View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Boston University Institutional Repository (OpenBU)

Boston University

OpenBU | **http://open.bu.edu**

Theses & Dissertations | Boston University Theses & Dissertations

2020

Studies in authentication

https://hdl.handle.net/2144/41697
Boston University

BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

STUDIES IN AUTHENTICATION

by

SOPHIA YAKOUBOV

B.S., Massachusetts Institute of Technology, 2011
M.S., Boston University, 2015

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2020

© 2020 by
SOPHIA YAKOUBOV
All rights reserved

Approved by

First Reader

Leonid Reyzin, PhD
Professor of Computer Science

Second Reader

Ran Canetti, PhD
Professor of Computer Science

Third Reader

Adam Smith, PhD
Professor of Computer Science

Fourth Reader

Claudio Orlandi, PhD
Associate Professor of Computer Science
Aarhus University

Acknowledgments

First and foremost, I would like to thank my advisor Leo Reyzin for being the smartest, nicest, most supportive advisor and mentor I could have hoped for. You gave me the freedom to pursue whatever directions and opportunities intrigued me — even when those opportunities took me all the way to Denmark a year before my graduation! — and continued to guide me through it all, reminding me of the big picture, both in terms of research and life in general.

Next, a huge "thank you!" to some of my other mentors: Paul Goldenberg, who was the first to show me what it was like to work on a math problem ("Raw Recruits", when I was 10) to which neither of us knew the answer — so much fun! Richard Stanley, who introduced me to academic research (which ended up being my passion, though I transitioned from combinatorics to cryptography), and who continued to patiently work with me even after I finished my undergraduate studies. Rob Cunningham, who was the best first boss anyone could have, and who supported me and encouraged me to grow at a time when I was less sure of myself. Ivan Damgård and Claudio Orlandi, who were the reason I came to Denmark, and who are a big part of why I love it there so much.

To my committee members — Leo Reyzin, Ran Canetti, Adam Smith and Claudio Orlandi — thank you for being there for me at these final stages of my time in graduate school!

Big thanks to all of my coauthors: Foteini Baldimtsi, Jan Camenisch, Ran Canetti, Rob Cunningham, Ivan Dåmgard, Maria Dubovitskaya, Pierre-Alain Dupont, Ben Fuller, Vijay Gadepally, Helene Haagh, Ariel Hamlin, Jonathan Herzog, Julia Hesse, Nick Hwang, Anna Lysyanskaya, Rebekah Mercer, Anca Nitulescu, Claudio Orlandi, David Pointcheval, Jill Poland, Ben Price, Michael Reschly, Leo Reyzin, Kai Samelin, Nabil Schear, Emily Shen, Adam Smith, Mayank Varia and Arkady Yerukhimovich.

No researcher is an island, and I would have gotten nowhere without all of you!

I would also like to thank all of my other friends and mentors, particularly those at Lincoln Laboratory, Boston University and Aarhus University. I did not get the chance to collaborate with all of you (yet!), but having your company, advice and support was an integral part of what made my grad school experience so great.

Finally, I would like to thank my mother, Nina Dubinsky, and my ex-partner, Ilya Lifshits, for being there for me throughout my time in graduate school. I could never have done it without you both.

STUDIES IN AUTHENTICATION

SOPHIA YAKOUBOV

Boston University, Graduate School of Arts and Sciences, 2020

Major Professor: Leonid Reyzin, PhD
Professor of Computer Science

ABSTRACT

This thesis presents advances in several areas of authentication.

First, we consider cryptographic accumulators, which are compact digital objects representing arbitrarily large sets. They support efficient proofs of membership (or, alternatively, of non-membership). We give the first definition of cryptographic accumulators in the UC framework, and construct two new accumulators: one uniquely suited for use in a revokable anonymous credential scheme, and one uniquely suited for use in a distributed system such as a blockchain-based PKI.

Next, we consider multi-designated verifier signatures (MDVS). An MDVS is a special kind of signature that can only be verified by parties explicitly specified by the signer; more than that, even if those designated verifiers wanted to prove to an external party (e.g. an adversary) that a certain message was signed by the signer, they should be unable to do so. This is crucial in contexts where off-the-record communication is desirable; the sender may not want to be provably linked to a possibly sensitive message, but still want the intended recipients to be able to verify the authenticity of the message. Existing literature defines and builds limited notions of MDVS, where the off-the-record property only holds when it is conceivable that all verifiers collude. We strengthen this property to support any subset of colluding verifiers, and give two constructions of our stronger notion of MDVS: one from functional

encryption, and one from standard primitives (but with a slightly larger signature size).

Finally, we consider fuzzy password authenticated key exchange (Fuzzy PAKE). PAKEs are protocols which enable two parties holding the same password (that is, the same potentially low-entropy, non-uniform string) to agree on a (high-entropy, uniform) secret key in a way that resists man-in-the-middle attacks and offline dictionary attacks on the password. We define Fuzzy PAKE, a special kind of PAKE where the passwords used for authentication may contain some errors. We provide the first efficient and general solutions to this problem that enable, for example, key agreement based on commonly used biometrics such as iris scans.

Contents

1 Introduction **1**

1.1 Advances in Accumulators 1

1.2 Advances in Designated Verifier Signatures 4

1.3 Advances in PAKE 6

2 Universally Composable Accumulators **9**

2.1 Introduction 9

2.1.1 Accumulator Applications 14

2.2 Revisiting Classical Accumulator Definitions 16

2.2.1 Notation and Algorithms 17

2.2.2 Security Definitions 21

2.3 Ideal Functionality for Accumulators 26

2.3.1 Modeling Decentralized Management 32

2.3.2 Modeling Non-Adaptive Soundness 32

2.3.3 Adding Privacy Properties 32

2.3.4 Discussion: Incorrect Accumulator and Witness Values 34

2.4 Equivalence Argument 35

2.5 Appendix A: Universally Composable Signatures 38

2.6 Appendix B: Universally Composable Zero-Knowledge 39

2.7 Appendix C: The RSA Accumulator 39

3 Accumulators with Applications to Anonymity-Preserving Revocation **43**

3.1 Introduction 43
3.1.1 Outline 44
3.2 Modular Accumulator Constructions 44
3.2.1 Leveraging Accumulators with Different Functionalities 44
3.2.2 Leveraging Less Secure Accumulators 49
3.3 Braavos: A Communication-Optimal Adaptively Sound Dynamic Accumulator 50
3.3.1 CLRSAB: A Communication-Optimal Non-Adaptively Sound Dynamic Accumulator 55
3.3.2 Adding Zero Knowledge to Braavos 60
3.4 BraavosB: Another Communication-Optimal Adaptively Sound Dynamic Accumulator 62
3.4.1 Range-RSA: A Dynamic Negative Accumulator 64
3.4.2 Range-RSA Accumulator Algorithms 67
3.4.3 BraavosB Soundness 67
3.4.4 Adding Zero Knowledge to BraavosB 67
3.5 Comparison with Other Constructions 72
3.6 Appendix A: Lower Bound on Total Communication in Negative Accumulators 75

4 Efficient Asynchronous Accumulators for Distributed PKI 76

4.1 Introduction 76
4.2 Definitions 79
4.3 Building Distributed Accumulators 80
4.4 Construction 80
4.5 Infrequent Membership Witness Updates 83
4.6 Limited Dynamism 84

4.7 Appendix 1: Algorithms 85
4.7.1 Accumulator Algorithms 85
4.7.2 Batch Witness Updates 88

5 Stronger Notions and Constructions for Multi-Designated Verifier Signatures 91

5.1 Introduction 91
5.1.1 A Motivating Example for MDVS 92
5.1.2 Flavors of Multi-Designated Verifier Signatures 96
5.1.3 Our Contributions 98
5.2 Multi-Designated Verifier Signatures 104
5.3 Standard Primitive-Based MDVS Constructions 111
5.3.1 New Primitive: Provably Simulatable Designated-Verifier Signatures (PSDVS) 112
5.3.2 Standard Primitive-Based MDVS Construction 117
5.3.3 Standard Primitive-Based PSDVS Construction 121
5.3.4 DDH and Paillier-Based PSDVS Construction 127
5.3.5 Sketch of a PSDVS Scheme Based on Prime Order Groups . . 139
5.4 FE-based Construction 140
5.4.1 Functional Encryption 142
5.4.2 The MDVS Construction 145
5.5 Appendix A: Instantiation of Non-Interactive ZK Proofs 148

6 Fuzzy Password Authenticated Key Exchange 153

6.1 Introduction 153
6.1.1 Our Contributions 155
6.2 Security Model 160
6.3 General Construction Using Garbled Circuits 165

6.3.1 Building Blocks 167
6.3.2 Construction 172
6.3.3 An Efficient Circuit f for Hamming Distance 179
6.4 Specialized Construction For Hamming Distance 181
6.4.1 Building Blocks 181
6.4.2 Construction 188
6.4.3 Security of $\mathsf{fPAKE}_{\mathsf{RSS}}$ 190
6.4.4 Further Discussion: Removing Modeling Assumptions 193
6.5 Comparison of fPAKE Protocols 193
6.6 Appendix A: Ideal UC Functionalities 196
6.7 Appendix B: Garbled Output Randomness: A New Yao's Garbled Circuit Definition 200
6.8 Appendix C: Proof of Theorem 12 202
6.9 Appendix D: Proof that $s\mathcal{F}^{P}_{\mathsf{RFE}}$ is Enough to Realize $\mathcal{F}^{P}_{\mathsf{fPAKE}}$ 212
6.10 Appendix E: A Concrete OT 214
6.11 Appendix F: Proof of Theorem 13 215
6.12 Appendix G: Proof of Theorem 14 230
6.13 Appendix H: A Natural (But Failed) Approach to fPAKE 238

References **240**

7 Curriculum Vitae **252**

List of Figures

0·1 Accumulator Notation xv

0·2 Signature Notation xv

0·3 Signature Notation xv

2·1 Accumulator Algorithms 21

2·2 The Collision-Freeness Game for Accumulators 25

2·3 Ideal Functionality $\mathcal{F}_{\mathsf{ACC}}$: Accumulator Manager / Witness Holder Interfaces 30

2·4 Ideal Functionality $\mathcal{F}_{\mathsf{ACC}}$: Third Parties Interfaces 31

2·5 Ideal Functionality for Digital Signatures 39

2·6 Algorithmic Ideal Functionality for Digital Signatures 40

2·7 Ideal Functionality for Zero Knowledge 40

2·8 RSA Accumulator Manager Algorithms 41

2·9 RSA Witness Holder Algorithms 42

2·10 RSA Verifier Algorithms 42

3·1 Modular Accumulator Derivations: Functionality 46

3·2 Modular Accumulator Derivations: Security 49

3·3 Braavos′ Algorithms 52

3·4 CLRSAB Algorithms 57

3·5 Reduction From the Non-Adaptive Soundness of CLRSAB to the Strong RSA Assumption 59

3·6 Construction B Accumulator Manager Algorithms 64

3·7 Construction B Witness Holder Algorithms 64
3·8 Construction B Third Party Algorithms 65
3·9 Range-RSA Accumulator Manager Algorithms 68
3·10 Range-RSA Witness Holder Algorithms 69
3·11 Range-RSA Third Party Algorithms 69
3·12 Accumulator Construction Comparison 73

4·1 The Distributed Accumulator 81

5·1 MDVS Constructions and Their Properties 98
5·2 Our MDVS Constructions and Building Blocks 99

6·1 Ideal Functionality fPAKE 163
6·2 A Modified `TestPwd` Interface to Allow for Different Leakage 164
6·3 Dual Execution Protocols 170
6·4 Ideal Functionality $\mathcal{F}^{P}_{\mathsf{RFE}}$ for Randomized Fuzzy Equality 174
6·5 A Protocol for Π_{RFE} Based on Yao's Garbled Circuits 175
6·6 Functionality $s\mathcal{F}^{P}_{\mathsf{RFE}}$ 178
6·7 The f circuit 180
6·8 Functionality $\mathcal{F}_{\mathsf{iPAKE}}$ 186
6·9 Protocol EKE2 187
6·10 Protocol $\mathsf{fPAKE}_{\mathsf{RSS}}$ 189
6·11 FPAKE Scheme Comparison 194
6·12 FPAKE Scheme Efficiency Comparison 195
6·13 Functionality $\mathcal{F}_{\mathsf{CRS}}$ 196
6·14 Functionality $\mathcal{F}_{\mathsf{RO}}$ 197
6·15 Functionality $\mathcal{F}_{\mathsf{IC}}$ 198
6·16 Functionality $\mathcal{F}_{\mathsf{OT}}$ 198

6·17 Functionality $\mathcal{F}_{\mathsf{pwKE}}$ 199
6·18 Garbled Output Randomness Game 200
6·19 Summary of Proof of Theorem 12 203
6·20 From Game $\mathbf{G}_0$ to Game $\mathbf{G}_1$ 204
6·21 Simulator $\mathcal{S}_{\mathsf{RFE}}$ for Π_{RFE} 212
6·22 Protocol $\mathsf{fPAKE}_{\mathsf{YGC}}$ realizing $\mathcal{F}^P_{\mathsf{fPAKE}}$ in the $s\mathcal{F}^P_{\mathsf{RFE}}$-hybrid model. 212
6·23 Simulator $\mathcal{S}_{\mathsf{fPAKE}}$ for $\mathsf{fPAKE}_{\mathsf{YGC}}$. 213
6·24 Output Tables for $\mathcal{F}^P_{\mathsf{fPAKE}}$ and $s\mathcal{F}^P_{\mathsf{RFE}}$ 214
6·25 A Concrete OT 215
6·26 Functionality $\mathcal{F}_{\ell\text{-}\mathsf{iPAKE}}$ 216
6·27 A UC Execution of EKE2 216
6·28 From Game $\mathbf{G}_0$ (left) to Game $\mathbf{G}_1$ 217
6·29 The Simulator $\mathcal{S}$ for the EKE2 Protocol 229
6·30 A UC Execution of $\mathsf{fPAKE}_{\mathsf{RSS}}$ 231
6·31 The Simulator $\mathcal{S}$ for $\mathsf{fPAKE}_{\mathsf{RSS}}$ 238
6·32 A First Natural Construction of FPAKE 239

List of Symbols

λ:	The security parameter.
D:	The domain of the accumulator (the set of elements that the accumulator can accumulate). Often, D includes all elements (e.g., $\{0,1\}^*$). Sometimes, D is more limited (e.g., primes of a certain size).
sk:	The accumulator manager's secret key or trapdoor. (The corresponding public key, if one exists, is not modeled here as it can be considered to be a part of the accumulator itself.)
t:	A discrete time / operation counter.
a_t:	The accumulator at time t.
m_t:	Any auxiliary values which might be necessary for the maintenance of the accumulator. These are typically held by the accumulator manager. Note that while the accumulator itself should be constant (or at least sub-linear) in size, m may be larger.
S_t:	The set of elements in the accumulated set at time t. Note that S_0 can be instantiated to be different, based on the initial sets supported by the accumulator in question. Most accumulators assume $S_0 = \emptyset$.
x, y:	Elements which might be added to or removed from the accumulator.
w_t^x:	A witness that element x is (or is not) in the accumulated set at time t.
$\mathsf{stts} \in \{\mathsf{in}, \mathsf{out}\}$:	A flag indicating of whether a given element is in the accumulated set or not.
$\mathsf{Op} \in \{\mathsf{Add}, \mathsf{Del}\}$:	A flag indicating of whether a given element is being added or deleted.
upmsg_t:	A broadcast message sent (by the accumulator manager, if one exists) at time t to all witness holders immediately after the accumulator has been updated. This message is meant to enable all witness holders to update the witnesses they hold for consistency with the new accumulator. It will often contain the new accumulator a_t, and the nature of the update itself (e.g., "x has been added and witness w_t^x has been produced"). It may also contain other information.
v:	A witness that the accumulator a_0 was generated correctly. (Only present in strong accumulators.)
v_t:	A witness that the accumulator a_t was updated correctly. (Only present in strong accumulators.)

Figure 0·1: Accumulator Notation (from Baldimtsi *et al.* (Baldimtsi et al., 2017))

pp:	The public parameters
msk:	The master secret key
spk:	The signer's public key
ssk:	The signer's secret key
vpk:	A verifier's public key
vsk:	A verifier's secret key
m:	A message
σ:	A signature

Figure 0·2: Signature Notation

pw:	The pass-string held by a party
sk:	The secret key derived by a party

Figure 0·3: Signature Notation

Chapter 1

Introduction

The field of cryptography addresses two broad areas — *privacy* (keeping data secret from unauthorized parties) and *authenticity* (ensuring that authorized parties can verify the validity or provenance of data). This thesis focuses on authenticity. There are many very different cryptographic primitives intended to provide authenticity in different contexts. The most famous of these are message authentication codes (MACs) and digital signatures, but there are many others, such as Merkle hash trees and key exchange. In this thesis, we study three authentication primitives: cryptographic accumulators, multi-designated verifier signatures (MDVS), and fuzzy password authenticated key exchange (Fuzzy PAKE). (Despite this apparent diversity of primitives, many authentication tools can, in fact, be considered in a unified framework: that of cryptographic accumulators.)

1.1 Advances in Accumulators

Chapters 2, 3 and 4 focus on cryptographic accumulators. Chapter 2 gives a new definition of accumulators in the universal composability framework. Chapter 3 gives a few accumulator constructions that are uniquely suited for use in revokable anonymous credential systems, and Chapter 4 gives an accumulator construction that is particularly well suited to use in distributed settings, e.g. on a blockcchain.

Accumulators, first introduced by Benaloh and de Mare (Benaloh and de Mare, 1994), are compact representations of arbitrarily large sets. Despite being small —

ideally constant-size relative to the size of the set they represent — they enable verification of statements about the set. Given a *membership witness* for some object x together with the accumulator, anyone can verify that x is in the accumulated set. If the accumulator is a *universal* accumulator (Li et al., 2007a), it also supports *non-membership* witnesses that can be used to verify that elements are not in the accumulated set. Typically, an accumulator is owned by an entity called an *accumulator manager* who can add elements to (and, if the accumulator is *dynamic* (Camenisch and Lysyanskaya, 2002), remove elements from) the set. If the accumulator is *strong* (Camacho et al., 2008), even a corrupt accumulator manager cannot forge a proof of (non-)membership.

