that the privacy threshold of the AoNT is at least d. For any set L of size d, let r denote the number of elements of $\mathbf{x}$ output by the AoNT distinguishing game (therefore $d-r$ elements of $\mathbf{b}$ are output by the game). Let ${}^{\mathsf{k}}\mathbf{b}$ denote the sub-vector of $\mathbf{b}$ of size $d-r$ containing all elements of $\mathbf{b}$ that are output by the AoNT distinguishing game, and let ${}^{\mathsf{u}}\mathbf{b}$ denote the sub-vector of size $n+r-d$ containing the elements that are not output. Similarly, let ${}^{\mathsf{k}}\mathbf{x}$ denote the sub-vector of $\mathbf{x}$ of size r that is output by the AoNT distinguishing game, and let ${}^{\mathsf{u}}\mathbf{x}$ denote the sub-vector of size $k-r$ containing the elements that are not output. Let $\mathbf{P}$ be the permutation matrix such that:

$$\mathbf{P} \begin{bmatrix} \mathbf{b} \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} {}^{\mathsf{u}}\mathbf{b} \\ {}^{\mathsf{k}}\mathbf{b} \\ {}^{\mathsf{k}}\mathbf{x} \\ {}^{\mathsf{u}}\mathbf{x} \end{bmatrix}.$$

Let ${}^{ku}\mathbf{G}$, ${}^{kk}\mathbf{G}$, ${}^{uu}\mathbf{G}$, ${}^{uk}\mathbf{G}$ be sub-matrices of $\mathbf{G}$, and let ${}^{k}\mathbf{a}$ and ${}^{u}\mathbf{a}$ be the sub-vectors of $\mathbf{a}$ such that:

$$\begin{bmatrix} \mathbf{I}_{n+r-d} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{d-r} & \mathbf{0} & \mathbf{0} \\ {}^{ku}\mathbf{G} & {}^{kk}\mathbf{G} & \mathbf{I}_r & \mathbf{0} \\ {}^{uu}\mathbf{G} & {}^{uk}\mathbf{G} & \mathbf{0} & \mathbf{I}_{k-r} \end{bmatrix} := \mathbf{P}\mathbf{M}\mathbf{P}^{\mathsf{T}} \quad \text{and} \quad \begin{bmatrix} {}^{u}\mathbf{b} \\ {}^{k}\mathbf{b} \\ {}^{k}\mathbf{a} \\ {}^{u}\mathbf{a} \end{bmatrix} := \mathbf{P} \begin{bmatrix} \mathbf{b} \\ \mathbf{a} \end{bmatrix}.$$

We thus have:

$$\begin{bmatrix} {}^{u}\mathbf{b} \\ {}^{k}\mathbf{b} \\ {}^{k}\mathbf{x} \\ {}^{u}\mathbf{x} \end{bmatrix} = \mathbf{P} \begin{bmatrix} \mathbf{b} \\ \mathbf{x} \end{bmatrix} = \mathbf{P}\mathbf{M} \begin{bmatrix} \mathbf{b} \\ \mathbf{a} \end{bmatrix} = \mathbf{P}\mathbf{M}\mathbf{P}^{\mathsf{T}}\mathbf{P} \begin{bmatrix} \mathbf{b} \\ \mathbf{a} \end{bmatrix} = \begin{bmatrix} \mathbf{I}_{n+r-d} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{d-r} & \mathbf{0} & \mathbf{0} \\ {}^{ku}\mathbf{G} & {}^{kk}\mathbf{G} & \mathbf{I}_r & \mathbf{0} \\ {}^{uu}\mathbf{G} & {}^{uk}\mathbf{G} & \mathbf{0} & \mathbf{I}_{k-r} \end{bmatrix} \begin{bmatrix} {}^{u}\mathbf{b} \\ {}^{k}\mathbf{b} \\ {}^{k}\mathbf{a} \\ {}^{u}\mathbf{a} \end{bmatrix} = \begin{bmatrix} {}^{u}\mathbf{b} \\ {}^{k}\mathbf{b} \\ {}^{ku}\mathbf{G}\,{}^{u}\mathbf{b} + {}^{kk}\mathbf{G}\,{}^{k}\mathbf{b} + {}^{k}\mathbf{a} \\ {}^{uu}\mathbf{G}\,{}^{u}\mathbf{b} + {}^{uk}\mathbf{G}\,{}^{k}\mathbf{b} + {}^{u}\mathbf{a} \end{bmatrix}.$$

Since $\mathbf{G}$ has ramp minimum distance d, the $r \times (n-(d-r))$ sub-matrix ${}^{ku}\mathbf{G}$ has rank r, thus ${}^{ku}\mathbf{G}\,{}^{u}\mathbf{b}$ is uniformly distributed. Therefore $\begin{bmatrix} {}^{k}\mathbf{b} \\ {}^{k}\mathbf{x} \end{bmatrix}$, which is the output of the AoNT distinction game, is uniformly distributed and independent from $\mathbf{a}$. This concludes the proof.

□

It remains to find a matrix with a desired ramp minimum distance. One way it so chose a random matrix, as show the following theorem.

Theorem 5.9. *For all $(n,k,d) \in \mathbb{N}^3$, and all prime powers q, a $k \times n$ matrix where all elements were chosen independently and uniformly at random over* $\mathrm{GF}(q)$*, has ramp minimum distance d with probability at least*

$$1 - \sum_{i=1}^{k} \binom{k}{i} (q-1)^i q^{\left(H_q\left(\frac{d-i}{n}\right)-1\right)n}$$

$$\text{where } H_q(x) := \begin{cases} 0 \text{ if } x = 0 \text{ or } x = 1; \\ x\log_q(q-1) - x\log_q(x) - (1-x)\log_q(1-x) \text{ if } 0 < x < 1. \end{cases}$$

Proof. Let $\mathrm{HW}(\mathbf{w})$ "Hamming weight" denote the number of non-zero entries of a vector $\mathbf{w}$. Let $\mathrm{HB}_q(n,r)$ "Hamming ball" denote the set of vectors of length n in $\mathrm{GF}(q)$ which have a Hamming weight of at most r.

Further, let $\mathrm{MC}_q(k)$ "minimal codewords" be the set of row vectors of length k in $\mathrm{GF}(q)$ whose leading non-zero entry is 1.

From the definition of ramp minimum distance, it follows that a $(\mathrm{GF}(q), n, k)$-linear block code with generator matrix $\mathbf{G}$ has ramp minimum distance d if:

$$\forall \mathbf{w} \in \mathrm{MC}_q(k) : \mathbf{w}\mathbf{G} \notin \mathrm{HB}_q(n, d - \mathrm{HW}(\mathbf{w})).$$

Since the coefficients of $\mathbf{G}$ are chosen independently and uniformly at random, the codeword $\mathbf{wG}$ is distributed uniformly. The probability that $\mathbf{wG}$ is in some Hamming ball of radius r thus is:

$$\Pr\left[\mathbf{wG} \in \mathrm{HB}_q(n,r)\right] = \frac{|\,\mathrm{HB}_q(n,r)|}{q^n} = \frac{\sum_{i=0}^{r}\binom{n}{i}(q-1)^i}{q^n} \leq \frac{q^{H_q\left(\frac{r}{n}\right)\cdot n}}{q^n} = q^{\left(H_q\left(\frac{r}{n}\right)-1\right)\cdot n}.$$

Using the union bound (Boole's inequality):

$$\begin{aligned}\Pr\left[\exists \mathbf{w} \in \mathrm{MC}_q(k) : \mathbf{wG} \in \mathrm{HB}_q(n, d-\mathrm{HW}(\mathbf{w}))\right] &\leq \sum_{\mathbf{w}\in\mathrm{MC}_q(k)} \Pr\left[\mathbf{wG} \in \mathrm{HB}_q(n, d-\mathrm{HW}(\mathbf{w}))\right] \\ &\leq \sum_{\mathbf{w}\in\mathrm{MC}_q(k)} q^{\left(H_q\left(\frac{d-\mathrm{HW}(\mathbf{w})}{n}\right)-1\right)\cdot n} \\ &= \sum_{i=1}^{k}\binom{k}{i}(q-1)^i q^{\left(H_q\left(\frac{d-i}{n}\right)-1\right)\cdot n}.\end{aligned}$$

The probability that $\mathbf{G}$ has ramp minimum distance d is therefore at least:

$$\begin{aligned}&\Pr\left[\mathbf{G} \text{ has ramp minimum distance } d\right] \\ &\quad= \Pr\left[\forall \mathbf{w} \in \mathrm{MC}_q(k) : \mathbf{wG} \notin \mathrm{HB}_q(n, d-\mathrm{HW}(\mathbf{w}))\right] \\ &\quad= 1 - \Pr\left[\exists \mathbf{w} \in \mathrm{MC}_q(k) : \mathbf{wG} \in \mathrm{HB}_q(n, d-\mathrm{HW}(\mathbf{w}))\right] \\ &\quad\geq 1 - \sum_{i=1}^{k}\binom{k}{i}(q-1)^i q^{\left(H_q\left(\frac{d-i}{n}\right)-1\right)\cdot n}.\end{aligned}$$

□

Unfortunately, we do not know of any efficient method to check whether a random matrix has a given ramp minimum distance. For practical parameters, however, it is feasible to generate and test such matrices with small values of k and d (e.g., less than 20).

Better AoNTs using our realization. Given a fixed size, it is sometimes possible to find matrices with a given ramp minimum distance but no matrix with the same (regular) minimum distance. Hence AoNTs based on matrices with a ramp minimum distance can achieve better parameters than previously known realizations. We now illustrate this fact with a numerical example. Let us determine the best message length k that a perfect AoNT with fixed parameters $n = 30$, $d = 12$, and $q = 2$ can achieve with both our realization and the standard realization. Both realizations will require a matrix with $(30 - k)$ rows and (ramp or regular, respectively) minimum distance $d = 12$. First, observe that there exists a 6×24 matrix over GF(2) with ramp minimum distance 12.[2] Hence using our realization, we can achieve $k = 6$. Plotkin [Plo60] showed that a

[2] Here is an example of such a matrix found using exhaustive search:

binary code with block length $2d$ and distance d can have at most $4d$ codewords. Hence there cannot exist a 6×24 matrix with (regular) minimum distance $d = 12$ (as it would generate a code with $2^6 = 64$ codewords, which is more than $4d = 48$). The best AoNT one can hope for using the standard realization thus has $k = 5$.

Statistical security. Theorem 5.9 stated that by choosing a random generator matrix, one can achieve a certain ramp minimum distance with a certain probability $(1 - \epsilon)$. If one uses our realization, but without checking that the matrix actually has the required ramp minimum distance, then the resulting AoNT will be perfectly secure with probability $(1 - \epsilon)$. (Note that this is different from saying that the AoNT is ϵ-secure, as the randomness used to generate the AoNT is not part of the distinguishing experiment.) In practice, one can make ϵ very small, e.g., $\epsilon < 2^{-\eta}$, and it might be acceptable to chose a random matrix and not check its properties to realize an AoNT.

5.3.3 Realizing a Perfectly Secure AoNT over a Small Field by Combining AoNTs

Designing perfectly-secure AoNTs over very small fields, e.g., GF(2), is hard. The previous realization does not scale well to large message lengths k and large privacy thresholds d; and realizations based on Shamir's secret sharing scheme (see Section 2.4.9.1) are always over large fields—using such a $(\mathrm{GF}(2^a), n, d, k)$-AoNT unmodified over GF(2) instead would result in a $(\mathrm{GF}(2), an, d, ak)$-AoNT with a poor privacy threshold d. The leakage of any GF(2) element means that the entire original $\mathrm{GF}(2^a)$ element is compromised. We now show how to combine the two approaches to realize a perfectly secure AoNT over a small field but with large k and d.

Our realization requires two AoNTs, a "fine-grained" one and a "coarse-grained" one, operating over a small field S and a large field L, respectively. We require that the number of elements of L be a power of that of S and that $k^{\mathfrak{s}} = \log(|L|)/\log(|S|)$ be true. We need to interpret a string of $k^{\mathfrak{l}}k^{\mathfrak{s}}$ elements from S as a string of $k^{\mathfrak{l}}$ elements of L, an operation we denote by S$\triangleright$L. The converse operation is denoted L$\triangleright$S.

The encoding function of our combined AoNT then works as follows. One first applies the coarse-grained AoNT to the whole data vector and then applies the fine-grained AoNT to each element of the result:

$$\mathsf{aenc}(\mathbf{a} \in S^{k^{\mathfrak{s}}k^{\mathfrak{l}}}) : \mathbf{x} \xleftarrow{\$} \mathsf{aenc}^{\mathfrak{l}}(\mathrm{S}\triangleright\mathrm{L}(\mathbf{a})); \forall j \in \{1, \ldots, n^{\mathfrak{l}}\} : \mathbf{b}[j] \xleftarrow{\$} \mathsf{aenc}^{\mathfrak{s}}(\mathrm{L}\triangleright\mathrm{S}(\mathbf{x}[j])); \text{ return } \mathbf{b}.$$

It's easy to see how the decoding function adec of the combined AoNT works and it is thus omitted.

Theorem 5.10. *Given a perfectly secure* $(S, n^{\mathfrak{s}}, d^{\mathfrak{s}}, k^{\mathfrak{s}})$*-AoNT* $(\mathsf{aenc}^{\mathfrak{s}}, \mathsf{adec}^{\mathfrak{s}})$ *and a perfectly secure* $(L, n^{\mathfrak{l}}, d^{\mathfrak{l}}, k^{\mathfrak{l}})$*-AoNT* $(\mathsf{aenc}^{\mathfrak{l}}, \mathsf{adec}^{\mathfrak{l}})$ *such that* $k^{\mathfrak{s}} = \log(|L|)/\log(|S|)$*, the*

$$\mathbf{G} = \begin{bmatrix} 111101101110000010111000 \\ 111111001111010101011111 \\ 011100111010001100001111 \\ 100110010110110011011010 \\ 110001100110100011000111 \\ 111110010100001001100110 \end{bmatrix}.$$

AoNT $(\mathsf{aenc}, \mathsf{adec})$ *described above is a perfectly secure* $(S, n^{\mathfrak{s}}n^{\mathfrak{l}}, (d^{\mathfrak{s}}+1)(d^{\mathfrak{l}}+1)-1, k^{\mathfrak{s}}k^{\mathfrak{l}})$*-AoNT.*

Proof. We now show that the combined scheme is secure. For any set E of at most $(d^{\mathfrak{s}}+1)(d^{\mathfrak{l}}+1)-1$ elements of $\{1,\ldots,k^{\mathfrak{s}}k^{\mathfrak{l}}\}$, and any two challenge messages $\mathbf{a}_1, \mathbf{a}_2 \in S^{k^{\mathfrak{s}}k^{\mathfrak{l}}}$; let E' be the following set: E' contains the integer $i \in \{1,\ldots,n^{\mathfrak{l}}\}$ if at least $(d^{\mathfrak{s}}+1)$ elements of $\{(i-1)n^{\mathfrak{s}}+1,\ldots,in^{\mathfrak{s}}\}$ are contained in E. Notice that E' has at most $d^{\mathfrak{l}}$ elements.

We now show $2n^{\mathfrak{l}}+2$ distributions, where any two consecutive distributions are perfectly indistinguishable. The first and the last distributions correspond to the two possible outputs of the AoNT distinguishing game; hence proving the claim.

Distribution $i \in \{1,\ldots,2n^{\mathfrak{l}}+2\}$ is $[\mathbf{b}]_E$, where $\mathbf{b}$ is calculated as follows:

$$\mathbf{x} \xleftarrow{\$} \begin{cases} \mathsf{aenc}^{\mathfrak{l}}(\mathrm{S}\triangleright\mathrm{L}(\mathbf{a}_1)) & \text{if } i \le n^{\mathfrak{l}}+1 \\ \mathsf{aenc}^{\mathfrak{l}}(\mathrm{S}\triangleright\mathrm{L}(\mathbf{a}_2)) & \text{otherwise.} \end{cases}$$

$$\forall j \in \{1,\ldots,n^{\mathfrak{l}}\} : \mathbf{x}'[j] \leftarrow \begin{cases} \mathbf{0} & \text{if } j \notin E' \wedge j+1 \le i \le j+1+n^{\mathfrak{l}} \\ \mathbf{x}[j] & \text{otherwise.} \end{cases}$$

$$\forall j \in \{1,\ldots,n^{\mathfrak{l}}\} : \mathbf{b}[j] \xleftarrow{\$} \mathsf{aenc}^{\mathfrak{s}}(\mathrm{L}\triangleright\mathrm{S}(\mathbf{x}'[j])).$$

It is easy to see that all two consecutive distributions except $(n^{\mathfrak{l}}+1)$ to $(n^{\mathfrak{l}}+2)$ are indistinguishable by the security property of the fine-grained scheme and since at most $d^{\mathfrak{s}}$ elements of $\mathbf{b}$ affect the output. The distributions $(n^{\mathfrak{l}}+1)$ and $(n^{\mathfrak{l}}+2)$ are indistinguishable because of the security property of the coarse-grained scheme and since at most $d^{\mathfrak{l}}$ elements of $\mathbf{x}$ affect the output.

□

Numerical example. Let us suppose that we are interested in a perfect AoNT that operates over $S = \mathrm{GF}(2)$ and that can store a cryptographic key of size $k = 256$ bits using at most $n = 8192$ bits (a kilobyte) of memory.

If we use a $(\mathrm{GF}(2^{10}), 819, 793, 26)$-AoNT built according to Franklin and Yung [FY92] unmodified over the field $\mathrm{GF}(2)$, we get a $(\mathrm{GF}(2), 8190, 793, 260)$-AoNT. This AoNT has a privacy threshold d of only 793 bits.

By combining a $(\mathrm{GF}(2), 32, 11, 8)$-AoNT (which can be found by exhaustive search) with a $(\mathrm{GF}(2^8), 255, 223, 32)$-AoNT built according to Franklin and Yung [FY92], one gets a $(\mathrm{GF}(2), 8160, 2687, 256)$-AoNT. This AoNT has a much better privacy threshold d of 2687, i.e., 2687 arbitrary bits may leak to the adversary.

5.3.4 Computationally Secure AoNT over a Large Field from a PRG

We now present a realization of a computationally secure AoNTs over a large field $\mathrm{GF}(2^\eta)$, where η is the security parameter. Our realization is optimal in the sense that it achieves both an optimal message length $k = n-1$ (thus an optimal rate $(n-1)/n$) and an optimal privacy threshold $d = n-1$. That is, the AoNT needs just a single additional element to encode a message and remains private even if the adversary obtains all but any one element.

Definition 5.11. *An ℓ-PRG where the output length is a multiple of the input length, i.e.,* $\mathsf{prg} : \mathrm{GF}(2^\eta) \mapsto \mathrm{GF}(2^\eta)^{\ell(\eta)/\eta}$*, is* KD-secure*, if for all $i = 1, \ldots, \ell(\eta)/\eta$, these ensembles are computationally indistinguishable:*

- $\{(x_1, \ldots, x_{i-1}, x'_i, x_{i+1}, \ldots, x_{\ell(\eta)/\eta})\}_{1^\eta}$ *where* $sk \xleftarrow{\$} \mathrm{GF}(2^\eta)$, $\mathbf{x} \leftarrow \mathsf{prg}(sk)$, *and* $x'_i \leftarrow x_i + sk$.
- $\{\mathbf{x}\}_{1^\eta}$ *where* $\mathbf{x} \xleftarrow{\$} \mathrm{GF}(2^\eta)^{\ell(\eta)/\eta}$.

Note that this property is somewhat reminiscent of the KDM-CCA2 security of encryption functions [CCS09].

Our realization, somewhat reminiscent of the OAEP realization of Canetti et al. [CDH+00], is as follows:

$$\mathsf{aenc}(\mathbf{m} \in \mathrm{GF}(2^\eta)^{\ell(\eta)/\eta}) : sk \xleftarrow{\$} \mathrm{GF}(2^\eta); \mathbf{x} \leftarrow \mathsf{prg}(sk); \mathbf{y} \leftarrow \mathbf{x} + \mathbf{m};$$
$$\text{return } \mathbf{y} || \big(sk + \sum_{i=1}^{\ell(\eta)/\eta} y_i\big).$$
$$\mathsf{adec}(\mathbf{y}||z) : \text{return } \mathbf{y} - \mathsf{prg}(z - \sum_{i=1}^{\ell(\eta)/\eta} y_i).$$

Theorem 5.12. *Given an ℓ-PRG that is both secure and KD-secure, the realization above yields a secure $(\mathrm{GF}(2^\eta), 1 + \ell(\eta)/\eta, \ell(\eta)/\eta, \ell(\eta)/\eta)$-AoNT.*

Proof. Recall that we need to prove that the output of the AoNT distinguishing game is computationally indistinguishable for any set L of size exactly $\ell(\eta)/\eta$. Let i denote the index that is missing from L. We treat now the two cases $i = \ell(\eta)/\eta + 1$ and $i \in \{1, \ldots, \ell(\eta)/\eta\}$ separately.

Case $i = \ell(\eta)/\eta + 1$. For any two challenge messages $\mathbf{m}$ and $\mathbf{m'}$, we consider the ensembles $\{\mathbf{y}\}_{1^\eta}$ computed as follows:

1. $sk \xleftarrow{\$} \mathrm{GF}(2^\eta)$ and $\mathbf{y} \leftarrow \mathbf{m} + \mathsf{prg}(sk)$.
2. $\mathbf{y} \xleftarrow{\$} \mathrm{GF}(2^\eta)^{\ell(\eta)/\eta}$.
3. $sk \xleftarrow{\$} \mathrm{GF}(2^\eta)$ and $\mathbf{y} \leftarrow \mathbf{m'} + \mathsf{prg}(sk)$.

Ensembles 1 and 2 on the one hand, and ensembles 2 and 3 on the other hand are computationally indistinguishable because the PRG is secure. Therefore ensembles 1 and 3, corresponding to the two ensembles output by the AoNT distinction game, are also computationally indistinguishable.

Case $i \in \{1, \ldots, \ell(\eta)/\eta\}$. For any two challenge messages $\mathbf{m}$ and $\mathbf{m'}$ we consider the ensembles $\{(y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_{\ell(\eta)/\eta}, z)\}_{1^\eta}$ computed as follows:

1. $sk \xleftarrow{\$} \mathrm{GF}(2^\eta)$; $\mathbf{x} \leftarrow \mathsf{prg}(sk)$; $\mathbf{y} \leftarrow \mathbf{m} + \mathbf{x}$; $z \leftarrow sk + \sum_j y_j$.
2. Idem, except that $z \xleftarrow{\$} \mathrm{GF}(2^\eta)$.
3. Idem, except that $\mathbf{y} \xleftarrow{\$} \mathrm{GF}(2^\eta)^{\ell(\eta)/\eta}$.
4. Idem, except that $\mathbf{y} \leftarrow \mathbf{m'} + \mathbf{x}$.
5. Idem, except that $z \leftarrow sk + \sum_j y_j$.

Ensembles 1 and 2 are computationally indistinguishable because the PRG is KD-secure. Indeed, for index i, let $\{(x_1, \ldots, x_{i-1}, e, x_{i+1}, \ldots, x_{\ell(\eta)/\eta})\}_{1^\eta}$ be the ensemble output from the PRG KD distinction game. Consider the ensemble $\{(y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_{\ell(\eta)/\eta}, z)\}_{1^\eta}$ where for $j \neq i : y_j \leftarrow x_j + m_j$ and $z \leftarrow \sum_{j \neq i} y_j + e + m_i$. If e is equal to $x_i + sk$, this is exactly ensemble 1; if e is random, this is exactly ensemble 2.

Ensembles 2 and 3 are computationally indistinguishable because the PRG is secure. Indeed, let $\{\mathbf{e}\}_{1^\eta}$ be the ensemble output by the PRG security game. Consider the ensemble $\{(y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_{\ell(\eta)/\eta}, z)\}_{1^\eta}$ where for $j \neq i : y_j \leftarrow e_j + m_j$ and $z \xleftarrow{\$} \mathrm{GF}(2^\eta)$. If $\mathbf{e}$ is equal to $\mathbf{x}$, this is exactly ensemble 2; if $\mathbf{e}$ is random, this is exactly ensemble 3.

Ensembles 3 and 4 are computationally indistinguishable because the PRG is secure. Indeed, let $\{\mathbf{e}\}_{1^\eta}$ be the ensemble output by the PRG security game. Consider the ensemble $\{(y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_{\ell(\eta)/\eta}, z)\}_{1^\eta}$ where for $j \neq i : y_j \leftarrow e_j + m'_j$ and $z \xleftarrow{\$} \mathrm{GF}(2^\eta)$. If $\mathbf{e}$ is random, this is exactly ensemble 3; if $\mathbf{e}$ is equal to $\mathbf{x}$, this is exactly ensemble 4.

Ensembles 4 and 5 are computationally indistinguishable because the PRG is KD-secure. Indeed, for index i, let $\{(x_1, \ldots, x_{i-1}, e, x_{i+1}, \ldots, x_{\ell(\eta)/\eta})\}_{1^\eta}$ denote the ensemble output by the PRG security game. Consider the ensemble $\{(y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_{\ell(\eta)/\eta}, z)\}_{1^\eta}$ where for $j \neq i : y_j \leftarrow x_j + m'_j$ and $z \leftarrow \sum_{j \neq i} y_j + e + m'_i$. If e is random, this is exactly ensemble 4; if e is equal to $x_i + sk$, this is exactly ensemble 5.

Hence ensembles 1 and 5, corresponding to the ensembles output by the AoNT distinction game, are computationally indistinguishable.

□

In Section 2.4.9.4 we observed that Canetti et al.'s [CDH+00] computationally-secure AoNT built by combining an exposure resilient function (ERF) with a pseudo-random generator (PRG) can have an essentially arbitrarily high message length k and message rate k/n, but cannot achieve a very high privacy threshold d.

6

Conventions for Usable Universal Composability

Over the last two decades, a series of composition frameworks have been proposed [Can00, Can01, PW00, Kue06, HS11, KT13, Mau11b, MR11, CDPW07, BPW07]. However, all these frameworks are either too restrictive or too general to be (easily) used in a formally correct way by protocol designers. As an example of the former, the *GNUC* framework by Hofheinz and Shoup [HS11] supports "top-down" corruption only, meaning that machines can be corrupted only if their parent machine is already corrupted as well, leaving out other forms of corruption. Both the UC and GNUC models fix a specific addressing mechanism directly in their underlying communication model based on party idenifiers (PIDs) and session identifiers (SIDs). While convenient for hierarchical protocol design, handling protocols with joint state then is out of scope or requires rather cumbersome extensions of the models [CR03, HS11], which in some cases causes problems (see the discussion in [KT08]). Moreover, as already discussed in [HS11], because of the runtime notions used in the GNUC model some natural functionalities cannot be realized, such as certain digital signature and public-key functionalities proposed in [Can00] (version of 2005). Conversely, the *IITM* model [Kue06, KT13] aims at highest possible generality, and only fixes as few details as possible. In particular, wiring, addressing, or corruption mechanisms are not specified in the framework. While this allows for general theorems, in particular composition theorems, it leaves a lot for the protocol designer to specify. Partly also the latest version of the UC framework, which is motivated by some fundamental flaws in previous versions, leaves more freedom to the protocol designers but thereby places more burden on them. The *Constructive Cryptography* framework by Maurer et al. [Mau11b, MR11] follows a very abstract approach and certain concepts, such as dynamic corruptions and runtime, are not yet modeled in the published literature.

Furthermore, none of the existing frameworks gives clear guidelines to protocol designers on how the framework has to be used, leading to numerous formally underspecified protocols in the literature and to protocols that are hard to compare. This makes it very difficult, if not impossible, to use such protocols as subroutines, hence defeating the very purpose of these frameworks, namely modular protocol design. For instance, it is often (e.g., [AMQR15, Wik04]) not clear against which types of adversaries the protocol is secure, as protocol designers only state security against "static corruption". Seemingly, they sometimes mean strong static corruption (the adversary may only corrupt parties

at the beginning of the session) and sometimes weak static corruption (the adversary may corrupt a machine upon its first activation). Moreover, runtime issues are largely ignored in the specification of protocols, even though, in some models they are quite subtle and tricky. For example, in the UC framework a machine is not allowed to send more bits to subroutines than it received on its I/O interface; if it does, it is forced to immediately stop its execution.

What is thus needed is a formally sound and expressive framework, which allows the protocol designer to specify protocols, on the one hand, in a flexible and convenient way, but, on the other hand, *enforces* fully specified protocols.

In this chapter, we provide an abstraction layer with templates on top of the IITM model with responsive environments (see Section 2.5.3 and [CEK+16a]) to allow for concise yet flexible protocol design, including handling of joint state. Finally, we provide a precise mapping from the templates to our model model for unambiguous and complete protocol specifications.

Roadmap. We provide the templates supporting a protocol designer in specifying ideal and real protocols in Section 6.1. We give the details of the machinery that maps specifications using these templates onto concrete interactive Turing machines in Section 6.2. We then introduce a programming language for functionalities and protocol machines in Section 6.3, and exemplify our conventions based on some examples in Section 6.4. Finally, we extend our conventions to joint state, and provide a detailed example in Section 6.5.

6.1 Templates for Real Protocols and Ideal Functionalities

In the following, we provide templates for efficiently and unambiguously specifying real as well as ideal protocols. We explain which parts of these protocols need to be specified by the protocol designer; all other parts will be formally defined once and forever in Section 6.2, where we formally specify the protocol systems induced by our templates. In Section 6.3, we will then suggest some concrete pseudocode elements that can be used to describe the different parts of our templates.

Before presenting the templates, we briefly sketch the overall structure of real and ideal protocols as well as the way corruption is handled, with full details provided in Section 6.2.

An *atomic real protocol* $\mathcal{R}$ is a system of ITMs of the form $\{M_{\mathsf{role}_1}, \ldots, M_{\mathsf{role}_n}\}$, where M_{role_i} is an ITM (in this case also called *real protocol machine*), which specifies the i-th role of the protocol $\mathcal{R}$, cf. Figure 6.2. As usual, roles are used to group protocol participants running the same code; however, it is up to the protocol designer to decide how and whether she wants to split the protocol participants into roles. Each machine in $\mathcal{R}$ has one bidirectional network tape to connect to the adversary. It also has I/O tapes to connect to other machines. However, no machine in $\mathcal{R}$ is connected via any tape to another machine in $\mathcal{R}$. In our conventions, I/O tapes are divided into parent tapes and subroutine tapes, to connect to higher-level protocols or the environment, and to subroutines, i.e., lower-level protocols, respectively. In a run of $\mathcal{R}$, by our conventions, each instance of M_{role_i} has an identity (ID) id, which is set upon the first activation of the instance, and cannot be changed later on. The ID is of the form (pid, sid), where pid

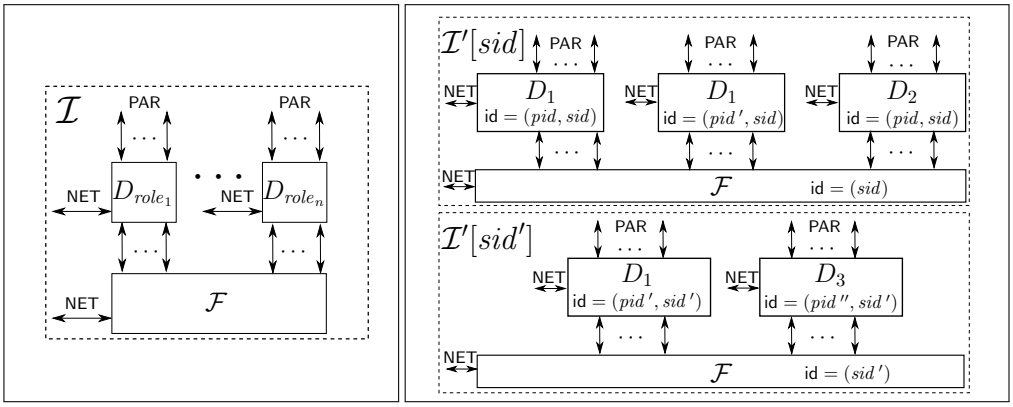


Fig. 6.1: Left: Static structure of an ideal protocol $\mathcal{I} = \{D_{role_1}, \ldots, D_{role_n}, \mathcal{F}\}$ with n roles and an ideal functionality $\mathcal{F}$. Dummies have an arbitrary number of PAR-tapes (and the same number of SUB-tapes connected to $\mathcal{F}$) for communication with higher level protocols or the environment.
Right: Example of a possible dynamic structure of an ideal protocol $\mathcal{I}' = \{D_1, D_2, D_3, \mathcal{F}\}$, where two sessions *sid* and *sid*′ were created by parties *pid*, *pid*′, and *pid*″. Note that instances of the same machine use the same tape names; mode **CheckAddress** is used to find the correct instance for each message. Note that *pid* takes part in two different roles in the same session *sid*.

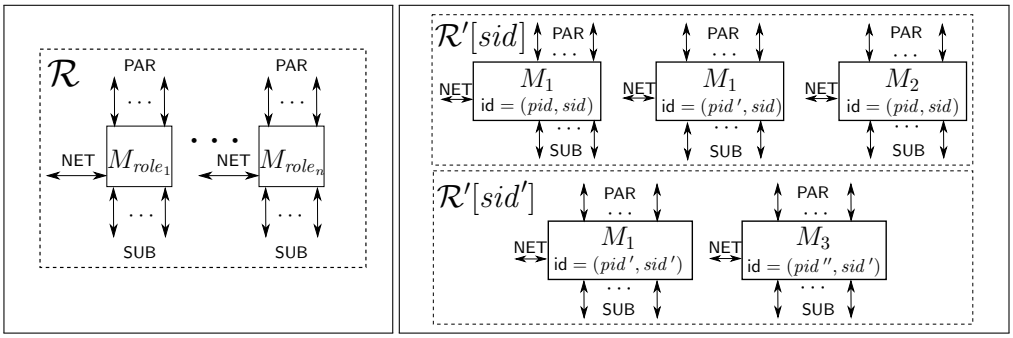


Fig. 6.2: Left: Static structure of an atomic real protocol $\mathcal{R} = \{M_{role_1}, \cdots, M_{role_n}\}$ with n roles. Each role has arbitrary numbers of SUB and PAR tapes to communicate with other protocols and the environment, respectively.
Right: Example of a possible dynamic structure of an atomic real protocol $\mathcal{R}' = \{M_1, M_2, M_3\}$ that is meant to realize the ideal protocol $\mathcal{I}' = \{D_1, D_2, D_3, \mathcal{F}\}$ from the right side of Figure 6.1. Note that every instance of a real protocol machine corresponds to one instance of a dummy, but the real protocol machines may additionally have subprotocols (as shown in Figure 6.3). Also note that, just as in Figure 6.1, instances of the same machine use the same tape names; mode **CheckAddress** is used to find the correct instance for each message.

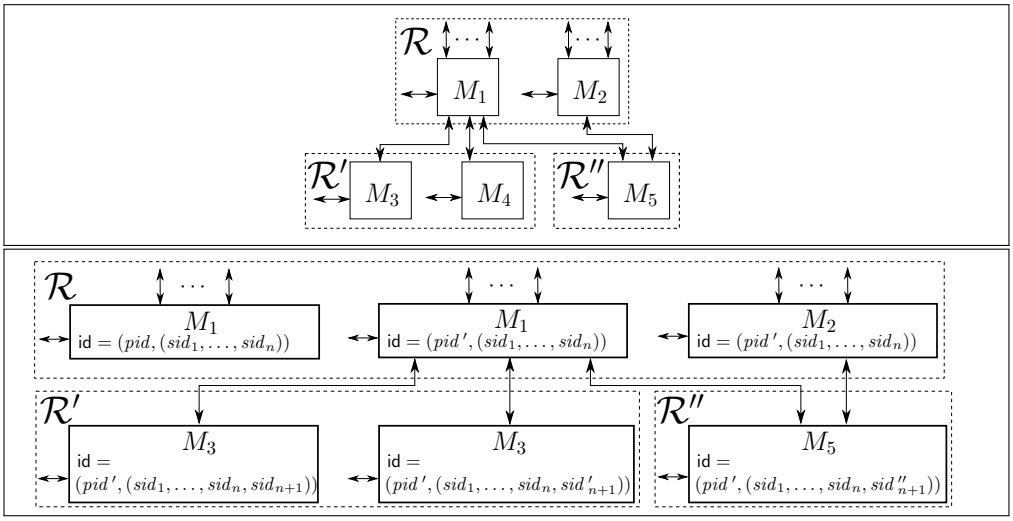


Fig. 6.3: Top: Example of the static structure of a real protocol $\mathcal{R} = \{M_1, M_2\}$, where M_1 uses the real protocol $\mathcal{R}' = \{M_3, M_4\}$ and both M_1 and M_2 use the real protocol $\mathcal{R}'' = \{M_5\}$.
Bottom: Example of a possible dynamic structure of the real protocol $\mathcal{R}$ and its subroutines $\mathcal{R}'$ and $\mathcal{R}''$ from the upper part of this figure. Note that instances of real protocol machines can only talk to each other if they belong to the same party. Also note that, just as in Figure 6.1, instances of the same machine use the same tape names; mode **CheckAddress** is used to find the correct instance for each message.

is a party ID (PID) and $sid = (sid_1, \ldots, sid_n)$ is the session ID (SID). We refer to the instance of M_{role_i} with identity id as $M_{\mathsf{role}_i}[\mathsf{id}]$. The structure of identities is such that any subroutine instance of $M_{\mathsf{role}_i}[\mathsf{id}]$ has an ID of the form $id' = (pid, (sid_1, \ldots, sid_n, sid_{n+1}))$, i.e., only the SID gets extended by sid_{n+1}. In particular, any subroutine instance of $M_{\mathsf{role}_i}[\mathsf{id}]$ has the same PID as $M_{\mathsf{role}_i}[\mathsf{id}]$. Similarly, a parent instance of $M_{\mathsf{role}_i}[\mathsf{id}]$ has ID $\mathsf{id}'' = (pid, (sid_1, \ldots, sid_{n-1}))$. We note that, while every ID identifies a unique instance of a single role, IDs are not necessarily unique across different roles. Hence, unlike other frameworks, it is possible for a single party pid to play different roles in the same session sid. See Figures 6.2 and 6.3 for an example.

An *ideal protocol* $\mathcal{I}$ is a system of ITMs of the form $\{D_{\mathsf{role}_1}, \ldots, D_{\mathsf{role}_n}, \mathcal{F}\}$. The ITMs $D_{\mathsf{role}_1}, \ldots, D_{\mathsf{role}_n}$ are called *dummies*, akin to the "ideal peers" in GNUC or "dummy parties" in UC. A dummy D_{role_i} corresponds to a role M_{role_i} in a real protocol. However, dummies essentially only act as forwarders between a higher-level protocol or the environment and the ideal functionality $\mathcal{F}$, which realizes the actual cryptographic task. Just like real protocol machines, a dummy has a network tape to the adversary as well as parent tapes. For every parent tape it has a corresponding subroutine tape to $\mathcal{F}$ which is used to forward messages. The functionality $\mathcal{F}$ has a network tape, no subroutine tapes, and for every subroutine tape of a dummy a corresponding parent tape. In a run of $\mathcal{I}$, each instance of dummy has an ID of the form $(pid, (sid_1, \ldots, sid_n))$ just as

their corresponding real machines. An instance of a dummy with such an ID talks to an instance of the ideal functionality $\mathcal{F}$ with ID $(sid_1, \ldots, sid_n)$. Note that the ID of an instance of $\mathcal{F}$ does not contain a specific PID since such an instance performs cryptographic tasks for all parties in session $(sid_1, \ldots, sid_n)$.

A *real protocol* is a composition of atomic real protocols and ideal protocols. We might have one or more higher-level (real) protocols which may use one or more real protocols as subroutines, which in turn may have subroutines, etc. (see Figure 6.3 for an example). Ideal protocols might be subroutines as well, in which case we have a hybrid system.

Figure 2.1 (on page 27) shows an example of an ideal protocol $\mathcal{I}$ which is realized by a real protocol $\mathcal{P} = \{\mathcal{R}, \mathcal{I}'\}$, where $\mathcal{R}$ makes use of the ideal protocol $\mathcal{I}'$ as a subroutine. The protocol $\mathcal{R} = \{M_1, M_2\}$ consists of two roles M_1 and M_2. In this case, the subroutine $\mathcal{I}'$ has two roles as well, but in general higher- and lower-level protocols are not required to have the same number of roles. Since $\mathcal{R}$ has two roles, the ideal protocol $\mathcal{I}$ it is supposed to realize has two roles as well. In Figure 6.1 an example for a possible instantiation of an ideal protocol $\mathcal{I}'$ with two sessions is given, while Figure 6.2 shows the corresponding instances of a real protocol $\mathcal{R}'$ which tries to realize $\mathcal{I}'$. In this case both protocols have three roles, which do not necessarily have instances in every session.

The adversary can explicitly corrupt real protocol machines and dummies by sending a corruption request on their network tape. Depending on whether (strong/weak) static corruption or dynamic corruption is modeled, such a corruption request might not always be possible, though. Also, the machine might reject the corruption request. In particular, a real protocol machine may first check the corruption status of its subroutines and then decide, depending on its own state, whether or not corruption is granted; dummies ask their ideal functionality whether corruption is allowed. For example, a real protocol machine might reject the corruption request if not all of its subroutine instances are corrupted. By this, bottom up corruption can be enforced. Once a corruption request is granted by a machine, the machine from then on is (explicitly) corrupted and just acts as a forwarder between the environment/the higher level protocol and its subroutines on the one hand and the adversary/simulator/environment on the other hand. In other words, the machine is in full control of the adversary/simulator/environment.

Every time a fresh instance enters **Compute** mode, it first asks the adversary with a special message whether he wants to corrupt this instance. The answer of the adversary is processed as described in the last paragraph, i.e., the instance may decide to reject a corruption request.

On the I/O interface, an environment or a higher-level protocol can ask at any time about the corruption status of (an instance of) a machine. If the machine was explicitly corrupted by the adversary, then the machine returns `true` immediately. Otherwise, the machine may check the corruption status of its subroutines first in order to determine the corruption status it returns. If, for example, one of its subroutine instances was corrupted, then the machine might return `true` as its corruption status. We emphasize that it might do so even if the machine itself was not (explicitly) corrupted by the adversary, and hence, follows its honest program. So, the status returned represents the overall status of the machine *and* its subroutines. This can be used to, e.g., model top-down corruption by considering a machine corrupted if at least one subroutine is corrupted. Then, as

Protocol Setup for $\mathcal{R} = \{M_{\mathsf{role}_1}, \ldots, M_{\mathsf{role}_n}\}$:

Participating roles: List of all roles participating in this protocol.
Corruption model: strong/weak static, dynamic with/without erasures.
Protocol parameters*: e.g., externally provided algorithms can parametrize a machine.

Realization M_{role_i} for each participating role role_i:

Implemented role: name of the role implemented by this machine, i.e., role_i.
Subroutines*: typically lists sub-functionalities if protocol is hybrid.
Internal state*: state variables used to store data across different invocations.
CheckIDformat*: algorithm to check format of ID.
Corruption behavior*: description of DetermineCorruptionStatus and AllowCorruption?.
Initialization*: this block is only executed the first time the machine receives a message; useful to assign initial values, etc.
MessagePreprocessing*: this block is executed every time a message is received.
Main: description of the actual behavior of the uncorrupted machine.

Fig. 6.4: Template for specifying ARPs (atomic real protocols). Blocks labeled with an asterisk are optional. **CheckIDformat** is part of the **CheckAddress** mode, whereas **Corruption behavior**, ..., **Main** are all executed within the **Compute** mode of the machine.

soon as a machine is corrupted by the adversary/simulator/environment, all higher level protocols that use this machine as subroutine (directly or indirectly) will automatically consider themselves to be corrupted, too.

The templates presented in the following will be used to specify the behavior of real protocol machines (i.e., the M_i), as well as ideal functionalities (i.e., $\mathcal{F}$). All other parts of Figure 2.1 (on page 27), in particular the behavior of dummies or the precise wiring of protocols and their subroutines is fixed in Section 6.2, and do not need to be specified by a designer. As for corruption, the protocol designer still has some freedom and can specify the desired behavior by certain algorithms she may specify as part of the protocol specification (otherwise defaults are used).

6.1.1 Specifying Real Protocols

Figure 6.4 shows the template for specifying (atomic) real protocols (ARPs) of the form $\mathcal{R} = \{M_{\mathsf{role}_1}, \ldots, M_{\mathsf{role}_n}\}$. This template consists of two parts: the protocol setup part, which specifies properties for the entire protocol, and the realizations of all roles M_{role_i} for $i = 1, \ldots, n$.

The protocol setup block of the protocol specification specifies the following properties:

Participating roles. This block simply lists the names of all roles participating in the protocol.

Corruption model. Here, one specifies the considered corruption model. Our conventions distinguish the following types of corruption: *strong static*, *weak static*, and *dynamic with/without erasures*. Each type makes fewer assumptions about the adversary, thus providing stronger security guarantees than the previous one.

Informally, *strong static corruption* means that the adversary has to determine upfront which instances are involved in a session and their corruption status. For *weak static corruption* the adversary can decide whether he wants to corrupt an

instance the first time this machine enters the **Compute** mode. He does not have to fix the instances involved in a session upfront. In the case of *dynamic corruption*, the adversary may corrupt every machine at any point in time; in this case, it needs to be specified whether one assumes secure erasures or not, i.e., whether temporary variables are assumed to be ephemeral or whether they are leaked to the adversary upon corruption.

Protocol parameters. If the given protocol is parametrized, the respective parameters as well as their required properties have to be specified here. This is often the case if $\mathcal{R}$ realizes an ideal protocol that outputs concrete cryptographic values. For instance, the realization of the (standard) signature functionality $\mathcal{F}_{\text{sig}}$ is typically parametrized by an EUF-CMA secure signature scheme, cf., e.g., [KT08] and Section 6.4.

Apart from the protocol setup, one has to specify each real procotol machine M_{role_i}, and hence, each role listed in the protocol setup.

Implemented role. This simply states which role is implemented by this machine. This name is used to uniquely identify each role.

Subroutines. Here the protocol designer lists all subroutines used by M_{role_i}. The machine will then be connected to all roles of that subroutine, and hence can (but does not need to) use all these roles. In most cases, the subroutines will be ideal functionalities, which then together with the higher-level protocol specify a hybrid system. If the protocol does not realize a message transmission functionality, this block typically at least contains a functionality for (insecure/secure/authenticated) channels between protocol participants.

Internal state. In this optional block one can declare state variables whose values are preserved across different activations of an instance of M_{role_i}. Such variables are always denoted by sans-serif fonts, e.g., a, b.

Besides the protocol-specific internal state variables, by our conventions, every machine has one default permanent variables that can be accessed by the protocol designer: A state variable id containing the identity of the machine of the form $(pid, (sid_1, \ldots, sid_n))$ as explained before. This variable is initialized by $\perp$. It must not be set by the protocol designer, but is assigned and changed according to our conventions by framework-specific procedures, cf. Section 6.2. Some additional framework-specific (and therefore hidden) state variables will be detailed in Section 6.2.

CheckIDformat. As mentioned before, by our conventions the **CheckAddress** mode of M_{role_i} ensures that every instance of M_{role_i} has an ID of the form $(pid, (sid_1, \ldots, sid_n))$. Often it is convenient to enforce more structure on the ID. For instance, it may often be convenient to require that $sid_n = ((pid_1, \ldots, pid_k), sid_n{}')$, in order to encode the identities of distinguished parties into the SID. For example, for the signing functionality $\mathcal{F}_{\text{sig}}$, this allows one to encode the identity of the signer into the SID, i.e., we would have $sid_n = ((pid_{\mathsf{signer}}), sid_n{}')$ in this case. By checking that a signing request was sent by a machine with party ID pid_{signer}, the functionality can now easily ensure that only the legitimate owner of the secret key can sign messages in any given session. We refer to Section 6.4 for details on this example.

Therefore, in this optional block of the machine specification the protocol designer can enforce such a structure by specifying a deterministic, polynomial-time

(in the size of its input and the security parameter), non-interactive algorithm **CheckIDformat**$(m, role)$, which takes the current input m and the role *role* that sent m (i.e., the tape that this message was received on) as input and outputs accept or reject. This algorithm is performed within the **CheckAddress** mode of M_{role_i} when the machine does not have an ID yet. (Once the machine has entered **Compute** for the first time and has set its ID, the machine, in **CheckAddress** mode, only accepts messages that are prefixed with this ID.) Although formally id has not been initialized during the execution of **CheckIDformat**, we allow an author to write id to refer to the ID which is currently being checked by this algorithm (and which is part of the message m). If this block is not specified, **CheckIDformat** always returns accept.

Corruption behavior. As mentioned before, an adversary may send a corruption request to a machine in order to (explicitly) corrupt a machine. However, the machine may reject this request (e.g., after having checked the corruption status of its subroutine instances). The protocol designer should specify the exact behavior upon receiving corruption requests by providing an algorithm AllowCorruption?; otherwise the machine behaves in a default way (see below). Also, as mentioned, the environment and higher-level protocols may ask for the corruption status of a machine. In order to determine this status, the machine may again consult its subroutine instances first. The protocol designer should therefore specify the exact behavior by providing an algorithm DetermineCorruptionStatus; again a default behavior is specified by our conventions if no algorithm is provided.

More specifically, the algorithm AllowCorruption?(id, internalState, *initialMessage*) is a deterministic, non-interactive algorithm which is executed every time the adversary tries to corrupt an *uncorrupted* machine. It gets the id and the current internal state (which consists of all variables specified in the *Internal state*-block) of this instance as input. Furthermore, if the instance was activated by an initial message for the first time in **Compute** mode and thus asked the adversary for its corruption state, then this initial message is also given to AllowCorruption? (otherwise *initialMessage* is $\perp$). Before the algorithm outputs a bit indicating whether or not the corruption request by the adversary is accepted, it may ask about the corruption status of its subroutine instances (this is the only exception to the non-interactivity of this algorithm and we define a special macro for it in Section 6.2 such that a protocol designer must never use a send-command in this algorithm). After the algorithm finished, the state of the machine is set to the one before executing this algorithm, i.e., this algorithm can not change the state of the machine. If not specified, this algorithm always returns true.

The algorithm DetermineCorruptionStatus(id, internalState) is a deterministic, non-interactive algorithm which is executed every time the environment/a higher level protocol asks for the corruption status of an *uncorrupted* machine. It gets the id and the internal state of this instance as input. It may ask for the corruption status of its subroutine instances (this is the only exception to the non-interactivity and we define a special macro for it in Section 6.2 such that a protocol designer must never use a send-command in this algorithm), before it outputs a bit indicating whether or not the machine considers itself as corrupted. After the algorithm finished, the state of the machine is set to the one before executing this algorithm,

i.e., this algorithm can not change the state of the machine. Note that if the machine was explicitly corrupted, i.e., it received a corruption request from the adversary which was accepted, then, by our conventions, true is returned in any case. Furthermore, if the algorithm DetermineCorruptionStatus outputs true once, then all following requests are answered with true immediately without calling DetermineCorruptionStatus again. Therefore, DetermineCorruptionStatus is invoked only if the machine has never considered itself corrupted before (either because it was explicitly corrupted or DetermineCorruptionStatus returned true) in order to determine the corruption status it returns. If not specified, DetermineCorruptionStatus always outputs false by default.

Initialization. If this optional non-interactive algorithm is specified, it is executed in mode **Compute** only upon receiving the first non–corruption-related message. (Being executed in mode **Compute** implies that the message received has been accepted in mode **CheckAddress** before.) This algorithm gets the current input m, the role *role* that sent this message (i.e., the tape that m was received on), the ID id and the internal state as input. Typically, it is used to assign initial values to variables or to set up key material used in the protocol. For this purpose, this algorithm may use the restricting `Respond` message to ask the adversary for some information necessary for initialization like, e.g., algorithms. Note that this is the only exception to the non-interactivity and only allowed because the adversary is forced to answer "immediately" such that the computation is not interrupted. More precisely, this algorithm may use the "send responsively" command defined in Section 6.3.

MessagePreprocessing. If this optional algorithm is specified, it is executed in mode **Compute** every time a non–corruption-related message is received. (The first such message will, however, first trigger the initialization explained above.) This algorithm gets the same input as **Initialization**. Note that this algorithm may send messages to other machines and thus potentially end the execution of the **Compute** mode. This algorithm is useful to specify certain tasks and checks that should be executed for all incoming non-corruption related messages in case the machine is not corrupted. For example, malformed message could be handled by **MessagePreprocessing**.

Main. This block specifies the main computation performed in mode **Compute**. It gets the same input as **Initialization** and **MessagePreprocessing** and is performed only after **Initialization** and **MessagePreprocessing** (if **MessagePreprocessing** did not stop the run by sending a message).

There are some framework specific messages that may not be sent in **MessagePreprocessing** and **Main**; these are noted in Section 6.2. In Section 6.3, we provide a convenient syntax for the specification of the blocks **Initialization**, **MessagePreprocessing**, and **Main**.

6.1.2 Specifying Ideal Functionalities

Recall that an ideal protocol $\mathcal{I}$ is a system of the form $\{D_{\mathsf{role}_1}, \ldots, D_{\mathsf{role}_n}, \mathcal{F}\}$. We next present a template for specifying ideal protocols in Figure 6.5. A protocol designer mainly has to specify the ideal functionality. The behavior of dummies is mostly fixed by

Protocol Setup for $\mathcal{I} = \{D_{\mathsf{role}_1}, \ldots, D_{\mathsf{role}_n}, \mathcal{F}\}$:

Participating roles: List of all roles participating in this protocol.
Corruption model: strong/weak static, dynamic.
Protocol parameters: e.g., externally provided algorithms can parametrize a machine.

Description of $\mathcal{F}$:

Internal state*: state variables used to store data across different invocations.
CheckIDformat$\{role_i, \ldots, role_j\}$*: algorithm to check the well-formedness of the session ID (executed by dummies in roles $role_i, \ldots, role_j$).
⋮
CheckIDformat$\{role_k, \ldots, role_l\}$*: algorithm to check the well-formedness of the session ID (executed by dummies in roles $role_k, \ldots, role_l$).
CheckIDformat$\{\}$*: algorithm to check the well-formedness of the session ID (executed by all remaining dummies).
CheckIDformatIdeal*: algorithm to check the well-formedness of the session ID (executed by ideal functionalities for messages from the adversary).
Corruption behavior*: description of AllowDummyCorruption? and LeakedData.
Initialization*: an algorithm that is only executed for the first message that is received; useful to assign initial values, etc.
MessagePreprocessing*: an algorithm that is executed every time a message is received.
Main: description of the actual behavior of the ideal functionality.

Fig. 6.5: Template for specifying ideal protocols. Blocks labeled with an asterisk are optional.

our conventions and does not have to be specified by the protocol designer. However, the protocol designer has to specify the list of roles, and hence, the number of dummies. Also, just as for real protocol machines, the protocol designer may want to impose specific restrictions on the form of IDs of dummies (and thus on the IDs of the ideal functionality $\mathcal{F}$) by specifying the algorithm **CheckIDformat**. Just as for real protocol machines, this algorithm is executed as part of the **CheckAddress** mode of a dummy. Otherwise, the dummies, including their behavior related to corruption, are fully specified by our conventions.

The blocks "Participating roles" and "Protocol parameters" in the protocol setup specification in Figure 6.5 as well as the blocks "Initialization", "MessagePreprocessing", and "Main" correspond to those for real protocols. The corruption model does not need to distinguish between dynamic corruption with and without erasures anymore, as explained below. The remaining blocks are explained next.

Internal state. Similar to real machines, there are two framework-specific state variables that are visible to the protocol designer: the identity id of the form $(sid_1, \ldots, sid_n)$ and a variable CorruptionSet, which keeps track of all corrupted *dummies* that are connected to this instance of the functionality, where a dummy is uniquely identified by a pair $(role, pid)$ describing the role und the PID of the dummy instance (the SID is the same as the one of the functionality). Again, id and CorruptionSet must not be set by the protocol designer, but are automatically changed by the framework.

CheckIDformat$\{$*set of roles*$\}$**.** This block specifies the **CheckIDformat** that is executed by the dummies in one of the roles listed in *set of roles*; the same conventions as for RPMs also apply here (in particular, we allow the usage of id to refer to the ID of

the dummy that is currently being checked. Note that this ID has a different format as the one of an ideal functionality since it includes a PID). A protocol designer is free to let an arbitrary number of roles use the same **CheckIDformat** by including them in the *set of roles*, or specify one **CheckIDformat** per role by defining only one role in *set of roles*. If no role is specified, then this **CheckIDformat** is executed by all roles that are not defined in any *set of roles* of another **CheckIDformat**, i.e., this can be used to define some default **CheckIDformat**. One role may not be specified in more than one *set of roles*, i.e., it has to be unambigous which **CheckIDformat** is used. If a role is not included in any *set of roles* and no default **CheckIDformat** is specified, it behaves as if **CheckIDformat** always returns accept.