In Chapter 2 (which is based on joint work with Foteini Baldimtsi and Ran Canetti (Baldimtsi et al.,)), we give the first definition of cryptographic accumulators in the UC (universal composability) framework. Accumulators are almost exclusively used as building blocks in real-world complex systems (as opposed to on their own). Having rigorous security analysis for such systems is crucial for their adoption and safe use in the real world, but it can turn out to be extremely challenging given their complexity. The UC framework is a paradigm designed to enable more modular security analyses and proofs, guaranteeing that any composition of primitives remains secure as long as the individual primitives have been proven secure in the UC framework. Our UC definition of accumulators covers different accumulator types (universal, dynamic, strong) concisely in a single functionality, and captures the two basic security properties of accumulators: correctness and soundness. Additionally, we prove the equivalence of our UC definition to standard accumulator definitions. This implies that existing popular accumulator schemes, such as the RSA accumulator, already meet our UC definition, and that the security proofs of existing systems that leverage such accumulators can be significantly simplified.

Chapters 3 and 4 give new accumulator constructions. Chapter 3 (which is based on joint work with Foteini Baldimtsi, Jan Camenisch, Maria Dubovitskaya, Anna Lysyanskaya, Leonid Reyzin and Kai Samelin (Baldimtsi et al., 2017)) focuses on building new accumulators from existing ones in a modular way. In this chapter, we show how accumulators with different functionalities and notions of security can be composed to enhance this functionality and security. This leads to the $\mathsf{Braavos}$, $\mathsf{Braavos}'$ and $\mathsf{BraavosB}$ accumulators. $\mathsf{Braavos}$ and $\mathsf{Braavos}'$ are ideally suited for use in the context of revokable anonymous credentials. They use signatures to bind random nonces to elements; the nonces are then accumulated in an RSA-based accumulator which is designed not to require update messages when additions take place, at the cost of only supporting the accumulation of random elements. $\mathsf{Braavos}$, $\mathsf{Braavos}'$ and $\mathsf{BraavosB}$ all have the advantage that they do not require that witnesses be updated when a new element is added. $\mathsf{Braavos}$ and $\mathsf{Braavos}'$ additionally have the properties that update messages do not reveal too much information about the updates made to the accumulated set, and that they support efficient zero knowledge proofs of member knowledge.

Chapter 4 (which is based on joint work with Leonid Reyzin (Reyzin and Yakoubov, 2016)) describes a different way to compose accumulators, which is reminiscent of the use of a binary counter. This leads to the $\mathsf{DistAccs}$ accumulator, which consists of a forrest of Merkle trees, one of each depth. An element is added in much the same way as 1 would be added to a binary number; we first try to place it in the depth-1 Merkle tree space. If that is occupied, we hash the new element with the element already there, and try to place that in the depth-2 tree space, etc. This accumulator is designed to minimize the necessary witness update frequency; an element's witness is its authenticating path, and it only needs to be updated when its tree is merged (which happens at most a logarithmic number of times). This makes $\mathsf{DistAccs}$ ideally suited

for use in a distributed setting, where different — possibly computationally weak — parties are responsible for the maintenance of different witnesses. One such setting is maintaining a PKI on a blockchain (Yakoubov et al., 2014). Note that, though the DistAccs accumulator does require witnesses to be updated more frequently than the Braavos, Braavos′ and BraavosB accumulators do, it is *strong* (that is, it does not require trust in the accumulator manager), which is crucial in distributed settings.

1.2 Advances in Designated Verifier Signatures

Chapter 5 (which is based on joint work with Ivan Damgård, Helene Haagh, Rebekah Mercer, Anca Nițulescu and Claudio Orlandi (Damgård et al., 2019)) focuses on Multi Designated Verifier Signatures (MDVS). An MDVS is a special kind of signature that can only be verified by parties explicitly specified by the signer; more than that, even if those designated verifiers wanted to prove to an external party (e.g. an adversary) that a certain message was signed by the signer, they should be unable to do so. This is crucial in contexts where off-the-record communication is desirable; the sender may not want to be provably linked to a possibly sensitive message, but still want the intended recipients to be able to verify the authenticity of the message.

We give stronger definitions and constructions of MDVS. Existing literature defines and builds limited notions of MDVS, where the off-the-record property only holds when all verifiers could conceivably collude. We strengthen this property to support any subset of colluding verifiers. We give two constructions of our stronger notion of MDVS: one from functional encryption, and one from standard primitives such as pseudorandom functions, pseudorandom generators, key agreement and NIZKs.

In the construction based on functional encryption, the signer produces a regular digital signature, and encrypts it using the functional encryption public encryption key. Each verifier has a functional decryption key that allows them to learn whether a

given ciphertext contains a signature that verifies either under their own verification key, or under the signer's. No verifier can convince an external party that the signature is the signer's, even if they hand over their secret verification key, since the ciphertext could just as easily contain a signature produced by the verifier and not the signer. In order to make sure that even colluding verifiers cannot convince an external party to accept the signature as the signer's, the ciphertext should encrypt a vector of signatures instead of just one; then, any third party will remain unsure whether that vector has only a valid signature from the signer, or one valid signature from each of the colluding designated verifiers.

The upside of this construction is that the signature size is equal to the upper bound on the size of conceivable groups of colluding verifiers, which may be smaller than the number of such verifiers. However, the downside is two-fold: it requires trusted setup, and it uses functional encryption, which requires strong computational assumptions. Our second MDVS construction does not require trusted setup, and uses only standard primitives. Informally, in this construction the signer produces a separate designated verifier signature for each verifier, and proves the consistency of this vector of signatures: that either they are all real signatures, or that they are all fake. (The proof is necessary to achieve *consistency*, which is the property that even a malicious signer should be unable to sow confusion by producing a signature that verifies for some — but not all — of the designated verifiers.) In order to support such proofs, we introduce a new primitive which we call a *provably simulatable* designated verifier signature. We show both generic and concretely efficient instantiations of this new primitive.

Designated Verifier Signatures as Accumulators Signatures fall very naturally into the accumulator framework; a public key can be considered as an accumulator value, and a signature can be a membership witness. Thinking of designated verifi-

cation as an extension of that makes perfect sense. Instead of having accumulator witnesses be publicly verifiable, we could consider a designated verifier accumulator where each witness is only verifiable by certain parties as specified by the accumulator manager.

1.3 Advances in PAKE

Chapter 6 (which is based on joint work with Pierre-Alain Dupont, Julia Hesse, David Pointcheval and Leonid Reyzin (Dupont et al., 2018)) focuses on Fuzzy Password Authenticated Key Exchange (Fuzzy PAKE). PAKEs are protocols which enable two parties holding the same password (that is, the same potentially low-entropy, non-uniform string) to agree on a (high-entropy, uniform) secret key in a way that resists man-in-the-middle attacks and offline dictionary attacks on the password.

We introduce and give a UC definition of Fuzzy PAKE, which is a special kind of PAKE where the passwords used for authentication may contain some errors. Verification should still succeed, as long as the passwords are within a certain distance of one another (for some notion of distance). This is particularly useful for key agreement based on commonly used biometrics such as iris scans, or on environmental readings.

We give two constructions of Fuzzy PAKE. The first is based on Yao's garbled circuits (Yao, 1986; Bellare et al., 2012) and oblivious transfer (see Chou and Orlandi (Chou and Orlandi, 2015) and references therein). The use of these techniques is standard in two-party computation. However, by themselves they give protocols secure only against honest-but-curious adversaries. In order to prevent malicious behavior of the players, one usually applies the cut-and-choose technique (Lindell and Pinkas, 2011), which is quite costly: to achieve an error probability of $2^{-\lambda}$, the number of circuits that need to be garbled increases by a factor of λ, and the number

of oblivious transfers that need to be performed increases by a factor of $\lambda/2$. We show that for our special case, to achieve malicious security, it suffices to repeat the honest-but-curious protocol twice (once in each direction), incurring only a factor of 2 overhead over the semi-honest case.

Our second construction is for the Hamming case: the two n-character passwords have low Hamming distance if not too many characters of one party's password are different from the corresponding characters of the other's password. The two parties execute a PAKE protocol for each position in the string, obtaining n values each that agree or disagree depending on whether the characters of the password agree or disagree in the corresponding positions. It is important that at this stage, agreement or disagreement at individual positions remains unknown to everyone; we therefore make use of a special variant of PAKE which we call *implicit-only PAKE* This first step upgrades Hamming distance over a potentially small alphabet to Hamming distance over an exponentially large alphabet. We then secret-share the ultimate output key into n shares using a robust secret sharing scheme, and encrypt each share using the output of the corresponding PAKE protocol.

The second construction is more efficient than the first in the number of rounds, communication, and computation. However, it works only for Hamming distance. Moreover, it has an intrinsic gap between functionality and security: if the honest parties need to be within distance δ to agree, then the adversary may break security by guessing a secret within distance 2δ.

Fuzzy PAKE as an Acccumulator PAKE does not naturally fall into the language of accumulators, and fitting it into this language may be more of a stretch. We do not consider PAKE as accumulators in Chapter 6 (or MDVS as an accumulator in Chapter 5), and it might seem that trying to view too many apparently different primitives through the same accumulator lens would lead to unnecessary

definitional complexity. While this may be true, the definitional complexity is offset by the value of seeing the space of possible authentication primitives and all its different axes in a very clear way. MDVS, viewed outside the context of accumulators, does not automatically provoke the question of whether it can be extended to support non-membership witnesses or deletions; inside the accumulator language, those are natural questions, and ones that I intend to explore in future work. Similarly, considering PAKE in the context of accumulators naturally leads to the question of whether popular accumulators can be extended to support verification in a way that is resilient to offline attacks.

Chapter 2

Universally Composable Accumulators

The contents of this section are a collaboration with Foteini Baldimtsi and Ran Canetti (Baldimtsi et al.,). The last section of the original paper is omitted.

2.1 Introduction

Accumulators, first introduced by Benaloh and de Mare (Benaloh and de Mare, 1994), are compact representations of arbitrarily large sets. Despite being small — ideally constant-size relative to the size of the set they represent! — they enable verification of statements about the set. Given a *membership witness* for some object x together with the accumulator, anyone can verify that x is in the accumulated set. If the accumulator is a *universal* accumulator (Li et al., 2007a), it also supports *non-membership* witnesses that can be used to verify that elements are not in the accumulated set. Typically, an accumulator is owned by an entity called an *accumulator manager* who can add elements to (and, if the accumulator is *dynamic* (Camenisch and Lysyanskaya, 2002), remove elements from) the set. If the accumulator is *strong* (Camacho et al., 2008), even a corrupt accumulator manager cannot forge a proof of (non-)membership.

Many crucial primitives are actually special cases of accumulators. For instance, digital signatures are accumulator schemes, where the signature verification key is the accumulator representing the set of signed messages, and the signatures are membership witnesses. The owner of the signing key is the accumulator manager, and she

can add elements to the set by signing them. Of course, she cannot *un*-sign elements (without publishing a revocation list, which is not constant in size), and she cannot produce a proof that a given element has not been signed, so this accumulator is neither dynamic nor universal. She can also always prove the membership of arbitrary elements, so this accumulator is not strong.

Another example of an accumulator is a Merkle hash tree. The tree root is the accumulator representing the set of leaf nodes, and the authenticating paths through the tree are membership witnesses. This accumulator supports both element addition and deletion, but when either of those events occur, all existing witnesses must be updated, requiring total work that is linear in the number of member elements. In many situations, this is prohibitively inefficient. The Merkle hash tree accumulator is strong, because all additions and deletions are publicly verifiable (by means of re-execution). Though the intuitive Merkle hash tree accumulator does not support proofs of non-membership, it can be modified to be universal (Camacho et al., 2008).

One construction of a universal, dynamic (but not strong) accumulator with efficient update algorithms is the RSA accumulator. It is the original accumulator introduced by Benaloh and de Mare (Benaloh and de Mare, 1994), augmented with dynamism by Camenisch and Lysyanskaya (Camenisch and Lysyanskaya, 2002), and with universality by Li, Li and Xue (Li et al., 2007a). It is one of the most popular accumulator constructions because of its compactness and efficiency.

Although accumulators are frequently analyzed as stand-alone primitives, they are *almost exclusively used as building blocks* in real-world complex systems, including anonymous credentials (Camenisch and Lysyanskaya, 2002; Nguyen, 2005; Camenisch et al., 2009; Baldimtsi et al., 2017), group signatures (Camenisch and Lysyanskaya, 2002) and, more recently, anonymous cryptocurrencies (Miers et al., 2013). Having rigorous security analysis for such systems is crucial for their adoption and safe use in

the real world, but it can turn out to be extremely challenging given their complexity. When a system consists of multiple building blocks, even if each one of them is proven secure independently, the security analysis of the whole needs to be done from scratch.

Universal Composability

Universally Composable (UC) security (Canetti, 2001), addresses this problem. Any protocol that has been shown to be UC-secure will maintain its security properties even when it is used concurrently with other arbitrary protocols as part of a larger system. This allows one to formally argue about the security of a complex scheme in a much simpler and cleaner way, as long as all the protocols used within it have already been proven to be UC-secure.

Showing that a protocol is UC-secure consists of two steps. First, we write out a set of instructions called the *ideal functionality*, which define how we would instantiate the primitive if we had an incorruptible third party to delegate its operation to. Second, we show that any attack an adversary carries out against the protocol, it can also carry out against the ideal functionality. This is done by arguing that for any efficient adversary and environment (which sets all parties' inputs, receives all parties' outputs and additionally receives information from the adversary), there exists a simulator such that the environment cannot tell the difference between interacting with the protocol and adversary, and interacting with the ideal functionality and simulator. This proves that any time it suffices to use our ideal functionality within a larger system, we can replace it with our protocol and the system will remain secure.

The modularity and the strong security guarantees provided by UC suggest that protocols should always be designed and proven secure in the UC framework. However, this is only the case for a small fraction of proposed cryptographic schemes. One roadblock to using the UC framework is that not all commonly used sub-protocols have UC definitions and proofs. Some such sub-protocols have already been defined

and analyzed in the UC framework (e.g. digital signatures (Canetti, 2001; Canetti, 2004), zero-knowledge proofs (Camenisch et al., 2011), etc.), but others have not. Cryptographic accumulators are one example of a very common primitive that has never been considered in the context of UC security.

Our Results

In this work, we make the following contributions:

1. We provide the first UC definition (ideal functionality) for cryptographic accumulators. There are many functionality flavors of accumulators: accumulators might support only additions, only deletions or both, and they might support proofs of membership, proofs of non-membership, or both. Our UC definition covers all of these possibilities in a modular way.
2. We then prove the equivalence of our UC definition to standard accumulator security definitions. This implies that existing secure accumulator constructions — such as the RSA accumulator (Benaloh and de Mare, 1994; Camenisch and Lysyanskaya, 2002; Li et al., 2007a) — are UC secure.
3. Finally, we discuss how our UC definition simplifies the proofs of security for schemes that use accumulators as a building block. First, we build an accumulator out of two weaker accumulators (as in (Baldimtsi et al., 2017), but with stronger privacy properties), and give a simple UC proof of security for that composite accumulator, which we call Braavos′. Then, we consider how UC simplifies proofs of security in more complex systems such as anonymous credentials.

Note that when defining a new ideal functionality, there are two possible scenarios: either existing constructions can be proven to securely realize the new functionality (as with digital signatures (Canetti, 2004)), or new constructions must be developed (as with commitment schemes (Canetti and Fischlin, 2001)). Our second contribu-

tion shows that our accumulator functionality is in the first scenario; popular, existing accumulator constructions already satisfy it. This greatly simplifies the security analysis of existing and future systems that use cryptographic accumulators as a building block.

Informally, two classical properties are considered for cryptographic accumulators. The first is correctness: for every element inside (or outside, for negative accumulators) the accumulated set, an honest witness holder can always prove membership (or non-membership, for negative accumulators) in the set. The second is soundness: for every element outside (or inside, for negative accumulators) the accumulated set, it is infeasible to prove membership (or non-membership, for negative accumulators).

Our ideal functionality is different than most ideal functionalities in that it requires as input from the simulator all of the accumulator algorithms (as previously done in the context of digital signatures (Canetti, 2004)). This is actually a very intuitive way to build an ideal functionality, since it only deviates from the algorithm outputs when necessary for correctness or soundness. We explain this in more detail in Sec. 2.3.

We chose not to incorporate secrecy or privacy requirements into our ideal functionality since they depend on specific applications and vary considerably; thus, they are best made separately, as an additional "layer" on top of the basic correctness guarantees captured in this work. Additionally, privacy-aware constructions often use accumulators and privacy-enhancing mechanisms (such as zero-knowledge proofs) as two separate modules, making the formalization here more conducive to modular analysis. We exemplify this point by sketching a modular analysis of the Baldimtsi *et al.* (Baldimtsi et al., 2017) construction of revocable anonymous credentials from zero knowledge proofs and accumulators.

Outline

We start by setting notation and presenting classical accumulator definitions (but with a twist) in Section 2.2. Then, in Section 2.3, we give an ideal UC functionality for accumulators that encompasses both of the properties listed above. In Section 2.4, we argue that any accumulator that has these properties meets our UC definition, and vice versa.

2.1.1 Accumulator Applications

To showcase the importance of a UC analysis for cryptographic accumulators we briefly discuss a few of the most interesting systems that use accumulators as a main building block. The security analysis of all the following systems would be much simpler when the underlying accumulator is UC-secure.

Access Control

Authentication of users is vital to most of the electronic systems we use today. It is usually achieved by giving the user a token, or *credential*, that the user must present to prove that she has permission to access a service. A naive construction for an access control system is to maintain a whitelist of authorized users (i.e., by storing their credentials). Whenever a user wants to access the system she just needs to present her credential, and as long as it is on the whitelist, the user will be given access. When a user needs to be revoked, her credential is just removed from the whitelist. Despite its simplicity, such a solution is not practical, since the size of the whitelist will have to grow linearly with the number of participating users. One might argue that expending an amount of resources that is linear in the number of participating users is something systems already do, but this is not the case for all systems, especially those the sole purpose of which is to control access to some resource.