CheckIDformatIdeal. Since all PAR tapes of an IF connect to dummies and we already specified **CheckIDformat** for those dummies, we can re-use this algorithm to check the IDs of message on PAR tapes. However, there may also be messages from the adversary on the NET tape of the ideal functionality. If such a message arrives and activates a fresh instance of the ideal functionality (i.e., this instance has never accepted any messages before), then, after the usual checks, this algorithm is executed to enforce a specific format for the session ID during the **CheckAddress** mode of the ideal functionality. For this algorithm the same conventions apply as for **CheckIDformat**. Note that, if this algorithm uses id to refer to the ID that is being checked, then id is of the form $(sid_1, \ldots, sid_n)$, unlike in the **CheckIDformat** algorithm.

Corruption behavior. As explained before, upon receiving a corruption request by the adversary, a dummy asks its functionality whether the request is granted. (If yes, the dummy is under complete control of the adversary.) For this purpose, the protocol designer can specify an algorithm AllowDummyCorruption?, similarly to AllowCorruption? in the case of real protocols.

To be more precise, AllowDummyCorruption?(pid, $role$, id, CorruptionSet, internalState, messageList) is a deterministic, non-interactice algorithm that may depend on the specific instance of the dummy (which is uniquely identified by its pid and $role$), the SID id, the currently corrupted dummy instances and the internal state of the ideal functionality. Additionally, it can depend on a list of all messages that were sent to or received from every dummy instance. Note that this also includes a special variable *initialMessage* from the corruption request of the dummy, which contains either the first message that was received by that dummy instance (if the dummy is fresh and currently determines its initial corruption) or $\perp$ of the dummy instance is no longer fresh. The algorithm is executed by the ideal functionality every time a dummy asks whether it may be corrupted.[1] It outputs either true or false and after that, the state of the ideal functionality is set to the state before the execution of this algorithm, i.e., this algorithm can not change the state of the ideal functionality. Since dummies already internally take care of dynamic and static corruption, it is not necessary to differentiate between those in this algorithm (i.e., it will only be

[1] Note that the ideal functionality first executes the **Initialization** algorithm, if this is the first message that was received in **Compute** mode. Neither **MessagePreprocessing** nor **Main** are executed upon such a request.

called if allowed by the current corruption mechanism). If this algorithm is not specified, it returns true by default.
Now, if the corruption request is accepted (according to AllowDummyCorruption?), the ideal functionality also decides which data is leaked to the adversary, i.e., which data is forwarded via the dummy to the adversary. The protocol designer can specify which data is leaked by the deterministic, poly-time, non-interactive algorithm LeakedData. More specifically, LeakedData(*pid*, *role*, id, CorruptionSet, internalState, messageList) gets the same input as AllowCorruption?. Note that by this algorithm the protocol designer can specify whether (dynamic) corruption with or without secure erasures is modeled, which is why this, unlike in the case of real protocols, is not explicitly stated in the protocol setup specification. If LeakedData is not specified, it will return the transcript of all communication between the corrupted instance of the dummy and the functionality, cf. also Section 6.2.3.3.

6.2 Mapping Templates to ITMs

In the following, we explain how the templates specified in Section 6.1 are mapped to actual systems in the sense of the IITM model with responsive environments. The resulting systems will be instantiations of the protocol systems in the responsive IITM model (see Section 2.5.3 and [CEK+16a]). Hence, all theorems in the responsive IITM model, such as composition theorems, will still hold true for these systems.

The mapping from templates to actual systems will fix all the framework-specific parts of our conventions which do not need to be specified by the protocol designer, thereby remedying the drawback of the original very general IITM model where the specific addressing conventions, the specific protocol structure for real and ideal protocols as well as corruption behavior are left open, and hence, left to the protocol designer.

As a result of this translation from templates to actual systems of ITMs, the protocol designer only has to take care of protocol specific aspects of a specification. Using our conventions, the modeling thus becomes close to that in other universal composability frameworks, making it easy for protocol designers to transition from those models to ours. However, as discussed in the introduction of this chapter our conventions have several advantages compared to other models.

6.2.1 Notation for the Formal Specification of ITMs

Before being able to formally specify how the resulting ITMs look like, we need to introduce some notation. This notation will only be used for this compilation, and partially in security proofs of protocols when specifying the simulators in full details. In Section 6.3, we will introduce some more abstract notation that allows for a more convenient and direct instantiation of our templates.

Message patterns. A message pattern *mp* is used to describe the format of a message $m \in \{0,1\}^* \cup \{\bot\}$. It is built from *local variables* (denoted in italic font), global variables (denoted in sans-serif font), `strings` (denoted in typewriter font) and special characters such as "(", ")", "," and "⊥".

Message patterns can be used to describe outgoing messages, in the following denoted by mp_{out}, and incoming messages, in the following denoted by mp_{in}. If a message pattern is used for sending, then the current values of global and local variables are inserted, while the remainder of the pattern stays as is (in particular, strings and special signs are not altered). The resulting message is then sent. If a message pattern is used for receiving, then, if a message m is received, it is matched against the pattern: After inserting the values of global variables and, if defined, those of local variables into mp_{in}, the resulting message must be the same as m except for undefined local variables, which match to an arbitrary text. After a successful match, all local variables contain the value that they matched on. The special symbol _ can be used in mp_{in} instead of an undefined local variable if the value that is matched on is not needed anymore.

Sending raw messages. When we write **send** mp_{out} **on** t, we mean that the message m that is created from mp_{out} at runtime is sent as-is on tape t.

Wait for. One can force an instance to continue its computation in mode **Compute** right where it stopped by using a wait-for command right after sending a message, such as **send** mp_{out} **on** t; **wait for** mp_{in} **on** t' **s.t.** $\langle condition \rangle$. More specifically, in mode **Compute** when executing **send** mp_{out} **on** t; **wait for** mp_{in} **on** t' **s.t.** $\langle condition \rangle$ the machine does the following: it outputs a message m (created from mp_{out} at runtime) on tape t and stops for this activation. In the next activation, when receiving a message and after this message has been accepted in mode **CheckAddress**, the machine, in mode **Compute**, will directly check whether it received a message of a form that matches mp_{in} and satisfies $\langle condition \rangle$ on input tape t'. If the message is valid, the computation continues at this point in the code. Otherwise, the machine drops the received message and stops for this activation without producing output; the instance now repeats this behavior until it received a message of the correct form on the correct tape.

Restricting messages. As explained in Section 2.5.3 and [CEK+16a], we consider a restriction relation R and responsive environments such that if a real or ideal protocol outputs a restricting message $x \in R[0]$ on a network tape, then the environment/adversary/simulator has to send a reply y on the corresponding tape with $(x, y) \in R$ immediately, i.e., without sending an incorrect message (wrong message according to R or wrong tape) to the protocol before.

Therefore we write **send responsively** mp_{out} **on** NET; **wait for** mp_{in} **on** NET **s.t.** $\langle condition \rangle$ to say that the machine sends a *restricting* message on NET (we typically only have one network output tape) and then waits to receive a response on the corresponding input network tape that matches with mp_{in} and satisfies $\langle condition \rangle$. This command has to be used if and only if a message $m \in R[0]$ is sent (i.e., when sending raw messages it is not possible to sent restricting messages with other commands or normal messages with this command) and the message pattern mp_{in} is usually defined in such a way that it accepts (some of the) possible answers to the restricting message. Furthermore, unlike the wait-for command for normal messages, if a message is received (after it was accepted in **CheckAddress** mode) which is not accepted by this wait-for command (either because the message m' does not match or because *condition* is not fulfilled), then the command automatically sends the first message m on NET again. Since our restriction guarantees that answers are always accepted in **CheckAddress**, this definition of the wait-for command ensures that the environment/adversary/simulator

has to provide an expected answer that fulfills *condition* before any instance in the protocol can continue.

Abort. We use the special keyword **abort** to say that a machine stops its computation at some point. More specifically, as soon as a machine in **Compute** mode reaches the **abort** command, it will produce empty output and thus stop its computation. Then, by definition, the master IITM is activated with empty input on the start tape.

6.2.2 Real protocols

In this section, we describe how the template in Figure 6.4 for real protocols is mapped to a (real protocol) system in the sense of the IITM model. We first describe how a real protocol system is structured and then detail the **CheckAddress** and **Compute** modes of the ITMs in such a system, including the behavior in case of corruption. While some of the following was already sketched in Section 2.5.3, we now provide full details.

Atomic real protocols. An *atomic real protocol* $\mathcal{R}$ (ARP) is a collection of what we call *real protocol machines* (RPM): we write $\mathcal{R} = \{M_{role_1}, \ldots, M_{role_n}\}$. See for example Figure 6.2. The RPM M_{role_i} is an ITM that is meant to model the i^{th} *role* of the ARP. Here n is the number of roles that an ARP offers. For example, a client-server ARP would have two roles, irrespective of the number of clients. A *(hybrid) real protocol* then is a composition of *atomic real protocols* (ARP) and *ideal protocols*.

Tapes. Recall that each ITM uses a pair of tapes to communicate with another ITM. In the following, for simplicity, we call a pair of related unidirectional tapes with opposite directions a bidirectional tape (or just a tape). In the basic IITM model, tapes are divided into NET (network) and I/O tapes. We further divide the latter into PAR (parent) and SUB (subroutine) tapes.

Each RPM has one network tape NET to communicate with the adversary, several PAR tapes to communicate with higher-level ARPs or the environment, and several SUB tapes to communicate with lower-level ARPs or ideal sub-protocols. ARPs have no internal tapes: RPM instances cannot communicate directly. More specifically, if an ARP machine M (according to its specification, Figure 6.4) has a subroutine, then, according to our conventions, we define M to have one SUB tape to every machine (role) in the subroutine; the subroutine can be a real protocol itself or an ideal protocol, where in the latter case M would be connected to all dummies of the ideal protocol. Conversely, every machine in the subroutine (ARP or dummy) has a PAR tape to M. To ensure connectability with arbitrary higher-level protocols, ARPs (and for ideal protocols also dummies and ideal functionalities) thus must be able to accommodate any number of roles in higher-level protocols and are therfore implicitly parametrized by the number of roles in a higher-level protocol. When connected as a subroutine to a specific higher-level protocol (or several higher-level protocols), then an ARP has exactly those PAR tapes needed to connect to the higher-level protocol(s) (refer to Figure 6.3 for an example).

To make our pseudocode in this section easier to read, we will not refer to tapes by giving them specific names but instead refer to the role *role* that they connect to. In particular, we will write NET if we want to denote the (single) network tape of a machine, SUB$[role_{ll}]$ to denote the tape that connects to the machine with role $role_{ll}$ in

one of the subroutines, and $\mathsf{PAR}[role_{hl}]$ to denote the tape that connects to a machine with role $role_{hl}$ in a higher level protocol. Note that $role_{hl}$ may be the environment which may play arbitrarily many roles (i.e., connect to an arbitrary number of PAR tapes). We will use a similar way to address different tapes in Section 6.3.

Check address mode of ARPs. As mentioned before, in our conventions, every instance of an RPM M has a unique ID id of the form $(pid, (sid_1, \ldots, sid_n))$, where pid is interpreted to be a party ID (PID) and $(sid_1, \ldots, sid_n)$ is interpreted to be a session ID (SID). This is ensured by an appropriate definition of the **CheckAddress** mode of an RPM, provided next. We note that the meaning of what a PID and SID is, is given solely by our rules regarding PIDs and SIDs.

Recall that by the IITM model, whenever a message m was written on an output tape (with name) t and if M has an input tape t, then all existing instances of M (in the order of their creation) are invoked in mode **CheckAddress** to check which instance accepts m. The first instance to accept m gets to process m in mode **Compute**. If no such instance exists, then a new one is created and run in mode **CheckAddress**. If this instance accepts, it gets to process m, and otherwise, m is dropped and the new instance is removed from the run.

Now, in the **CheckAddress** mode, an instance of the RPM M does the following, as specified in detail in Figure 6.6. There are two cases: $\mathsf{id} = \bot$, i.e., the instance of M does not have an ID yet, or $\mathsf{id} \neq \bot$, i.e., an ID has been set.

We first consider the case $\mathsf{id} = \bot$. This case means that the instance of M was just created and invoked in mode **CheckAddress** in order to check whether this new instance accepts the incoming message, and hence, gets to process this message in mode **Compute**. More specifically, upon receiving a message m on some tape t, the instance of M checks that m is prefixed with an identifier id, i.e., $m = (id, m')$. For messages that arrive on NET or an PAR tape, id must be of the form $(pid, (sid_1, \ldots, sid_n))$. For messages that arrive on a SUB tape, id must be of the form $(pid, (sid_1, \ldots, sid_{n+1}))$. Additionally, the deterministic, polynomial-time (in the length of its input and the size of the security parameter), non-interactive algorithm **CheckIDformat** specified for the given role (cf. Figure 6.4) is performed, given (m, t) as input. It outputs either accept or reject; if this algorithm is not specified, then it always outputs accept by default. If this algorithm accepts m as well, then the instance of M accepts m in mode **CheckAddress**. Hence, according to the IITM model, this instance of M then gets to process m on tape t in mode **Compute**; otherwise this instance of M is removed again from the run and m is dropped. In mode **Compute** (as explained below), M will store the ID $(pid, (sid_1, \ldots, sid_n))$ in id.

We now consider the case that $\mathsf{id} \neq \bot$. This means that the instance of M has already accepted a message before in mode **CheckAddress** and set id to an ID of the form $(pid, (sid_1, \ldots, sid_n))$. As already mentioned in Section 6.1, we refer to this instance of M by $M[\mathsf{id}]$. Now, upon receiving a message m on NET or an PAR tape in mode **CheckAddress**, $M[\mathsf{id}]$ accepts m if and only if it parses as (id, m'). If m is received on a SUB tape, then m is parsed as $((pid, (sid_1, \ldots, sid_{n+1})), m')$ for some sid_{n+1} and some m'.

As specified in mode **Compute** below, the messages $M[\mathsf{id}]$ outputs on NET and PAR tapes will always be prefixed by id. The messages $M[\mathsf{id}]$ outputs on SUB tapes will always be of the form $((pid, (sid_1, \ldots, sid_{n+1})), m')$ with $\mathsf{id} = (pid, (sid_1, \ldots, sid_n))$. In

Upon receiving a message m on tape t in mode **CheckAddress** do the following.

- If $\mathsf{id} = \bot$ (i.e., this is the first message the instance received):
  - If t is the NET or a PAR tape: parse the message as $m = (id, m')$ for $id = (pid, (sid_1, \ldots, sid_n))$ and some $n \geq 1$; otherwise return false. Return whatever **CheckIDformat**(m, t) does.
  - If t is a SUB tape: parse the message as $m = ((pid, (sid_1, \ldots, sid_{n+1})), m')$ for some $n \geq 1$; otherwise return false. Return whatever **CheckIDformat**(m, t) does.
- Else ($\mathsf{id} \neq \bot$, i.e., this is not the first message the instance received and accepted, and hence, the ID of the machine is stored in id):
  - If t is the NET or a PAR tape: check whether the message pareses as $m = (\mathsf{id}, m')$, if it does, return true, otherwise return false.
  - If t is a SUB tape: check whether the message parses as $m = (id', m')$ where $id' = (pid, (sid_1, \ldots, sid_{n+1}))$ and $(pid, (sid_1, \ldots, sid_n)) = \mathsf{id}$, if it does, return true, otherwise return false.

Fig. 6.6: The **CheckAddress** mode of real protocol machines (RPMs).

particular, $M[\mathsf{id}]$ can send messages only under its PID pid and SID $(sid_1, \ldots, sid_n)$. Also, $M[\mathsf{id}]$ addresses subroutine instances always by appending another SID sid_{n+1} to its ID id.

We finally note that by the definition of the **CheckAddress** mode every instance of an RPM indeed has a *unique* ID (of the required form).[2]

Compute mode of ARPs. As already explained, the **Compute** mode of an ITM specifies the actual computation performed by (an instances of) the ITM. Our conventions restrict the behavior of RPMs in a specific way mainly to guarantee the desired behavior in terms of addressing of other machines and corruption. Also, our generic description of the **Compute** mode of an RPM includes the algorithms for initizalization, message preprocessing, and main specified in the template of an RPM in Figure 6.4.

We provide the formal specification of **Compute** for RPMs in Figures 6.7–6.8. When activated for the first time in **Compute** mode with a message, an RPM instance first initializes its id based on the message in the described way. The instance then asks the adversary about its corruption status, as explained in detail in Section 6.2.2.1. If the message is a regular message (i.e., not corruption-related), uncorrupted instances execute the **Initialization**, **MessagePreprocessing**, and **Main** algorithms. These three algorithms must be specified by the protocol designer (see Figure 6.4) and constitute the "inner shell" of the RPM. For subsequent regular messages, instances only execute the **MessagePreprocessing** and **Main** algorithms. For special corrupt messages, or in case they are corrupted, instances behave as explained in Section 6.2.2.1.

The three algorithms **Initialization**, **MessagePreprocessing**, and **Main** can read id and have access to the internal state specified in the template, but cannot access any other (framework-specific) state including corruption-related state. The **Initialization** algorithm is non-interactive, i.e., it may not send messages to other machines or end the run. However, sometimes it is convenient to let the adversary initialize the internal state of an instance. For this purpose, we make an exception to the non-interactivity

[2] Remember that this only means that no two instances of a machine M can have the same ID. However, there might be multiple RPM instances with the same ID in a protocol, as IDs are not required to be unique across roles.

and allow **Initialization** to send a restricting message (specified by a message pattern mp_{out}) to the adversary, using the

send responsively mp_{out} **to** NET
wait for mp_{in} **from** NET **s.t.** ⟨condition⟩

construct.[3] We note that this has to be answered by the adversary immediately since it is a restricting message. Formally, by the definition of this construct (which is given in Section 6.3), the message $(\texttt{Respond}, m)$, where m is built according to the pattern mp_{out}, is sent until the adversary responds with a message that matches the pattern mp_{in} such that *condition* is fulfilled. This is why we allow this message in **Initialization**: it is guaranteed that the execution will finish and no other machine of the protocol will be activated in the mean time. Of course, a protocol designer is free to also use this special message in **MessagePreprocessing** and **Main** to get information from the adversary; however, it may not be used to model real network traffic (as motivated in Section 2.5.3). Finally, none of **Initialization**, **MessagePreprocessing**, and **Main** may send any of the following framework specific messages (for $id = (pid, sid)$ or $id = sid$, where pid is some PID and sid is some SID as described above, $b \in \{\mathsf{true}, \mathsf{false}\}$, and $s \in \{0,1\}^* \cup \{\bot\}$):

- $(id, \texttt{CorruptionStatus}?)$
- $(id, (\texttt{CorruptionStatus}, b))$
- $(id, (\texttt{CorruptionStatus}, b, s))$
- $(id, (\texttt{CorrStatRestricting}, b, s))$
- $(id, \texttt{CorruptionList}?)$
- $(id, \texttt{AmICorrupted}?)$
- $(id, \texttt{CorruptMe}?)$

As already mentioned above, whenever an RPM instance $M[\mathsf{id}]$ outputs a message on the NET or a PAR tape, then that message must be prefixed by id; messages output on a SUB tape must be prefixed by some identity id' of the intended subroutine machine, where id' extends id with some sid_{n+1}. While every message that is sent by our framework already fulfills this requirement, it must also hold for all messages that are sent by the protocol designer in **MessagePreprocessing** and **Main**. Note that we introduce a special syntax for sending messages in these algorithms in Section 6.3, which already takes care of this requirement by implicitly adding the correct prefix to every message. So, as long as a protocol designer uses this syntax, he does not have to care about this requirement at all and can focus his attention entirely on the actual contents of the message.

6.2.2.1 Corruption of RPMs

Recall that the IITM model does not fix the corruption behavior of protocols. We thus need to define it as part of our conventions. Like in the UC and GNUC models, we consider a central adversary that controls all dishonest ITM instances. As a first

[3] Note that here we use the notation conventions for the "inner shell" introduced in Section 6.3 here. These are slightly different than the general conventions introduced at the beginning of Section 6.2.1 since messages do not explicitly have to be prefixed with id; this is taken care of by the framework.

State variable id $\in \{0,1\}^* \cup \{\bot\} = \bot$. {*Identity of the instance.*
State variable corr $\in \{\mathsf{false}, \mathsf{true}, \bot\} = \bot$. {*Corruption status.*
State variable subcorr $\in \{\mathsf{false}, \mathsf{true}\} = \mathsf{false}$. {*Consider instance corrupted?*
State variable init $\in \{\mathsf{false}, \mathsf{true}\} = \mathsf{false}$. {*"Inner shell" initialized?*
State variable internalState. {*State of "inner shell" of instance as per template in Sec. 6.1.*
State variable transcript. {*Transcript of all actions of "inner shell".*

Upon receiving a message $m = ((pid, (sid_1, \ldots, sid_n)), m')$ from the NET or a PAR tape t, or a message $m = ((pid, (sid_1, \ldots, sid_{n+1})), m')$ from a SUB tape t do:

if id $= \bot$**:**
  id $\leftarrow (pid, (sid_1, \ldots, sid_n))$. {*Initialize identity of instance.*
if corr $= \bot$**:** {*Initialize corruption status.*
  if $t = \mathsf{NET} \wedge m' = (\texttt{SetCorruptionStatus}, b)$**:** {*First message sets corruption status.*
    corr $\leftarrow b \wedge$ AllowCorruption?(id, internalState, m).
      {*AllowCorruption? is only executed if* $b = \mathsf{true}$.
    send (id, (CorruptionStatus, corr, $\bot$)) **on** NET.
  else: {*For all other first messages.*
    send responsively (id, CorruptMe?) **on** NET;
    wait for (id, (SetCorruptionStatus, b)) **on** NET.
    corr $\leftarrow b \wedge$ AllowCorruption?(id, internalState, m).
      {*AllowCorruption? is only executed if* $b = \mathsf{true}$.
    if corr $= \mathsf{true}$**:** {*Else: continue processing* (m, t).
      if $t = \mathsf{PAR}[role_{hl}] \wedge m' = \texttt{CorruptionStatus}?$**:**
        {*This message has to be processed after the notification.*
        send responsively (id, (CorrStatRestricting, true, (m, t))) **on** NET;
        wait for (id, OK) **on** NET. {*Continue processing* m.
      else:
        send (id, (CorruptionStatus, true, (m, t))) **on** NET.
else if corr $= \mathsf{false} \wedge t = \mathsf{NET} \wedge m' = (\texttt{SetCorruptionStatus}, \mathsf{true})$**:** {*Corruption request.*
  corr $\leftarrow$ AllowCorruption?(id, internalState, $\bot$).
  if static corruption is specified**:**
    corr $\leftarrow \mathsf{false}$.
  if corr $= \mathsf{true}$**:**
    if secure erasures are specified**:**
      send (id, (CorruptionStatus, true, internalState)) **on** NET.
    else:
      send (id, (CorruptionStatus, true, transcript)) **on** NET.
  else:
    send (id, (CorruptionStatus, false, $\bot$)) **on** NET.
else if corr $= \mathsf{false} \wedge t = \mathsf{NET} \wedge m' = (\texttt{SetCorruptionStatus}, \mathsf{false})$**:**
    {*(Useless) corruption request.*
  send (id, (CorruptionStatus, false, $\bot$)) **on** NET.

Continued in Figure 6.8...

Fig. 6.7: The **Compute** mode of real protocol machines (RPMs) and dummies.

...continued from Figure 6.7.

if $t = \mathsf{PAR}[role_{hl}] \wedge m' = \texttt{CorruptionStatus}?$: {*Query corruption status.*
  subcorr $\leftarrow$ subcorr $\vee$ corr $\vee$ DetermineCorruptionStatus(id, internalState).
    {*DetermineCorruptionStatus is only executed if* subcorr = corr = false.
  send (id, (CorruptionStatus, subcorr)) **on** $\mathsf{PAR}[role_{hl}]$.
if corr = true: {*Corrupted instances are multiplexers.*
  if $t = \mathsf{PAR}[role_{hl}]$:
    send (id, (par, $role_{hl}, m'$)) **on** NET.
  else if $t = \mathsf{NET} \wedge m' = (\texttt{par}, role_{hl}, m'')$:
    send (id, m'') **on** $\mathsf{PAR}[role_{hl}]$.
  else if this is an instance of an RPM:
    if $t = \mathsf{SUB}[role_{ll}]$:
      send (id, (sub, $role_{ll}, sid_{n+1}, m'$)) **on** NET.
    else if $t = \mathsf{NET} \wedge m' = (\texttt{sub}, role_{ll}, lastsid, m'')$:
      send ((pid, ($sid_1, \ldots, sid_n$), $lastsid$)), m'') **on** $\mathsf{SUB}[role_{ll}]$.
  else if this is an instance of a dummy:
    if $t = \mathsf{SUB}[role_{ll}]$:
      send (id, (sub, $role_{ll}, m'$)) **on** NET.
    else if $t = \mathsf{NET} \wedge m' = (\texttt{sub}, role_{ll}, m'')$:
      send (id, m'') **on** $\mathsf{SUB}[role_{ll}]$.
else: {*Honest behaviour.*
  Append (m, t) to transcript.
  if init = false:
    init $\leftarrow$ true. {*Initialize internal state upon first regular message.*
    Call **Initialization**(m, t, id, internalState, transcript).
  Call **MessagePreprocessing**(m, t, id, internalState, transcript).
  Call **Main**(m, t, id, internalState, transcript).
abort. {*In case no message was sent.*

Fig. 6.8: The **Compute** mode of real protocol machines (RPMs) and dummies (continued).

idea, it seems the adversary should have full control over ITMs that he *corrupts*. This however would disrupt the execution of the system of ITMs and thus not meaningfully model reality. For example, providing the adversary with full control over an instance $M[\mathsf{id}] = M[(pid, (sid_1, \ldots, sid_n)]$, would allow the adversary to send messages under IDs different than id, and hence, impersonate parties other than pid. Also, this would allow the adversary to access subroutines for arbitrary other parties and in arbitrary subroutines. In the case of RPMs, we therefore instead fix the "outer shell" of instances and give the adversary control only over the "inner shell". The adversary thus has just enough power so that our conventions meaningfully model dishonest behavior in reality.

In a nutshell, each RPM instance keeps track of its corruption status. When honest, it executes the code specified by the **Initialization**, **MessagePreprocessing** and **Main** algorithms. When corrupt, it first sends its internal state (as specified in the template in Figure 6.4) to the adversary and then instead acts as a (bidirectional) multiplexer between the NET tape on one hand, and the PAR and SUB tapes on the other, which is equivalent to giving the adversary control over the "inner shell" of the instance. The adversary can corrupt an instance by sending a special message on its NET tape. It is important that the environment is aware of the corruption status of top-level ITMs: in the ideal world, we must limit the amount of corruptions that can be performed by

the simulator in order to guarantee a meaningful indistinguishability experiment for universal composability. To that effect, in our conventions the environment is allowed to query the corruption status of top-level RPMs. This will force a simulator to keep the corruption status returned by dummies (in the ideal world) and the corresponding RPMs (in the real world) in sync.

Corruption state. Every instance of an RPM has a special state variable $\mathsf{corr} \in \{\mathsf{true}, \mathsf{false}, \bot\}$, not accessible by the "inner shell", which is initially $\bot$ and indicates whether it is corrupted. As detailed below, upon the first activation in mode **Compute** the adversary is asked by the RPM whether to corrupt the instance, thereby causing corr to be set to true or false. If dynamic corruptions are allowed, the adversary can corrupt the instance also at a later point in time by sending a special corruption message to it. *Transient* corruptions, where the adversary can un-corrrupt a party (cause corr to switch back to false) are also conceivable, but not further modelled here.

As sketched above, once an instance is corrupted it acts as a multiplexer for regular messages between the NET tape on one hand, and the PAR and SUB tapes on the other. Messages on NET carry a prefix containing the source/destination tape, and in the case of a SUB tape, also the sid_{n+1} of the subroutine. The corrupted instance still handles corruption-related messages (e.g., `CorruptionStatus`?) directly and analogously to the honest instance.