Cryptographic accumulators enable more efficient access control systems. Instead of keeping a whitelist, an accumulator can be used to maintain the set of authorized users. Whenever a user is given access to the resource, she is given a credential that can be seen as an accumulator membership witness. One possible construction uses the digital signature accumulator together with a blacklist of revoked users, which grows linearly with the number of revocations. This construction is the one most commonly used in public key infrastructures (PKIs), where a certificate revocation list (CRL) that contains the revoked certificates is published periodically. This solution is more efficient, since usually the number of revoked users is much smaller than the number of total users in the system. However, it is still not ideal, since the blacklist can grow to significant size. A dynamic accumulator — which supports both element additions and deletions while remaining small — is a much better solution.

Anonymous Credentials

The inefficiency of the naive whitelist and blacklist solutions for access control becomes even more problematic when anonymity is considered as a goal of the system: if a user wishes to anonymously show that her credential is on a whitelist (or not on a blacklist), then she would have to perform a zero-knowledge proof of membership (or non-membership) which would require cost linear to the size of the corresponding list. Given how expensive zero knowledge proofs usually are, it is important to avoid doing work linear in the number of valid or revoked members in a system. To avoid this inefficiency, *anonymous credentials schemes* (the most prominent solution for anonymous user authentication) make use of dynamic cryptographic accumulators as an essential building block to allow for efficient proofs of membership (and practical user revocation) (Camenisch and Lysyanskaya, 2002; Nguyen, 2005; Camenisch et al., 2009). Idemix (Camenisch and Van Herreweghen, 2002), the leading anonymous credential system by IBM, is such an example of an anonymous credential scheme that

employs cryptographic accumulators for user membership management (Baldimtsi et al., 2017).

Cryptocurrencies

As discussed above, when a proof of membership (or non-membership) needs to be done in zero-knowledge, the naive whitelist and blacklist solutions are not realistic. Anonymous cryptocurrencies, like anonymous credentials, require such zero-knowledge proofs. In order to prove that a payment is valid (and is not a double-spend), when a user wishes to spend a coin that she owns, she must first prove that her coin does not belong in a list of previously spent coins. To ensure anonymity, such a proof must be done in zero-knowledge. Universal cryptographic accumulators are used in Zerocoin (Miers et al., 2013) to maintain the set of spent coins while enabling efficient zero-knowledge proofs of non-membership.

Group Signatures

Accumulators have been suggested for building other cryptographic primitives such as group signatures. In a group signature scheme, the group manager maintains a list of valid group members, and periodically grants (or revokes) membership. There has been much research on the topic of group signatures, and a number of efficient schemes have been proposed. One of the first practical solutions supporting revocation uses cryptographic accumulators for user revocation (Camenisch and Lysyanskaya (Camenisch and Lysyanskaya, 2002), building on the ACJT group signature scheme (Ateniese et al., 2000)).

2.2 Revisiting Classical Accumulator Definitions

We first discuss accumulator terminology and notation and review accumulator algorithms. Then, in Section 2.2.2, we revise the classical accumulator definitions of

security to be more modular, and to support a wider range of accumulator functionalities. These changes make the transition to the UC model more clear and natural.

2.2.1 Notation and Algorithms

An accumulator is a compact representation of a set $S = \{x_1, \dots x_n\}$, which can be used to prove statements about the underlying set. Different accumulator types and properties have been considered in the literature. Here, we use the terminology and definitions of Baldimtsi *et al.* (Baldimtsi et al., 2017), who provide a modular view of accumulator functionalities. Like them, we consider four basic types of accumulators:

- *Static accumulator*: represents a fixed set.
- *Additive accumulator*: supports only addition of elements to the set.
- *Subtractive accumulator*: supports only deletion of elements from the set.
- *Dynamic accumulator* (Camenisch and Lysyanskaya, 2002): supports both additions and deletions.

Note that a trivial way to achieve deletions and additions is by re-instantiating the accumulator with the updated set. Although simple, this takes a polynomial amount of time in the number of element additions or deletions which have been performed up until that point. For practical applications a dynamic accumulator should support both additions and deletions in time which is either independent of the number of operations performed altogether, or at least sublinear in this number.

In addition to considering the types of modifications we can make to accumulated sets, we also consider the types of proofs (membership proofs, non-membership proofs, or both) accumulators support.

- *Positive accumulator*: supports membership proofs.
- *Negative accumulator*: supports non-membership proofs.
- *Universal accumulator* (Li et al., 2007a): supports both types of proofs.

We consider three types of parties in the accumulator setting. The *accumulator*

manager is a special party who is the "owner" of the accumulated set: she creates the accumulator, adds and deletes elements, and creates membership and non-membership witnesses. A *witness holder*, or *user*, is responsible for an accumulated element (i.e. she owns a credential in a system for which an accumulator is used). She is interested in being able to prove the (non-)membership of that element to others, so she maintains the witness for that element, by updating it when/if necessary. Finally, a *verifier* is any third party who is only interested in checking the proofs of (non-)membership (e.g. a gatekeeper checking credentials).

We now describe the algorithms performed by each party, and summarize them in Figure 2·1. In Figure 0·1 we summarize the notation used to describe the different accumulator algorithm input and output parameters.

Accumulator Manager Algorithms

The following are algorithms performed by the accumulator manager who creates the accumulator and maintains it as required. If the accumulator is additive, she can add elements to it by calling the Update algorithm with $\mathsf{Op} = \mathsf{Add}$. If the accumulator is subtractive, she can delete elements by calling Update with $\mathsf{Op} = \mathsf{Del}$. If it is dynamic, she can do both. If the accumulator is positive, the accumulator manager can create membership witnesses by calling $\mathsf{WitCreate}$ with $\mathsf{stts} = \mathsf{in}$ (where stts is a variable representing the *status* of an element, which can be in or out of the set); if it is negative she can create non-membership witnesses by calling $\mathsf{WitCreate}$ with $\mathsf{stts} = \mathsf{out}$. If it is universal, she can do both.

- $\mathsf{Gen}(1^\lambda, S_0) \rightarrow (sk, a_0, m_0)$ outputs the accumulator manager's secret key sk, the accumulator a_0 (representing the initial set $S_0 \subseteq D$ of elements in the accumulator, where D is the domain of the accumulator[1]), and an auxiliary

[1]The allowable S_0 sets vary from accumulator to accumulator. There are accumulators that support only $S_0 = \emptyset$; others support any polynomial-size S_0, and yet others support any S_0 that can be expressed as a polynomial number of ranges.

value m_0 necessary for the maintenance of the accumulator (i.e. one could think of m_t being the accumulator manager's memory or storage at step t).

- $\mathsf{Update}(\mathsf{Op}, sk, a_t, m_t, x) \rightarrow (a_{t+1}, m_{t+1}, w^x_{t+1}, \mathsf{upmsg}_{t+1})$ updates the accumulator by either adding or deleting an element. If $\mathsf{Op} = \mathsf{Add}$ it adds the element $x \in D$ to the accumulator and outputs the updated accumulator value a_{t+1} and auxiliary value m_{t+1}, as well as the membership witness w^x_{t+1} for x and an update message upmsg_{t+1}, which enables witness holders to bring their witnesses up to date. If $\mathsf{Op} = \mathsf{Del}$ then it deletes the element x from the accumulator and outputs a_{t+1}, m_{t+1} and upmsg_{t+1} as before, as well as a *non*-membership witness w^x_{t+1}.
- $\mathsf{WitCreate}(\mathsf{stts}, sk, a_t, m_t, x, (\mathsf{upmsg}_1, \ldots, \mathsf{upmsg}_t)) \rightarrow w^x_t$ creates a (non-) membership witness. If $\mathsf{stts} = \mathsf{in}$ it generates a membership witness w^x_t for x, and if $\mathsf{stts} = \mathsf{out}$ it generates a non-membership witness. (Of course, this algorithm should only succeed in generating a valid membership witness if x is actually in the set, and in generating a non-membership witness if x is not in the set.)

Remark 1. *The parameters sk, m and* upmsg *are optional for some accumulator constructions. We mark such parameters in grey. For instance, in a Merkle hash tree accumulator there is no secret key sk, and in a digital signature accumulator there is no auxiliary value m or update messages* upmsg*. Notice that the* WitCreate *algorithm takes in both the auxiliary value m and the update messages, which seems redundant; after all, the update messages can always be kept as part of m. The reason we provide the algorithm with both arguments is to account for scenarios which do not use any auxiliary storage.*

Remark 2. *The notion of a public key is absent on the above definition. One can consider the accumulator value a to be the "public key" of the scheme, since it is used for verification. In fact, in the digital signature accumulator construction, the public verification key is equal to the accumulator value. However, unlike a typical public key, the accumulator value can evolve over time.*

Witness Holder Algorithms

Witness holders are interested in proving the (non-)membership of certain elements, and thus maintain witnesses for those elements. They use a witness update algorithm WitUp to sync their witnesses with the accumulator when additions or deletions occur.

- $\mathsf{WitUp}(\mathsf{stts}, x, w_t^x, \mathsf{upmsg}_{t+1}) \to w_{t+1}^x$ updates the membership witness for element x (if $\mathsf{stts} = \mathsf{in}$) or the non-membership witness if $\mathsf{stts} = \mathsf{out}$. The updates use the update messages upmsg, which contain information about changes to the accumulator value (e.g. that a given element was added, what the new accumulator value is, etc).

Verifier/Third Party Algorithms

The last category of accumulator users are the *verifiers* (or third parties) who are only interested in checking proofs of (non-)membership. They do so by calling the $\mathsf{VerStatus}$ algorithm.

- $\mathsf{VerStatus}(\mathsf{stts}, a_t, x, w_t^x) \to \phi$ checks whether the membership witness (if $\mathsf{stts} = \mathsf{in}$) or the non-membership witness (if $\mathsf{stts} = \mathsf{out}$) for element x is valid; it returns $\phi = 1$ if it is, and $\phi = 0$ if it is not.

If the accumulator is *strong* (Definition 4), the accumulator should be secure even against a cheating accumulator manager. That is, all modifications that an accumulator manager makes to the accumulator should be publicly verifiable. The differences in the algorithms are as follows: (a) Gen and Update also output a value v, which essentially is a proof that an accumulator was created/updated correctly. (b) Additional verification algorithms VerGen and $\mathsf{VerUpdate}$ can be used to check these proofs.

Algorithm	Inputs	Outputs
Accumulator Manager Algorithms		
Gen	$1^\lambda, S_0$	sk, a_0, m_0, v
Update	$\mathsf{Op}_t, sk, a_t, m_t, x$	$a_{t+1}, m_{t+1}, w^x_{t+1}, \mathsf{upmsg}_{t+1}, v_{t+1}$
WitCreate	$\mathsf{stts}, sk, a_t, m_t, (\mathsf{upmsg}_1, \ldots, \mathsf{upmsg}_t), x$	w^x_t
Witness Holder Algorithms		
WitUp	$\mathsf{stts}, x, w^x_t, \mathsf{upmsg}_{t+1}$	w^x_{t+1}
Verifier or Third Party Algorithms		
VerStatus	$\mathsf{stts}, a_t, x, w^x_t$	$\phi \in \{0,1\}$
Additional Third Party Algorithms in Strong Accumulators		
VerGen	$1^\lambda, S_0, a_0, v$	$\phi \in \{0,1\}$
VerUpdate	$\mathsf{Op}_t, a_t, a_{t+1}, x, v_{t+1}$	$\phi \in \{0,1\}$

Figure 2·1: Accumulator Algorithms. In static accumulators, the Update, WitUp and VerUpdate algorithms do not exist. In additive accumulators, Op is required to be equal to Add everywhere. In subtractive accumulators, Op is required to be equal to Del. In dynamic accumulators, Op can be either. In positive accumulators, stts is required to be equal to in everywhere. In negative accumulators, stts is required to be equal to out. In universal accumulators, stts can be either.

2.2.2 Security Definitions

A cryptographic accumulator should satisfy two basic security properties: correctness and soundness. In this section, we review the classical correctness and soundness properties of accumulators (stated, for instance, by Ghosh *et al.* (Ghosh et al., 2016)). We revise these classical definitions in several ways.

1. We explicitly consider the correctness of the witness update algorithm, which (Ghosh et al., 2016) consider only as an efficiency shortcut, and thus exclude from their definitions. Since the update algorithm is used in practice, we believe it is important to include in the formal definitions.
2. We allow the generation of membership witnesses during addition (or non-membership witnesses during deletion) as is commonly done in practice, while (Ghosh et al., 2016) only considers the generation of witnesses from a fixed accumulator state. Because of this, we have two separate notions of correctness — *correctness* and *creation-correctness*.

Correctness Definitions

Definitions 7 and 2 give the correctness requirements for the more general case of a universal dynamic accumulator. Informally, an accumulator is *correct* or *creation-correct* if an up-to-date version of a witness produced by Update or $\mathsf{WitCreate}$, respectively, can be used to verify the (non-)membership of the corresponding element. It is easy to adapt our definition for cases of additive/subtractive or positive/negative. To get a definition for an additive accumulator, restrict all instances of Op to be equal to Add; to get a definition for a subtractive accumulator, restrict all instances of Op to be equal to Del. Similarly, to get a definition for a positive accumulator, restrict all instances of stts to be equal to in; to get a definition for a negative accumulator, restrict all instances of stts to be equal to out.

Definition 1 (Correctness)**.** *A universal dynamic accumulator is* correct *for a given domain D of elements if an up-to-date witness w^x corresponding to value x can always be used to verify the (non-)membership of x in an up-to-date accumulator a. More formally, there exists a negligible function ν in the security parameter λ such that for all:*

- *security parameters λ,*
- *initial sets $S_0 \subseteq D$,*
- *values $x \in D$,*
- *positive integers t polynomial in λ,*
- *positive integers t_x such that $1 \leq t_x \leq t$,*
- *operations* $\mathsf{Op} \in \{\mathsf{Add}, \mathsf{Del}\}$ *(with* $\mathsf{stts} = \mathsf{in}$ *if* $\mathsf{Op} = \mathsf{Add}$ *and* $\mathsf{stts} = \mathsf{out}$ *if* $\mathsf{Op} = \mathsf{Del}$*),*
- *lists of tuples* $[(y_1, \mathsf{Op}_1), \ldots, (y_{t_x-1}, \mathsf{Op}_{t_x-1})]$, $[(y_{t_x+1}, \mathsf{Op}_{t_x+1}), \ldots, (y_t, \mathsf{Op}_t)]$, *where*
  - $y_i \in D$ *and* $\mathsf{Op}_i \in \{\mathsf{Add}, \mathsf{Del}\}$ *for* $i \in [1, \ldots, t_x - 1, t_x + 1, \ldots, t]$*;*
  - *If* $\mathsf{Op} = \mathsf{Add}$*, then* (x, Del) *does not appear in* $[(y_{t_x+1}, \mathsf{Op}_{t_x+1}), \ldots, (y_t, \mathsf{Op}_t)]$*; and*
  - *If* $\mathsf{Op} = \mathsf{Del}$*, then* (x, Add) *does not appear in* $[(y_{t_x+1}, \mathsf{Op}_{t_x+1}), \ldots, (y_t, \mathsf{Op}_t)]$*,*

The following holds:

$$\Pr\left[\begin{array}{l}(a_0, sk) \leftarrow \mathsf{Gen}(1^\lambda, S_0);\\ (a_i, m_i, w_i^{y_i}, \mathsf{upmsg}_i) \leftarrow \mathsf{Update}(\mathsf{Op}_i, sk, a_{i-1}, m_{i-1}, y_i) \textit{ for } i \in [1, \ldots, t_x - 1];\\ (a_{t_x}, m_{t_x}, w_{t_x}^x, \mathsf{upmsg}_{t_x}) \leftarrow \mathsf{Update}(\mathsf{Op}, sk, a_{t_x-1}, m_{t_x-1}, x);\\ (a_i, m_i, w_i^{y_i}, \mathsf{upmsg}_i) \leftarrow \mathsf{Update}(\mathsf{Op}_i, sk, a_{i-1}, m_{i-1}, y_i) \textit{ for } i \in [t_x + 1, \ldots, t];\\ w_i^x \leftarrow \mathsf{WitUp}(\mathsf{stts}, x, w_{i-1}^x, \mathsf{upmsg}_i) \textit{ for } i \in [t_x + 1, \ldots, t]) :\\ \mathsf{VerStatus}(\mathsf{stts}, a_t, x, w_t^x) = 1\end{array}\right] \geq 1 - \nu(\lambda)$$

Definition 2 (Creation-Correctness)**.** *A universal dynamic accumulator is* creation-correct *for a given domain D of elements if an up-to-date witness w^x* created by the WitCreate algorithm — not by the Update algorithm! — *corresponding to value x can always be used to verify the (non-)membership of x in an up-to-date accumulator a.*

More formally, there exists a negligible function ν in the security parameter λ such that for all

- *security parameters λ,*
- *initial sets $S_0 \subseteq D$,*
- *values $x \in D$,*
- *positive integers t polynomial in λ,*
- *positive integers t_x such that $1 \leq t_x \leq t$,*
- *statuses $\mathsf{stts} \in \{\mathsf{in}, \mathsf{out}\}$, and*
- *lists of values $[(y_1, \mathsf{Op}_1), \ldots, (y_t, \mathsf{Op}_t)]$, where*
  - *$y_i \in D$ and $\mathsf{Op}_i \in \{\mathsf{Add}, \mathsf{Del}\}$ for $i \in [1, \ldots, t]$;*
  - *If $\mathsf{stts} = \mathsf{in}$*
    - *either (a) $x \in S_0$, or (b) (x, Add) appears in $[(y_1, \mathsf{Op}_1), \ldots, (y_{t_x-1}, \mathsf{Op}_{t_x})]$ and was not followed by (x, Del), and*
    - *(x, Del) does not appear in $[(y_{t_x+1}, \mathsf{Op}_{t_x+1}), \ldots, (y_t, \mathsf{Op}_t)]$;*
  - *If $\mathsf{stts} = \mathsf{out}$*
    - *either (a) $x \notin S_0$, or (b) (x, Del) appears in $[(y_1, \mathsf{Op}_1), \ldots, (y_{t_x-1}, \mathsf{Op}_{t_x})]$ and was not followed by (x, Add), and*
    - *(x, Add) does not appear in $[(y_{t_x+1}, \mathsf{Op}_{t_x+1}), \ldots, (y_t, \mathsf{Op}_t)]$;*

The following holds:

$$\Pr\left[\begin{array}{l}(a_0, sk) \leftarrow \mathsf{Gen}(1^\lambda, S_0);\\ (a_i, m_i, w_i^{y_i}, \mathsf{upmsg}_i) \leftarrow \mathsf{Update}(\mathsf{Op}_i, sk, a_{i-1}, m_{i-1}, y_i) \textit{ for } i \in [1, \ldots, t_x];\\ w_{t_x}^x \leftarrow \mathsf{WitCreate}(\mathsf{stts}, sk, a_t, m_t, x);\\ (a_i, m_i, w_i^{y_i}, \mathsf{upmsg}_i) \leftarrow \mathsf{Update}(\mathsf{Op}_i, sk, a_{i-1}, m_{i-1}, y_i) \textit{ for } i \in [t_x + 1, \ldots, t];\\ w_i^x \leftarrow \mathsf{WitUp}(\mathsf{stts}, x, w_{i-1}^x, \mathsf{upmsg}_i) \textit{ for } i \in [t_x + 1, \ldots, t]) :\\ \mathsf{VerStatus}(\mathsf{stts}, a_t, x, w_t^x) = 1\end{array}\right] \geq 1 - \nu(\lambda)$$

Soundness Definitions

Classically, *collision-freeness* (Bari and Pfitzmann, 1997) is the soundness definition for accumulators. Collision-freeness informally requires that for any element not in the accumulated set it should be hard to find a membership witness. For negative and universal accumulators, collision-freeness can be extended to require that for any element in the accumulated set it should be hard to find a non-membership witness. Another formalization of accumulator soundness for universal accumulators is *undeniability* (Lipmaa, 2012), which requires that for any element (regardless of its presence in the accumulated set) it be hard to find *both* a membership witness and a non-membership witness.

In this paper, we choose to use collision-freeness, since undeniability is not meaningful for positive or negative accumulators, which only support proofs of membership or proofs of non-membership but not both. Definition 3 gives the collision-freeness definition for a universal dynamic accumulator. This definition can be converted to work for positive, negative, additive or subtractive accumulators in the usual way (by limiting the possible values of Op or stts).

Definition 3 (Collision-Freeness)**.** *A universal dynamic accumulator is* collision-free *for a given domain D of elements if it is hard to fabricate a (non-)membership witness w for a value x that is not (or, respectively, is) in the accumulated set. More formally, consider the collision-freeness game described in Figure 2·2. An accumulator is collision-free if for any sufficiently large security parameter λ, for any probabilistic polynomial-time adversary $\mathcal{A}_{\mathsf{ColFree}}$, there exists a negligible function ν in the security parameter λ such that the probability that $\mathcal{A}_{\mathsf{ColFree}}$ wins the game is less than $\nu(\lambda)$.*

Non-Adaptive Soundness

In the collision-freeness game of Figure 2·2, the adversary is able to choose elements to add and delete adaptively. However, this notion of collision-freeness (or *sound-*

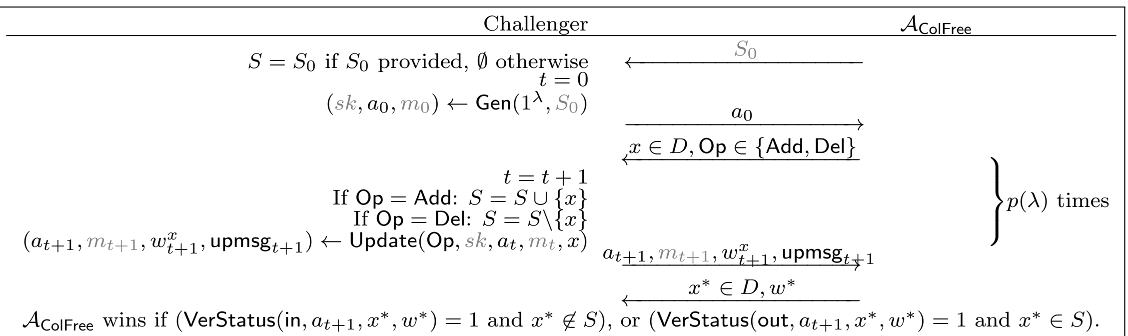


Figure 2·2: The Collision-Freeness Game for Accumulators

ness) is quite strong. In a *non-adaptive*[2] version of the game, the adversary would be required to commit to all elements it intends to add before seeing a_0. Certain accumulators can only be shown to meet non-adaptive soundness. One example of such an accumulator is the CLRSAB accumulator, which was informally introduced as a brief remark by Camenisch and Lysyanskaya (Camenisch and Lysyanskaya, 2002) and formally described by Baldimtsi *et al.* (Baldimtsi et al., 2017). Note that, in particular, a non-adaptively sound accumulator can always be used to accumulate random values, since it makes no difference whether random values are chosen beforehand or on-the-fly.

Strength

Typically, accumulators are not required to be secure against cheating accumulator managers, since in many scenarios the entity that manages the set (and thus the accumulator) is trusted. When that is not the case (e.g. in many blockchain applications), a *strong* accumulator can be used. A strong accumulator provides guarantees even against a cheating accumulator manager. Informally, an accumulator is *strong* if all of the modifications an accumulator manager makes to the accumulator are verifiable.

[2]Note that this does not refer to non-adaptive corruptions, as in the context of MPC; it is not corruptions that are non-adaptive, but the choice of accumulated elements.

Definition 4 (Strength)**.** *An accumulator is* strong *for a given domain D of elements if an adversary cannot win the game described in Figure 2·2 with non-negligible probability even if it is modified as follows: instead of asking the challenger to run* Gen *and* Update*, the adversary runs them locally and sends the challenger the updated accumulator values together with witnesses v. The challenger aborts if* VerGen *or* $\mathsf{VerUpdate}$ *return* 0*.*

We must also ensure the correctness of the VerGen and $\mathsf{VerUpdate}$ algorithms.

Definition 5 (Strength Correctness)**.** *Informally, an accumulator has* strength correctness *if* VerGen *and* $\mathsf{VerUpdate}$ *run on honestly generated inputs and outputs of* Gen *and* Update *always return* 1*.*

2.3 Ideal Functionality for Accumulators

Universally Composable (UC) security, proposed by Canetti (Canetti, 2001) and described briefly in Section 3.1, requires a different flavor of definitions than those described in Section 2.2. A UC definition of security for some primitive consists of a set of instructions called an *ideal functionality* which achieves the goals of the primitive when carried out by an incorruptible third party. Informally, to show that a candidate protocol *securely realizes* the ideal functionality, it must be shown that any adversary in a real execution of the protocol can be simulated by a corresponding ideal world adversary in an interaction with the incorruptible third party running the ideal functionality.

Definition 6 ((Canetti, 2001, Page 12))**.** *Let* $\mathsf{exec}_{\Pi,\mathcal{A},\mathcal{Z}}$ *denote the random variable (over the local random choices of all the involved machines) describing the output of environment $\mathcal{Z}$ when interacting with adversary $\mathcal{A}$ and parties running protocol* Π*. Protocol* Π *UC-emulates ideal functionality $\mathcal{F}$ if for any adversary $\mathcal{A}$ there exists a simulator $\mathcal{SIM}$ such that, for any environment $\mathcal{Z}$ the distributions of* $\mathsf{exec}_{\Pi,\mathcal{A},\mathcal{Z}}$ *and* $\mathsf{exec}_{\mathcal{F},\mathcal{SIM},\mathcal{Z}}$ *are indistinguishable. That is, on any input, the probability that $\mathcal{Z}$ outputs* 1 *after interacting with $\mathcal{A}$ and parties running* Π *differs by at most a negligible amount from the probability that $\mathcal{Z}$ outputs* 1 *after interacting with $\mathcal{SIM}$ and $\mathcal{F}$.*

In this section we present our ideal functionality $\mathcal{F}_{\mathsf{ACC}}$ for an accumulator.

Like (Canetti, 2004), we discuss several candidate ideal functionalities in order to build up the intuition for how we arrived at the ideal functionality described in Fig. 2·3 and 2·4.

First Attempt

A naive first attempt at an accumulator functionality might ignore the accumulator and witness objects altogether, instead functioning as a simple set manager. It would allow the accumulator manager to add and remove elements from the set, and answer 'yes' or 'no' to membership (or non-membership) queries. These queries could optionally be parametrized by timestamps, so as to allow queries about all states of the set, past and present. However, this simple ideal functionality definition fails to support one of the basic modular operations of accumulators. Recall that an accumulator is an object that evolves by time, i.e. at time t it might represent a different set from what it used to represent at time t'. Thus, if we do not consider explicit accumulator objects, then it is impossible to talk about committing to a given set by committing to an accumulator value at a specific time.

Second Attempt - Explicitly Modeling Accumulator Values

A second attempt might be to add explicit accumulator values, without modeling witnesses. So, a membership query would now have the form, 'is this element a (non-)member under this accumulator value?'. However, the absence of explicit witness objects also limits the modular use of accumulators significantly. Specifically, not having explicit witness objects would not work when the ability to verify the (non)membership of certain elements should be secret-shared or otherwise restricted. (For instance, perhaps I should be able to demonstrate my membership in some organization - such as the gym - but any third party shouldn't be able to test my

membership without my help, because that would be a violation of my privacy.) Adding these privacy features to an accumulator system would require re-designing and re-proving the accumulator system from scratch if witness objects were not part of the ideal functionality. If witness and accumulator objects are modeled explicitly, however, existing accumulator systems can simply be combined with existing off-the-shelf primitives such as secret sharing, encryption, or commitment. In other words, having the functionality give binary answers to membership queries is over-idealization; it is a good way to model accumulators on their own, but it does not lend itself to use by other protocols that need actual accumulator and witness values to operate.

Final Attempt

Our ideal functionality for accumulators $\mathcal{F}_{\mathsf{ACC}}$ is described in Figures 2·3 and 2·4 and provides interfaces for all of the algorithms in Figure 2·1. (Note that in the functionality the accumulator manager interfaces ignore all queries for which the querier's identity is not encoded in the functionality session id *sid*.)

We loosely base $\mathcal{F}_{\mathsf{ACC}}$ on the ideal functionality for digital signatures described by Canetti (Canetti, 2004). Canetti actually gave two different functionalities for digital signatures, which we recall for completeness in Appendix 2.5. The first one (Figure 2·5) asks the ideal world adversary for a *verification key*; while the second (Figure 2·6) asks the ideal world adversary for a *verification algorithm.* Similarly, Camenisch *et al.* (Camenisch et al., 2019) give functionalities for signatures, non-interactive zero knowledge proofs and for commitments that are explicitly parameterized by the protocol algorithms. Using a given deterministic signature verification algorithm, rather than allowing the ideal world adversary to make each verification decision, achieves two goals:

- It forces verification decisions to be consistent.

- It makes combining UC signatures and zero knowledge proofs of signature knowledge in a black box way simpler.

For these reasons, we chose to define our $\mathcal{F}_{\mathsf{ACC}}$ to receive explicit algorithms from the ideal world adversary. Thus, instead of asking the ideal world adversary to provide updated accumulator states, witnesses and verification decisions, our ideal world adversary provides *all* accumulator algorithms to the functionality (Step 1e in Figure 2·3).[3] This is a very intuitive way to define an ideal functionality: it explicitly uses the accumulator algorithms except where it needs to modify their behavior to match what is demanded by correctness or soundness. If an ideal execution (that uses the ideal functionality) is indistinguishable from a real execution, that means that the algorithms' behavior did not need any modification.

Just like in the context of digital signatures, if the algorithms are modeled explicitly, usage within multi-party computation (MPC) protocols or in larger zero-knowledge-based systems such as Zcash can be done in a modular way, using existing components.

In addition to the benefits listed above, this also allows us more flexibility to add privacy features to the ideal functionality, as discussed in Section 2.3.3.

Remark 3. *Note that inputs belonging to anyone but the accumulator manager ($\mathcal{AM}$) can be misinformed (just like parties are frequently misinformed about verification keys in signature schemes, in the absence of a PKI). In order to capture such cases, we require parties to provide all inputs to witness holder and third party algorithms, instead of having some inputs, such as the accumulator value, implicitly stored by the ideal functionality.*

The ideal functionality described in Figures 2·3 and 2·4 is really an entire "menu" of functionalities covering all different types of accumulators: additive, subtractive, dynamic, positive, negative and universal and finally strong accumulators. More ex-

[3]These algorithms will, among other things, check that elements being added are in the domain D of the accumulator in question.

1. GEN: Upon getting (GEN, sid, S_0) as first activation from $\mathcal{AM}$ ...
   (a) Initialize an operation counter $t = 0$.
   (b) Initialize an empty list $\mathbf{A}$. This list will be used to keep track of all accumulator states.
   (c) Initialize an empty map $\mathbf{S}$, and set $\mathbf{S}[0] = S_0$. (If S_0 was not provided, use $\emptyset$.) This map will be used to map operation counters to current accumulated sets.
   (d) Send (GEN, sid) to Adversary $\mathcal{A}_{\mathsf{Ideal}}$.
   (e) Get $(\mathsf{ALGORITHMS}, sid, (\mathsf{Gen}, \mathsf{Update}, \mathsf{WitCreate}, \mathsf{WitUp}, \mathsf{VerStatus}, \mathsf{VerGen}, \mathsf{VerUpdate}))$ from Adversary $\mathcal{A}_{\mathsf{Ideal}}$. This includes all of the accumulator algorithms; their expected input output behavior is described in Figure 2·1. All of them should be polynomial-time; we restrict the verification algorithms $\mathsf{VerStatus}$, VerGen, $\mathsf{VerUpdate}$ to be deterministic.
   (f) Run $(sk, a_0, m_0, v) \leftarrow \mathsf{Gen}(1^\lambda, S_0)$.
   (g) Verify that $\mathsf{VerGen}(S_0, a_0, v) = 1$. If not, output $\perp$ to $\mathcal{AM}$ and halt. (This ensures strength.) Otherwise, continue.
   (h) Store sk, m_0; add a_0 to $\mathbf{A}$.
   (i) Output $(\mathsf{ALGORITHMS}, sid, S_0, (\mathsf{Gen}, \mathsf{Update}, \mathsf{WitCreate}, \mathsf{WitUp}, \mathsf{VerStatus}, \mathsf{VerGen}, \mathsf{VerUpdate})$ to $\mathcal{AM}$.
2. UPDATE: Upon getting $(\mathsf{UPDATE}, sid, \mathsf{Op}, x)$ from $\mathcal{AM}$ ...
   (a) Increment the operation counter: $t = t + 1$.
   (b) Set $\mathbf{S}[t] = \mathbf{S}[t-1]$.
   (c) Run $(a_t, m_t, w_t^x, \mathsf{upmsg}_t, v_t) \leftarrow \mathsf{Update}(\mathsf{Op}, sk, a_{t-1}, m_{t-1}, x)$.
   (d) If $\mathsf{Op} = \mathsf{Add}$:
      i. Verify that $\mathsf{VerStatus}(\mathsf{in}, a, x, w_t) = 1$. If not, output $\perp$ to $\mathcal{AM}$ and halt. (This ensures correctness.) Otherwise, continue.
      ii. If $x \notin \mathbf{S}[t]$, add x to $\mathbf{S}[t]$.
   (e) If $\mathsf{Op} = \mathsf{Del}$:
      i. Verify that $\mathsf{VerStatus}(\mathsf{out}, a, x, w_t) = 1$. If not, output $\perp$ to $\mathcal{AM}$ and halt. (This ensures negative correctness.) Otherwise, continue.
      ii. If $x \in \mathbf{S}[t]$, remove x from $\mathbf{S}[t]$.
   (f) Verify that $\mathsf{VerUpdate}(\mathsf{Op}, a_{t-1}, a_t, x, v_t) = 1$. If not, output $\perp$ to $\mathcal{AM}$ and halt. (This ensures strength.) Otherwise, continue.
   (g) Store m_t, upmsg_t; add a_t to $\mathbf{A}$.
   (h) Output $(\mathsf{UPDATE}, sid, \mathsf{Op}, a_t, x, w_t, \mathsf{upmsg}_t)$ to $\mathcal{AM}$.
3. $\mathsf{WITCREATE}$: Upon getting $(\mathsf{WITCREATE}, sid, \mathsf{stts}, x)$ from $\mathcal{AM}$ ...
   (a) Run $w \leftarrow \mathsf{WitCreate}(\mathsf{stts}, sk, a_t, m_t, x, (\mathsf{upmsg}_1, \ldots, \mathsf{upmsg}_t))$
   (b) If $\mathsf{stts} = \mathsf{in}$:
   If $x \in \mathbf{S}[t]$, verify that $\mathsf{VerStatus}(\mathsf{in}, a_t, x, w) = 1$. If not, output $\perp$ to $\mathcal{AM}$ and halt. (This ensures creation-correctness.) Otherwise, continue.
   (c) If $\mathsf{stts} = \mathsf{out}$:
   If $x \notin \mathbf{S}[t]$, verify that $\mathsf{VerStatus}(\mathsf{out}, a_t, x, w) = 1$. If not, output $\perp$ to $\mathcal{AM}$ and halt. (This ensures negative-creation-correctness.) Otherwise, continue.
   (d) Output $(\mathsf{WITNESS}, sid, \mathsf{stts}, x, w)$ to $\mathcal{AM}$.
4. WITUP: Upon getting $(\mathsf{WITUP}, sid, \mathsf{stts}, a_{old}, a_{new}, x, w_{old}, (\mathsf{upmsg}_{old+1}, \ldots, \mathsf{upmsg}_{new}))$ from any party $\mathcal{H}$ ...
   (a) Run $w_{new} \leftarrow \mathsf{WitUp}(\mathsf{stts}, x, w_{old}, (\mathsf{upmsg}_{old+1}, \ldots, \mathsf{upmsg}_{new}))$
   (b) If $a_{old} \in \mathbf{A}$, $a_{new} \in \mathbf{A}$ and $old < new$:
      i. If $\mathsf{stts} = \mathsf{in}$, $\mathsf{VerStatus}(\mathsf{in}, a_{old}, x, w_{old}) = 1$, $x \in \mathbf{S}[t]$ for $t \in [old, \ldots, new]$, $\mathsf{upmsg}_{old+1}, \ldots, \mathsf{upmsg}_{new}$ match the stored values and $\mathsf{VerStatus}(\mathsf{in}, a_{new}, x, w_{new}) = 0$, output $\perp$ to $\mathcal{P}$ and halt. (This ensures correctness.) Otherwise, continue.
      ii. If $\mathsf{stts} = \mathsf{out}$, $\mathsf{VerStatus}(\mathsf{out}, a_{old}, x, w_{old}) = 1$, $x \notin \mathbf{S}[t]$ for $t \in [old, \ldots, new]$, $\mathsf{upmsg}_{old+1}, \ldots, \mathsf{upmsg}_{new}$ match the stored values and $\mathsf{VerStatus}(\mathsf{out}, a_{new}, x, w_{new}) = 0$, output $\perp$ to $\mathcal{P}$ and halt. (This ensures negative correctness.) Otherwise, continue.
   (c) Output $(\mathsf{UPDATEDWITNESS}, sid, \mathsf{stts}, a_{old}, a_{new}, x, w_{old}, (\mathsf{upmsg}_{old+1}, \ldots, \mathsf{upmsg}_{new}), w_{new})$ to $\mathcal{H}$.