Setting the initial corruption status. When an RPM instance receives its first message m, it saves m and the tape t it arrived on, and gives the adversary the chance to corrupt it before processing the message by sending a *restricting* message $(\mathsf{id}, \texttt{CorruptMe?})$ on NET. The instance then *immediately* receives $(\mathsf{id}, (\texttt{SetCorruptionStatus}, b))$ on NET: here we rely on the fact that the adversary and the environment are *responsive*, i.e., they are not allowed to activate any other RPM instance before responding as sketched in Section 2.5.3 and explained in detail in [CEK+16a]. If $b = \mathsf{false}$, the instance sets corr to false. If $b = \mathsf{true}$, the instance calls an algorithm AllowCorruption? (see Figure 6.4) to determine whether to set corr to true or false (see also below). In any of these cases, if corr is set to false, the instance processes the saved first message m as if it was just received on tape t; if corr is set to true, then one has to distinguish two cases. If the original message m was received on a PAR tape and $m = (\mathsf{id}, \texttt{CorruptionStatus?})$, then the adversary is notified via a restricting message $(\mathsf{id}, (\texttt{CorrStatRestricting}, \mathsf{true}, (m, t)))$ on NET such that he knows that the corruption request was successful. He then has to answer with $(\mathsf{id}, \texttt{OK})$ "immediately", i.e., without activating other machines in the protocol. As soon as this response is received, the initial message m is processed. This is done because otherwise the environment/higher level protocols could not hope to obtain the real corruption status of fresh instances. In any other case, the instance sends $(\mathsf{id}, (\texttt{CorruptionStatus}, \mathsf{true}, (m, t)))$ on NET, containing the internal state of the instance — which at this point consists of the initial message and tape.

Note that there is a special case to this rule: if the first message received by an instance is $m = (\mathsf{id}, (\texttt{SetCorruptionStatus}, b))$ on NET, then the instance does not send $(\mathsf{id}, \texttt{CorruptMe?})$ but instead treats this initial message also as response to $(\mathsf{id}, \texttt{CorruptMe?})$. If the instance determines that it is not corrupted, it will then send an acknowledgment $(\mathsf{id}, \texttt{CorruptionStatus}, \mathsf{false}, \bot)$ to the adversary instead of processing

the initial message. This rule is primarily for convenience, since it simplifies the specification of simulators.

Allowing corruption. The protocol-designer–specified predicate AllowCorruption?(id, internalState, *initialMessage*) is a deterministic, non-interactive algorithm that is used to determine whether to allow the corruption of an RPM instance or refuse it. It may depend on the ID id and all variables defined in internalState. Furthermore, if this algorithm is called because the adversary tries to corrupt a fresh instance (i.e., this is the first time that the instance gets a (id, (`SetCorruptionStatus`, b)) message), then this algorithm may also depend on the initial message that was first accepted by this instance; this message is stored as is (i.e., including the prefix which contains id) in *initialMessage*. Otherwise, *initialMessage* is $\perp$ and hence can be used to distinguish these two cases. This predicate allows, for example, the modeling of bottom-up corruption (an instance can only be corrupted if all of its subroutines are) or incorruptible machines (such as secure hardware tokens). Inside the AllowCorruption?, the instance may query the corruption status of an instance with $id' = (\mathsf{id}, sid_n)$ and role $role_{ll}$ in a subroutine $\mathcal{S}$ via the corr($\mathcal{S}[sid_{n+1}], role_{ll}$) macro. Formally, the macro corr($\mathcal{S}[sid_{n+1}], role_{ll}$) does the following: ($id'$, `CorruptionStatus`?) is sent out on the corresponding SUB tape. Our conventions ensure that an answer is returned immediately; the machine cannot receive any other message in the meantime, as explained in the next paragraph. Note that this is the only exception to the non-interactivity of this algorithm; a protocol designer must not send any messages himself but only use this macro. If this algorithm is not specified, then it always outputs true by default and thus allows every corruption request.

Querying the corruption status. An instance of an RPM can be queried about its corruption status on any PAR tape, and hence, by a higher-level protocol or the environment. When an instance receives a message (id, `CorruptionStatus`?) on a PAR tape and it was corrupted by the adversary at some point in the past, then it returns (id, `CorruptionStatus`, true) on the same PAR tape. If it was not explicitly corrupted before, it may consider itself corrupt nevertheless by calling the DetermineCorruptionStatus predicate (as specified by the protocol designer, see Figure 6.4) and respond with a message (id, `CorruptionStatus`, b) to the PAR tape where $b \in \{\mathsf{true}, \mathsf{false}\}$ is the output of DetermineCorruptionStatus. As we do not consider transient corruptions, if the instance has sent $b = \mathsf{true}$ in the past, it must always send $b = \mathsf{true}$. (This is ensured independently of how the DetermineCorruptionStatus predicate is specified, see Figures 6.7–6.8.) If a `CorruptionStatus`?-query arrives before the instance is initialized, i.e., corr $= \perp$, then it first asks the adversary for its corruption status (which is then determined immediately).

Formally, the DetermineCorruptionStatus(id, internalState) predicate is a deterministic, non-interactive algorithm which may depend on the ID id and every variable in internalState and outputs either true or false. Like for AllowCorruption?, in the specification of the predicate DetermineCorruptionStatus the protocol designer may use the macro corr($\mathcal{S}[sid_{n+1}], role_{ll}$) to check for the corruption status of its subroutines instances. Again, this is the only exception to the non-interactivity; a protocol designer must not send any messages himself but may only use this macro. If this algorithm is not specified, then it always outputs false.

As mentioned above, an instance of an RPM that has not been corrupted explicitly by the adversary (i.e., corr still is false) may still return true as its corruption status via the DetermineCorruptionStatus algorithm. This allows a simulator to corrupt the corresponding dummy in the ideal world. For instance, this might be useful when a machine cannot perform its task anymore because too many of its subroutines have already been corrupted. Note that even if true is returned as the corruption status, the instance itself still behaves honestly (and is not controlled by the adversary) because corr = false.

Our conventions regarding corruption ensure that `CorruptionStatus`?-queries from higher-level protocols/the environment (and hence, also those queries done with the macro $\mathsf{corr}(\mathcal{S}[sid_{n+1}], role_{ll})$) are answered immediately. When answering such queries subroutine instances (recursively) could be asked about their corruption status. Now, some of these subroutine instances might not have been initialized yet, which means that the adversary is queried to determine the value of corr for these instances. These queries are restricting messages, and hence, because environments and simulators are responsive, they have to be answered immediately, i.e., environments/simulators may not send a message to any other instance of the protocol before answering such queries. Even if the adversary corrupts a fresh instance, this message is answered immediately nevertheless since he is only notified of the successful corruption by a restricting message. So altogether, `CorruptionStatus`?-queries are answered immediately. In particular, the adversary cannot change the corruption status of subroutine RPM instances that were already processed during such a query. Thus `CorruptionStatus`?-queries are atomic, thereby removing complex edge cases in corruption-related algorithms from consideration.[4]

Specific conventions for different types of corruption. As discussed earlier, we support various types of corruption. Depending on the concrete type being used to model a concrete protocol, different specific conventions apply.

Dynamic corruptions. Recall that for dynamic corruptions, the adversary is allowed to corrupt RPM instances even after their initialization. So, if the adversary sends a corruption request $(\mathsf{id}, (\texttt{SetCorruptionStatus}, \mathsf{true}))$ at any point after the instance was created, AllowCorruption? is invoked and if this procedure returns true, then corr is set to true.

If an instance is corrupted, its internal state as specified in the template, i.e., the state variables accessible to its "inner shell", are sent to the adversary. In a setting when *secure erasures* are *not* allowed, one must assume that all computations, temporary variables,

[4] Note that a protocol designer is able to abuse the freedom of our framework to disrupt the atomicity of `CorruptionStatus`? by using the $\mathsf{corr}(\mathcal{S}[sid_{n+1}], role_{ll})$ macro for an sid_{n+1} which will not be accepted by the **CheckIDformat** of a fresh instance. The protocol designer is also able to explicitly reject the `CorruptionStatus`? message in **CheckIDformat**; in both cases the environment gets control and is free to do anything, while instances that are still waiting for an answer become stuck forever. However, these problems depend entirely on the definitions of the protocol designer; the adversary is never able to disrupt a `CorruptionStatus`? request himself. Furthermore, these problems are artificial and should never occur in a well defined protocol; in particular, they can easily be spotted and corrected in the definition of the protocol.

random coins, sent and received messages, and previous assignments of all variables are still part of the state sent to the adversary. To ease the burden of protocol designers, we introduce an extra state variable transcript, not readable by the "inner shell", that records everything the instance does. Upon corruption, the value of transcript is sent to the adversary in lieu of the internal state. The exact format of these notifications is the same as the one specified in the *Setting the initial corruption status* paragraph, but instead of (m, t) it includes either internalState or transcript.

Formally, the adversary can also try to send the (useless) message (id, (`SetCorruptionStatus`, false)) to an instance that already determined its initial corruption state. This message is always answered with (id, `CorruptionStatus`, false, $\perp$) such that a protocol designer does not have to care about this special case.

Weak static corruptions. For (weak) static corruptions, RPM instances can be corrupted only during their initialization, i.e., when corr $= \perp$. Therefore, once corr has been set to true or false after the initial query to the adversary, it is made sure that it cannot be changed anymore (independently of how AllowCorruption? is defined). If the adversary tries to send a corruption message after the initial corruption is fixed, it will always be answered with (id, `CorruptionStatus`, false, $\perp$).

Strong static corruptions. For strong static corruptions, the adversary has to fix all RPM instance of one session, i.e., those with the same SID, and fix their corruption status at the beginning of the session. For this purpose, in addition to applying the conventions for weak static corruptions, all RPMs in an ARP are surrounded with an ITM called M_{corrupt}: an ARP $\mathcal{P} = \{M_1, \ldots, M_n\}$ thus becomes a system $\mathcal{P}' = \{M_{\text{corrupt}}, \mathcal{P}\}$, where the machine M_{corrupt} acts as a filter around all external tapes of $\{M_1, \ldots, M_n\}$. There is one instance of M_{corrupt} per *sid*.

Upon receiving the first message on a filtered NET, an PAR, or a SUB tape, $M_{\text{corrupt}}[sid]$ *restrictively* asks the adversary for a list containing the corruption status of all RPM instances in that session. $M_{\text{corrupt}}[sid]$ then checks the corruption status of all instances against the list (it forwards `CorruptMe`?-queries and `CorruptionStatus`?-queries and the corresponding responses while doing so) and finally forwards the initial message. Whenever another message arrives on a filtered tape, $M_{\text{corrupt}}[sid]$ re-checks the corruption status of all instances. In case any corruption status check fails, $M_{\text{corrupt}}[sid]$ stops forwarding messages forever to prevent the adversary from gaining any illicit advantage in the indistinguishability experiment.

Discussion. By using the algorithms AllowCorruption? and DetermineCorruptionStatus to define corruption, our model becomes very flexible compared to other models. A protocol designer can easily define both top-down and bottom-up corruption, which are usually hardwired in many models. For top-down corruption, one uses the DetermineCorruptionStatus to answer true as soon as at least one of the subroutine instances is corrupted. In this case, the adversary is still free to corrupt any instance I, but by doing so, all higher-level instances that use I (directly or via other subprotocols) will consider themselves to be corrupted, too. For bottom-up corruption, one uses AllowCorruption? to only allow corruption of some instance I if all instances of its subroutines (that are used by an honest I) are also corrupted.

The main advantage of this approach is that one can specify a very fine grained corruption model. For example, it is possible to combine a protocol $\mathcal{P}$ with bottom-up

and a protocol $\mathcal{P}'$ with top-down corruption, model honest (uncorruptible) parties, or only allow corruption after some setup is completed.

6.2.3 Ideal protocols

Recall that in ideal protocols (IPs), the input is handed to a trusted ITM instance called an ideal functionality (IF) instance. To simplify the corruption model, as usual, additionally dummy ITMs are introduced that act as a filter to the PAR tapes of the IF: when a dummy instance is honest, it transparently forwards messages to and from the IF, but when corrupt, it gives the adversary direct access to the IP's input/output and to the IF.

We now present the overall structure of IPs. Next we define dummies and IFs.

6.2.3.1 Structure

Recall that an ideal protocol $\mathcal{I}$ is a system (in the sense of Section 2.5.3) of the form $\{D_{\mathsf{role}_1}, \ldots, D_{\mathsf{role}_n}, \mathcal{F}\}$, where $D_{\mathsf{role}_1}, \ldots, D_{\mathsf{role}_n}, \mathcal{F}$ are ITMs. The ITMs $D_{\mathsf{role}_1}, \ldots, D_{\mathsf{role}_n}$ are called *dummies*. A dummy D_{role_i} corresponds to a role M_{role_i} in a real protocol. However, dummies essentially only act as forwarders between a higher-level protocol (or the environment) and the ideal functionality (IF) $\mathcal{F}$, which realizes the actual cryptographic task. Just like real protocol machines, a dummy has a network tape to the adversary as well as PAR tapes. For every PAR tape it has a corresponding SUB tape to $\mathcal{F}$ (in order to forward messages). The functionality $\mathcal{F}$ has a network tape, no SUB tapes, and for every subroutine tape of a dummy a corresponding PAR tape. The dummies' SUB tapes and the IF's PAR tapes are the only internal tapes of the IP. In Figure 6.1, the left side shows the structure of an ideal protocol with n dummies, while the right side shows several instances of a sample ideal protocol. This example contains the dummies D_1, D_2, and D_3.

As further explained below, in a run of $\mathcal{I}$, each instance of a dummy has an ID of the form $(pid, (sid_1, \ldots, sid_n))$ just as their corresponding real machines. An instance of a dummy with such an ID talks to an instance of the ideal functionality $\mathcal{F}$ with ID $(sid_1, \ldots, sid_n)$. Note that the ID of an instance of $\mathcal{F}$ does not contain a specific PID since such an instance performs cryptographic tasks for all parties in session $(sid_1, \ldots, sid_n)$.

If an ARP realizes an IP, the two must have an identical set of external PAR tapes. Furthermore there is one dummy in the IP for each RPM in the ARP, and each dummy has the same PAR tapes as the corresponding RPM.

IPs designed to be secure against strong static corruption, additionally consist of an M_{corrupt} ITM that surrounds the dummies and the IF, i.e., $\mathcal{I}' = \{M_{\text{corrupt}}, \mathcal{I}\}$. That M_{corrupt} is similar to the one for ARPs, except that it has additional network tapes to accommodate the IF, and has no subroutine tapes.

6.2.3.2 Dummies

In a nutshell, when honest, dummy instances forward all protocol messages between their PAR tapes and the corresponding SUB tapes, and thus directly forward the IP's input to the IF, and the IF's output to the IP's output. When corrupted, dummy instances

Upon receiving a message m on tape t in mode **CheckAddress** do the following.

- If $\mathsf{id} = \bot$ (i.e., this is the first message the instance received):
  Parse the message as $m = (id, m')$ for $id = (pid, (sid_1, \ldots, sid_n))$ and some $n \geq 1$; otherwise return false. Return whatever $\mathbf{CheckIDformat}(m, t)$ does.
- Else ($\mathsf{id} \neq \bot$, i.e., this is not the first message the instance received and the ID of the dummy has been stored in id before):
  Parse the message as $m = (\mathsf{id}, m')$; otherwise return false.

Fig. 6.9: The **CheckAddress** mode of dummies.

act as a multiplexer for regular messages between their NET tape on one hand and their PAR- and SUB-tapes on the other, giving the adversary direct access to the inputs and outputs of the IF and the IP.

Dummies are very similar to RPMs, and thus their **CheckAddress** and **Compute** are similar as well. In particular, the ITM that specifies a dummy in mode **Compute** is defined in the same way as the one for RPM, namely it is the one defined in Figures 6.7–6.8; the **CheckAddress** is defined in a slightly different way. However, except for the algorithm **CheckIDformat**, which is defined in the specification of an ideal protocol (see Figure 6.5), all other algorithms that the protocol designer has to specify for RPMs, are now fixed for dummies in a specific way. So, there is almost nothing left for the protocol designer to specify.

We now specify the ITM that specifies a dummy in detail. That is, we give a specification of the **CheckAddress** and the **Compute** modes of such a machine.

Identity and Check address. Just as for RPMs, the ID of a dummy instance is of the form $(pid, (sid_1, \ldots, sid_n))$. Unlike RPMs, the ID written on and expected to receive on a SUB tape is of this form as well, rather than of the form $(pid, (sid_1, \ldots, sid_{n+1}))$. This is because a dummy merely acts as a forwarder between the environment/higher level protocol and the IF; the messages sent on SUB tapes are those forwarded to the IF. Accordingly, the **CheckAddress** mode of dummies is defined as specified in Figure 6.9.

Compute. The **Compute** mode of dummies is a special case of the **Compute** mode of RPMs, where, as mentioned, most algorithms that for RPMs would be specified by the protocol designer are now fixed in a specific way (see Figure 6.10). In particular, the **Initialization** and **MessagePreprocessing** algorithms are no-ops. In the **Main** algorithm, honest instances forward regular messages received on one of their PAR tapes to the corresponding SUB tape, and vice versa. The algorithm AllowCorruption? of dummies is, unlike the one specified by protocol designers, interactive: It asks the IF that belongs to the dummy by sending $((pid, \mathsf{id}), (\texttt{AmICorrupted?}, initialMessage))$ about whether or not to allow corruption, and in case the corruption request is granted, which information to forward to the adversary/simulator/environment; this information is then written to internalState and thus it will be given to the adversary (see Figure 6.10 and Figures 6.7–6.8). The algorithm DetermineCorruptionStatus is the default one, i.e., it returns false, which means that `CorruptionStatus`?-requests are always answered with the corruption status corr of the dummy (see Figures 6.7–6.8). The algorithms are defined in detail in Figure 6.10. The actual **Compute** mode of the ITM describing a dummy is specified in Figures 6.7–6.8 (the same as the one for RPMs).

```
function AllowCorruption?(id, internalState, initialMessage):
    Let role_ll be one SUB tape to the IF.
    send (id, (AmICorrupted?, initialMessage)) on SUB[role_ll];
    wait for (id, (CorruptionStatus, b, leakage)) on SUB[role_ll].
    if b = true:
        internalState ← leakage.   {Upon corruption, get internal state from ideal functionality.
    return b.
function Main(m, t, id, internalState, transcript):                {Honest dummy is a forwarder.
    if t = PAR[role_hl]:
        send m on SUB[role_hl].
    else if t = SUB[role_ll]:
        send m on SUB[role_ll].
```

Fig. 6.10: Algorithms in **Compute** mode of dummies.

6.2.3.3 Ideal functionalities

Recall that IFs codify the actual behavior of the IP. In the following, we precisely specify the ITM that defines the IF obtained from the specification of an ideal protocol (see Figure 6.5). For this purpose, we have to specify the **CheckAddress** and **Compute** modes of this ITM. This includes the specification of corruption. Formally, the **CheckAddress** and **Compute** modes are defined in Figures 6.11 and 6.12.

Check address. As mentioned before, the id of IFs consists only of an $sid = (sid_1, \ldots, sid_n)$. Hence, this ID does not contain a PID as an IF handles the tasks of all parties in session sid. Messages received or sent on a PAR tape are prefixed with (pid, id), while messages sent or received on the NET tape must be prefixed with id. This leads to the following definition of the **CheckAddress** mode: If a fresh copy (i.e., $\mathsf{id} = \bot$) of an IF received a message on a PAR tape, it first tries to parse it as $((pid, sid), m)$ and then simulates the **CheckIDformat** of the dummy that belongs to this tape, where the simulation assumes that the message was received on the corresponding PAR tape of the dummy.[5] If a fresh copy of an IF received a message on NET, then the algorithm **CheckIDformatIdeal** is used to check whether the ID has the expected format. We note that this format is different to what dummies expect on NET as the ID of an IF does not contain a PID, which is why we use a second algorithm for this case instead of reusing **CheckIDformat**. Otherwise, **CheckIDformatIdeal** is essentially the same as **CheckIDformat**; in particular, both get the same input and both have the same default (i.e., they always accept if not specified). If an already existing copy (i.e., $\mathsf{id} \neq \bot$) of an IF receives a message on PAR, it tries to parse it as $((pid, \mathsf{id}), m)$ and accepts iff this is possible. If an already existing copy of an IF receives a message on NET, it tries to parse it as (id, m) and accepts iff this is possible.

Compute and Corruption. The **Compute** mode of IFs is greatly simplified compared to that of RPMs, as IF instances cannot be corrupted directly (only dummies can). The **Compute** mode of IFs is formally defined in Figure 6.12. When activated for the first

[5] For usual definitions of **CheckIDformat** this step will accept since the message was already checked by the dummy that sent this message. In particular, the IF will always be able to parse the message since it is guaranteed to have the correct format. However, we perform this step nervertheless since this makes the IF a σ-session version as defined in [CEK+16a].

Upon receiving a message m on tape t in mode **CheckAddress** do the following.

- If $\mathsf{id} = \bot$ (i.e., this is the first message the instance received):
  - If t is a PAR tape: parse the message as $m = (id, m')$ for $id = (pid, (sid_1, \ldots, sid_n))$ and some $n \geq 1$; otherwise return false. Simulate **CheckIDformat**(m, t') of the dummy that is connected to t, where t' is the PAR tape (of the dummy) that corresponds to the tape t.
  - If t is the NET tape: parse the message as $m = (id, m')$ for $id = (sid_1, \ldots, sid_n)$ and some $n \geq 1$; otherwise return false. Return whatever **CheckIDformatIdeal**(m, t) does.
- Else ($\mathsf{id} \neq \bot$, i.e., this is not the first message the instance received and the ID of the IF has been stored in id before):
  - If t is a PAR tape: parse the message as $m = ((pid, \mathsf{id}), m')$; if this works, return true, otherwise return false.
  - If t is the NET tape: parse the message as $m = (\mathsf{id}, m')$; if this works, return true, otherwise return false.

Fig. 6.11: The **CheckAddress** mode of ideal functionalities (IFs).

time in the **Compute** mode, IF instances first initialize their id based on the received message. If the message was a regular message, they then execute the **Initialization**, **MessagePreprocessing**, and **Main** algorithms, which are all specified by the protocol designer. If the first message was the special $((pid, \mathsf{id}), (\texttt{AmICorrupted?}, initialMessage))$ request from a dummy, then only **Initialization** is executed before proceeding as mentioned below. For subsequent regular messages, IF instances only execute the **MessagePreprocessing** and **Main** algorithms. The same restrictions as with real protocols also apply for the three algorithms of IFs, however, unlike ARPs, these algorithms are allowed to access corruption-related state variables. This allows the instance to, e.g., provide extra capabilities to the adversary in the face of dummy corruption.

Note that all messages that are sent on the NET tape must be prefixed by id, while all messages sent on a PAR tape (to a dummy) must be prefixed with (pid, id) for some pid. Again, just as for ARPs, this requirement is automatically taken care of by the syntax presented in Section 6.3, so as long as a protocol designer uses this syntax, he does not have to care about this requirement at all.

In the **Main** algorithm, it is sometimes convenient to run *externally-provided algorithms*. For example, in the signature functionality, we let the adversary choose an algorithm (upon initialization) that $\mathcal{F}_{\text{sig}}$ uses to "sign" messages. Formally, the IP must contain a polynomial p_{rt} in its protocol parameters that will bound the runtime of the externally provided algorithms. When the IF instance with ID id in a run with security parameter η executes such an algorithm alg with input i, which we denote by $\mathsf{alg}^{(p_{\text{rt}})}(i)$, it will run alg for up to $p_{\text{rt}}(|1^\eta| + |id| + |i|)$ steps[6] and else abort it with output $\bot$. We also note that such algorithms are not allowed to send messages.

When receiving an $((pid, \mathsf{id}), (\texttt{AmICorrupted?}, initialMessage))$-message from a dummy instance, the IF instance will run an AllowDummyCorruption? predicate (as specified by the protocol designer, see Figure 6.5) to decide whether to allow the dummy to be corrupted, and if yes, it runs the LeakedData algorithm (as also specified by the protocol designer) to determine what information it should leak to the adversary. More specifically, the deterministic, non-interactive algorithm AllowDummyCorruption?$(pid, role, \mathsf{id},$

[6] We include the ID of the IF in the polynomial to allow for more elegant and natural joint-state realizations. See Section 6.5.2 for an example.

CorruptionSet, internalState, messageList) outputs either true or false and can depend on the PID and the role of the dummy, the session ID id, the currently corrupted dummy instances, the internal state of the IF, and a list of all messages that were sent to or received from every dummy instance. Note that this list also includes the current `AmICorrupted`?-message and thus the *initialMessage*, which either contains the initial message that was first accepted by the dummy (if the dummy instance currently determines its initial corruption status) or $\perp$ (if the dummy instance is no longer fresh). Note that the initial message is stored as is in *initialMessage*, i.e., including the prefix. If AllowDummyCorruption? is not specified, the IF behaves as if true is always returned. The deterministic, non-interactive algorithm LeakedData($pid, role$, id, CorruptionSet, internalState, messageList) outputs a bitstring and may depend on the same information as AllowDummyCorruption?. If LeakedData is not specified, then messageList$[pid, role]$, which contains a transcript of all messages sent to and received from the dummy instance $(pid, role)$, is leaked. Note that an `AmICorrupted`?-message is answered immediately by an IF, i.e., no other instance of the protocol is activated during this request.[7]

As mentioned above, IFs keep track of the list of corrupt dummy instances in an implicit CorruptionSet state variable, where a dummy instance is specified by the pair $(role, pid)$, which consists of its role and PID (its SID is the same as the one of the IF instance). IFs also keep track of all inputs received and outputs sent to a given dummy instance $(role, pid)$ in an implicit messageList$[(role, pid)]$ variable. Per default, i.e., if not specified by the protocol designer, LeakedData returns messageList$[(role, pid)]$ corresponding to the dummy instance $(role, pid)$ that is being corrupted.

6.3 Programming Language for the Templates

We next introduce convenient notation for specifying the **Initialization**, **MessagePreprocessing**, and **Main** blocks of the machine specifications from Section 6.1. We only introduce notation that is important to provide unique interfaces to protocol designers; the remaining parts of these algorithms can be described in any pseudocode. The main difference between this notation and the one presented in Section 6.2.1 is that protocol designers do not have to worry about prefixing messages with the correct ID. This is automatically taken care of, such that one only has to specify the "payload" of the message. Note that all commands in this section can be distinguished from those in Section 6.2.1 since they send/receive messages *to*/*from* roles, but not *on* tapes.

On a high level, each of the algorithms **Initialization**, **MessagePreprocessing**, and **Main** consists of multiple blocks, where each block specifies for which messages it is executed, the computations it performs, and the messages it sends out. Figures 6.13 and 6.15 provide examples.

More precisely, a single block has the following form:

- A block starts with a header of one of the following forms:

[7] The environment/adversary/simulator may be activated since **Initialization** is executed before an `AmICorrupted`? is processed. However, **Initialization** may only send a restricting message, which gives the claim.

State variable $\mathsf{id} \in \{0,1\}^* \cup \{\bot\} = \bot$. {*Identity of the instance.*
State variable $\mathsf{CorruptionSet} \subset \{0,1\}^* \times \{0,1\}^* = \emptyset$. {*Set of corrupted dummies.*
State variable internalState. {*State of "inner shell" of instance as per template in Sec. 6.1.*
State variable $\mathsf{messageList} \in ((\{0,1\}^*)^2 \mapsto \{0,1\}^*)$.
{*All messages received/sent per dummy instance. If a key does not exist yet, ϵ is returned.*

Upon receiving a message $m = ((pid, (sid_1, \ldots, sid_n)), m')$ from tape $t \neq \mathsf{NET}$ or a message $m = ((sid_1, \ldots, sid_n), m')$ from tape $t = \mathsf{NET}$ do:
if $\mathsf{id} = \bot$**:**
  $\mathsf{id} \leftarrow (sid_1, \ldots, sid_n)$. {*Initialize identity of instance.*
  Call **Initialization**$(m, t, \mathsf{id}, \mathsf{internalState})$.
if $t \neq \mathsf{NET} \wedge m' = (\texttt{AmICorrupted}?, initialMessage)$**:**
  Let *role* be the role of the dummy attached to the tape t.
  Append $(\texttt{recv}, t, m')$ to $\mathsf{messageList}[(role, pid)]$.
  $b \leftarrow \mathsf{AllowDummyCorruption}?(m, t, \mathsf{id}, \mathsf{internalState}, \mathsf{CorruptionSet}, \mathsf{messageList})$.
  $leakage \leftarrow \bot$.
  if $b = \mathsf{true}$**:**
    Add $(role, pid)$ to CorruptionSet.
    $leakage \leftarrow \mathsf{LeakedData}(m, t, \mathsf{id}, \mathsf{internalState}, \mathsf{CorruptionSet}, \mathsf{messageList})$.
  $m'' \leftarrow (\texttt{CorruptionStatus}, b, leakage)$.
  Append (send, t, m'') to $\mathsf{messageList}[(role, pid)]$.
  send $((pid, \mathsf{id}), m'')$ **on** t.
else:
  if $t \neq \mathsf{NET}$**:**
    Let *role* be the role of the dummy attached to the tape t.
    Append $(\texttt{recv}, t, m')$ to $\mathsf{messageList}[(role, pid)]$.
  Call **MessagePreprocessing**$(m, t, \mathsf{id}, \mathsf{internalState}, \mathsf{CorruptionSet})$.
    {*Before sending a message m'' on tape t'' belonging to a dummy in role role, append $(\texttt{sent}, t'', m'')$ to* $\mathsf{messageList}[(role, pid)]$. *Also goes for* **Main**.
  Call **Main**$(m, t, \mathsf{id}, \mathsf{internalState}, \mathsf{CorruptionSet})$.
  abort. {*In case no message was sent.*

Fig. 6.12: The **Compute** mode of ideal functionalities (IFs).