Figure 2·3: Ideal Functionality $\mathcal{F}_{\mathsf{ACC}}$: Accumulator Manager / Witness Holder Interfaces

plicitly, by default, if all of the text (except for the text colored by pink) is considered, the ideal functionality describes a dynamic, universal accumulator. By restricting Op

1. $\mathsf{VERSTATUS}$: Upon getting $(\mathsf{VERSTATUS}, sid, \mathsf{stts}, a, \mathsf{VerStatus}', x, w)$ from any party $\mathcal{P}$ ...
   (a) If $\mathsf{VerStatus}' = \mathsf{VerStatus}$ and there exists a t such that $a = a_t \in \mathbf{A}$:
      i. Let t be the largest such number.
      ii. If $\mathsf{stts} = \mathsf{in}$:
         A. If $\mathcal{AM}$ not corrupted, $x \notin \mathbf{S}[t]$ and $\mathsf{VerStatus}(\mathsf{in}, a_t, x, w) = 1$, output $\perp$ to $\mathcal{P}$ and halt. (This ensures collision-freeness.) Otherwise, continue.
         B. Set $\phi = \mathsf{VerStatus}(\mathsf{in}, a_t, x, w)$.
      iii. If $\mathsf{stts} = \mathsf{out}$:
         A. If $\mathcal{AM}$ not corrupted, $x \in \mathbf{S}[t]$ and $\mathsf{VerStatus}(\mathsf{out}, a_t, x, w) = 1$, output $\perp$ to $\mathcal{P}$ and halt. (This ensures negative collision-freeness.) Otherwise, continue.
         B. Set $\phi = \mathsf{VerStatus}(\mathsf{out}, a_t, x, w)$.
   (b) Otherwise, set $\phi = \mathsf{VerStatus}'(\mathsf{stts}, a, x, w)$.
   (c) Output $(\mathsf{VERIFIED}, sid, \mathsf{stts}, a, \mathsf{VerStatus}', x, w, \phi)$ to $\mathcal{P}$.
2. VERGEN: Upon getting $(\mathsf{VERGEN}, sid, S, a, v, \mathsf{VerGen}')$ from any party $\mathcal{P}$ ...
   (a) Set $\phi = \mathsf{VerGen}'(S, a, v)$.
   (b) Output $(\mathsf{VERIFIED}, sid, S, a, v, \mathsf{VerGen}', \phi)$ to $\mathcal{P}$.
3. $\mathsf{VERUPDATE}$: Upon getting $(\mathsf{VERUPDATE}, sid, \mathsf{Op}, a, a', x, v_t, \mathsf{VerUpdate}')$ from any party $\mathcal{P}$ ...
   (a) Set $\phi = \mathsf{VerUpdate}'(\mathsf{Op}, a, a', x, v_t)$.
   (b) Output $(\mathsf{VERIFIED}, sid, \mathsf{Op}, a, a', x, v_t, \mathsf{VerUpdate}', \phi)$ to $\mathcal{P}$.

Figure 2·4: Ideal Functionality $\mathcal{F}_{\mathsf{ACC}}$: Third Parties Interfaces

to be only Add or only Del we could make it additive or subtractive instead of dynamic; by restricting stts to be only in or only out we could make it positive or negative instead of universal. Figure 2·3 describes the ideal functionality interfaces for the accumulator manager and witness holders; Figure 2·4 describes the interfaces for third parties.

We use color coding to describe different types of accumulators within the same functionality. If the ideal functionality is limited to the black text, it describes a positive additive accumulator. Actions that are present only in subtractive accumulators are colored green. Actions that are present only in negative accumulators are colored blue. Finally, actions that are present only in strong accumulators are colored pink; actions *not* present in strong accumulators are colored orange.

We use $\mathcal{F}_{\mathsf{ACC}}$ to refer to the universal dynamic accumulator functionality. We add Add, Del, in and out to the subscript to denote additive, subtractive, positive and negative accumulators, respectively. We add other parameters to the subscript (e.g. 'STRONG') to denote other properties.

2.3.1 Modeling Decentralized Management

If the accumulator is strong, it may make sense to allow anyone to perform an accumulator update, instead of restricting the ability to perform such updates to the accumulator manager. We model this by making a few changes to the functionality. First, the GEN, UPDATE and $\mathsf{WITCREATE}$ interfaces of the ideal functionality no longer only accept invocations by $\mathcal{AM}$. Additionally, instead of having a strict ordering of update operations, we might allow parties to perform an update on any accumulator state, resulting in a tree of states. The functionality will be modified to perform the appropriate checks and record-keeping.

2.3.2 Modeling Non-Adaptive Soundness

We model non-adaptive soundness (Section 2.2.2) by making two simple changes to the ideal functionality. First, when sending the GEN command to the ideal functionality (in Step 1 of Figure 2·3), the accumulator manager $\mathcal{AM}$ is expected to provide a set of all elements that will ever be added or deleted. (This can be done e.g. by providing a PRF seed.) Second, if even one element outside of that set is added or deleted, nothing is guaranteed; the functionality simply runs the algorithms it was given, without performing any checks.

2.3.3 Adding Privacy Properties

Our ideal functionality as stated in Figures 2·3 and 2·4 does not make any attempt to hide anything about the accumulated set from any accumulator user. In this section, we discuss how we add such privacy properties to the ideal functionality.

Add-Delete Unlinkability

In certain scenarios it is desirable that an adversary should not be able to link an addition of an element to a deletion of the same element later on. Such a property is

relevant when accumulators are used as an anonymous revocation mechanism where the revocation information should not allow anyone to determine that the user revoked just now was the user who joined two hours ago, and not the user who joined four hours ago (Baldimtsi et al., 2017). We do not formally model add-delete unlinkability; instead, we define a stronger property which we call *hiding update-message (HUM).*

Hiding Update-Message (HUM)

Informally, an accumulator is *hiding update-message*, or *HUM*, if given all of the update messages produced in the course of an execution, it is impossible to tell whether one specific update message corresponds to the addition / deletion of element x_0 or element x_1 for $x_0, x_1 \in D$.

We can incorporate HUM into our ideal functionality by placing limitations on the algorithm Update provided by the ideal world adversary. We require Update to consist of two sub-algorithms: one sub-algorithm — Update_1 — which receives no input at all except for randomness, and produces the update message; and a second sub-algorithm — Update_2 — which can receive state from Update_1 as well as all of the other inputs typically provided to Update, and produces all the other outputs of Update. This forces update messages to reveal nothing about the added / deleted element.

Note that this modification is very strong, since it forces the update messages to *statistically* hide the elements; constructions where the elements are only computationally hidden would not meet this definition. This modification trivially implies the add-delete unlinkability property described above, since update messages now contain no information at all about the elements.

Remark 4. *We clearly need to withhold x from Update_1, in order to guarantee that the update message does not reveal x. However, we could consider allowing Update_1 to see the other inputs to Update. This would not work because if we give Update_1 access*

to the accumulator a or the auxiliary value m, then the update message it produces might contain arbitrary information about the set of elements accumulated prior to the current operation. In particular, the update message might reveal which elements were added / deleted previously, breaking the HUM property.

Zero-Knowledge

Ghosh *et al.* (Ghosh et al., 2016) define the notion of a *zero-knowledge accumulator*, which requires that accumulator and witness values reveal nothing about the accumulated set (other than the element to which the witness corresponds). We can incorporate ZK by placing limitations on the Update and $\mathsf{WitCreate}$ algorithms provided by the ideal world adversary, just like we did for the HUM property. We can require each algorithm to consist of two sub-algorithms: one which does not require any set-dependent inputs and produces the accumulator and witness values (as necessary), and a second sub-algorithm (which can receive state from the first) which produces all other values.

2.3.4 Discussion: Incorrect Accumulator and Witness Values

If an incorrect accumulator value (or verification algorithm $\mathsf{VerStatus}'$) is provided to the verification interface, we allow the party making the query to control the verification verdict, via $\mathsf{VerStatus}'$. This models the fact that any party can issue verification queries for accumulator values of their choice — for instance, for accumulator values which they may have generated themselves, and for which they control the accumulated set.

If an incorrect witness for a member element is provided to the verification interface, we allow the ideal world adversary to control the verification verdict (via the algorithm $\mathsf{VerStatus}$ it provides during the generation phase). This models the fact that we only require the ideal world adversary to be unable to come up with a witness for a *non*-member (or a non-membership witness for a member); we do not

require that an adversary be unable to come up with a witness for a member (or a non-membership witness for a non-member). For instance, it may be possible to modify valid witnesses to obtain other witnesses for the same element. Note also that multiple witnesses can be generated for the same element by means of the $\mathsf{WitCreate}$ interface.

2.4 Equivalence Argument

Like Canetti (Canetti, 2004), we prove that satisfying our UC definition for dynamic universal accumulators is the same as satisfying the classical definition.[4]

Theorem 1. *Let* $\Pi_{\mathsf{ACC}} = (\mathsf{Gen}, \mathsf{Update}, \mathsf{WitCreate}, \mathsf{WitUp}, \mathsf{VerStatus})$ *be a universal dynamic accumulator scheme, and let* $\mathsf{VerStatus}$ *be deterministic. Then* Π_{ACC} *securely realizes* $\mathcal{F}_{\mathsf{ACC}}$ *if and only if* Π_{ACC} *satisfies Definitions 7, 2 and 3.*

Proof. Our proof follows the structure of the proof of Canetti (Canetti, 2004) (pages 12-14).

1. We start by assuming that Π_{ACC} does not satisfy Definitions 7, 2 and 3. We then show that Π_{ACC} also does not securely realize $\mathcal{F}_{\mathsf{ACC}}$. To do this, we build an environment $\mathcal{Z}$ and an adversary $\mathcal{A}_{\mathsf{Real}}$ such that for any simulator $\mathcal{SIM}$, $\mathcal{Z}$ can distinguish between interacting with $\mathcal{A}_{\mathsf{Real}}$ and Π_{ACC}, and interacting with $\mathcal{SIM}$ and $\mathcal{F}_{\mathsf{ACC}}$. Like the environment of Canetti (Canetti, 2004), our environment does not corrupt any parties, and does not send any messages to the adversary. Because all accumulator operations are non-interactive, meaning that they are run locally by individual parties, no messages are exchanged in the real world. So, the adversary $\mathcal{A}_{\mathsf{Real}}$ is never activated.
   (a) Assume Π_{ACC} is not correct (i.e. does not satisfy Definition 7). That is, there exists a security parameter λ, an initial set $S_0 \subseteq D$, a value $x \in D$, an operation $\mathsf{Op} \in \{\mathsf{Add}, \mathsf{Del}\}$ (with $\mathsf{stts} = \mathsf{in}$ if $\mathsf{Op} = \mathsf{Add}$ and $\mathsf{stts} = \mathsf{out}$ if $\mathsf{Op} = \mathsf{Del}$) and a list of values $[(y_1, \mathsf{Op}_1), \ldots, (y_{t_x-1}, \mathsf{Op}_{t_x-1})]$, $[(y_{t_x+1}, \mathsf{Op}_{t_x+1}), \ldots, (y_t, \mathsf{Op}_t)]$, where
      - $y_i \in D$ and $\mathsf{Op}_i \in \{\mathsf{Add}, \mathsf{Del}\}$ for $i \in [1, \ldots, t_x - 1, t_x + 1, \ldots, t]$;

[4] This proof also implies that satisfying our UC definition for additive or subtractive, positive or negative accumulators is the same as satisfying the classical definition; however, it does not imply anything for strong accumulators. We leave that up to future work.

- If $\mathsf{Op} = \mathsf{Add}$, then (x, Del) does not appear in $[(y_{t_x+1}, \mathsf{Op}_{t_x+1}), \ldots, (y_t, \mathsf{Op}_t)]$; and
- If $\mathsf{Op} = \mathsf{Del}$, then (x, Add) does not appear in $[(y_{t_x+1}, \mathsf{Op}_{t_x+1}), \ldots, (y_t, \mathsf{Op}_t)]$,

such that with non-negligible probability, the honestly-produced witness for x against accumulator a_t will not verify.

Our environment $\mathcal{Z}$ will send the following commands to some party $\mathcal{AM}$, where sid encodes the identity of $\mathcal{AM}$:

- (GEN, sid, S_0),
- $(\mathsf{UPDATE}, sid, \mathsf{Op}_1, y_1), \ldots, (\mathsf{UPDATE}, sid, \mathsf{Op}_{t_x-1}, y_{t_x-1})$,
- $(\mathsf{UPDATE}, sid, \mathsf{Op}, x)$, and
- $(\mathsf{UPDATE}, sid, \mathsf{Op}_{t_x+1}, y_{t_x+1}), \ldots, (\mathsf{UPDATE}, sid, \mathsf{Op}_t, y_t)$.

As a result of the third step, $\mathcal{Z}$ will learn a_{t_x} and $w^x_{t_x}$. As a result of the fourth step, $\mathcal{Z}$ will learn a_t and $t - t_x$ update messages $(\mathsf{upmsg}_{t_x+1}, \ldots, \mathsf{upmsg}_t)$. It then sends $(\mathsf{WITUP}, \mathsf{stts}, sid, a_{t_x}, a_t, x, w^x_{t_x}, (\mathsf{upmsg}_{t_x+1}, \ldots, \mathsf{upmsg}_t))$ to some party $\mathcal{H}$ (where possibly $\mathcal{H} = \mathcal{AM}$), and receives w^x_t back. Finally, it sends $(\mathsf{VERSTATUS}, sid, \mathsf{stts}, a_t, \mathsf{VerStatus}' = \mathsf{VerStatus}(\cdot, \cdot, \cdot, \cdot), x, w^x_t)$ to some party $\mathcal{P}$ (which may be the same party or not). $\mathcal{Z}$ outputs the returned verdict ϕ.

In the real world, ϕ will be 0 with non-negligible probability according to our assumption.

In the ideal world, if no error messages are returned, ϕ will always be 1, since in WitUp, we will always hit Item 4(b)i or 4(b)ii of Figure 2·3, and there the first three listed conditions will be satisfied.

(b) Assume Π_{ACC} is not creation-correct (i.e. does not satisfy Definition 2). $\mathcal{Z}$ can distinguish between the real and ideal worlds in a way very similar to that described above.

(c) Assume Π_{ACC} is not collision-free (i.e. does not satisfy Definition 3). That is, there exists an adversary $\mathcal{A}_{\mathsf{ColFree}}$ that can forge a (non-)membership witness for a non-member (or member, respectively) x with non-negligible probability. Our $\mathcal{Z}$ will use $\mathcal{A}_{\mathsf{ColFree}}$ to generate inputs for $\mathcal{AM}$. Having received x^*, w^* from $\mathcal{A}_{\mathsf{ColFree}}$, $\mathcal{Z}$ will compute ϕ_{in} by calling $(\mathsf{VERSTATUS}, sid, \mathsf{in}, a_t, x^*, w^*)$, and ϕ_{out} by calling $(\mathsf{VERSTATUS}, sid, \mathsf{out}, a_t, x^*, w^*)$. $\mathcal{Z}$ will then output 1 if x^* was in the accumulated set and $\phi_{out} = 1$ or if x^* was not in the accumulated set and $\phi_{in} = 1$, and will output 0 otherwise.

In the real world, if $\mathcal{A}_{\mathsf{ColFree}}$ met the collision-freeness win conditions, $\mathcal{Z}$ will output 1 with non-negligible probability according to our assumption. In the ideal world, both ϕ_{in} and ϕ_{out} will always be 0 or $\perp$, since we will satisfy the first two conditions in Item 1(a)iiA (or Item 1(a)iiiA, if $\mathsf{stts} = \mathsf{out}$) of VERSTATUS in Figure 2·4. If the third condition is satisfied too, $\perp$ will be returned. If it is not, 0 will be returned, as a result of Item 1(a)iiB (or Item 1(a)iiiB, if $\mathsf{stts} = \mathsf{out}$) in Figure 2·4.

2. We now prove the other direction. Assume that Π_{ACC} does not securely realize $\mathcal{F}_{\mathsf{ACC}}$. That is, there exists an adversary $\mathcal{A}_{\mathsf{Real}}$ such that for any simulator $\mathcal{SIM}$, there exists an environment $\mathcal{Z}$ that can distinguish between interacting with $\mathcal{A}_{\mathsf{Real}}$ and Π_{ACC}, and interacting with $\mathcal{SIM}$ and $\mathcal{F}_{\mathsf{ACC}}$. We show that if that is the case, Π_{ACC} must also violate Definition 7, 2 or 3. We pick a simulator $\mathcal{SIM}$ that proceeds as follows, running an internal copy of $\mathcal{A}_{\mathsf{Real}}$:
   - Inputs from $\mathcal{Z}$ is forwarded to $\mathcal{A}_{\mathsf{Real}}$. Outputs from $\mathcal{A}_{\mathsf{Real}}$ is forwarded to $\mathcal{Z}$.
   - $\mathcal{SIM}$ handles corruptions according to the standard corruption model (Canetti, 2001).
   - Upon receiving (GEN, sid) from $\mathcal{F}_{\mathsf{ACC}}$, $\mathcal{SIM}$ sends the actual accumulator algorithms back as $(\mathsf{GEN}, sid, (\mathsf{Gen}, \mathsf{Update}, \mathsf{WitCreate}, \mathsf{WitUp}, \mathsf{VerStatus}))$.

This simulator guarantees that the real and ideal worlds will be distributed identically, *unless* one of the following causes $\mathcal{F}_{\mathsf{ACC}}$ to return $\perp$:

- In Update, $\mathcal{F}_{\mathsf{ACC}}$ hits Item 2(d)i or 2(e)i of Figure 2·3. If this happens, correctness (Definition 7) is violated.
- In WitCreate, $\mathcal{F}_{\mathsf{ACC}}$ hits Item 3b or 3c of Figure 2·3. If this happens, creation-correctness (Definition 2) is violated.
- In VerStatus, $\mathcal{F}_{\mathsf{ACC}}$ hits Item 1(a)iiA or 1(a)iiiA of Figure 2·4. If this happens, collision-freeness (Definition 3) is violated.
- In WitUp, $\mathcal{F}_{\mathsf{ACC}}$ hits Item 4(b)i or 4(b)ii of Figure 2·3. If this happens, either correctness or creation-correctness is violated.

In order for $\mathcal{Z}$ to distinguish between the real and ideal worlds, one of the above must happen with non-negligible probability, and thus either Definition 7, 2 or 3 must be violated with non-negligible probability.

□

We can modify the theorem and proof to also prove equivalence between classical

and UC definitions for strong accumulators.

Corollary 1. *Let* $\Pi_{\mathsf{ACC}} = (\mathsf{Gen}, \mathsf{Update}, \mathsf{WitCreate}, \mathsf{WitUp}, \mathsf{VerStatus}, \mathsf{VerGen}, \mathsf{VerUpdate})$ *be a strong universal dynamic accumulator scheme, and let* VerStatus, VerGen *and* VerUpdate *be deterministic. Then* Π_{ACC} *securely realizes* $\mathcal{F}_{\mathsf{ACC},STRONG}$ *if and only if* Π_{ACC} *satisfies Definitions 7, 2 and 4.*

Proof. The proof is very similar to that of Theorem 1 above, with a few changes. The changes are in Steps 1c and 2 of the proof.

In Step 1c of the proof above, instead of calling $\mathcal{A}_{\mathsf{ColFree}}$, we call $\mathcal{A}_{\mathsf{Strength}}$ which runs Gen and Update itself. The environment $\mathcal{Z}$ computes its output exactly as before. In the ideal world, both ϕ_{in} and ϕ_{out} will always be 0 or $\perp$, since we will satisfy the first condition in Item 1(a)iiA (or Item 1(a)iiiA, if $\mathsf{stts} = \mathsf{out}$) of VERSTATUS (ignoring the condition that $\mathcal{AM}$ is not corrupted, which does not apply for a strong accumulator). If the third condition is satisfied too, $\perp$ will be returned. If it is not, 0 will be returned, as a result of Item 1(a)iiB (or 1(a)iiiB, if $\mathsf{stts} = \mathsf{out}$) of Figure 2·4.

In Step 2 of the proof above, $\mathcal{SIM}$ includes VerGen and VerUpdate in the list of algorithms it sends to the ideal functionality. Then, in the list of things that might cause $\mathcal{F}_{\mathsf{ACC},STRONG}$ to return $\perp$, we replace the third bullet with the following:

- In VerStatus, $\mathcal{F}_{\mathsf{ACC}}$ hits Item 1(a)iiA or 1(a)iiiA of Figure 2·4. If this happens, strength (Definition 4) is violated.

We also add the following:

- In Gen, $\mathcal{F}_{\mathsf{ACC},STRONG}$ returns $\perp$ at Item 1g of Figure 2·3. If this happens, strength correctness (Definition 5) is violated.
- In Update, $\mathcal{F}_{\mathsf{ACC},STRONG}$ returns $\perp$ at Item 2f of Figure 2·3. If this happens, strength correctness (Definition 5) is violated.

□

2.5 Appendix A: Universally Composable Signatures

In this appendix (specifically, in Figures 2·5 and 2·6), we describe the two digital signature ideal functionalities described by Canetti (Canetti, 2001; Canetti, 2004). The first does not require the simulator to provide the signing and verification algorithms explicitly at key generation time; the second does. Both ideal functionalities require the verifier to provide the verification key (or verification algorithm) when using the

1. **Key Generation:** Upon getting (KEYGEN, sid) from a party Signer ...
   (a) If this is not the first KeyGen command, ignore this command. Otherwise, continue.
   (b) If sid does not encode Signer's identity, ignore this command. Otherwise, continue.
   (c) Initialize an empty map $\mathbf{W}$.
   (d) Send (KEYGEN, sid) to Adversary $\mathcal{A}_{\mathsf{Ideal}}$.
   (e) Get (VERKEY, sid, vk) from Adversary $\mathcal{A}_{\mathsf{Ideal}}$.
   (f) Record vk.
   (g) Send (VERKEY, sid, vk) to Signer.
2. **Signature Generation:** Upon getting (SIGN, sid, x) from a party Signer ...
   (a) Verify that sid encodes Signer's identity. If not, ignore this command. Otherwise, continue.
   (b) Send (SIGN, sid, x) to Adversary $\mathcal{A}_{\mathsf{Ideal}}$.
   (c) Get ($\mathsf{SIGNATURE}, sid, x, \sigma$) from Adversary $\mathcal{A}_{\mathsf{Ideal}}$.
   (d) Verify that $(x, \sigma) \notin \mathbf{W}$ or $\mathbf{W}[(x, \sigma)] = 1$. If not, send $\perp$ to Signer and halt. Otherwise, continue.
   (e) If $(x, \sigma) \notin \mathbf{W}$, record $\mathbf{W}[(x, \sigma)] = 1$.
   (f) Output ($\mathsf{SIGNATURE}, sid, x, \sigma$) to Signer.
3. **Signature Verification:** Upon getting ($\mathsf{VERIFY}, sid, x, \sigma, vk$) from a party $\mathsf{Verifier}$ ...
   (a) Send ($\mathsf{VERIFY}, sid, x, \sigma, vk$) to Adversary $\mathcal{A}_{\mathsf{Ideal}}$.
   (b) Get ($\mathsf{VERIFIED}, sid, x, \sigma, vk, \phi$) from Adversary $\mathcal{A}_{\mathsf{Ideal}}$.
   (c) If $(x, \sigma) \in \mathbf{W}$: let $\phi' = \mathbf{W}[(x, \sigma)]$.
   (d) Else:
      i. If the signer is not corrupted, vk is the recorded public key, and $(x, \sigma) \notin \mathbf{W}$, set $\phi' = 0$.
      ii. Else, let $\phi' = \phi$.
      iii. Record $\mathbf{W}[(x, \sigma)] = \phi'$.
   (e) Output ($\mathsf{VERIFIED}, sid, x, \sigma, vk, \phi'$) to $\mathsf{Verifier}$.

Figure 2·5: Ideal Functionality for Digital Signatures (Canetti, 2004)

verification interface. This models the fact that the verifier might be misinformed about the verification key if a PKI is not available.

2.6 Appendix B: Universally Composable Zero-Knowledge

In this appendix (in Figure 2·7) we recall the ideal functionality $\mathcal{F}_{ZK}$ from (Canetti, 2001) which is parameterized by a binary relation R that takes in an element x and a witness w. It expects a single input ($\mathsf{PROVE}, sid, x, w$) from Prover (where sid encodes the identities of Prover and $\mathsf{Verifier}$). If $R(x, w) = 1$ then $\mathcal{F}_{ZK}$ will output ($\mathsf{VERIFIED}, sid, x$) to $\mathsf{Verifier}$[5].

2.7 Appendix C: The RSA Accumulator

In this appendix (in Figures 2·8, 2·9 and 2·10), we review the RSA dynamic universal accumulator, which has been shown to meet the classical accumulator definitions. By

[5] Corruption is also modeled; if Prover is corrupt, the adversary learns the prover's witness w.

1. **Key Generation:** Upon getting (KEYGEN, sid) from a party Signer ...
   (a) If this is not the first KeyGen command, ignore this command. Otherwise, continue.
   (b) If sid does not encode Signer's identity, ignore this command. Otherwise, continue.
   (c) Initialize an empty list $\mathbf{W}$ of signed messages.
   (d) Send (KEYGEN, sid) to Adversary $\mathcal{A}_{\mathsf{Ideal}}$.
   (e) Get $(\mathsf{ALGORITHMS}, sid, \mathsf{Sign}, \mathsf{Verify})$ from Adversary $\mathcal{A}_{\mathsf{Ideal}}$, where Sign is a polynomial-time algorithm and Verify is a polynomial-time *deterministic* algorithm.
   (f) Send $(\mathsf{ALGORITHMS}, sid, \mathsf{Verify})$ to Signer.
2. **Signature Generation:** Upon getting (SIGN, sid, x) from a party Signer ...
   (a) Verify that sid encodes Signer's identity. If not, ignore this command. Otherwise, continue.
   (b) Let $\sigma = \mathsf{Sign}(x)$.
   (c) Verify that $\mathsf{Verify}(x, \sigma) = 1$. If not, send $\perp$ to Signer and halt. Otherwise, continue.
   (d) Output $(\mathsf{SIGNATURE}, sid, x, \sigma)$ to Signer.
   (e) Record x in $\mathbf{W}$.
3. **Signature Verification:** Upon getting $(\mathsf{VERIFY}, sid, x, \sigma, \mathsf{Verify}')$ from a party Verifier ...
   (a) If $\mathsf{Verify}' = \mathsf{Verify}$, the signer is not corrupted, $\mathsf{Verify}(x, \sigma) = 1$ and $x \notin \mathbf{W}$, send $\perp$ to signer and halt. (This violates soundness.) Otherwise, continue.
   (b) $\phi = \mathsf{Verify}'(x, \sigma)$.
   (c) Output $(\mathsf{VERIFIED}, sid, x, \sigma, \mathsf{Verify}', \phi)$ to Verifier.

Figure 2·6: Ideal Functionality for Digital Signatures with Algorithms Provided by the Adversary (Canetti, 2001) (2005 version)

$\mathcal{F}_{ZK}^{R}$ is parameterized by a binary relation R. It proceeds as follows.

1. Upon getting $(\mathsf{PROVE}, sid, x, w)$ from Prover, Ignore it unless $sid = (\mathsf{Prover}, \mathsf{Verifier}, sid')$ for some Verifier. Next, if $R(x, w) = 1$, send the output $(\mathsf{VERIFIED}, sid, x)$ to Verifier; otherwise, do nothing. From now on, ignore PROVE inputs.
2. Upon getting $(\mathsf{CORRUPTPROVER}, sid)$ from Adversary $\mathcal{A}_{\mathsf{Ideal}}$, send w to Adversary $\mathcal{A}_{\mathsf{Ideal}}$. If Adversary $\mathcal{A}_{\mathsf{Ideal}}$ now provides a value (x', w') such that $R(x', w')$ holds, and no output was yet sent to Verifier, send $(\mathsf{VERIFIED}, sid, x')$ to Verifier.

Figure 2·7: Ideal Functionality for Zero Knowledge (Canetti, 2001) (2005 version)

Theorem 1, it follows that this accumulator also meets our UC definition.

The RSA accumulator is described across several papers. It was introduced by Benaloh and de Mare (Benaloh and de Mare, 1994), augmented with dynamism by Camenisch and Lysyanskaya (Camenisch and Lysyanskaya, 2002), and with universality by Li, Li and Xue (Li et al., 2007a).

$\mathsf{Gen}(1^\lambda, S_0)$:

1. Select two λ-bit safe primes $p = 2p' + 1$ and $q = 2q' + 1$ where p' and q' are also prime, and let $n = pq$. (Consider n to be public knowledge from hereon out; it is actually a part of the accumulator value a, but for simplicity we will not refer to it as such.)
2. Let $sk = p'q'$.
3. Select a random integer $a' \leftarrow \mathbb{Z}_n^*$.
4. Let $a_0 = (a')^2 \bmod n$. (Like n, this initial accumulator a_0 will be part of all future accumulator values, but for simplicity we will not refer to it as such.)
5. Let $m_0 = S_0$ be the set of member elements.
6. Return (sk, a_0, m_0).

$\mathsf{Update}(\mathsf{Op}, sk, a_t, m_t, x)$:

1. Let $m_{t+1} = m_t$.
2. Check that $x \in D$ (that is, that x is an odd prime). If not, return $\perp$.
3. If $\mathsf{Op} = \mathsf{Add}$:
   (a) If $x \in m_{t+1}$ (x is already a member):
      i. Let $w_{t+1}^x = \mathsf{WitCreate}(\mathsf{in}, sk, a_t, m_t, x)$.
      ii. Let $a_{t+1} = a_t$.
      iii. Let $\mathsf{upmsg}_t = (\mathsf{Add}, \perp)$.
   (b) Else (x is not yet a member):
      i. Add x to m_{t+1}.
      ii. Let $w_{t+1}^x = a_t$.
      iii. Let $a_{t+1} = a_t^x \bmod n$.
      iv. Let $\mathsf{upmsg}_{t+1} = (\mathsf{Add}, x)$.
   (c) Return $(a_{t+1}, m_{t+1}, w_{t+1}^x, \mathsf{upmsg}_{t+1})$.
4. If $\mathsf{Op} = \mathsf{Del}$:
   (a) If $x \notin m_t$ (x is already a non-member):
      i. Let $a_{t+1} = a_t$.
      ii. Let $\mathsf{upmsg}_{t+1} = (\mathsf{Del}, \perp)$.
   (b) Else (x is a member):
      i. Remove x from m_{t+1}.
      ii. Let $a_{t+1} = a_t^{x^{-1} \bmod sk} \bmod n$.
      iii. Let $\mathsf{upmsg}_{t+1} = (\mathsf{Del}, (a_{t+1}, x))$.
   (c) Let $w_{t+1}^x = \mathsf{WitCreate}(\mathsf{out}, sk, a_{t+1}, m_{t+1}, x)$.
   (d) Return $(a_{t+1}, m_{t+1}, w_{t+1}^x, \mathsf{upmsg}_{t+1})$.

$\mathsf{WitCreate}(\mathsf{stts}, sk, a_t, m_t, x)$:

1. If $x \notin D$ or $x \notin m_t$, FAIL.
2. If $\mathsf{stts} = \mathsf{in}$:
   (a) Let $w_t^x = a_t^{x^{-1} \bmod p'q'} \bmod n$, and return w_t^x.
3. If $\mathsf{stts} = \mathsf{out}$:
   (a) Compute Bezout coefficients α, β such that $\alpha(\prod_{y \in m_{t+1}} y) + \beta x = 1$. (Given that $x \neq y$ for all $y \in m_{t+1}$, since both x and all y are prime, such α and β are guaranteed to exist and be efficiently computable.)
   (b) Let $r_t^x = \alpha$, and $s_t^x = a_0^{-\beta} \bmod n$.
   (c) Return $w_t^x = (r_t^x, s_t^x)$.

   Verifying that the witness holds, we have

$$a_t^{r_t^x} = a_0^{r_t^x \prod_{y \in m_{t+1}} y} = a_0^{\alpha \prod_{y \in m_{t+1}} y} = a_0^{1 - \beta x} = a_0 (a_0^{-\beta})^x = a_0 (s_t^x)^x (\bmod n).$$

Figure 2·8: RSA Accumulator Manager Algorithms

WitUp(stts, x, w_t^x, upmsg$_{t+1}$):

1. Parse (Op, upmsg′) = upmsg.
2. If stts = in:
   (a) If Op = Add:
      i. Parse y = upmsg′.
      ii. If $y = \perp$: let $w_{t+1}^x = w_t^x$.
      iii. Else: let $w_{t+1}^x = (w_t^x)^y \bmod n$.
      iv. Return w_{t+1}^x.
   (b) If Op = Del:
      i. If upmsg′ $= \perp$: let $w_{t+1}^x = w_t^x$.
      ii. Else:
         A. Parse (a_{t+1}, y) = upmsg′.
         B. Compute Bezout coefficients α, β such that $\alpha x + \beta y = 1$. (Given that $x \neq y$, since both x and y are prime, such α and β are guaranteed to exist.)
         C. Let $w_{t+1}^x = (w_t^x)^\beta a_{t+1}^\alpha \bmod n$.
      iii. Return w_{t+1}^x.
3. If stts = out:
   (a) Parse $(r_t^x, s_t^x) = w_t^x$.
   (b) If Op = Add:
      i. Parse y = upmsg′.
      ii. If $y = \perp$: let $w_{t+1}^x = w_t^x$.
      iii. Else:
         A. Find α and β such that $\alpha y + \beta x = 1$.
         B. $r_{t+1}^x = \alpha r_t^x \bmod x$.
         C. Find a value v such that $r_{t+1}^x y = r_t^x + vx$.
         D. $s_{t+1}^x = s_t^x a_t^v \bmod n$.
         E. Let $w_{t+1}^x = (r_{t+1}^x, s_{t+1}^x)$.
         F. Return w_{t+1}^x.

      Verifying that the new witness holds, we have

$$a_{t+1}^{r_{t+1}} = a_t^{r_{t+1} y} = a_t^{r_t + rx} = a_t^{rx} a_t^{r_t}$$

$$= a_t^{rx} s^x a_0 = (a_t^r s)^x a_0 = s_{t+1}^x a_0 (\mathrm{mod} n).$$

   (c) Op = Del:
      i. If upmsg′ $= \perp$: let $w_{t+1}^x = w_t^x$.
      ii. Else:
         A. Find a value v such that $0 \leq r_t^x y - vx < 2^\lambda$ (this is efficiently doable).
         B. $r_{t+1}^x = r_t^x y - vx$.
         C. $s_{t+1}^x = s_t^x a_{t+1}^x a_0$.
         D. Let $w_{t+1}^x = (r_{t+1}^x, s_{t+1}^x)$.
         E. Return w_{t+1}^x.