- **recv** mp_{in} **from** NET **s.t.** ⟨condition⟩, if the message is to be received on the network tape.
- **recv** mp_{in} **from** $(pid, role, role_{hl})$ **s.t.** ⟨condition⟩, if an ideal functionality expects a message from a dummy instance with PID *pid* in role *role*; here and in the following, $role_{hl}$ specifies the role of the sender in the higher-level protocol which used the dummy. Recall that if the ID of the IF instance is $(sid_1, \ldots, sid_n)$, then such a message is received via the dummy instance $D_{\mathsf{role}_{role}}[(pid, (sid_1, \ldots, sid_n))]$. Also, recall that a dummy $D_{\mathsf{role}_{role}}$ in role *role* has several PAR tapes (and corresponding SUB tapes), one for each role of a higher-level protocol connected to the dummy. For the above header, the message is expected on the tape corresponding to role $role_{hl}$ of the higher level protocol.
- **recv** mp_{in} **from** $(\mathsf{PAR}, role_{hl})$ **s.t.** ⟨condition⟩, if a real protocol machine expects a message from a higher-level protocol in role $role_{hl}$.
- **recv** mp_{in} **from** $(\mathcal{S}[sid_{n+1}], role_{ll})$ **s.t.** ⟨condition⟩, if an RPM expects a message from its subroutine $\mathcal{S}$ with session ID extension sid_{n+1}, where the sender takes role $role_{ll}$ in $\mathcal{S}$. If the instance of the RPM has ID $(pid, (sid_1, \ldots, sid_n))$, then

this means that this instance expects a message from the subroutine instance with ID $(pid, (sid_1, \ldots, sid_{n+1}))$ and role $role_{ll}$. Note that this uniquely identifies the instance of the subroutine from which a message is expected.

In all cases, the message pattern mp_{in} specifies the format and structure the incoming message has to satisfy (see Section 6.2.1 for a definition of message patterns). As described in Section 6.2.2 and Section 6.2.3, incoming messages always have the form $m = (id, m')$, i.e., they need to be prefixed either by id or by an extension thereof. This structure does not need to be enforced by mp_{in}, but the protocol designer only has to specify the format of m'; the correct prefixing is taken care of by the framework. Hence, m' (not m) will be matched with mp_{in}. Also note that, similar to message patterns, if one uses an undefined *local variable* to describe, e.g., the $role_{hl}$, then this variable will accept any value and store it. Finally, ⟨condition⟩ specifies a condition that has to be satisfied for entering this block. This condition might depend on the incoming message itself, as well as on the internal state of the instance of the machine (again see Figures 6.13 and 6.15 for examples).

- The header of a block is followed by ⟨*code*⟩, describing the actual computations performed by this block. This can be any arbitrary probabilistic algorithm. We use the command **abort** to say that the machine aborts by producing empty output, in which case the environment gets activated upon empty input. Note that this implies that only interactive algorithms may use this command.
- The activation of an instance ends as soon as the machine sends a message (or executes **abort**). The notation here is analogous to the header of the block and has one of the following forms (see again Figures 6.13 and 6.15 for examples):
  - **send** mp_{out} **to** NET, if the message is to be sent on the network tape.
  - **send** mp_{out} **to** $(pid, role, role_{hl})$, if an IF sends a message (via the corresponding dummy) on an PAR tape to an instance of a higher-level protocol with PID pid in role $role_{hl}$ that accessed the functionality via the dummy in role $role$. This message will be sent to the dummy instance $D_{\mathsf{role}_{role}}[(pid, (sid_1, \ldots, sid_n))]$, where $(sid_1, \ldots, sid_n)$ is the ID of the IF, on the tape corresponding to the higher-level protocol role $role_{hl}$.
  - **send** mp_{out} **to** $(\mathsf{PAR}, role_{hl})$, if an RPM sends a message on a PAR tape to an instance of role $role_{hl}$ in the higher-level protocol.
  - **send** mp_{out} **to** $(\mathcal{S}[sid_{n+1}], role_{ll})$, if an RPM sends a message to role $role_{ll}$ in session $sid = (sid_1, \ldots, sid_{n+1})$ of the subprotocol $\mathcal{S}$.
  - **reply** mp_{out}, if the message is to be sent back exactly to the sender of the message upon which the machine was activated.

  Similar to before, the outgoing message pattern mp_{out} only needs to specify the payload m' of an outgoing message. The correct prefixing by the ID according to Section 6.2.2 and Section 6.2.3 is handled by the framework.

Whenever an instance of a machine gets activated and enters one of the parts of the specification (i.e., **Initialization**, **MessagePreprocessing**, or **Main**), it will not continue at the point where it stopped, but go through the entire sequence of blocks starting with the first one, and enter the first block for which the message pattern, the sender and the specified condition are satisfied. Note that the order of the blocks can

therefore determine the behavior of a machine if incoming messages could satisfy the templates and conditions of multiple blocks, and we thus recommend to avoid this, e.g., by adding headers specifying the type of every message.

Sometimes it is convenient not to go through all blocks again but start right at the line where a message was sent, preserving all values of local variables. For this, a protocol designer may also use a variant of the *wait for* command defined in Section 6.2.1:

send mp_{out} **to** *dest*;
wait for mp_{in} **from** *source* **s.t.** ⟨*condition*⟩

This command essentially does the same as the *wait for* command defined in Section 6.2.1, but automatically prefixes messages with the right ID and removes the prefix of incoming messages. Furthermore, *dest* and *source* are not tape names but instead use the same syntax as the *send* and *recv* commands from above (e.g., if an RPM wants to send a message to role $role_{ll}$ in session $sid = (sid_1, \ldots, sid_{n+1})$ of the subprotocol $\mathcal{S}$, it uses $dest = (\mathcal{S}[sid_{n+1}], role_{ll})$).

A protocol designer may also send a generic restricting message on the NET tape, which can be used to get, e.g., algorithms and key material from the environment/adversary/simulator without influencing any other instances of the protocol. For this, as already explained in Section 6.2.2, he may use the following command:

send responsively mp_{out} **to** NET
wait for mp_{in} **from** NET **s.t.** ⟨condition⟩

Formally, this command first prefixes m, which is built according to the pattern mp_{out}, with both the id and the special string `Respond`, i.e., it actually sends the (raw-)message $(\mathsf{id}, (\texttt{Respond}, m))$. This is done because every message that is prefixed with an ID and the special string `Respond` has to be answered immediately (with an arbitrary message) by the definition of our restriction, see [CEK+16a] for the exact definition. Then, just as the command defined in Section 6.2.1, this command will wait for a response, check whether the response matches mp_{in} and *condition* is fulfilled, and then, if this is the case, continue the computation at this point. If one of the checks fails, it sends the message $(\mathsf{id}, (\texttt{Respond}, m))$ again (thus still requiring an immediate answer) and waits for the next answer. This guarantees that no other instance of the protocol will do anything until a correct message was sent by the adversary to the instance that used this command. Note that a protocol designer can use m to create arbitrary restricting messages, which can be used to, e.g., model different requests for keys and algorithms. We want to emphasize that a protocol designer has to pay very close attention when using this construct: On the one hand, this command must never be used to model real network traffic, since it will then overly restrict the distinguishing environment, i.e., some (real) attacks will no longer be possible in the model. On the other hand, it is possible to define a message pattern or *condition* in such a way that an environment does not know which message has to be sent to continue the run (e.g. by requiring a witness to an NP problem).

6.4 An Example Functionality and its Realization: Digital Signatures

In this section, we provide an example for how to apply our model and conventions to actual functionalities: we specify the digital signature functionality $\mathcal{F}_{\text{sig}}$ and its realization P_{sig}. On a high level, the ideal functionality $\mathcal{F}_{\text{sig}}$ has to guarantee that a verification request for a message/signature pair only succeeds if the message had been signed using the functionality. This behavior implies perfect unforgeability of the signature scheme.

For digital signatures, there are two roles: the role of a signer, who can sign arbitrary messages, and the role of a verifier, who can verify given message/signature pairs. Accordingly, the ideal system has the form $\mathcal{I}_{\text{sig}} = \{D_{\text{sig}}^{\mathsf{signer}}, D_{\text{sig}}^{\mathsf{verifier}}, \mathcal{F}_{\text{sig}}\}$, where $D_{\text{sig}}^{\mathsf{signer}}$ and $D_{\text{sig}}^{\mathsf{verifier}}$ are the respective dummies.

6.4.1 The Ideal Functionality $\mathcal{F}_{\text{sig}}$

Figures 6.13–6.14 specify $\mathcal{F}_{\text{sig}}$ using our conventions presented before. The ID of $\mathcal{F}_{\text{sig}}$ is of the form $(sid_1, \ldots, (pid, sid_n'))$ where (pid, sid_n') describes the session paramaters of $\mathcal{F}_{\text{sig}}$: The first component pid describes the owner of the key in this session. Only the owner may sign messages, but everyone is allowed to verify them. The second component sid_n' can be used to model different keys per person, e.g., for different email addresses or domains that belong to the same person. Of course, a protocol can model a single key by only using a fixed value for sid_n'.

Upon its first activation (with a message not related to corruption), the **Initialization** block of the machine asks the adversary for signing and verification algorithms, as well as for the keys that should be used by the functionality. As the adversary might submit algorithms that cannot be evaluated efficiently, the functionality needs to be parametrized by a polynomial p, which is used to truncate the running time of those algorithms. For instance, $\mathsf{sig}^{(\mathsf{p})}(msg, \mathsf{sk})$ means that $\mathcal{F}_{\text{sig}}$ runs sig on the given inputs for at most p steps; if the algorithm has not terminated by then, $\mathcal{F}_{\text{sig}}$ aborts sig and sets its output to $\perp$, cf. also Section 6.2.3.3.

Upon being initialized by the signer (`InitSign`), the functionality returns the public encryption key pk to the signer, whose identity is encoded in the last component of the SID. Now, whenever receiving a `Sign`-request for a message msg from the signer (but not from any other party), the functionality will sign msg and check that the resulting signature σ is valid. If this is the case, σ is returned to the signer, otherwise it returns $\perp$.

Upon receiving a `Verify` request for a message/signature pair from an arbitrary party for a given public key, $\mathcal{F}_{\text{sig}}$ distinguishes two cases: If the submitted public key is not the one of the signer or the signer has been corrupted, $\mathcal{F}_{\text{sig}}$ just verifies the signature and returns the result. If the submitted public key belongs to the signer, in order to guarantee unforgeability, $\mathcal{F}_{\text{sig}}$ only returns true if the signature is valid and the message had been signed by the functionality before.

This basic behavior of $\mathcal{F}_{\text{sig}}$ is in line with previous definitions [KT08, Can03]. We slightly deviate from the standard definition to allow for an elegant joint state formulation

Protocol Setup:

Participating roles: signer, verifier.
Corruption model: dynamic.
Protocol parameters:
– $\mathsf{p} \in \mathbb{Z}[x]$. {*Polynomial that bounds the runtime of algorithms provided by the adversary.*

Continued in Figure 6.14...

Fig. 6.13: The ideal signature functionality $\mathcal{F}_{\text{sig}}$.

and realization, cf. Section 6.5.2. We therefore modify the functionality such that a party sending input to $\mathcal{F}_{\text{sig}}$ will always obtain a response (potentially $\perp$ for invalid requests), which seems convenient anyway.

6.4.2 Realizing $\mathcal{F}_{\text{sig}}$

Similar to previous realizations of $\mathcal{F}_{\text{sig}}$, our realization P_{sig} is parametrized by a signature scheme $\Sigma = (\mathsf{gen}, \mathsf{sig}, \mathsf{ver})$. As we shall see, this scheme must be EUF-CMA secure (c.f. Section 2.7).

Figure 6.15 specifies the realization P_{sig} of $\mathcal{F}_{\text{sig}}$. According to the two roles in the protocol, the real protocol has the form $\mathsf{P}_{\text{sig}} = \{M_{\mathsf{signer}}, M_{\mathsf{verifier}}\}$. The protocol is parametrized by the aforementioned signature scheme $\Sigma = (\mathsf{gen}, \mathsf{sig}, \mathsf{ver})$. As can be seen, the realization of $\mathcal{F}_{\text{sig}}$ essentially consists of executing the algorithms of Σ.

We now obtain the following theorem, the proof of which is similar to the one found in [KT08], and therefore omitted.

Theorem 6.1. *Let* p *be a polynomial and let* $\Sigma = (\mathsf{gen}, \mathsf{sig}, \mathsf{ver})$ *be a digital signature scheme such that* $(\mathsf{gen}^{(\mathsf{p})}, \mathsf{sig}^{(\mathsf{p})}, \mathsf{ver}^{(\mathsf{p})})$ *is* correct*; then the protocol* $\mathsf{P}_{\text{sig}} = \{M_{\mathsf{signer}}, M_{\mathsf{verifier}}\}$ *realizes* $\mathcal{I}_{\text{sig}} = \{D_{\mathsf{signer}}, D_{\mathsf{verifier}}, \mathcal{F}_{\text{sig}}\}$ *if and only if* $(\mathsf{gen}^{(\mathsf{p})}, \mathsf{sig}^{(\mathsf{p})}, \mathsf{ver}^{(\mathsf{p})})$ *is EUF-CMA secure.*

6.5 Joint State

Composition theorems (for unbounded self-composition) state that it suffices to prove that a real protocol realizes an ideal functionality in a single session in order to conclude that multiple sessions of the real protocol realize multiple sessions of the ideal functionality. The problem is that this requires the states of the different sessions of the protocols/functionalities to be disjoint. In particular, the random coins used in different sessions have to be chosen independently. This, for example for digital signatures or public-key encryption, means that a party would have to choose new key pairs for *every* session, which is completely impractical.

Canetti and Rabin [CR03] therefore proposed composition theorems with joint state, or joint state (composition) theorems for short, to solve this problem (see also [KT08, KT09]). These theorems allow multiple protocol sessions to share common state. For example, in a joint state realization of $\mathcal{F}_{\text{sig}}$ the signing/verification keys of one party are used in all sessions.

...continued from Figure 6.13.

Description of $\mathcal{F}_{\text{sig}}$:

Internal state:

- $(\mathsf{sig}, \mathsf{ver}, \mathsf{pk}, \mathsf{sk}) \in (\{0,1\}^* \cup \{\bot\})^4 = (\bot, \bot, \bot, \bot)$. {*Algorithms and key pair.*
- $\mathsf{pidowner} \in \{0,1\}^* \cup \{\bot\} = \bot$. {*Party ID of the key owner.*
- $\mathsf{msglist} \subseteq \{0,1\}^* = \emptyset$. {*Set of recorded messages.*
- $\mathsf{StatusSign} \in \{0,1\}^* \cup \{\bot\} = \bot$. {*The signing status.*

CheckIDformat{signer}: accept iff id has the form $(pid, (sid_1, \ldots, sid_{n-1}, (pid, sid_n{}')))$.[a]
CheckIDformat{verifier}: accept iff id has the form $(pid, (sid_1, \ldots, sid_{n-1}, (pid', sid_n{}')))$.
CheckIDformatIdeal: accept iff id has the form $(sid_1, \ldots, sid_{n-1}, (pid, sid_n{}'))$.
Corruption behavior: LeakedData: if the signer is corrupted, return StatusSign. Otherwise return $\bot$.
Initialization:

send responsively `InitMe` **to** NET;
wait for $(\texttt{Init}, (sig, ver, pk, sk))$ **from** NET.
$(\mathsf{sig}, \mathsf{ver}, \mathsf{pk}, \mathsf{sk}) \leftarrow (sig, ver, pk, sk)$.
Parse id as $(sid_1, \ldots, sid_{n-1}, (pid, sid_n{}'))$.
$\mathsf{pidowner} \leftarrow pid$.

MessagePreprocessing:

recv _ **from** $(pid, \mathsf{verifier}, \_)$ **s.t.** $(pid, \mathsf{verifier}) \in \mathsf{CorruptionSet}$:
reply $\bot$. {*Prevent corrupted users from verifying. Otherwise $\mathcal{A}$ may be able to test if messages are in* msglist.

Main:

recv `InitSign` **from** $(\mathsf{pidowner}, \mathsf{signer}, \_)$:
$\mathsf{StatusSign} \leftarrow \texttt{ready}$.
reply $(\texttt{InitSign}, \texttt{success}, \mathsf{pk})$. {*Successful initialization. Note that* signer *can submit* `InitSign` *multiple times, always with the same effect.*

recv $(\texttt{Sign}, msg)$ **from** $(\mathsf{pidowner}, \mathsf{signer}, \_)$ **s.t.** $\mathsf{StatusSign} = \texttt{ready}$:
$\sigma \leftarrow \mathsf{sig}^{(\mathsf{p})}(msg, \mathsf{sk})$.
$b \leftarrow \mathsf{ver}^{(\mathsf{p})}(msg, \sigma, \mathsf{pk})$. {*Sign and check that verification succeeds.*
if $\sigma = \bot \vee b \neq \mathsf{true}$:
reply $(\texttt{Signature}, \bot)$. {*Signing or verification test failed.*
else:
add msg **to** msglist.
reply $(\texttt{Signature}, \sigma)$. {*Record msg for verification and return signature.*

recv $(\texttt{Verify}, msg, \sigma, pk)$ **from** $(pid, \mathsf{verifier}, \_)$:
$b \leftarrow \mathsf{ver}^{(\mathsf{p})}(msg, \sigma, pk)$. {*Verify signature.*
if $pk = \mathsf{pk} \wedge b = \mathsf{true} \wedge msg \notin \mathsf{msglist} \wedge (\mathsf{pidowner}, \mathsf{signer}) \notin \mathsf{CorruptionSet}$:
reply $(\texttt{VerResult}, \mathsf{false})$. {*Prevent forgery.*
else:
reply $(\texttt{VerResult}, b)$. {*Return verification result.*

recv _ **from** $(pid, role_F, \_)$:
reply $\bot$. {*Default answer if request failed. By this, uncorrupted users will always obtain a response.*

[a] Recall that, for **CheckIDformat**, id does *not* refer to the ID of the ideal functionality $\mathcal{F}_{\text{sig}}$ but instead refers to the ID of the dummy that executes **CheckIDformat**.

Fig. 6.14: The ideal signature functionality $\mathcal{F}_{\text{sig}}$ (continued). Note that AllowDummyCorruption? is not specified, which means that we take the default behavior for this algorithm, i.e., it always returns true.

Protocol Setup:

Participating roles: signer, verifier.
Corruption model: dynamic with erasures.
Protocol parameters:
- a signature scheme $\Sigma = (\mathsf{gen}, \mathsf{sig}, \mathsf{ver})$ (which should be EUF-CMA secure).

Realization of M_{signer}:

Implemented role: signer.
Internal state:
- $(\mathsf{sk}, \mathsf{pk}) \in (\{0,1\}^* \cup \{\bot\})^2 = (\bot, \bot)$. {*Key pair.*
- $\mathsf{StatusSign} \in \{\bot, \texttt{ready}\} = \bot$. {*The signing status.*

CheckIDformat: accept iff id can be parsed as $(pid, (sid_1, \ldots, sid_{n-1}, (pid, sid_n')))$.
Initialization:
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{gen}(1^\eta)$.
Main:

recv `InitSign` **from** (PAR, _):
  $\mathsf{StatusSign} \leftarrow \texttt{ready}$.
  reply $(\texttt{InitSign}, \texttt{success}, \mathsf{pk})$.
{*Only the specified owner in the session may get* pk *or sign.*

recv $(\texttt{Sign}, msg)$ **from** (PAR, _) **s.t.** $\mathsf{StatusSign} = \texttt{ready}$:
  $\sigma \leftarrow \mathsf{sig}(msg, \mathsf{sk})$.
  reply $(\texttt{Signature}, \sigma)$.
{*Sign msg, return signature.*

recv _ **from** (PAR, _):
  reply $\bot$.
{*Error message if not initialized.*

Realization of M_{verifier}:

Implemented role: verifier.
CheckIDformat: accept iff id can be parsed as $(pid, (sid_1, \ldots, sid_{n-1}, (pid', sid_n')))$.
Main:

recv $(\texttt{Verify}, msg, \sigma, pk)$ **from** (PAR, _):
  $b \leftarrow \mathsf{ver}(msg, \sigma, pk)$.
  reply $(\texttt{VerResult}, b)$.
{*Verify, return result.*

recv _ **from** (PAR, _):
  reply $\bot$.
{*Answer all other cases with an error message.*

Fig. 6.15: The realization $\mathsf{P}_{\mathrm{sig}}$ of the ideal signature protocol $\mathcal{I}_{\mathrm{sig}}$.

In the UC and GNUC models, extensions to the frameworks are needed to properly model joint state. Also, specific composition theorems with joint state are needed. Conversely, such a theorem follows immediately from the general composition theorem available in the IITM model, as shown in [KT08, KT13].

One important reason for this is that in the IITM model the addressing of machines is not fixed a priori, but can be specified flexibly using the **CheckAddress** mode. In other models, a hierarchical addressing of machines with PIDs and SIDs is hard-wired into the models, which imposes a tree (or forest) of instances. However, a joint state realization has to handle requests for different sessions in one session-independent instance. In particular, the ID of an instance of the joint state realization will typically be a PID

and this instance takes care of all requests for this party in all sessions.[8] Hence, this breaks the addressing conventions, particularly, the tree structure, and thus requires changes to the models themselves.

Another reason for the smooth treatment of joint-state in the IITM model is that in the IITM model the composition theorems are very general. They apply to a very broad class of protocol systems. Hardly any assumption about a real or ideal protocol system is made (see also Section 2.5.3 and [CEK+16a]). In particular, a real protocol can be formulated in such a way that it realizes the ideal protocol with joint state. The composition theorems of the IITM model then still apply because the joint state realization still belongs to the class of real protocols handled by the composition theorems of the IITM model.

So, the responsive IITM model itself can seamlessly deal with joint state, and this benefit is also shared by the IITM model with responsive environments [CEK+16a]. However, to facilitate the design process of protocols in the responsive IITM model, in Section 6.2 we defined specific instantiations for real and ideal protocols, including the **CheckAddress** mode that they use. These conventions are meant to be close to common conventions for real and ideal protocols, and thus again enforce a hierarchical addressing mechanism and a tree structure for instances. Therefore, for the same reason mentioned before, within the class of real protocols defined according to our conventions it is not possible to specify joint-state realizations.[9]

In this section, we therefore extend our conventions to also deal with joint-state realizations. We emphasize that the class of real protocols defined by these extended conventions is still a subclass of the general class of real protocols in the sense of the responsive IITM model. So, still there is no need to extend or modify the responsive IITM model itself or to prove new composition theorems, as the extended conventions are just a specific instantiation of the responsive IITM model and the theorems proved within this model. See [CEK+16a] for a more detailed discussion.

In what follows, we first introduce our conventions for joint-state protocols/realizations, and then present an example joint-state protocol for signatures to illustrate these conventions.

6.5.1 Conventions for Joint-State Protocols

In what follows, we present our conventions for joint-state protocols. We first start with the general structure of these protocols and provide an overview. We then define the different components in detail and also introduce useful syntax to specify joint-state protocols.

[8] For example, for signatures, the joint state realization has to handle signing and verification requests for a party *pid* coming from different sessions. Hence, there will be one instance of the joint state realization per PID, rather than per PID and SID.

[9] Note that typically a protocol designer would first design and prove the security of the protocols in the hierarchical setting and only later ideal functionalities for certain primitives will be replaced by (often already existing) joint-state realizations in order to obtain realistic implementations. In other words, for most cryptographic tasks a protocol designer just relies on the existence of joint-state realizations but does not have to consider joint-state herself.

Structure and Overview. A joint-state protocol system is a generalization of a real protocol system, in that instances thereof handle multiple sessions. A joint-state protocol system consists not only of atomic real-protocols (ARPs) and/or ideal protocols (IPs), but, importantly, also of *atomic joint-state protocols* (AJSPs), which in turn consist of a collection of *joint-state ITMs* (JSMs). In Figure 6.17, $\mathcal{J}$ is a joint-state protocol, $\mathcal{T}$ is an AJSP with two JSMs, i.e., $\mathcal{T} = \{W_1, W_2\}$, and $\mathcal{I}$ and $\mathcal{I}'$ are IPs. Typically, $\mathcal{I}$ and $\mathcal{I}'$ coincide up to the names of tapes. In particular, if $\mathcal{I} = \{D_1, \ldots, D_l, \mathcal{F}\}$, then $\mathcal{I}' = \{D'_1, \ldots, D'_l, \mathcal{F}'\}$, where D_i coincides with D'_i and $\mathcal{F}$ with $\mathcal{F}'$ up to tape names. In the example presented in Section 6.5.2, both IFs coincide with $\mathcal{I}_{\text{sig}}$ (up to tape names). Roughly speaking, the purpose of $\mathcal{T}$ is to realize multiple sessions of $\mathcal{I}$ in just one session of $\mathcal{I}'$.

More precisely, let us assume that we have a higher-level protocol $\mathcal{P}$ which uses $\mathcal{I} = \{D_1, \ldots, D_l, \mathcal{F}\}$ (or its joint-state realization $\mathcal{J}$), as its subroutine. Let us assume that the RPM M_i describes the ith role of $\mathcal{P}$. Recall that according to our conventions, if $M_i[(pid, (sid_1, \ldots, sid_{n-1}))]$, i.e., party pid in the ith role of session $(sid_1, \ldots, sid_{n-1})$, wants to send a request to, say the jth role of its subroutine, then it sends a request to $D_j[(pid, (sid_1, \ldots, sid_n))]$, for some sid_n. This request is forwarded by this dummy instance, if not corrupted, to $\mathcal{F}[(sid_1, \ldots, sid_n)]$.

The number n of roles in $\mathcal{T} = \{W_1, \ldots, W_l\}$, i.e., the number of JSMs in such a protocol, coincides with the number of roles (dummies) in the ideal functionality $\mathcal{I}$ it is supposed to realize. As defined precisely below, the ID of an instance of W_j is of the form (pid, sid_n). This instance handles all requests for party pid in *any* session $(sid_1, \ldots, sid_{n-1})$ of the higher-level protocol run by pid. That is, if $M_i[(pid, (sid_1, \ldots, sid_{n-1}))]$ (see above) sends a request to $D_j[(pid, (sid_1, \ldots, sid_n))]$, then, if $\mathcal{I}$ is replaced by its joint-state realization, this request is handled by $W_j[(pid, sid_n)]$. Usually, $W_j[(pid, sid_n)]$ uses another ideal protocol $\mathcal{I}' = \{D'_1, \ldots, D'_l, \mathcal{F}'\}$ as subroutine to process this request, where $\mathcal{I}$ and $\mathcal{I}'$ coincide except for renamed tapes. In such a case, $W_j[(pid, sid_n)]$ uses the dummy instance $D'_j[(pid, sid_n)]$, which in turn uses $\mathcal{F}'[sid_n]$. See Figure 6.16 for an example.

Note that sid_n is set by the higher-level protocol. However, $\mathcal{I}$, and analogously its joint-state realization $\mathcal{J}$, might expect a certain format (specified in **CheckIDformat**). For example, $\mathcal{I}/\mathcal{J}$ might require sid_n to be $\perp$. In this case, the above means that there is (at most) one instance $W_j[(pid, \perp)]$ of W_j per party pid and role j in every run. These instances use only one instance, namely $\mathcal{F}[\perp]$ (and its dummies) to handle all instances $\mathcal{F}[(sid_1, \ldots, sid_{n-1}, \perp)]$ (and their dummies) for all $(sid_1, \ldots, sid_{n-1}, \perp)$.

As is clear from the above, the main difference between an RPM and a JSM is that a JSM has to handle multiple sessions. Also, an instance of a JSM invokes a (instance of a) subroutine with its own ID (e.g., $W_j[(pid, sid_n)]$ invokes $D'_j[(pid, sid_n)]$, rather than $D'_j[(pid, sid_n, sid_{n+1})]$, which we would require from an RPM instance). While the latter is not essential, the fact that one JSM instance takes care of multiple sessions is crucial and this is the core of every joint-state realization. In the IITM model, this is easy to model. We merely have to modify the **CheckAddress** mode of JSM compared to RPMs and in the **Compute** store the appropriate ID.

So, for the specification of joint-state realization we have to (slightly) break out of our conventions for RPMs. But we do not have to break out of the responsive IITM model itself: joint-state realizations are still real protocols in the sense of the responsive

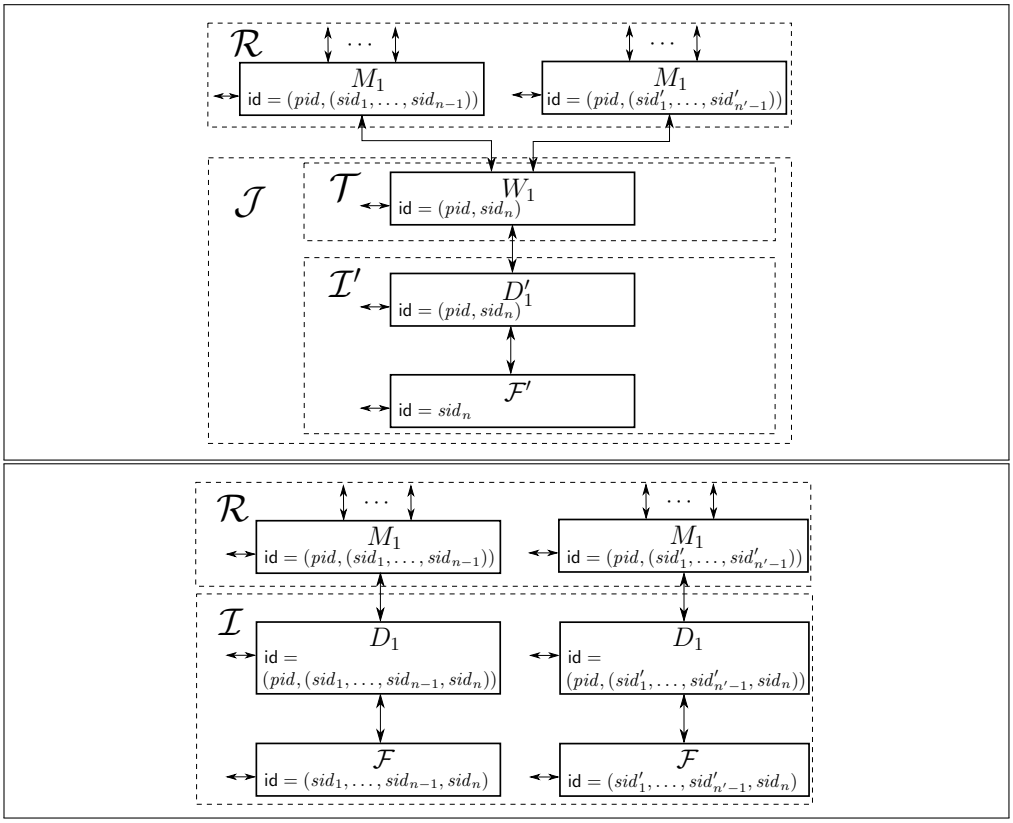


Fig. 6.16: Example for the dynamic structure of an (atomic) real protocol $\mathcal{R} = \{M_1\}$, which uses an ideal functionality $\mathcal{I} = \{D_1, \mathcal{F}\}$ (bottom) or its joint state realization $\mathcal{J} = \{\mathcal{T}, \mathcal{I}'\}$ (top), where $\mathcal{T} = \{W_1\}$ is an atomic joint state protocol and $\mathcal{I}' = \{D'_1, \mathcal{F}'\}$ is an ideal protocol. There are two instances of M_1 in different sessions, potentially with SIDs of different lengths, which both want to talk to a subroutine whose SID extends their own SID by sid_n. If these instances send messages to $\mathcal{I}$, they will access different instances of the dummy and the ideal functionality; however, if they send messages to the joint-state realization $\mathcal{J}$, they both access the same instances of W_1, the dummy and the ideal functionality.

IITM model (see Section 2.5.3 and [CEK+16a]). In particular, the composition theorem still applies. To obtain such theorem, other models needed to be extended, since, as explained before, it is not possible to directly specify joint-state realizations in such models. Also, this theorem is not a trivial consequence of the composition theorems in these model, but required a dedicated proof.

In what follows, we define joint-state protocols and joint-state machines in more detail.

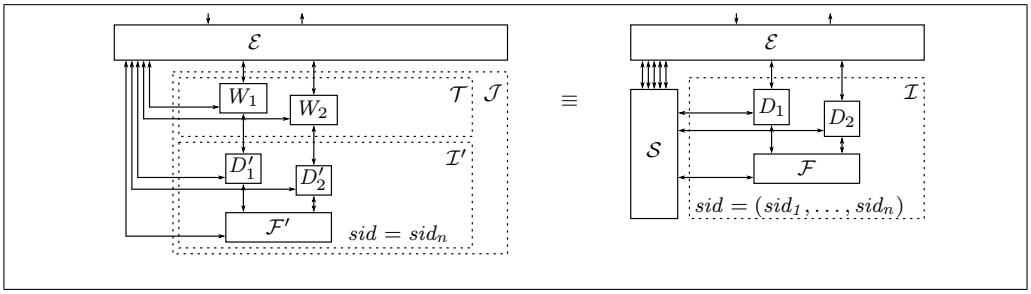


Fig. 6.17: Universal composability with joint state ($\mathcal{J} \leq \mathcal{I}$) and an example for the usual internal structure of joint-state protocols. Here we show the strong simulatabilty setting (which is equivalent to the UC and dummy UC settings).

Upon receiving a message m on tape t in mode **CheckAddress** do the following.

- If $\mathsf{id} = \bot$ (i.e., this is the first message the instance received):
  - If t is the NET or a SUB tape: parse the message as $m = (id, m')$ for $id = (pid, sid_n)$; otherwise return false. Return whatever **CheckIDformat**(m, t, id) does.
  - If t is a PAR tape: parse the message as $m = ((pid, (sid_1, \ldots, sid_n)), m')$ for some $n \geq 1$; otherwise return false. Return whatever **CheckIDformat**$(m, t, (pid, sid_n))$ does.
- Else ($\mathsf{id} \neq \bot$, i.e., this is not the first message the instance received and id was set in mode **Compute** before):
  - If t is the NET or a SUB tape: Check if the message parses as $m = (\mathsf{id}, m')$; if it does, return true; otherwise return false.
  - If t is a PAR tape: Check if the message parses as $m = (id', m')$ where $id' = (pid, (sid_1, \ldots, sid_n))$ and $(pid, sid_n) = \mathsf{id}$; if it does, return true; otherwise return false.

Fig. 6.18: The **CheckAddress** mode of joint-state machines.

Atomic Joint-State Protocols. As already mentioned, an AJSP is a collection of *joint-state ITMs* (JSM): we write $\mathcal{T} = \{W_1, \ldots, W_n\}$. See for example Figure 6.17. JSMs are similar to real protocol machines (RPMs), except that one instance of a JSM is active in multiple sessions.

The identity id of a JSM instance consists of a party identity and a shortened session identity: $\mathsf{id} = (pid, sid_n)$. The instance is active in all sessions $(sid_1, \ldots, sid_n)$. The session identifier of subroutine instances is just sid_n, rather than (sid_n, sid_{n+1}), for some sid_{n+1}, as also mentioned above. The **CheckAddress** and **Compute** of JSMs are adapted from those of RPMs accordingly; see Figures 6.18 and 6.19–6.20 for the formal definition. As can be seen from Figures 6.19–6.20, the **Compute** mode coincides with the one for RPMs, except for the handling of IDs.

Template for AJSPs. The template for ARPs in Figure 6.4 is also used for AJSPs.

Wiring. ARPs and AJSPs use the same wiring for PAR tapes and the NET tape, however, SUB tapes are handled differently. Formally, if W is a machine in an AJSP which has an (ideal, atomic real or atomic joint-state) protocol $\mathcal{Q}$ listed as subroutine, then, for every machine M in $\mathcal{Q}$, W has as many SUB tapes as it has PAR tapes and all of them connect to a PAR tape of M. This allows W to use different tapes if different machines in a higher level protocol activate W; in this case for the machine M it looks

State variable id $\in \{0,1\}^* \cup \{\bot\} = \bot$. {*Identity of the instance.*
State variable corr $\in \{$false, true, $\bot\} = \bot$. {*Corruption status.*
State variable subcorr $\in \{$false, true$\}$ = false. {*Consider instance corrupted?*
State variable init $\in \{$false, true$\}$ = false. {*"Inner shell" initialized?*
State variable internalState. {*State of "inner shell" of instance as per template in Fig. 6.4.*
State variable transcript. {*Transcript of all actions of "inner shell".*

Upon receiving a message $m = ((pid, sid_n), m')$ from the NET or a SUB tape t,
or a message $m = ((pid, (sid_1, \ldots, sid_n)), m')$ from a PAR tape t do:
if id $= \bot$**:**
  id $\leftarrow (pid, sid_n)$. {*Initialize identity of instance.*
if corr $= \bot$**:** {*Initialize corruption status.*
  if $t =$ NET $\wedge\, m' =$ (SetCorruptionStatus, b)**:** {*First message sets corruption status.*
    corr $\leftarrow b \wedge$ AllowCorruption?(id, internalState, m).
      {*AllowCorruption? is only executed if* $b =$ true.
    send (id, (CorruptionStatus, corr, $\bot$)) **on** NET.
  else: {*For all other first messages.*
    send responsively (id, CorruptMe?) **on** NET;
    wait for (id, (SetCorruptionStatus, b)) **on** NET.
    corr $\leftarrow b \wedge$ AllowCorruption?(id, $\bot$).
      {*AllowCorruption? is only executed if* $b =$ true.
    if corr $=$ true**:** {*If* corr $=$ false*: continue processing* (m, t).
      if $t =$ PAR$[role_{hl}] \wedge\, m' =$ CorruptionStatus?**:**
        {*This message is processed after the notification.*
        send responsively (id, (CorrStatRestricting, true, (m, t))) **on** NET;
        wait for (id, OK) **on** NET. {*Continue processing* m.
      else:
        send (id, (CorruptionStatus, true, (m, t))) **on** NET.
else if corr $=$ false $\wedge\, t =$ NET $\wedge\, m' =$ (SetCorruptionStatus, true)**:** {*Dynamic corruption.*
  corr $\leftarrow b \wedge$ AllowCorruption?(id, internalState, $\bot$).
  if static corruption is specified**:**
    corr $\leftarrow$ false.
  if corr $=$ true**:**
    if secure erasures are possible**:**
      send (id, (CorruptionStatus, true, internalState)) **on** NET.
    else:
      send (id, (CorruptionStatus, true, transcript)) **on** NET.
  else:
    send (id, (CorruptionStatus, false, $\bot$)) **on** NET.
else if corr $=$ false $\wedge\, t =$ NET $\wedge\, m' =$ (SetCorruptionStatus, false)**:**
    {*(Useless) corruption request.*
  send (id, (CorruptionStatus, false, $\bot$)) **on** NET.

Continued in Figure 6.20...

Fig. 6.19: The **Compute** mode of joint-state machines.

...continued from Figure 6.19.

if $t = \mathsf{PAR}[role_{hl}] \wedge m' = \texttt{CorruptionStatus}?$: {*Query status questions to subroutine.*
  subcorr $\leftarrow$ subcorr $\vee$ corr $\vee$ DetermineCorruptionStatus($(pid, (sid_1, \ldots, sid_n))$, internalState).[a]
    {*DetermineCorruptionStatus is only executed if* subcorr = corr = false.
  send $((pid, (sid_1, \ldots, sid_n)), (\texttt{CorruptionStatus}, \text{subcorr}))$ **on** t.
if corr = true: {*Corrupted instances are multiplexers.*
  if $t = \mathsf{PAR}[role_{hl}]$:
    send $(\mathsf{id}, (\texttt{par}, role_{hl}, (sid_1, \ldots, sid_n), m'))$ **on** NET.
  else if $t = \mathsf{NET} \wedge m' = (\texttt{par}, role_{hl}, (sid_1, \ldots, sid_n), m'')$:
    {*Note that* sid_n *must be the same as the one in* id.
    send $((pid, (sid_1, \ldots, sid_n)), m'')$ **on** $\mathsf{PAR}[role_{hl}]$.
  else if $t = \mathsf{SUB}[role_{ll}]$:
    send $(\mathsf{id}, (\texttt{sub}, role_{ll}, m'))$ **on** NET.
  else if $t = \mathsf{NET} \wedge m' = (\texttt{sub}, role_{ll}, m'')$:
    send (id, m'') **on** $\mathsf{SUB}[role_{ll}]$.
else: {*Honest behaviour.*
  Append (m, t) to transcript.
  if init = false:
    init $\leftarrow$ true. {*Initialize internal state upon first regular message.*
    Call **Initialization**$(m, t, \mathsf{id}, \text{internalState}, \text{transcript})$.
  Call **MessagePreprocessing**$(m, t, \mathsf{id}, \text{internalState}, \text{transcript})$.
  Call **Main**$(m, t, \mathsf{id}, \text{internalState}, \text{transcript})$.
abort. {*In case no message was sent.*

[a] Note that DetermineCorruptionStatus not only gets id but the full prefix of the message. Thus, it may depend on the exact session that is simulated by this single joint-state instance.

Fig. 6.20: The **Compute** mode of joint-state machines (continued).

like different higher level protocols try to access it, instead of only a single machine W. In the next paragraph, we introduce a simple syntax to use these different SUB tapes.

Additional syntax for joint-state machines. As JSMs differ from RPMs, it is necessary to introduce additional syntax on top of what we defined in Section 6.3 for the sending and receiving of messages by JSMs. In particular, we need different commands for sending and receiving messages on PAR and SUB tapes, while the commands for sending and receiving messages on NET tapes are the same. As before, the framework takes care of the correct prefixing of messages, so as to not overburden the protocol designer.

Additional types of receive blocks in JSMs:

- **recv** mp_{in} **from** $(\mathsf{PAR}, sid, role_{hl})$ **s.t.** $\langle$condition$\rangle$, if the message is to be received from a higher-level protocol with role $role_{hl}$ in session sid. So, if the higher-level protocol sends a message of the form $((pid, (sid_1, \ldots, sid_n)), m')$, then mp_{in} is matched with m' and the local variable sid is set to $(sid_1, \ldots, sid_{n-1})$.
- **recv** mp_{in} **from** $(\mathcal{I}, role_{hl})$ **s.t.** $\langle$condition$\rangle$, if the message is to be received from the subroutine ideal protocol $\mathcal{I}$ in the higher-level role $role_{hl}$; the lower-level role is the same as the role of this joint-state machine instance. Recall that the instance which sent this message has, according to our conventions, the same ID as the

instance of the joint-state realization, and therefore, no ID information needs to be provided.

Additional types of send blocks in JSMs:

- **send** mp_{out} **to** $(\mathsf{PAR}, sid, role_{hl})$, if the message is to be sent to a higher-level protocol with role $role_{hl}$ in the session sid. Recall that an instance of the joint-state realization has an ID of the form (pid, sid_n), but it can send message to instances of higher-level protocols where the messages are prefixed with $(pid, (sid_1, \ldots, sid_n))$. To do so, in the send block specified above, this means that sid is defined to be $(sid_1, \ldots, sid_{n-1})$, i.e., sid denotes the SID of the higher level protocol instance.
- **send** mp_{out} **to** $(\mathcal{I}, role_{hl})$, if the message is to be sent to the subroutine ideal protocol $\mathcal{I}$ in the higher-level role $role_{hl}$; again, the lower-level role is the same as the role of this joint-state machine instance and the IDs of the instance of the joint-state machine and the instance of the subroutine coincide, so this information does not have be mentioned in the send block.

6.5.2 Joint State Realization of Signatures

We now construct a joint-state realization $\mathcal{J}_{\text{sig}}$ of the signature ideal functionality $\mathcal{I}_{\text{sig}}$. The joint state protocol $\mathcal{J}_{\text{sig}}$ consists of an AJSP $\mathcal{T}_{\text{sig}}$ and another copy $\mathcal{I}'_{\text{sig}}$ of the signature ideal functionality: $\mathcal{J}_{\text{sig}} = \{\mathcal{T}_{\text{sig}}, \mathcal{I}'_{\text{sig}}\}$. Further, the AJSP $\mathcal{T}_{\text{sig}}$ consists of two JSMs: W_{signer} and W_{verifier}, corresponding to the two roles in $\mathcal{I}'_{\text{sig}}$.

Recall that there is one instance of $\mathcal{I}_{\text{sig}}$ per $sid = (sid_1, \ldots, sid_n)$ in the ideal world, but only one instance of $\mathcal{I}'_{\text{sig}}$ per sid_n in the joint-state world. The AJSP $\mathcal{T}_{\text{sig}}$ acts as an adapter so that one instance of $\mathcal{I}'_{\text{sig}}$ can emulate multiple instances of $\mathcal{I}_{\text{sig}}$. The rough idea of the construction of $\mathcal{T}_{\text{sig}}$ is that if $\mathcal{T}_{\text{sig}}$ receives a message x to be signed or verified, it forwards (sid, x) to $\mathcal{I}'_{\text{sig}}$, instead of just x, thereby binding the SID to the signature, and hence preventing that signatures generated in one session can be successfully used in another session (otherwise, one could easily distinguish the joint-state protocol from the ideal protocol since the ideal protocol can not share state between sessions). We provide the formal construction of $\mathcal{T}_{\text{sig}}$ in Figures 6.21–6.22.

Theorem 6.2. *For every polynomial p', there is a polynomial p such that the joint-state protocol $\mathcal{J}_{\text{sig}} = \{\mathcal{T}_{\text{sig}}, \mathcal{I}'_{\text{sig}}\}$ (where $\mathcal{I}'_{\text{sig}}$ uses the polynomial p' to bound the runtime of externally provided algorithms) realizes $\mathcal{I}_{\text{sig}}$ (where $\mathcal{I}_{\text{sig}}$ uses the polynomial p to bound the runtime of externally provided algorithms), i.e.,*

$$\mathcal{J}_{\text{sig}} = \{\mathcal{T}_{\text{sig}}, \mathcal{I}'_{\text{sig}}\} = \{\{W_{\mathsf{signer}}, W_{\mathsf{verifier}}\}, \mathcal{I}'_{\text{sig}}\} \leq \mathcal{I}_{\text{sig}}.$$

The proof of the above theorem is straightforward and therefore omitted.

By the construction of $\mathcal{J}_{\text{sig}}$: in every run of $\mathcal{J}_{\text{sig}}$, for every pid and ${sid_n}'$, we have at most one instance of $W_{\mathsf{signer}}[(pid, (pid, {sid_n}'))]$ but potentially several instances of $W_{\mathsf{verifier}}[(pid', (pid, {sid_n}'))]$, one for each pid'. These instances use $\mathcal{F}'_{\text{sig}}[(pid, {sid_n}')]$ to handle all requests in sessions $(sid_1, \ldots, sid_{n-1}, (pid, {sid_n}'))$ of $\mathcal{I}_{\text{sig}}$. In particular, there is only one key pair generated for each pair $(pid, {sid_n}')$ in the joint state world, while there is a new key pair in the ideal world for *every* session $(sid_1, \ldots, sid_{n-1}, (pid, {sid_n}'))$.

Protocol Setup:

Participating roles: signer, verifier.
Corruption model: dynamic with erasures.

Realization of the joint state machine W_{signer} for the signer role:

Implemented role: signer.
Subroutines: $\mathcal{I}_{\mathrm{sig}}$ from Figure 6.13, denoted $\mathcal{I}'_{\mathrm{sig}}$.
Internal state:
- lastSid $\in \{0,1\}^* \cup \{\bot\} = \bot$. {*The sid of the last message from* PAR.
- initedSids $\subset \{0,1\}^* \cup \{\bot\} = \emptyset$. {*List of sids that completed initialization.*

CheckIDformat: accept iff id has the form $(pid, (pid, sid_n'))$.[a]
Corruption behavior: AllowCorruption? and DetermineCorruptionStatus:
return corr($\mathcal{I}'_{\mathrm{sig}}$, signer).
Main:

recv `InitSign` **from** (PAR, sid, $role_{hl}$) **s.t.** lastSid $= \bot$:
lastSid $\leftarrow sid$.
send `InitSign` **to** ($\mathcal{I}'_{\mathrm{sig}}$, $role_{hl}$). {$\mathcal{F}_{\mathrm{sig}}$ *will respond consistently in case of duplicate queries.*

recv (`InitSign`, `success`, pk) **from** ($\mathcal{I}'_{\mathrm{sig}}$, $role_{hl}$) **s.t.** lastSid $\neq \bot$:
Insert lastSid into initedSids.
($lastsid$, lastSid) $\leftarrow$ (lastSid, $\bot$).
send (`InitSign`, `success`, pk) **to** (PAR, $lastsid$, $role_{hl}$).

recv (`Sign`, x) **from** (PAR, sid, $role_{hl}$) **s.t.** lastSid $= \bot \wedge sid \in$ initedSids:
lastSid $\leftarrow sid$.
send (`Sign`, (sid, x)) **to** ($\mathcal{I}'_{\mathrm{sig}}$, $role_{hl}$). {*Prepend sid to message.*

recv (`Signature`, σ) **from** ($\mathcal{I}'_{\mathrm{sig}}$, $role_{hl}$) **s.t.** lastSid $\neq \bot$:
($lastsid$, lastSid) $\leftarrow$ (lastSid, $\bot$).
send (`Signature`, σ) **to** (PAR, $lastsid$, $role_{hl}$).

recv _ **from** (PAR, sid, $role_{hl}$) **s.t.** lastSID $= \bot$: {*Handle malformed messages or sign requests if uninitialized.*
lastSid $\leftarrow sid$.
send $\bot$ **to** ($\mathcal{I}'_{\mathrm{sig}}$, $role_{hl}$). {$\mathcal{F}_{\mathrm{sig}}$ *might initialize.*

recv $\bot$ **from** ($\mathcal{I}'_{\mathrm{sig}}$, $role_{hl}$) **s.t.** lastSid $\neq \bot$:
($lastsid$, lastSid) $\leftarrow$ (lastSid, $\bot$).
send $\bot$ **to** (PAR, $lastsid$, $role_{hl}$).

[a] Recall that by the **CheckAddress** mode of JSMs id is of the form (pid, sid_n). Now, **CheckIDformat** additionally imposes the structure (pid, sid_n') on sid_n.

Continued in Figure 6.22...

Fig. 6.21: The atomic joint state protocol $\mathcal{T}_{\mathrm{sig}}$ in the joint state realization for $\mathcal{I}_{\mathrm{sig}}$.

Hence, if $\mathcal{I}_{\mathrm{sig}}$ is used by some higher level protocol and one would replace $\mathcal{I}_{\mathrm{sig}}$ by its realization $\mathsf{P}_{\mathrm{sig}}$ (the composition theorems allow one to do this), then this would typically yield a completely impractical realization. If the higher level uses $\mathcal{J}_{\mathrm{sig}}$ instead and in $\mathcal{J}_{\mathrm{sig}}$ one replaced $\mathcal{I}'_{\mathrm{sig}}$ by its realization (also $\mathsf{P}_{\mathrm{sig}}$ but with tapes renamed), then this would yield a realistic realization.

As already mention in Section 6.4, $\mathcal{I}_{\mathrm{sig}}$ requires sid_n to be of the form (pid, sid_n') instead of just pid because this way a higher-level protocol can handle different keys used for different purposes by the same pid: The SID sid_n' might for example be a domain name (e.g., www.example.com or subdomain.example.com) or a protocol name

...continued from Figure 6.21.

Realization of the joint state machine W_{verifier} for the verifier role:

Implemented role: verifier.
Subroutines: $\mathcal{I}_{\text{sig}}$ from Figure 6.13, denoted $\mathcal{I}'_{\text{sig}}$.
CheckIDformat: accept iff id has the form $(pid, (pid', sid_n'))$.
Corruption behavior: AllowCorruption? and DetermineCorruptionStatus:
  return corr$(\mathcal{I}'_{\text{sig}}, \mathsf{verifier})$.
Internal state:
– lastSid $\in \{0,1\}^* \cup \{\bot\} = \bot$. {*The sid of the last message from* PAR.
Main:
  recv $(\texttt{Verify}, x, \sigma, pk)$ **from** $(\mathsf{PAR}, sid, role_{hl})$ **s.t.** lastSid $= \bot$**:**
    lastSid $\leftarrow sid$.
    send $(\texttt{Verify}, (sid, x), \sigma, pk)$ **to** $(\mathcal{I}'_{\text{sig}}, role_{hl})$. {*Prepend sid to message.*

  recv $(\texttt{VerResult}, b)$ **from** $(\mathcal{I}'_{\text{sig}}, role_{hl})$ **s.t.** lastSid $\neq \bot$**:**
    $(lastsid, \mathsf{lastSid}) \leftarrow (\mathsf{lastSid}, \bot)$.
    send $(\texttt{VerResult}, b)$ **to** $(\mathsf{PAR}, lastsid, role_{hl})$.

  recv _ **from** $(\mathsf{PAR}, sid, role_{hl})$ **s.t.** lastSID $= \bot$**:** {*Handle malformed messages.*
    lastSid $\leftarrow sid$.
    send $\bot$ **to** $(\mathcal{I}'_{\text{sig}}, role_{hl})$. {$\mathcal{F}_{\text{sig}}$ *might initialize.*

  recv $\bot$ **from** $(\mathcal{I}'_{\text{sig}}, role_{hl})$ **s.t.** lastSid $\neq \bot$**:**
    $(lastsid, \mathsf{lastSid}) \leftarrow (\mathsf{lastSid}, \bot)$.
    send $\bot$ **to** $(\mathsf{PAR}, lastsid, role_{hl})$.

Fig. 6.22: The atomic joint state protocol $\mathcal{T}_{\text{sig}}$ in the joint state realization for $\mathcal{I}_{\text{sig}}$ (continued).

(TLS or SSH), and the party uses different keys for different domains and different keys for different protocols. So, in the joint state realization $\mathcal{J}_{\text{sig}}$ uses one instance of $\mathcal{F}_{\text{sig}}$ for every party *pid* and domain only, while in the ideal world there is an instance of $\mathcal{F}_{\text{sig}}$ for every party *pid*, domain, *and* every SID.

7

Concluding Remarks

In this thesis we studied the design of practical and secure protocols. We now briefly discuss a number of open problems which we did not cover.

We designed a (sub-)protocol for arithmetic circuit evaluation in which one can securely employ an output from the circuit as an input to another (sub-)protocol and vice-versa, by using commitments and zero-knowledge proofs. Our solution requires separate commitments to the value being shared in both sub-protocols. To further improve efficiency, it would be desirable for the sub-protocols to share the same commitments, unfortunately the UC model does not allow this. Future work should design better UC functionalities that allow such sharing. Furthermore, even if our protocol is the fastest of its kind, in many real-world applications one needs protocols with even shorter runtime: achieving better efficiency, especially for devices with limited computational capabilities, is therefore another good future research direction.

The design techniques we used in our 2-server TPASS protocol were specific to two parties (or a small number of parties). However, those techniques are not tailored to protocols with a large number of parties, and providing techniques for the design of practical protocols of this kind is an open problem. As a starting point, we suggest improving the multi-server TPASS protocol of Camenisch et al. [CLLN14] to be resistant against adaptive/transient corruptions.

In the area of modeling imperfectly erasable memory, we did not model all types of memory that appear in practice. We believe it would be interesting to expand the modeling of those memories, for example, by modeling multi-use memory, i.e., memory that can be overwritten; and modeling adaptive leakage, i.e., memory that can leak multiple times. Other interesting tasks are a systematic study of the best possible protocols that realize perfectly erasable memory in different settings.

We proposed conventions for writing protocols in a flexible and convenient manner in the responsive IITM model. Future work should provide a library of common functionalities in the framework, which could be cherry-picked as needed when designing higher-level protocols, provide writing conventions for formulating simulators and security proofs, and extend the conventions to global setup.