      Verifying that the new witness holds, we have

$$a_{t+1}^{r_{t+1}} = a_{t+1}^{r_t y - rx} = s_t^x a_0 a_{t+1}^{-rx} = (s_t a_{t+1}^{-1})^x a_0 = s_{t+1}^x a_0 (\mathrm{mod} n).$$

Figure 2·9: RSA Witness Holder Algorithms

VerStatus(stts, a_t, x, w_t^x):

1. If stts = in:
   (a) Return 1 if $a_t = (w_t^x)^x \bmod n$.
   (b) Return 0 otherwise.
2. If stts = out:
   (a) Parse $(r_t^x, s_t^x) = w_t^x$.
   (b) Return 1 if $a_t^{r_t^x} = (s_t^x)^x a_0 (\mathrm{mod} n)$.
   (c) Return 0 otherwise.

Figure 2·10: RSA Verifier Algorithms

Chapter 3

Accumulators with Applications to Anonymity-Preserving Revocation

The contents of this section are a collaboration with Foteini Baldimtsi, Jan Camenisch, Maria Dubovitskaya, Anna Lysyanskaya, Leonid Reyzin and Kai Samelin (Baldimtsi et al., 2017). The last sections (dealing with the details of integrating the $\mathsf{Braavos}$ accumulator into anonymous credentials, and with implementation) are omitted, as they are largely not my work and are not necessary for understanding the earlier sections. This chapter also contains some contributions from (Baldimtsi et al.,), most of which was covered in the previous chapter. In particular, I choose to describe the $\mathsf{Braavos}'$ accumulator in this chapter instead of the previous one, since it is closely related to the $\mathsf{Braavos}$ accumulator introduced here.

3.1 Introduction

As explained in the previous chapter, a natural use-case for accumulators is anonymous credentials. In this chapter, we introduce a number of accumulators which are ideally suited for use in anonymous credential systems. There are a number of properties which make an accumlator suited for use in such systems:

1. They should not require witness (or *credential*) holders to update their witnesses too often; ideally, an addition of a new element should require no modification to existing witnesses.
2. They should support efficient zero-knowledge proofs of element and witness

knowledge.

3. The update messages required for witness maintenance should not reveal too much information (that is, the accumulator should either be add-revoke unlinkable or update message hiding (HUM)).

The accumulators we introduce in this chapter — Braavos, Braavos′ and BraavosB— all share the first two properties listed above. Braavos and Braavos′ also achieve the last one. We build these accumulators in an intuitive, modular way, by combining other accumulators.

3.1.1 Outline

In Section 3.2, we describe how to combine accumulators to obtain new ones with better functionality and security. In Section 3.3, we introduce the Braavos and Braavos′ accumulators; in Section 3.4, we introduce the BraavosB accumulator.

3.2 Modular Accumulator Constructions

In this section, we introduce the idea of combining different accumulators to obtain new accumulators with different properties. This technique can lead to the creation of more efficient accumulators, such as the Braavos and Braavos′ accumulators described in Section 3.3 and the BraavosB accumulator described in Section 3.4. We describe how this can lead to not only enhanced security and efficiency (as in the case of Braavos), but also to richer functionality. In this chapter, we refer to the definitions introduced in Section 2.2.1.

3.2.1 Leveraging Accumulators with Different Functionalities

Notice that, though the notion of a subtractive accumulator helps us draw a more complete mental picture of the accumulator space, there are conceptual equivalences that allow us to ignore subtractive accumulators from hereon out. Let S be the

accumulated set of elements, and $\overline{S}$ be the complement of that set ($\overline{S} = D \setminus S$, where D is the domain of all accumulatable elements).[1]

Notice that, conceptually,

- a positive additive accumulator of S is the same as a negative subtractive accumulator of $\overline{S}$.
- a negative additive accumulator of S is the same as a positive subtractive accumulator of $\overline{S}$.
- a universal additive accumulator of S is the same as a universal subtractive accumulator of $\overline{S}$.

Next, we discuss combining simple positive and negative accumulators to obtain universal and dynamic accumulators. An example of a simple positive accumulator is any digital signature scheme; the accumulator value is the verification key, and a membership witness for x is a signature on x (Pöhls and Samelin, 2014). Simple negative accumulators include the Merkle tree construction over ranges (Camacho et al., 2008) and the range-RSA accumulator introduced in Appendix 3.4.1.

For the purpose of this discussion, we assume that all accumulators under consideration have $D = \{0, 1\}^*$ (that is, they can can hold arbitrary elements). We also assume that all of these accumulators are used to accumulate *sets*, not *multi-sets* (that is, an element in the accumulated set is not added again unless it was previously deleted).

Figure 3·1 gives an illustration of our derivations. The proofs of the correctness and soundness of these constructions are easy exercises, and are therefore omitted. We include only the proof of soundness of construction B given its relevance to one of our constructions (it is used in Section 3.4).

[1] Note that as long as S is polynomial in size, $\overline{S}$ can be expressed as a polynomial number of ranges.

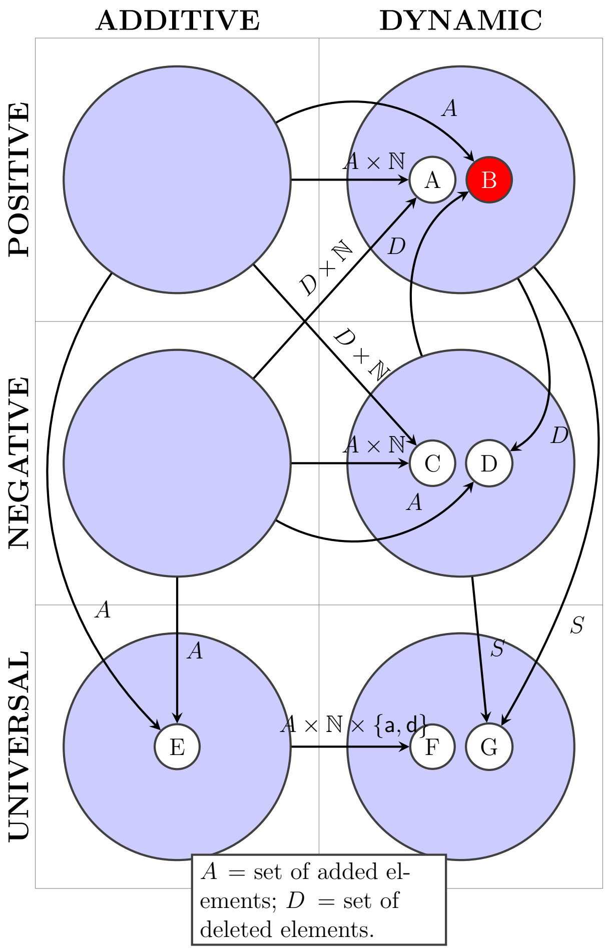


Figure 3·1: Modular accumulator derivations in terms of functionality. Large circles represent a space of accumulator constructions (e.g., the top left-most large circle contains all positive accumulators). Each dot (labeled A-G) within a large circle represents a construction of the type corresponding to the large circle. Arrows denote the modular usage of accumulators of the type corresponding to their start-point to build an accumulator of the type corresponding to their end-point. Arrow labels denote the types of objects being accumulated by the start-point accumulator. S denotes the set of current members, $\mathbb{N}$ the set of natural numbers (used for indexing), and $\{\mathsf{a}, \mathsf{d}\}$ the set of possible actions ('add' or 'delete').

Adding Dynamism We can build a *dynamic positive accumulator* ACC (construction A in Figure 3·1) out of a positive accumulator $\mathsf{ACC_P}$ and a negative accumulator $\mathsf{ACC_N}$ by adding indexing to the elements. When a new element x is added to ACC, the pair $(x, 1)$ is added to $\mathsf{ACC_P}$. Then, when the element is deleted, the pair $(x, 1)$ is added to $\mathsf{ACC_N}$. Next time x is added, it is added as $(x, 2)$; each time the element is added and deleted, the index is incremented. (Notice that this requires the accumulator manager to maintain an auxiliary storage m the size of which is linear in $|A|$, where A is the set of all elements ever added.) Proving the membership of x then consists of producing an index i and proving that $((x, i) \in \mathsf{ACC_P}) \wedge ((x, i) \notin \mathsf{ACC_N})$.

Similarly, we can build a *dynamic negative accumulator* ACC (construction C in Figure 3·1) out of a positive accumulator $\mathsf{ACC_P}$ and a negative accumulator $\mathsf{ACC_N}$. However, the roles of the two accumulators are reversed; when an element x is added to ACC, (x, i) is added to $\mathsf{ACC_N}$ for the appropriate index i. When the element is deleted, (x, i) is added to $\mathsf{ACC_P}$. Proving the non-membership of x then consists of producing an index i and proving that $((x, 1) \notin \mathsf{ACC_N}) \vee (((x, i-1) \in \mathsf{ACC_P}) \wedge ((x, i) \notin \mathsf{ACC_N}))$.

Flipping the Sign of a Dynamic Accumulator There is an alternative way to build dynamic positive (or negative) accumulators; however, it assumes the existence of a dynamic negative (or positive, respectively) accumulator. We can build a dynamic positive accumulator ACC (construction B in Figure 3·1) out of a positive accumulator $\mathsf{ACC_P}$ and a dynamic negative accumulator $\mathsf{ACC_N}$ by adding all added elements to $\mathsf{ACC_P}$, and adding all deleted elements to $\mathsf{ACC_N}$. When an element that has previously been deleted is re-added, it is removed from $\mathsf{ACC_N}$. Proving the membership of x consists of proving that $(x \in \mathsf{ACC_P}) \wedge (x \notin \mathsf{ACC_N})$. The construction of a dynamic negative accumulator (construction D in Figure 3·1) out of a simple negative accumulator and a dynamic positive accumulator mirrors this one, and we will not discuss it further.

Theorem 2. *Construction B is an adaptively sound positive dynamic accumulator if* $\mathsf{ACC_P}$ *is a adaptively sound positive accumulator and* $\mathsf{ACC_N}$ *is an adaptively sound negative dynamic accumulator.*

Proof. The proof consists of a reduction to the adaptive soundness of either $\mathsf{ACC_P}$ or $\mathsf{ACC_N}$. If an adversary produces a witness for an element that is not a member of the accumulated set, then if that element was never added the adversary has succeeded in breaking the adaptive soundness of $\mathsf{ACC_P}$, and if that element was deleted the adversary has succeeded in breaking the adaptive soundness of $\mathsf{ACC_N}$. □

Instantiating Construction B In Section 3.4, we describe a communication-optimal instantiation of construction B which we call BraavosB. BraavosB uses digital signatures as the positive accumulator $\mathsf{ACC_P}$, and another new accumulator called range-RSA (described in Section 3.4.1) as the dynamic negative accumulator $\mathsf{ACC_N}$. The communication complexity of BraavosB rivals that of Braavos, which is the focal construction of this paper and is described in detail in Section 3.3. However, though (asymptotically) the communication costs of Braavos and BraavosB are equally small, Braavos has several advantages over BraavosB. One of these is that Braavos supports more efficient zero knowledge proofs of member knowledge. Another is that BraavosB requires the accumulator manager to store an amount of information linear in the number of deleted elements, while Braavos only requires the accumulator manager to store a constant amount of information.

Adding Universality Additive (construction E in Figure 3·1) and dynamic (construction G in Figure 3·1) universal accumulators can be built by combining a positive and negative accumulator of the same type in a straightforward way; both the positive and negative accumulators are used to accumulate the elements in the set.

A dynamic universal accumulator (construction F in Figure 3·1) can also be built out of an additive universal accumulator $\mathsf{ACC_U}$ in a manner similar to those used to produce constructions A and C. Each element x, when seen for the first time, is

assigned a counter $i = 1$. When x is added, a tuple of the form (x, i, a) is added to the additive universal accumulator. When x is deleted, a tuple of the form (x, i, d) is added to the additive universal accumulator, and the counter i is incremented. Proving the membership of x then consists of producing the counter i and proving that $((x, i, \mathsf{a}) \in \mathsf{ACC_U}) \wedge ((x, i, \mathsf{d}) \notin \mathsf{ACC_U})$. Proving the non-membership of x consists of proving that $((x, 1, \mathsf{a}) \notin \mathsf{ACC_U}) \vee (((x, i-1, \mathsf{d}) \in \mathsf{ACC_U}) \wedge ((x, i, \mathsf{a}) \notin \mathsf{ACC_U}))$.

3.2.2 Leveraging Less Secure Accumulators

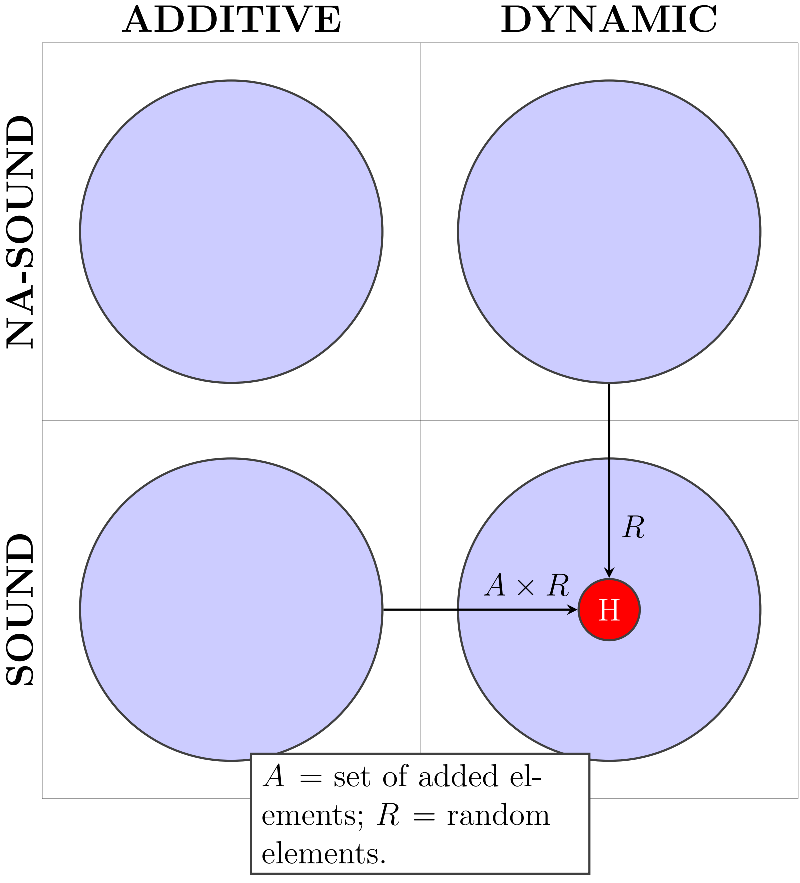


Figure 3·2: Modular accumulator derivations in terms of security. All accumulators in this diagram are positive. The notation is the same as that used in Figure 3·1; additionally, R denotes random elements. A specific (particularly efficient) instantiation of construction H is discussed further in Section 3.3.

In addition to considering combining accumulators with different functionalities, we can consider combining accumulators with different security properties. Given an (adaptively) sound positive additive accumulator ACC_A and non-adaptively sound (NA-sound) positive dynamic accumulator ACC_NA, we can build an adaptively sound dynamic accumulator ACC, as shown in Figure 3·2. [2] We call this construction "Construction H". When an element x is added, the accumulator manager selects a random element r from the domain D of ACC_NA. She then adds r to ACC_NA, and (x, r) to ACC_A. (Recall that random elements can always be safely accumulated in non-adaptively sound accumulators, since those random elements can be chosen without using any information about the accumulator.) When deleting x, the accumulator manager removes r from ACC_NA. Proving the membership of x in ACC consists of producing an r and proving that $(r \in \mathsf{ACC}_\mathsf{NA}) \wedge ((x, r) \in \mathsf{ACC}_\mathsf{A})$.

Note that, in order to support deletions, the accumulator manager must store a mapping from every element x to the corresponding r. This can be avoided by having the accumulator manager use a pseudorandom function F_s (where s is the secret pseudorandom function seed) to select an r corresponding to a given x: $r = \mathsf{F}_s(x)$. Even though this causes elements added to ACC_NA to be computed rather than chosen at random (therefore seemingly requiring adaptive soundness rather than non-adaptive soundness), non-adaptive soundness is still sufficient because of the indstinguishability of the pseudorandom and random cases.

3.3 Braavos: A Communication-Optimal Adaptively Sound Dynamic Accumulator

In this section we introduce the Braavos (Baldimtsi et al., 2017) and Braavos′ (Baldimtsi et al.,) accumulators, which are instantiations of construction H from Figure 3·2.

[2]Shamir and Tauman (Shamir and Tauman, 2001) achieve a similar goal of construct chosen message unforgeable signatures from random message unforgeable ones by using a different technique.

$\mathsf{Braavos}$ and $\mathsf{Braavos}'$ are adaptively sound positive dynamic accumulators derived from an adaptively sound positive additive accumulator $\mathsf{ACC_A}$ (instantiated as a signature scheme SIG) and a non-adaptively sound positive dynamic accumulator $\mathsf{ACC_{NA}}$ (instantiated using CLRSAB, described in Section 3.3.1).

As a recap, informally, they both work as follows. When a new element x is added, a random value r_x is chosen to correspond to it, and the pair (x, r_x) is accumulated in $\mathsf{ACC_A}$. A proof of membership for x consists of the value r_x, a proof of membership of (x, r_x) in $\mathsf{ACC_A}$ (which is simply a digital signature), and a proof of membership of r_x in $\mathsf{ACC_{NA}}$. Then, when the element x is deleted, r_x is removed from $\mathsf{ACC_{NA}}$ (so a proof of membership of r_x in $\mathsf{ACC_{NA}}$ can no longer be produced).

The difference between $\mathsf{Braavos}$ and $\mathsf{Braavos}'$ is simply that $\mathsf{Braavos}$ uses a pseudorandom function to choose the random values r_x (as $r_x = \mathsf{F}_s(x)$), whereas $\mathsf{Braavos}'$ picks a fresh truly random r upon every addition. Figure 3·3 describes the $\mathsf{Braavos}'$ algorithms.