Throughout the development of this thesis, we were surprised that cryptographic protocol design and proof is still a largely manual process. Protocol design shares some

similarities with software engineering, except for the absence of tools one has come to expect in the latter. Even if the design of a practical automatic proof creator or checker seems out of immediate reach, we believe that the development of simple tools will be of tremendous help for further protocol design. For example, it would be useful, given their description using our conventions of Chapter 6, to be able to actually run a protocol on the one hand, and an ideal functionality with a simulator on the other hand. Thereby protocol designers could write automated unit tests to see if the two actually produce identical (or indistinguishable) output for a particular distinguishing strategy, thus freeing them for manually performing this tedious task.

References

AMQR15. Dirk Achenbach, Jörn Müller-Quade, and Jochen Rill. Universally composable firewall architectures using trusted hardware. Cryptology ePrint Archive, Report 2015/099, 2015. http://eprint.iacr.org/.

BBCM95. Charles H. Bennett, Gilles Brassard, Claude Crépeau, and Ueli M. Maurer. Generalized privacy amplification. *IEEE Transactions on Information Theory*, 41(6):1915–1923, 1995.

BCC+11. S. Babbage, D. Catalano, C. Cid, B. de Weger, O. Dunkelman, C. Gehrmann, L. Granboulan, T. Güneysu, J. Hermans, T. Lange, A. Lenstra, C. Mitchell, M. Näslund, P. Nguyen, C. Paar, K. Paterson, J. Pelzl, T. Pornin, B. Preneel, C. Rechberger, V. Rijmen, M. Robshaw, A. Rupp, M. Schläffer, S. Vaudenay, F. Vercauteren, and M. Ward. ECRYPT II Yearly Report on Algorithms and Keysizes, 2011.

BCL+05. Boaz Barak, Ran Canetti, Yehuda Lindell, Rafael Pass, and Tal Rabin. Secure computation without authentication. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 361–377. Springer, August 2005.

BDN+11. W. E. Burr, D. F. Dodson, E. M. Newton, R. A. Perlner, W. T. Polk, S. Gupta, and E. A. Nabbus. Electronic authentication guideline. NIST Special Publication 800-63-1, 2011.

BDOZ11. Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 169–188. Springer, May 2011.

Ber05. Daniel J Bernstein. Cache-timing attacks on aes, 2005.

BH93. Donald Beaver and Stuart Haber. Cryptographic protocols provably secure against dynamic adversaries. In Rainer A. Rueppel, editor, *EUROCRYPT'92*, volume 658 of *LNCS*, pages 307–323. Springer, May 1993.

BIB89. Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In Piotr Rudnicki, editor, *8th ACM PODC*, pages 201–209. ACM, August 1989.

BJKS03. John Brainard, Ari Juels, Burton S. Kaliski Jr., and Michael Szydlo. A new two-server approach for authentication with short secrets. In *Proceedings of the 12th USENIX Security Symposium (SECURITY 2003)*, pages 201–214, Washington, DC, USA, August 5–9, 2003. USENIX Association.

BJSL11. Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 11*, pages 433–444. ACM Press, October 2011.

BM84. G. R. Blakley and Catherine Meadows. Security of ramp schemes. In G. R. Blakley and David Chaum, editors, *CRYPTO'84*, volume 196 of *LNCS*, pages 242–268. Springer, August 1984.

BPW07. M. Backes, B. Pfitzmann, and M. Waidner. The reactive simulatability (RSIM) framework for asynchronous systems. *Information and Computation*, 205(12):1685–1720, 2007.

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.

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.

Can03. Ran Canetti. Universally composable signatures, certification and authentication. Cryptology ePrint Archive, Report 2003/239, 2003. http://eprint.iacr.org/2003/239.

CC06. Hao Chen and Ronald Cramer. Algebraic geometric secret sharing schemes and secure multi-party computations over small fields. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 521–536. Springer, August 2006.

CCGS10. Jan Camenisch, Nathalie Casati, Thomas Groß, and Victor Shoup. Credential authenticated identification and key exchange. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 255–276. Springer, August 2010.

CCS09. Jan Camenisch, Nishanth Chandran, and Victor Shoup. A public key encryption scheme secure against key dependent chosen plaintext and adaptive chosen ciphertext attacks. In Antoine Joux, editor, *EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 351–368. Springer, April 2009.

CDH+00. Ran Canetti, Yevgeniy Dodis, Shai Halevi, Eyal Kushilevitz, and Amit Sahai. Exposure-resilient functions and all-or-nothing transforms. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 453–469. Springer, May 2000.

CDN01. Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 280–299. Springer, May 2001.

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, February 2007.

CEGL08a. Ran Canetti, Dror Eiger, Shafi Goldwasser, and Dah-Yoh Lim. How to protect yourself without perfect shredding. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 511–523. Springer, July 2008.

CEGL08b. Ran Canetti, Dror Eiger, Shafi Goldwasser, and Dah-Yoh Lim. How to protect yourself without perfect shredding. Cryptology ePrint Archive, Report 2008/291, 2008. http://eprint.iacr.org/2008/291.

CEK+16a. Jan Camenisch, Robert R. Enderlein, Stephan Krenn, Ralf Küsters, and Daniel Rausch. Universal composition with responsive environments. Cryptology ePrint Archive, Report 2016/034, 2016. http://eprint.iacr.org/2016/034.

CEK+16b. Jan Camenisch, Robert R. Enderlein, Stephan Krenn, Ralf Küsters, Daniel Rausch, and Oana Ciobotaru. Universal composability—conventions for complete and unambiguous protocol specifications. In submission, 2016.

CES13. Jan Camenisch, Robert R. Enderlein, and Victor Shoup. Practical and employable protocols for UC-secure circuit evaluation over $\mathbb{Z}_n$. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 19–37. Springer, September 2013.

CF01. Ran Canetti and Marc Fischlin. Universally composable commitments. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 19–40. Springer, August 2001.

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.

CHK05a. Ran Canetti, Shai Halevi, and Jonathan Katz. Adaptively-secure, non-interactive public-key encryption. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 150–168. Springer, February 2005.

CHK+05b. Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. Universally composable password-based key exchange. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 404–421. Springer, May 2005.

CKL06. Ran Canetti, Eyal Kushilevitz, and Yehuda Lindell. On the limitations of universally composable two-party computation without set-up assumptions. *Journal of Cryptology*, 19(2):135–167, April 2006.

CKS11. Jan Camenisch, Stephan Krenn, and Victor Shoup. A framework for practical universally composable zero-knowledge protocols. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 449–467. Springer, December 2011.

CLLN14. Jan Camenisch, Anja Lehmann, Anna Lysyanskaya, and Gregory Neven. Memento: How to reconstruct your secrets from a single password in a hostile environment. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 256–275. Springer, August 2014.

CLN12. Jan Camenisch, Anna Lysyanskaya, and Gregory Neven. Practical yet universally composable two-server password-authenticated secret sharing. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12*, pages 525–536. ACM Press, October 2012.

CLOS02. Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *34th ACM STOC*, pages 494–503. ACM Press, May 2002.

CR03. Ran Canetti and Tal Rabin. Universal composition with joint state. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 265–281. Springer, August 2003.

CS97. J. Camenisch and M. Stadler. Proof Systems for General Statements about Discrete Logarithms. *Institute for Theoretical Computer Science, ETH Zürich, Tech. Rep.*, 260, 1997.

CS98. Ronald Cramer and Victor Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 13–25. Springer, August 1998.

CS99. Ronald Cramer and Victor Shoup. Signature schemes based on the strong RSA assumption. In *ACM CCS 99*, pages 46–51. ACM Press, November 1999.

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, August 2003.

CW79. Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.

DCFIJ99. Giovanni Di Crescenzo, Niels Ferguson, Russell Impagliazzo, and Markus Jakobsson. How to forget a secret. In *STACS 99*, pages 500–509. Springer, 1999.

DFK+06. Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 285–304. Springer, March 2006.

DG03. Mario Di Raimondo and Rosario Gennaro. Provably secure threshold password-authenticated key exchange. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 507–523. Springer, May 2003.

DJ03. Ivan Damgård and Mads Jurik. A length-flexible threshold cryptosystem with applications. In Reihaneh Safavi-Naini and Jennifer Seberry, editors, *ACISP 03*, volume 2727 of *LNCS*, pages 350–364. Springer, July 2003.

DK10. Ivan Damgård and Marcel Keller. Secure multiparty AES. In Radu Sion, editor, *FC 2010*, volume 6052 of *LNCS*, pages 367–374. Springer, January 2010.

DKL+12. Ivan Damgård, Marcel Keller, Enrique Larraia, Christian Miles, and Nigel P. Smart. Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In Ivan Visconti and Roberto De Prisco, editors, *SCN 12*, volume 7485 of *LNCS*, pages 241–263. Springer, September 2012.

DKL+13. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, September 2013.

DN03. Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 247–264. Springer, August 2003.

DO10a. Ivan Damgård and Claudio Orlandi. Multiparty computation for dishonest majority: From passive to active security at low cost. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 558–576. Springer, August 2010.

DO10b. Ivan Damgård and Claudio Orlandi. Multiparty computation for dishonest majority: from passive to active security at low cost. Cryptology ePrint Archive, Report 2010/318, 2010. http://eprint.iacr.org/2010/318.

DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, August 2012.

DY05. Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In Serge Vaudenay, editor, *PKC 2005*, volume 3386 of *LNCS*, pages 416–431. Springer, January 2005.

EMC. EMC Corporation. RSA distributed credential protection. http://www.emc.com/security/rsa-distributed-credential-protection.htm.

FK00. Warwick Ford and Burton S. Kaliski Jr. Server-assisted generation of a strong secret from a password. In *9th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2000)*, pages 176–180, Gaithersburg, MD, USA, June 4–16, 2000. IEEE Computer Society.

FY92. Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *24th ACM STOC*, pages 699–710. ACM Press, May 1992.

GMR89. Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

GMT. Peter Gaži, Ueli Maurer, and Björn Tackmann. Manuscript.

GMW87. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

Gol01. Oded Goldreich. *Foundations of Cryptography*, volume 1. Cambridge University Press, 2001. Cambridge Books Online.

Gos12. Jeremi M. Gosney. Password cracking HPC. Passwords^12 Conference, 2012.

Gut96. Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the Sixth USENIX Security Symposium, San Jose, CA*, volume 14, 1996.

HJKY95. Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In Don Coppersmith, editor, *CRYPTO'95*, volume 963 of *LNCS*, pages 339–352. Springer, August 1995.

HLP15. Carmit Hazay, Yehuda Lindell, and Arpita Patra. Adaptively secure computation with partial erasures. Cryptology ePrint Archive, Report 2015/450, 2015. http://eprint.iacr.org/2015/450.

Hof11. Dennis Hofheinz. Possibility and impossibility results for selective decommitments. *Journal of Cryptology*, 24(3):470–516, July 2011.

HS11. Dennis Hofheinz and Victor Shoup. GNUC: A new universal composability framework. Cryptology ePrint Archive, Report 2011/303, 2011. http://eprint.iacr.org/2011/303.

HS15. Dennis Hofheinz and Victor Shoup. GNUC: A new universal composability framework. *Journal of Cryptology*, 28(3):423–508, July 2015.

IPS08. Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure arithmetic computation with no honest majority. Cryptology ePrint Archive, Report 2008/465, 2008. http://eprint.iacr.org/2008/465.

IPS09. Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure arithmetic computation with no honest majority. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 294–314. Springer, March 2009.

Jab01. David P. Jablon. Password authentication using multiple servers. In David Naccache, editor, *CT-RSA 2001*, volume 2020 of *LNCS*, pages 344–360. Springer, April 2001.

JKK14. Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 233–253. Springer, December 2014.

JL00. Stanislaw Jarecki and Anna Lysyanskaya. Adaptively secure threshold cryptography: Introducing concurrency, removing erasures. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 221–242. Springer, May 2000.

JL09. Stanislaw Jarecki and Xiaomin Liu. Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 577–594. Springer, March 2009.

JS07. Stanislaw Jarecki and Vitaly Shmatikov. Efficient two-party secure computation on committed inputs. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 97–114. Springer, May 2007.

Kah96. David Kahn. *The Codebreakers: The comprehensive history of secret communication from ancient times to the internet.* Simon and Schuster, 1996.

KL15. Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography.* CRC Press, 2015.

KMTG05. Jonathan Katz, Philip D. MacKenzie, Gelareh Taban, and Virgil D. Gligor. Two-server password-only authenticated key exchange. In John Ioannidis, Angelos

Keromytis, and Moti Yung, editors, *ACNS 05*, volume 3531 of *LNCS*, pages 1–16. Springer, June 2005.

Kre12. S. Krenn. *Bringing Zero-Knowledge Proofs of Knowledge to Practice*. PhD thesis, Université de Fribourg, 2012.

KshS12. Benjamin Kreuter, abhi shelat, and Chih hao Shen. Billion-gate secure computation with malicious adversaries. Cryptology ePrint Archive, Report 2012/179, 2012. http://eprint.iacr.org/2012/179.

KT08. Ralf Kuesters and Max Tuengerthal. Joint state theorems for public-key encryption and digital signature functionalities with local computation. Cryptology ePrint Archive, Report 2008/006, 2008. http://eprint.iacr.org/2008/006.

KT09. Ralf Küsters and Max Tuengerthal. Computational soundness for key exchange protocols with symmetric encryption. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *ACM CCS 09*, pages 91–100. ACM Press, November 2009.

KT13. Ralf Kuesters and Max Tuengerthal. The IITM model: a simple and expressive model for universal composability. Cryptology ePrint Archive, Report 2013/025, 2013. http://eprint.iacr.org/2013/025.

Kue06. Ralf Kuesters. Simulation-based security with inexhaustible interactive turing machines. Cryptology ePrint Archive, Report 2006/151, 2006. http://eprint.iacr.org/2006/151.

Lim08. Dah-Yoh Lim. *The paradigm of partial erasures*. PhD thesis, Massachusetts Institute of Technology, 2008.

LP07. Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 52–78. Springer, May 2007.

LPS08. Yehuda Lindell, Benny Pinkas, and Nigel P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *SCN 08*, volume 5229 of *LNCS*, pages 2–20. Springer, September 2008.

Mau02. Ueli M. Maurer. Indistinguishability of random systems. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 110–132. Springer, April / May 2002.

Mau10. Ueli Maurer. Constructive cryptography - a primer (invited paper). In Radu Sion, editor, *FC 2010*, volume 6052 of *LNCS*, page 1. Springer, January 2010.

Mau11a. Ueli Maurer. Constructive cryptography – a new paradigm for security definitions and proofs. In *Theory of Security and Applications (TOSCA 2011)*, volume 6993 of *Lecture Notes in Computer Science*, pages 33–56. Springer-Verlag, April 2011.

Mau11b. Ueli Maurer. Constructive Cryptography - A New Paradigm for Security Definitions and Proofs. In Sebastian Mödersheim and Catuscia Palamidessi, editors, *TOSCA 2011*, volume 6993 of *LNCS*, pages 33–56. Springer, 2011.

MR11. Ueli Maurer and Renato Renner. Abstract cryptography. In Bernard Chazelle, editor, *ICS 2011*, pages 1–21. Tsinghua University Press, January 2011.

MSJ02. Philip D. MacKenzie, Thomas Shrimpton, and Markus Jakobsson. Threshold password-authenticated key exchange. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 385–400. Springer, August 2002.

Nie03. J. B. Nielsen. *On Protocol Security in the Cryptographic Model*. PhD thesis, BRICS, Computer Science Department, University of Aarhus, 2003.

NNOB12. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, August 2012.

NO09. Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 368–386. Springer, March 2009.

Pai99. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 223–238. Springer, May 1999.

Ped92. Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 129–140. Springer, August 1992.

Plo60. Morris Plotkin. Binary codes with specified minimum distance. *Information Theory, IRE Transactions on*, 6(4):445–450, 1960.

PM99. Niels Provos and David Mazières. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91. USENIX, 1999.

PSSW09. Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, December 2009.

PW00. Birgit Pfitzmann and Michael Waidner. Composition and integrity preservation of secure reactive systems. In S. Jajodia and P. Samarati, editors, *ACM CCS 00*, pages 245–254. ACM Press, November 2000.

PW01. Birgit Pfitzmann and Michael Waidner. A model for asynchronous reactive systems and its application to secure message transmission. In *2001 IEEE Symposium on Security and Privacy*, pages 184–200. IEEE Computer Society Press, May 2001.

RBC13. Joel Reardon, David A. Basin, and Srdjan Capkun. SoK: Secure data deletion. In *2013 IEEE Symposium on Security and Privacy*, pages 301–315. IEEE Computer Society Press, May 2013.

RCB12. Joel Reardon, Srdjan Capkun, and David Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *Proceedings of the 21st USENIX conference on Security symposium*, pages 17–17. USENIX Association, 2012.

RRBC13. Joel Reardon, Hubert Ritzdorf, David A. Basin, and Srdjan Capkun. Secure data deletion from persistent media. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 271–284. ACM Press, November 2013.

Sho01. Victor Shoup. A proposal for an ISO standard for public key encryption. Cryptology ePrint Archive, Report 2001/112, 2001. http://eprint.iacr.org/2001/112.

Sho04. Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004. http://eprint.iacr.org/2004/332.

Sin11. Simon Singh. *The code book: the science of secrecy from ancient Egypt to quantum cryptography.* Anchor, 2011.

SK05. Michael Szydlo and Burton S. Kaliski Jr. Proofs for two-server password authentication. In Alfred Menezes, editor, *CT-RSA 2005*, volume 3376 of *LNCS*, pages 227–244. Springer, February 2005.

Wik04. Douglas Wikström. A universally composable mix-net. In Moni Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 317–335. Springer, February 2004.

WJ79. S. S. Wagstaff Jr. Greatest of the Least Primes in Arithmetic Progressions Having a Given Modulus. *Mathematics of Computation*, 33(147):pp. 1073–1080, 1979.

Yee94. Bennet Yee. *Using secure coprocessors.* PhD thesis, CMU, 1994.

YT95. Bennet Yee and J Doug Tygar. Secure coprocessors in electronic commerce applications. In *Proceedings of The First USENIX Workshop on Electronic Commerce, New York, New York*, 1995.

Appendix

A

Formal Definition of Ideal Functionalities

In this appendix, we recall the definition of the ideal functionalities we use as subroutines in our realizations, namely: the common reference string ideal functionality ($\mathcal{F}_{\text{crs}}^{D}$), the ideal functionalities for authenticated channels ($\mathcal{F}_{\text{ac}}$) and for one-sided–authenticated channels ($\mathcal{F}_{\text{osac}}$), and the special ideal functionality for zero-knowledge proofs of existence for one verifier ($\mathcal{F}_{\text{gzk}}$) and two verifiers ($\mathcal{F}_{\text{gzk}}^{2\text{v}}$). We adapted some of the functionalities to suit the needs of our protocols.

A.1 Common Reference Strings $\mathcal{F}_{\text{crs}}^{D}$

Here we describe the ideal functionality for common reference strings $\mathcal{F}_{\text{crs}}^{D}$ for a distribution D. Recall that we make use of two distributions in this thesis: $\mathcal{F}_{\text{crs}}^{\mathbb{G}^3}$ outputs a CRS uniformly distributed over $\mathbb{G}^3$ and $\mathcal{F}_{\text{crs}}^{\text{gzk}}$ outputs a CRS according to the distribution required by Camenisch et al.'s protocol π for zero-knowledge proofs of existance [CKS11]. The structure of $\mathcal{F}_{\text{crs}}^{D}$ is the same as what is defined in Canetti's UC paper [Can00].

Interfaces. $\mathcal{F}_{\text{crs}}^{D}$ is a two-interface system:

- The network interface, connected to the ideal adversary/simulator.
- The $\mathcal{U}$-interface, connected to the ideal peers of all the parties. This interface is multiplexed; we assume that headers are added to all messages to enable proper routing.

State. The ideal functionality is stateful and maintains the following data structures:

- $\underline{\textit{Seen}}$: a subset of $\{0,1\}^*$. Keeps track of which messages were accepted.
- $\underline{x}$: the stored CRS, initially $\underline{x} = \bot$.

We model the single-session variant of $\mathcal{F}_{\text{crs}}^{D}$, the session ID $sid = \underline{sid}$ is thus fixed.

Reacting to messages. $\mathcal{F}_{\text{crs}}^{D}$ reacts to messages as follows.

1. Receive $\langle \text{GetCRS:}pid, sid \rangle$ on $\mathcal{U}$ (from a party pid),
   such that $\{\text{“GetCRS:}pid\text{”}\} \cap \underline{\textit{Seen}} = \emptyset$:

Insert "GetCRS:*pid*" into $\underline{Seen}$.
If $\underline{x} = \perp$, then randomly choose $\underline{x}$ according to distribution D.
Send $\langle$GetCRS:*pid*, *sid*, $\underline{x}\rangle$ on network.

2. Receive $\langle$Deliver:*pid*, *sid*$\rangle$ on network,
such that $\{$"Deliver:*pid*"$\} \cap \underline{Seen} = \emptyset$
and $\{$"GetCRS:*pid*"$\} \subset \underline{Seen}$:

Insert "Deliver:*pid*" into $\underline{Seen}$.
Send $\langle$Deliver:*pid*, *sid*, $\underline{x}\rangle$ on $\mathcal{U}$ (to party *pid*).

A.2 Authenticated Channels $\mathcal{F}_{\text{ac}}$

Here we describe the ideal functionality for single-use authenticated channels $\mathcal{F}_{\text{ac}}$. The structure is the same as the one defined in Hofheinz and Shoup's GNUC paper [HS11].

Interfaces. $\mathcal{F}_{\text{ac}}$ is a three-interface system:

- The network interface, connected to the ideal adversary/simulator.
- The $\mathcal{P}$-interface, connected to the ideal peer of the sender.
- The $\mathcal{Q}$-interface, connected to the ideal peer of the receiver.

State. The ideal functionality is stateful and maintains the following data structures:

- $\underline{Seen}$: a subset of $\{0,1\}^*$. Keeps track of which messages were accepted.
- $\underline{x}$: the message that is to be sent, where $\underline{x} \in \{0,1\}^*$.

We model the single-session variant of $\mathcal{F}_{\text{ac}}$, the session ID $sid = \underline{sid}$ is thus fixed.

Reacting to messages. $\mathcal{F}_{\text{ac}}$ reacts to messages as follows.

1. Receive $\langle$Send, *sid*, $x\rangle$ on $\mathcal{P}$,
such that $\{$"Send"$\} \cap \underline{Seen} = \emptyset$:

Insert "Send" into $\underline{Seen}$.
Store the message: $\underline{x} \leftarrow x$.
Send $\langle$Send, *sid*, $\underline{x}\rangle$ on network.

2. Receive $\langle$Ready, *sid*$\rangle$ on $\mathcal{Q}$,
such that $\{$"Ready"$\} \cap \underline{Seen} = \emptyset$:

Insert "Ready" into $\underline{Seen}$.
Send $\langle$Ready, *sid*$\rangle$ on network.

3. Receive $\langle$Done, *sid*$\rangle$ on network,
such that $\{$"Done"$\} \cap \underline{Seen} = \emptyset$
and $\{$"Send", "Ready"$\} \subset \underline{Seen}$:

Insert "Done" into $\underline{Seen}$.
Send $\langle$Done, *sid*$\rangle$ on $\mathcal{P}$.

4. Receive $\langle$Deliver, *sid*$\rangle$ on network,
   such that $\{$"Deliver"$\} \cap \underline{Seen} = \emptyset$
   and $\{$"Send", "Ready"$\} \subset \underline{Seen}$:

   Insert "Deliver" into $\underline{Seen}$.
   Send $\langle$Deliver, *sid*, $\underline{x}\rangle$ on $\mathcal{Q}$.

5. Receive $\langle$Corrupt:$\mathcal{R}$, *sid*$\rangle$ on network,
   where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$,
   such that $\{$"Corrupt:$\mathcal{R}$"$\} \cap \underline{Seen} = \emptyset$:

   Insert "Corrupt:$\mathcal{R}$" into $\underline{Seen}$.
   Send $\langle$Corrupt:$\mathcal{R}$, *sid*$\rangle$ on $\mathcal{R}$.

6. Receive $\langle$Reset, *sid*, $x\rangle$ on network,
   such that $\{$"Reset"$\} \cap \underline{Seen} = \emptyset$
   and $\{$"Corrupt:$\mathcal{P}$"$\} \subset \underline{Seen}$:

   Insert "Reset" into $\underline{Seen}$.
   Store a new message: $\underline{x} \leftarrow x$.
   Send $\langle$Reset, *sid*$\rangle$ on network.

A.3 One-Sided–Authenticated Channels $\mathcal{F}_{\text{osac}}$

Here we describe our ideal functionality for multi-use one-sided–authenticated channels $\mathcal{F}_{\text{osac}}$. The structure is similar to the regular $\mathcal{F}_{\text{ac}}$ with the obvious extensions for multi-use, but we added an additional Hijack instruction to model the fact that the first message from the user is not authenticated.

Interfaces. $\mathcal{F}_{\text{osac}}$ is a three-interface system:

- The network interface, connected to the ideal adversary/simulator.
- The $\mathcal{U}$-interface, connected to the ideal peers of the users. This interface is multiplexed; we assume that headers are added to all messages to enable proper routing.
- The $\mathcal{Q}$-interface, connected to the ideal peer of the initial receiver.

State. The ideal functionality is stateful and maintains the following data structures:

- $\underline{Seen}$: a subset of $\{0,1\}^*$. Keeps track of which messages were accepted.
- $\underline{xc}$: an associative array between an integer and the message that is to be sent to the server.
- $\underline{xs}$: an associative array between an integer and the message that is to be sent to the user.
- $\underline{U}$: a user in the system. Keeps track of which user initiated a query. If $\underline{U} = \mathcal{A}$, $\mathcal{F}_{\text{osac}}$ sends/receives messages on the network interface instead of the $\mathcal{U}$-interface as written.

We model the single-session variant of $\mathcal{F}_{\text{osac}}$, the session ID *sid* = $\underline{sid}$ is thus fixed.

Reacting to messages. $\mathcal{F}_{\text{osac}}$ reacts to messages as follows.

Message from user to server. For messages that the user sends to the server, $\mathcal{F}_{\text{osac}}$ proceeds similarily to a multi-session $\mathcal{F}_{\text{ac}}$, except that the first message might be hijacked by $\mathcal{A}$.

1. Receive $\langle \text{Send:c:0}, sid, x \rangle$ on $\mathcal{U}$ (from a user pid_u),
   such that $\{\text{“Send:c:0”}\} \cap \underline{Seen} = \emptyset$:

   Insert “Send:c:0” into $\underline{Seen}$.
   Store the message: $\underline{xc}[0] \leftarrow x$. Record the user: $\underline{U} \leftarrow pid_u$.
   Send $\langle \text{Send:c:0}, sid, x \rangle$ on network.

2. Receive $\langle \text{Send:c:}qid, sid, x \rangle$ on $\mathcal{U}$ (from the user $\underline{U}$),
   where $qid \in \mathbb{N}^*$,
   such that $\{\text{“Send:c:}qid\text{”}\} \cap \underline{Seen} = \emptyset$,
   and $\{\text{“Done:c:}(qid-1)\text{”}\} \subset \underline{Seen}$:

   Insert “Send:c:qid” into $\underline{Seen}$.
   Store the message: $\underline{xc}[qid] \leftarrow x$.
   Send $\langle \text{Send:c:}qid, sid, x \rangle$ on network.

3. Receive $\langle \text{Ready:c:0}, sid \rangle$ on $\mathcal{Q}$,
   such that $\{\text{“Ready:c:0”}\} \cap \underline{Seen} = \emptyset$:

   Insert “Ready:c:0” into $\underline{Seen}$.
   Send $\langle \text{Ready:c:0}, sid \rangle$ on network.

4. Receive $\langle \text{Ready:c:}qid, sid \rangle$ on $\mathcal{Q}$,
   where $qid \in \mathbb{N}^*$,
   such that $\{\text{“Ready:c:}qid\text{”}\} \cap \underline{Seen} = \emptyset$,
   and $\{\text{“Deliver:c:}(qid-1)\text{”}\} \subset \underline{Seen}$:

   Insert “Ready:c:qid” into $\underline{Seen}$.
   Send $\langle \text{Ready:c:}qid, sid \rangle$ on network.

5. Receive $\langle \text{Done:c:}qid, sid \rangle$ on network,
   such that $\{\text{“Done:c:}qid\text{”}\} \cap \underline{Seen} = \emptyset$
   and $\{\text{“Send:c:}qid\text{”}, \text{“Ready:c:}qid\text{”}\} \subset \underline{Seen}$:

   Insert “Done:c:qid” into $\underline{Seen}$.
   Send $\langle \text{Done:c:}qid, sid \rangle$ on $\mathcal{U}$ (to the user $\underline{U}$).

6. Receive $\langle \text{Deliver:c:}qid, sid \rangle$ on network,
   such that $\{\text{“Deliver:c:}qid\text{”}\} \cap \underline{Seen} = \emptyset$
   and $\{\text{“Send”}, \text{“Ready”}\} \subset \underline{Seen}$:

   Insert “Deliver:c:qid” into $\underline{Seen}$.
   Send $\langle \text{Deliver:c:}qid, sid, \underline{xc}[qid] \rangle$ on $\mathcal{Q}$.

Message from server to user. For messages that the server sends to the user, $\mathcal{F}_{\text{osac}}$ proceeds similarly as for a multi-session $\mathcal{F}_{\text{ac}}$, i.e., there is no hijack. The server can start sending messages to the user only after it received the first message from the user.

7. Receive $\langle$Send:s:0, $sid, x\rangle$ on $\mathcal{Q}$,
   such that $\{$"Send:s:0"$\} \cap \underline{Seen} = \emptyset$
   and $\{$"Deliver:c:0"$\} \subset \underline{Seen}$:

   Insert "Send:s:0" into $\underline{Seen}$.
   Store the message: $\underline{xs}[0] \leftarrow x$.
   Send $\langle$Send:s:0, $sid, x\rangle$ on network.

8. Receive $\langle$Send:s:$qid, sid, x\rangle$ on $\mathcal{Q}$,
   where $qid \in \mathbb{N}^*$,
   such that $\{$"Send:s:qid"$\} \cap \underline{Seen} = \emptyset$,
   and $\{$"Done:s:$(qid-1)$"$\} \subset \underline{Seen}$:

   Insert "Send:s:qid" into $\underline{Seen}$.
   Store the message: $\underline{xs}[qid] \leftarrow x$.
   Send $\langle$Send:s:$qid, sid, x\rangle$ on network.

9. Receive $\langle$Ready:s:0, $sid\rangle$ on $\mathcal{U}$ (from the user $\underline{U}$),
   such that $\{$"Ready:s:0"$\} \cap \underline{Seen} = \emptyset$
   and $\{$"Done:c:0"$\} \subset \underline{Seen}$:

   Insert "Ready:s:0" into $\underline{Seen}$.
   Send $\langle$Ready:s:0, $sid\rangle$ on network.

10. Receive $\langle$Ready:s:$qid, sid\rangle$ on $\mathcal{U}$ (from the user $\underline{U}$),
    where $qid \in \mathbb{N}^*$,
    such that $\{$"Ready:s:qid"$\} \cap \underline{Seen} = \emptyset$,
    and $\{$"Deliver:s:$(qid-1)$"$\} \subset \underline{Seen}$:

    Insert "Ready:s:qid" into $\underline{Seen}$.
    Send $\langle$Ready:s:$qid, sid\rangle$ on network.

11. Receive $\langle$Done:s:$qid, sid\rangle$ on network,
    such that $\{$"Done:s:qid"$\} \cap \underline{Seen} = \emptyset$
    and $\{$"Send:s:qid", "Ready:s:qid"$\} \subset \underline{Seen}$:

    Insert "Done:s:qid" into $\underline{Seen}$.
    Send $\langle$Done:s:$qid, sid\rangle$ on $\mathcal{Q}$.

12. Receive $\langle$Deliver:s:$qid, sid\rangle$ on network,
    such that $\{$"Deliver:s:qid"$\} \cap \underline{Seen} = \emptyset$
    and $\{$"Send", "Ready"$\} \subset \underline{Seen}$:

    Insert "Deliver:s:qid" into $\underline{Seen}$.
    Send $\langle$Deliver:s:$qid, sid, \underline{xs}[qid]\rangle$ on $\mathcal{U}$ (to user $\underline{U}$).

Corruption. $\mathcal{F}_{\text{osac}}$ reacts to corruption, reset, and hijack messages as follows.

13. Receive $\langle$Corrupt:$\mathcal{Q}$, *sid*$\rangle$ on network,
    such that $\{$"Corrupt:$\mathcal{Q}$"$\} \cap \underline{Seen} = \emptyset$:

    Insert "Corrupt:$\mathcal{Q}$" into $\underline{Seen}$.
    Send $\langle$Corrupt:$\mathcal{Q}$, *sid*$\rangle$ on $\mathcal{Q}$.

14. Receive $\langle$Corrupt:$\mathcal{U}$, *sid*$\rangle$ on network,
    such that $\{$"Corrupt:$\mathcal{U}$"$\} \cap \underline{Seen} = \emptyset$
    and $\{$"Send:c:0"$\} \subset \underline{Seen}$:

    Insert "Corrupt:$\mathcal{U}$" into $\underline{Seen}$.
    Send $\langle$Corrupt:$\mathcal{U}$, *sid*$\rangle$ on $\mathcal{U}$ (to the user $\underline{U}$).

15. Receive $\langle$Reset:c:*qid*, *sid*, $x\rangle$ on network,
    where $qid \in \mathbb{N}$,
    such that $\{$"Reset:c:*qid*"$\} \cap \underline{Seen} = \emptyset$,
    and $\{$"Corrupt:$\mathcal{U}$"$\} \subset \underline{Seen}$:

    Insert "Reset:c:*qid*" into $\underline{Seen}$.
    Store a new message: $\underline{xc}[qid] \leftarrow x$.
    Send $\langle$Reset:c:*qid*, *sid*$\rangle$ on network.

16. Receive $\langle$Reset:s:*qid*, *sid*, $x\rangle$ on network,
    where $qid \in \mathbb{N}$,
    such that $\{$"Reset:s:*qid*"$\} \cap \underline{Seen} = \emptyset$,
    and $\{$"Corrupt:$\mathcal{Q}$"$\} \subset \underline{Seen}$:

    Insert "Reset:s:*qid*" into $\underline{Seen}$.
    Store a new message: $\underline{xs}[qid] \leftarrow x$.
    Send $\langle$Reset:s:*qid*, *sid*$\rangle$ on network.

17. Receive $\langle$Hijack, *sid*, $x\rangle$ on network,
    such that $\{$"Hijack", "Done:c:0", "Deliver:c:0"$\} \cap \underline{Seen} = \emptyset$
    and $\{$"Send:c:0"$\} \subset \underline{Seen}$:

    Insert "Hijack" into $\underline{Seen}$.
    Change $\underline{U}[qid] \leftarrow \mathcal{A}$. Store a new message: $\underline{xc}[qid] \leftarrow x$.
    Send $\langle$Hijack, *sid*, $x\rangle$ on network.

Runtime estimate of realization. While we do not provide a realization of this functionality, for the purposes of estimating the efficiency, we assume that the client does one encryption using a CCA-2 cryptosystem during initialization, and the server one decryption. Thereafter, they use symmetric cryptography.

A.4 Zero-Knowledge Proofs of Existence for One Verifier $\mathcal{F}_{\text{gzk}}$

The functionality $\mathcal{F}_{\text{gzk}}$ is a tool which allows one to simplify the security proof of protocols which use zero-knowledge proofs of *existence*. This functionality was proposed by Camenisch, Krenn, and Shoup [CKS11]. The two major differences between $\mathcal{F}_{\text{gzk}}$ and the traditional functionality for zero-knowledge proofs of knowledge (e.g., the one defined by Hofheinz and Shoup [HS11] in the GNUC model) is that the former *1)* does not check its inputs and *2)* does not allow the adversary to extract the witnesses quantified by $\exists$.

One must be careful with this functionality, since it is not intended to be used like a regular ideal functionality. Indeed the functionality is quite useless by itself. However, by using the special composition theorem by Camenisch, Krenn, and Shoup, one can prove that if the $\mathcal{F}_{\text{gzk}}$-hybrid protocol is secure against a weak class of environments called *nice* environments, and the protocol is such that honest provers never try to prove incorrect statements, then the modified protocol in which all instances of $\mathcal{F}_{\text{gzk}}$ have been replaced by the zero-knowledge protocol π described by Camenisch, Krenn, and Shoup is secure in the UC-sense.

In the definition here, unlike the definition of Camenisch et al., the adversary learns the statement that was proven, i.e., authenticated (or one-sided–authenticated) channels are sufficient in the realization of π.

$\mathcal{F}_{\text{gzk}}$ is designed to be used in a setting where adaptive corruption with erasures are allowed; and $\mathcal{F}_{\text{gzk}}$ is parameterized by a binary predicate $R : \big(x, (w_{\not\exists}, w_{\exists})\big) \mapsto \{0,1\}$, and a leakage function $\omega : (x, w_{\not\exists}) \mapsto \{0,1\}^*$.

Interfaces. $\mathcal{F}_{\text{gzk}}$ is a three-interface system:

- The network interface, connected to the ideal adversary/simulator.
- The $\mathcal{P}$-interface, connected to the ideal peer of the prover.
- The $\mathcal{Q}$-interface, connected to the ideal peer of the verifier.

State. The ideal functionality is stateful and maintains the following data structures:

- $\underline{Seen}$: a subset of $\{0,1\}^*$. Keeps track of which messages were accepted.
- $\underline{x}$: the statement that is to be proven. This can be used as the first argument to the binary predicate R.
- $\underline{w_{\not\exists}}$: the witnesses whose knowledge is proven. This and the witnesses whose existence is proven can be used as the second argument to the binary predicate R.

We model the single-session variant of $\mathcal{F}_{\text{gzk}}$, the session ID $sid = \underline{sid}$ is thus fixed.

Reacting to messages. $\mathcal{F}_{\text{gzk}}$ reacts to messages as follows.

1. Receive $\langle \text{Send}, sid, x, w_{\not\exists}, w_{\exists} \rangle$ on $\mathcal{P}$,
   such that $\{\text{“Send”}\} \cap \underline{Seen} = \emptyset$:

   Insert “Send” into $\underline{Seen}$.
   Store the instance and all witnesses quantified by $\not\exists$: $\underline{x} \leftarrow x$ and $\underline{w_{\not\exists}} \leftarrow w_{\not\exists}$.
   Send $\langle \text{Send}, sid, \omega(x, w_{\not\exists}) \rangle$ on network.

The ideal functionality $\mathcal{F}_{\text{gzk}}$, being *gullible*, does not check if the predicate holds, i.e., if $R\big(x, (w_{\triangleright}, w_{\exists})\big) = 1$. Usually in an $\mathcal{F}_{\text{gzk}}$-hybrid protocol, the environment is restricted to being *nice*, and the honest parties never prove false statements, so $\mathcal{F}_{\text{gzk}}$ should never see false statements.

2. Receive $\langle \text{Ready}, sid \rangle$ on $\mathcal{Q}$,
   such that $\{\text{“Ready”}\} \cap \underline{Seen} = \emptyset$:

   Insert “Ready” into $\underline{Seen}$.
   Send $\langle \text{Ready}, sid \rangle$ on network.

3. Receive $\langle \text{Lock}, sid \rangle$ on network,
   such that $\{\text{“Lock”}\} \cap \underline{Seen} = \emptyset$
   and $\{\text{“Send”}, \text{“Ready”}\} \subset \underline{Seen}$:

   Insert “Lock” into $\underline{Seen}$.
   Send $\langle \text{Lock}, sid, \underline{x} \rangle$ on network.

4. Receive $\langle \text{Done}, sid \rangle$ on network,
   such that $\{\text{“Done”}\} \cap \underline{Seen} = \emptyset$
   and $\{\text{“Lock”}\} \subset \underline{Seen}$:

   Insert “Done” into $\underline{Seen}$.
   Send $\langle \text{Done}, sid \rangle$ on $\mathcal{P}$.

5. Receive $\langle \text{Deliver}, sid \rangle$ on network,
   such that $\{\text{“Deliver”}\} \cap \underline{Seen} = \emptyset$
   and $\{\text{“Lock”}\} \subset \underline{Seen}$:

   Insert “Deliver” into $\underline{Seen}$.
   Send $\langle \text{Deliver}, sid, \underline{x} \rangle$ on $\mathcal{Q}$.

6. Receive $\langle \text{Corrupt:}\mathcal{R}, sid \rangle$ on network,
   where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$,
   such that $\{\text{“Corrupt:}\mathcal{R}\text{”}\} \cap \underline{Seen} = \emptyset$:

   Insert “Corrupt:$\mathcal{R}$” into $\underline{Seen}$.
   Send $\langle \text{Corrupt:}\mathcal{R}, sid \rangle$ on $\mathcal{R}$.

7. Receive $\langle \text{Reset}, sid, x, w_{\triangleright}, w_{\exists} \rangle$ on network,
   such that $\{\text{“Reset”}, \text{“Lock”}\} \cap \underline{Seen} = \emptyset$
   and $\{\text{“Corrupt:}\mathcal{P}\text{”}\} \subset \underline{Seen}$:

   Insert “Reset” into $\underline{Seen}$.
   Store the instance and all witnesses quantified by $\triangleright$: $\underline{x} \leftarrow x$ and $\underline{w_{\triangleright}} \leftarrow w_{\triangleright}$.
   Send $\langle \text{Reset}, sid \rangle$ on network.

8. Receive $\langle \text{Expose}, sid \rangle$ on network,
   such that $\{\text{“Expose”}, \text{“Lock”}\} \cap \underline{Seen} = \emptyset$
   and $\{\text{“Send”}, \text{“Corrupt:}\mathcal{P}\text{”}\} \subset \underline{Seen}$:

   Insert “Expose” into $\underline{Seen}$.
   Send $\langle \text{Expose}, sid, \underline{x}, \underline{w_{\not\ni}} \rangle$ on network.

A.4.1 GNUC Formalism

For completeness, we provide here the formal definition of $\mathcal{F}_{\text{gzk}}$ in the GNUC formalism. It is easy to see that the special composition theorem of Camenisch et al. still holds despite the translation to the GNUC framework.

$\mathcal{F}_{\text{gzk}}$ is parametrized by a binary predicate x and a leakage function ω.

- `send` : Accept $\langle \texttt{send}, x, w_k, w_e \rangle$ from $\mathcal{U}$. Store the instance and all witnesses quantified by $\not\ni$: $\bar{x} \leftarrow x$ and $\bar{w}_k \leftarrow w_k$. The ideal functionality $\mathcal{F}_{\text{gzk}}$, being *gullible*, does not check if the predicate holds. Send $\langle \texttt{send}, \omega(x, w_k) \rangle$ to $\mathcal{A}$.
- `ready` : Accept $\langle \texttt{ready} \rangle$ from $\mathcal{P}$. Send $\langle \texttt{ready} \rangle$ to $\mathcal{A}$.
- `lock` [`send` $\wedge$ `ready`] : Accept $\langle \texttt{lock} \rangle$ from $\mathcal{A}$. Send $\langle\rangle$ to $\mathcal{A}$.
- `done` [`lock`] : Accept $\langle \texttt{done} \rangle$ from $\mathcal{A}$. Send $\langle \texttt{done} \rangle$ to $\mathcal{U}$.
- `deliver` [`lock`] : Accept $\langle \texttt{deliver}, L \rangle$ from $\mathcal{A}$, where $L = \omega(\bar{x}, \bar{w}_k) \vee$ [`corrupt:Q`]. Send $\langle \texttt{deliver}, \bar{x} \rangle$ to $\mathcal{P}$.
- `corrupt:P` : Accept a special $\langle \texttt{corrupt} \rangle$ message from $\mathcal{U}$. Send $\langle \texttt{corrupt:P} \rangle$ to $\mathcal{A}$ together with an invitation for the message $\langle \texttt{expose} \rangle$.
- `corrupt:Q` : Accept a special $\langle \texttt{corrupt} \rangle$ message from $\mathcal{P}$. Send $\langle \texttt{corrupt:Q} \rangle$ to $\mathcal{A}$.
- `reset` [$\neg$`lock` $\wedge$ `corrupt:P`] : Accept $\langle \texttt{reset}, x, w_k, w_e \rangle$ from $\mathcal{A}$. The ideal functionality $\mathcal{F}_{\text{gzk}}$, being *gullible*, does not check if the predicate holds. Store the instance and all witnesses quantified by $\not\ni$: $\bar{x} \leftarrow x$ and $\bar{w}_k \leftarrow w_k$. Send $\langle\rangle$ to $\mathcal{A}$.
- `expose` [`send` $\wedge$ $\neg$`lock` $\wedge$ `corrupt:P`] : Accept $\langle \texttt{expose} \rangle$ from $\mathcal{A}$. Send $\langle \texttt{expose}, \bar{x}, \bar{w}_k \rangle$ to $\mathcal{A}$.

A.5 Zero-Knowledge Proofs of Existence for Two Verifiers $\mathcal{F}_{\text{gzk}}^{2\text{v}}$

Our functionality $\mathcal{F}_{\text{gzk}}^{2\text{v}}$ is very similar to $\mathcal{F}_{\text{gzk}}$. It is intended to be used when the prover needs to simultaneously prove the same statement to two different verifiers (and erase the same values in both proofs). An honest execution of $\mathcal{F}_{\text{gzk}}^{2\text{v}}$ is similar to running two sessions of $\mathcal{F}_{\text{gzk}}$ in parallel with the same statement and with two different verifiers, except that both sessions share a single Lock message. Of course, if the prover is corrupted, he can prove different statements to the two verifiers.

Interfaces. $\mathcal{F}_{\text{gzk}}^{2\text{v}}$ is a four-interface system:

- The network interface, connected to the ideal adversary/simulator.
- The $\mathcal{U}$-interface, connected to the ideal peer of the prover.
- The $\mathcal{P}$-interface, connected to the ideal peer of the first verifier.
- The $\mathcal{Q}$-interface, connected to the ideal peer of the second verifier.

State. The ideal functionality is stateful and maintains the following data structures:

- $\underline{\mathit{Seen}}$: a subset of $\{0,1\}^*$. Keeps track of which messages were accepted.
- $\underline{x}^{\mathcal{P}}, \underline{x}^{\mathcal{Q}}$: the statement that is to be proven. This can be used as the first argument to the binary predicate R.
- $\underline{w_{\triangleright}}^{\mathcal{P}}, \underline{w_{\triangleright}}^{\mathcal{Q}}$: the witnesses whose knowledge is proven. This and the witnesses whose existence is proven can be used as the second argument to the binary predicate R.

We model the single-session variant of $\mathcal{F}_{\text{gzk}}^{\text{2v}}$, the session ID $sid = \underline{sid}$ is thus fixed.

Reacting to messages. $\mathcal{F}_{\text{gzk}}^{\text{2v}}$ reacts to messages as follows.

1. Receive $\langle \text{Send:}\mathcal{R}, sid, x, w_{\triangleright}, w_{\exists} \rangle$ on $\mathcal{U}$,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$,
such that $\{\text{“Send:}\mathcal{R}\text{”}\} \cap \underline{\mathit{Seen}} = \emptyset$:

   Insert “Send:$\mathcal{R}$” into $\underline{\mathit{Seen}}$.
   Store the instance and all witnesses quantified by $\triangleright$: $\underline{x}^{\mathcal{R}} \leftarrow x$ and $\underline{w_{\triangleright}}^{\mathcal{R}} \leftarrow w_{\triangleright}$.
   Send $\langle \text{Send:}\mathcal{R}, sid, \omega(x, w_{\triangleright}) \rangle$ on network.

   The ideal functionality $\mathcal{F}_{\text{gzk}}^{\text{2v}}$, being *gullible*, does not check if the predicate holds, i.e., if $R\big(x, (w_{\triangleright}, w_{\exists})\big) = 1$. Usually in an $\mathcal{F}_{\text{gzk}}^{\text{2v}}$-hybrid protocol, the environment is restricted to being *nice*, and the honest parties never prove false statements, so $\mathcal{F}_{\text{gzk}}^{\text{2v}}$ should never see false statements.

2. Receive $\langle \text{Ready:}\mathcal{R}, sid \rangle$ on $\mathcal{R}$,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$,
such that $\{\text{“Ready:}\mathcal{R}\text{”}\} \cap \underline{\mathit{Seen}} = \emptyset$:

   Insert “Ready:$\mathcal{R}$” into $\underline{\mathit{Seen}}$.
   Send $\langle \text{Ready:}\mathcal{R}, sid \rangle$ on network.

3. Receive $\langle \text{Lock:}\mathcal{R}, sid \rangle$ on network,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\} \wedge \big(\big(\{\text{“Send:}\mathcal{P}\text{”}, \text{“Send:}\mathcal{Q}\text{”}, \text{“Ready:}\mathcal{P}\text{”}, \text{“Ready:}\mathcal{Q}\text{”}\} \subset \underline{\mathit{Seen}} \wedge \underline{x}^{\mathcal{P}} = \underline{x}^{\mathcal{Q}}\big) \vee (\text{“Corrupt:}\mathcal{U}\text{”} \in \underline{\mathit{Seen}})\big)$,
such that $\{\text{“Lock:}\mathcal{R}\text{”}\} \cap \underline{\mathit{Seen}} = \emptyset$,
and $\{\text{“Send:}\mathcal{R}\text{”}, \text{“Ready:}\mathcal{R}\text{”}\} \subset \underline{\mathit{Seen}}$:

   Insert “Lock:$\mathcal{R}$” into $\underline{\mathit{Seen}}$.
   Send $\langle \text{Lock:}\mathcal{R}, sid, \underline{x}^{\mathcal{R}} \rangle$ on network.

   When the user is honest, we make sure here that the two protocols are synchronized. This way, in the realization, the user can erase data in both protocols simultaneously.

4. Receive $\langle \text{Done:}\mathcal{R}, sid \rangle$ on network,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\} \wedge \big(\big(\{\text{“Lock:}\mathcal{P}\text{”}, \text{“Lock:}\mathcal{Q}\text{”}\} \subset \underline{\mathit{Seen}}\big) \vee (\text{“Corrupt:}\mathcal{U}\text{”} \in \underline{\mathit{Seen}})\big)$,
such that $\{\text{“Done:}\mathcal{R}\text{”}\} \cap \underline{\mathit{Seen}} = \emptyset$,
and $\{\text{“Lock:}\mathcal{R}\text{”}\} \subset \underline{\mathit{Seen}}$:

Insert "Done:$\mathcal{R}$" into $\underline{\mathit{Seen}}$.
Send $\langle$Done:$\mathcal{R}$, $\mathit{sid}\rangle$ on $\mathcal{U}$.

5. Receive $\langle$Deliver:$\mathcal{R}$, sid, $\underline{x}^{\mathcal{R}}$, $L\rangle$ on network,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\} \wedge L = \omega(\underline{x}^{\mathcal{R}}, \underline{w}_{\rightarrow}^{\mathcal{R}}) \vee$ "Corrupt:$\mathcal{R}$" $\in \underline{\mathit{Seen}} \wedge$
$\big(\big(\{$"Lock:$\mathcal{P}$", "Lock:$\mathcal{Q}$"$\} \subset \underline{\mathit{Seen}}\big) \vee \big($"Corrupt:$\mathcal{U}$" $\in \underline{\mathit{Seen}}\big)\big)$,
such that $\{$"Deliver:$\mathcal{R}$"$\} \cap \underline{\mathit{Seen}} = \emptyset$,
and $\{$"Lock:$\mathcal{R}$"$\} \subset \underline{\mathit{Seen}}$:

Insert "Deliver:$\mathcal{R}$" into $\underline{\mathit{Seen}}$.
Send $\langle$Deliver:$\mathcal{R}$, sid, $\underline{x}^{\mathcal{R}}\rangle$ on $\mathcal{R}$.

6. Receive $\langle$Corrupt:$\mathcal{T}$, $\mathit{sid}\rangle$ on network,
where $\mathcal{T} \in \{\mathcal{U}, \mathcal{P}, \mathcal{Q}\}$,
such that $\{$"Corrupt:$\mathcal{T}$"$\} \cap \underline{\mathit{Seen}} = \emptyset$:

Insert "Corrupt:$\mathcal{T}$" into $\underline{\mathit{Seen}}$.
Send $\langle$Corrupt:$\mathcal{T}$, $\mathit{sid}\rangle$ on $\mathcal{T}$.

7. Receive $\langle$Reset:$\mathcal{R}$, sid, x, $w_{\rightarrow}$, $w_{\exists}\rangle$ on network,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$,
such that $\{$"Reset:$\mathcal{R}$", "Lock"$\} \cap \underline{\mathit{Seen}} = \emptyset$,
and $\{$"Corrupt:$\mathcal{U}$"$\} \subset \underline{\mathit{Seen}}$:

Insert "Reset:$\mathcal{R}$" into $\underline{\mathit{Seen}}$.
Store the instance and all witnesses quantified by $\rightarrow$: $\underline{x}^{\mathcal{R}} \leftarrow x$ and $\underline{w}_{\rightarrow}^{\mathcal{R}} \leftarrow w_{\rightarrow}$.
Send $\langle$Reset:$\mathcal{R}$, $\mathit{sid}\rangle$ on network.

8. Receive $\langle$Expose:$\mathcal{R}$, $\mathit{sid}\rangle$ on network,
where $\mathcal{R} \in \{\mathcal{P}, \mathcal{Q}\}$,
such that $\{$"Expose:$\mathcal{R}$", "Lock"$\} \cap \underline{\mathit{Seen}} = \emptyset$,
and $\{$"Send", "Corrupt:$\mathcal{U}$"$\} \subset \underline{\mathit{Seen}}$:

Insert "Expose:$\mathcal{R}$" into $\underline{\mathit{Seen}}$.
Send $\langle$Expose:$\mathcal{R}$, sid, $\underline{x}^{\mathcal{R}}$, $\underline{w}_{\rightarrow}^{\mathcal{R}}\rangle$ on network.

Realization. $\mathcal{F}_{\text{gzk}}^{\text{2v}}$ is realized by running two independent instances of the π protocol by Camenisch, Krenn, and Shoup [CKS11]—one instance with each verifier. However, the prover waits until he got a reply from both verifiers before erasing the witnesses and sending out the last message in each proof instance.

Curriculum Vitae

Robert Richard Enderlein
Born on 2 July 1987 in [redacted].
Nationality: Swiss and German.

Education

[redacted]–2016 **Eidgenössische Technische Hochschule Zürich (ETH Zurich).**
Ph.D. candidate in the Computer Science Department.

[redacted] **École Polytechnique Fédérale de Lausanne (EPFL).**
Master of Science in Communication Systems.

[redacted]

[redacted]

[redacted]

Work experience

2016– [redacted]

[redacted]–2016 **IBM Research GmbH** in Zurich.
Pre-doctoral researcher in the Computer Science Department.

[redacted]

[redacted]

[redacted]

Refereed Publications

- J. Camenisch, R. R. Enderlein, G. Neven.
  Two-Server Password-Authenticated Secret Sharing UC-Secure Against Transient Corruptions.
  PKC 2015: 283–307.
- J. Camenisch, M. Dubovitskaya, R. R. Enderlein, A. Lehmann, G. Neven, C. Paquin, F.-S. Preiss.
  Concepts and languages for privacy-preserving attribute-based authentication.
  J. Information Security and Applications 19(1): 25–44 (2014).
- J. Camenisch, R. R. Enderlein, V. Shoup.
  Practical and Employable Protocols for UC-Secure Circuit Evaluation over $\mathbb{Z}_n$.
  ESORICS 2013: 19–37. **Best student paper.**
- J. Camenisch, M. Dubovitskaya, R. R. Enderlein, G. Neven.
  Oblivious Transfer with Hidden Access Control from Attribute-Based Encryption.
  SCN 2012: 559–579.
- S. Obermeier, R. Schierholz, H. Hadeli, R. Enderlein, A. Hristova, T. Locher.
  Secure Management of Certificates for Industrial Control Systems.
  Security and Management (SAM) 2012.

Manuscripts in Submission

- J. Camenisch, R. R. Enderlein, U. Maurer.
  Memory Erasability Amplification.
- J. Camenisch, R. R. Enderlein, S. Krenn, R. Küsters, D. Rausch, O. Ciobotaru.
  Universal Composability—Conventions for Complete and Unambiguous Protocol Specifications.
- J. Camenisch, R. R. Enderlein, S. Krenn, R. Küsters, D. Rausch.
  Universal Composition with Responsive Environments.
  Cryptology ePrint Archive 2016/034.

Book Chapters

- P. Bichsel, J. Camenisch, M. Dubovitskaya, R. R. Enderlein, S. Krenn, I. Krontiris, A. Lehmann, G. Neven, C. Paquin, F.-S. Preiss, K. Rannenberg, A. Sabouri.
  An Architecture for Privacy-ABCs.
  Attribute-based Credentials for Trust: 11–78.
- P. Bichsel, J. Camenisch, M. Dubovitskaya, R. R. Enderlein, S. Krenn, A. Lehmann, G. Neven, F.-S. Preiss.
  Cryptographic Protocols Underlying Privacy-ABCs.
  Attribute-based Credentials for Trust: 79–108.