We aim for $\mathsf{Braavos}$ and $\mathsf{Braavos}'$ to have two properties: (1) communication optimality (Camacho, 2009) (as described in Section 3.6), and (2) efficient zero knowledge proofs, as described in Section 3.3.2. Our choice of underlying adaptively sound accumulator $\mathsf{ACC_A}$ in $\mathsf{Braavos}$ is the CL signature scheme (Camenisch and Lysyanskaya, 2003), because it supports efficient zero knowledge proofs of knowledge of a signature on a committed value.

Communication optimality is achieved in both $\mathsf{Braavos}$ and $\mathsf{Braavos}'$ thanks to the use of digital signatures as $\mathsf{ACC_A}$ and careful choice of $\mathsf{ACC_{NA}}$ — update messages should only be required when deletions occur.

Because of the different mechanisms used to choose the random values r accumulated in $\mathsf{ACC_{NA}}$, $\mathsf{Braavos}$ and $\mathsf{Braavos}'$ have different advantages. $\mathsf{Braavos}$ saves on accumulator manager storage requirements, because choosing the random value r_x as

$\mathsf{Gen}(1^\lambda, S = \emptyset)$:

1. $(\mathsf{SIG}.sk, \mathsf{SIG}.a_0) \leftarrow \mathsf{SIG}.\mathsf{Gen}(1^\lambda, \emptyset)$
2. $(\mathsf{CLRSAB}.sk, \mathsf{CLRSAB}.a_0, \mathsf{CLRSAB}.\mathsf{upmsg}_0) \leftarrow \mathsf{CLRSAB}.\mathsf{Gen}(1^\lambda, \emptyset)$
3. Set
   (a) $sk \leftarrow (\mathsf{SIG}.sk, \mathsf{CLRSAB}.sk)$,
   (b) $a_0 \leftarrow (\mathsf{SIG}.a_0, \mathsf{CLRSAB}.a_0)$,
   (c) $\mathsf{upmsg}_0 \leftarrow \mathsf{CLRSAB}.a_0$
   (d) Instantiate m_0 as an empty map.
4. Return $(sk, a_0, \mathsf{upmsg}_0, m_0)$

$\mathsf{Update}(\mathsf{Op}_t, sk, a_t, m_t, x)$:

1. If $\mathsf{Op}_t = \mathsf{Add}$ and $x \notin m_t$:
   (a) Pick r_x at random from the domain D_{CLRSAB} of the CLRSAB accumulator. (We require the domain to be large enough that the probability of picking the same element twice is negligible.)
   (b) Set $m_{t+1} = m_t$
   (c) Set $m_{t+1}[x] = r_x$
   (d) $\mathsf{CLRSAB}.w_{t+1}^{r_x} \leftarrow \mathsf{CLRSAB}.\mathsf{Update}(\mathsf{Add}, \mathsf{CLRSAB}.sk, \mathsf{CLRSAB}.a_t, r_x)$
   (e) $\mathsf{SIG}.w_{t+1}^{(x,r_x)} \leftarrow \mathsf{SIG}.\mathsf{Update}(\mathsf{Add}, \mathsf{SIG}.sk, \mathsf{SIG}.a_0, (x, r_x))$
   (f) Set $\mathsf{CLRSAB}.a_{t+1} = \mathsf{CLRSAB}.a_t$.
   (g) Set $a_{t+1} = (\mathsf{SIG}.a_0, \mathsf{CLRSAB}.a_{t+1})$
   (h) Set $w_{t+1}^x = (r_x, \mathsf{CLRSAB}.w_{t+1}^{r_x}, \mathsf{SIG}.w_{t+1}^{(x,r_x)})$
   (i) Set $\mathsf{upmsg}_{t+1} = \perp$
   (j) Return $(a_{t+1}, m_{t+1}, w_{t+1}^x, \mathsf{upmsg}_{t+1})$
2. If $\mathsf{Op}_t = \mathsf{Del}$ and $x \in m_t$:
   (a) Set $r_x = m_t[x]$
   (b) Set $m_{t+1} = m_t$
   (c) Delete x from m_{t+1}
   (d) $(\mathsf{CLRSAB}.a_{t+1}, \mathsf{CLRSAB}.\mathsf{upmsg}_{t+1}) \leftarrow \mathsf{CLRSAB}.\mathsf{Update}(\mathsf{Add}, \mathsf{CLRSAB}.sk, \mathsf{CLRSAB}.a_t, r_x)$
   (e) Set $a_{t+1} = (\mathsf{SIG}.a_0, \mathsf{CLRSAB}.a_{t+1})$
   (f) Set $\mathsf{upmsg}_{t+1} = \mathsf{CLRSAB}.\mathsf{upmsg}_{t+1}$
   (g) Return $(a_{t+1}, m_{t+1}, \mathsf{upmsg}_{t+1})$

$\mathsf{WitCreate}(\mathsf{stts}, sk, a_t, m_t, x)$:

1. If $\mathsf{stts} = \mathsf{in}$ and $x \in m_t$:
   (a) Set $r_x = m_t[x]$
   (b) $\mathsf{SIG}.w_t^{(x,r_x)} \leftarrow \mathsf{SIG}.\mathsf{WitCreate}(\mathsf{in}, \mathsf{SIG}.sk, \mathsf{SIG}.a_0, (x, r_x))$
   (c) $\mathsf{CLRSAB}.w_t^{r_x} \leftarrow \mathsf{CLRSAB}.\mathsf{WitCreate}(\mathsf{in}, \mathsf{CLRSAB}.sk, \mathsf{CLRSAB}.a_t, r_x)$
   (d) Set $w_t^x = (r_x, \mathsf{CLRSAB}.w_t^{r_x}, \mathsf{SIG}.w_t^{(x,r_x)})$
   (e) Return w_t^x

$\mathsf{WitUp}(\mathsf{stts}, x, w_t^x = (r_x, \mathsf{CLRSAB}.w_t^{r_x}, \mathsf{SIG}.w_t^{(x,r_x)}), \mathsf{upmsg}_{t+1})$:

1. If $\mathsf{upmsg}_{t+1} \neq \perp$: (This update message corresponds to a deletion)
   (a) $\mathsf{CLRSAB}.w_{t+1}^{r_x} = \mathsf{CLRSAB}.\mathsf{WitUp}(\mathsf{in}, r_x, \mathsf{CLRSAB}.w_t^{r_x}, \mathsf{upmsg}_{t+1})$
2. Else: $w_{t+1}^x = w_t^x$
3. Return w_{t+1}^x

$\mathsf{VerStatus}(\mathsf{in}, a_t = (\mathsf{SIG}.a_t, \mathsf{CLRSAB}.a_t), x, w_t^x = (r_x, \mathsf{CLRSAB}.w_t^{r_x}, \mathsf{SIG}.w_t^{(x,r_x)}))$:

1. Return 1 if both of the following are 1, and 0 otherwise:
   - $\mathsf{SIG}.\mathsf{VerStatus}(\mathsf{in}, \mathsf{SIG}.a_0, (x, r_x), \mathsf{SIG}.w_t^{(x,r_x)})$
   - $\mathsf{CLRSAB}.\mathsf{VerStatus}(\mathsf{out}, \mathsf{CLRSAB}.a_t, r_x, \mathsf{CLRSAB}.w_t^{r_x})$

Figure 3·3: Braavos′ algorithms. We omit parameters unnecessary for the SIG and CLRSAB accumulator algorithms.

$r_x = \mathsf{F}_s(x)$ means that the random values do not have to be remembered, and can instead be recomputed on the fly. Therefore, while Braavos′ requires $O(A)$ accumulator manager storage (where A is the number of elements that have ever been added), Braavos only requires a constant amount of accumulator manager storage.

On the other hand, $\mathsf{Braavos}'$ naturally has the hiding update-message (HUM) property, while $\mathsf{Braavos}$ only has a weaker property called add-delete unlinkability (Baldimtsi et al., 2017). Because no update messages are required when additions take place and because when deletions take place, the update messages are only tied to random elements, no deletion can be linked to an addition. However, because $\mathsf{Braavos}$ uses the same random value $r_x = \mathsf{F}_s(x)$ for every re-addition of the same element x, two deletions of the same element are clearly linkable[3]; in $\mathsf{Braavos}'$, that is not the case. Because of $\mathsf{Braavos}'$'s use of fresh randomness for every re-addition of x, $\mathsf{Braavos}'$ achieves the HUM property, random choice of r.

Intuitively, both $\mathsf{Braavos}$ and $\mathsf{Braavos}'$ are secure because if an element was never added then no signature on it has ever been produced, and every time an element x is removed, all random values r_x that have been signed with x are in the CLRSAB accumulator, so no proof of non-membership for any such r_x can be produced.

More formally, we leverage our UC definitions from Chapter 2 in the following theorem and proof. Let $\mathcal{F}_{\mathsf{ACC},\mathsf{in},HUM}$ be our accumulator functionality $\mathcal{F}_{\mathsf{ACC}}$ for a dynamic, positive, HUM accumulator. That is, $\mathcal{F}_{\mathsf{ACC},\mathsf{in},HUM}$ is $\mathcal{F}_{\mathsf{ACC}}$ restricted to $\mathsf{stts} = \mathsf{in}$, and requiring the simulator to provide Update in two parts, as necessary for HUM (described in Section 2.3.3). Similarly, let $\mathcal{F}_{\mathsf{ACC},\mathsf{in},\mathsf{Add}}$ be our accumulator functionality $\mathcal{F}_{\mathsf{ACC}}$ for a positive additive accumulator, and let $\mathcal{F}_{\mathsf{ACC},\mathsf{in},\mathsf{Add},NA}$ be our accumulator functionality $\mathcal{F}_{\mathsf{ACC}}$ for a positive additive accumulator that is non-adaptively sound (Section 2.3.2).

Theorem 3. *The* $\mathsf{Braavos}'$ *accumulator described in Figure 3·3 securely realizes* $\mathcal{F}_{\mathsf{ACC},\mathsf{in},HUM}$ *as long as* SIG *securely realizes* $\mathcal{F}_{\mathsf{ACC},\mathsf{in},\mathsf{Add}}$ *with no update messages, and* CLRSAB *securely realizes* $\mathcal{F}_{\mathsf{ACC},\mathsf{in},\mathsf{Add},NA}$ *with no update messages for additions.*

We can prove Theorem 3 very simply using the fact that both SIG and CLRSAB

[3] Adding zero knowledge proofs would not resolve this issue — that random value cannot be hidden within a zero knowledge proof in any straightforward way, since it must be used to update CLRSAB witnesses.

are UC-secure (that is, by operating in the double-$\mathcal{F}_{\mathsf{ACC}}$-hybrid model). Before our UC definitions, a proof of security would involve a multi-step security reduction of the new accumulator to one of the old ones.

Proof. The simulator for the new accumulator uses its two inner simulators to obtain algorithms for the inner accumulators, composes them as in Figure 3·3, and submits those to the ideal functionality. (Since the CLRSAB accumulator is only non-adaptively sound, the simulator also pre-selects the random values that are to be accumulated in the CLRSAB accumulator.)[4]

In Section 2.3.3, we described how in order to modify the UC functionality $\mathcal{F}_{\mathsf{ACC}}$ to be HUM, we require that the simulator provide the algorithm Update in two parts: one sub-algorithm (let's call it Update_1) which only receives randomness and produces the update message; and a second sub-algorithm (let's call it Update_2) which produces all the other outputs of Update, and is additionally allowed to depend on the state of Update_1. If the update being performed is an addition, we do not need Update_1 at all, since no update message is necessary; we simply set $\mathsf{Update}_2(\mathsf{Add}, sk, a_t, m_t, x) = \mathsf{Update}(\mathsf{Add}, sk, a_t, m_t, x)$. If the update being performed is a deletion, $\mathsf{Update}_1(\mathsf{Del}, sk, a_t, m_t)$ gets a random pre-selected value and performs a CLRSAB deletion on it; it then passes the random value it deleted as $\mathsf{state}_{\mathsf{Update}_1}$ to $\mathsf{Update}_2(\mathsf{Del}, sk, a_t, m_t, x, \mathsf{state}_{\mathsf{Update}_1})$ which does the rest of the work.

The views of the environment $\mathcal{Z}$ in the real and ideal worlds will be identical in the so-called double-$\mathcal{F}_{\mathsf{ACC}}$-hybrid model, since the sub-accumulator functionalities guarantee that if an element was never added then no signature on it exists, and every time an element x is removed, all random values r_x that have been signed with x are not in the set accumulated in CLRSAB, so no proof of membership for any such r_x can be produced. □

Security of the Braavos accumulator follows trivially from the security of Braavos′ and pseudo-randomness of F.[5]

[4] Notice that this works regardless of how the simpler accumulators are implemented (simply software vs. hardward vs. distributed protocols), since they satisfy the UC definition.

[5] The Braavos and Braavos′ accumulators use Camenisch-Lysyanskaya (CL) signatures (Camenisch and Lysyanskaya, 2003) as the underlying positive accumulator $\mathsf{ACC}_{\mathsf{A}}$, and the CLRSAB accumulator as the underlying dynamic positive non-adaptively sound accumulator $\mathsf{ACC}_{\mathsf{NA}}$, both of which rely on the strong RSA assumption. This implies that Braavos′ relies only on that assumption, and that Braavos additionally only requires the pseudo-randomness of F.

The challenge that remains is finding a communication-optimal, dynamic, non-adaptively sound accumulator $\mathsf{ACC}_{\mathsf{NA}}$. $\mathsf{ACC}_{\mathsf{NA}}$ should only require membership witness updates upon element deletions, not element additions. In Section 3.3.1, we describe CLRSAB, which is exactly such an accumulator.

3.3.1 CLRSAB: A Communication-Optimal Non-Adaptively Sound Dynamic Accumulator

In this section, we formally describe the CLRSAB accumulator, which was informally introduced by Camenisch and Lysyanskaya (Camenisch and Lysyanskaya, 2002) in a remark on page 12. The CLRSAB accumulator is similar to the standard RSA accumulator (Camenisch and Lysyanskaya, 2002), which evolves the accumulator value (as well as all membership witnesses) with every addition and deletion. The CLRSAB accumulator, unlike the RSA accumulator, evolves the accumulator value with every deletion only. However, the price is that, as far as we can tell, the CLRSAB accumulator is only non-adaptively sound.

The RSA Accumulator In order to understand the CLRSAB accumulator, it helps to understand the RSA accumulator first. Its value is a quadratic residue a modulo n, where n is an RSA modulus: $n = pq$, where $p = 2p' + 1$ and $q = 2q' + 1$ for prime p, p', q, and q'. The domain D of the RSA accumulator consists of all odd positive prime integers x.[6]

During the addition of x to the accumulator, the new accumulator value is computed as $a_{t+1} = a_t^x \bmod n$. The membership witness w for x is then defined to be the old accumulator value a_t. A membership verification consists of checking that $a = w^x \bmod n$. When another element y is added to the accumulator, the member-

[6]Note that p' or q' cannot themselves be accumulated, since $(p')^{-1} \bmod p'q'$ and $(q')^{-1} \bmod p'q'$ do not exist; however, that only happens with negligible probability in the adaptive soundness game, since if the adversary finds p' or q', he or she has succeeded in factoring n.

ship witness for x is updated by taking $w_{t+1} = w_t^y \bmod n$.

When an element y is deleted, the accumulator manager (who knows the trapdoor $p'q'$) computes the new accumulator as $a_{t+1} = a_t^{y^{-1} \bmod p'q'} \bmod n$. The membership witness w for x can then be updated using the Bezout coefficients α and β such that $\alpha x + \beta y = 1$. (Recall that the domain D of the accumulator contains only odd prime numbers, so such α and β are guaranteed to exist.) The new witness is computed as $w_{t+1} = w_t^{\beta} a_{t+1}^{\alpha} \bmod n$. The RSA accumulator is more formally described in Section 2.7.

The CLRSAB Accumulator The CLRSAB accumulator preserves the relationship between the accumulator value and the witnesses, but avoids computing a new accumulator value and updating witnesses during each addition. Instead, during the addition of odd prime x the accumulator manager keeps the accumulator constant, and computes the membership witness w for x as $w = a^{x^{-1} \bmod p'q'} \bmod n$. Notice that this eliminates the need for updating existing membership witnesses during additions. The process for proving membership and for deletions is the same as in the RSA accumulator. The algorithms of the CLRSAB accumulator are detailed in Figure 3·4.

CLRSAB Soundness The RSA accumulator is adaptively sound, meaning that an adversary cannot find a membership witness for an element that is not a member even if she chooses which elements should be added, optionally based on accumulator and witness values she has previously seen.

The CLRSAB accumulator is non-adaptively sound, meaning that an adversary cannot find a membership witness for an element that is not a member if she chooses all elements to add prior to seeing any accumulator information. In particular, the CLRSAB accumulator is sound when only random elements are added to the accumu-

Gen($1^\lambda, \emptyset$):

1. Select two λ-bit safe primes $p = 2p' + 1$ and $q = 2q' + 1$ where p' and q' are also prime, and let $n = pq$. (Consider n to be public knowledge from hereon out; it is actually a part of the accumulator value a, but for simplicity we will not refer to it as such.)
2. Let $sk = p'q'$.
3. Select a random integer $a' \leftarrow \mathbb{Z}_n^*$.
4. Let $a = (a')^2 \bmod n$.
5. Return (sk, a).

Update(Op, sk, a, x):

1. Check that $x \in D$ (that is, that x is an odd prime). If not, FAIL.
2. If Op = Add:
   (a) Let $w = a^{x^{-1} \bmod p'q'} \bmod n$.
   (b) Return $(a, w, \mathsf{upmsg} = \bot)$.
3. If Op = Del:
   (a) Let $a = a^{x^{-1} \bmod p'q'} \bmod n$.
   (b) Let $\mathsf{upmsg} = (a, x)$.
   (c) Return $(a, w = \bot, \mathsf{upmsg})$.

WitUp(stts = in, x, w, upmsg):

1. If upmsg = $\bot$: return w. (This was an addition, and no update is necessary.)
2. Parse $(a, y) = \mathsf{upmsg}$.
3. Compute Bezout coefficients α, β such that $\alpha x + \beta y = 1$. (Given that $x \neq y$, since both x and y are prime, such α and β are guaranteed to exist.)
4. Let $w = w^\beta a^\alpha \bmod n$.
5. Return w.

VerStatus(stts = in, a, x, w):

1. Return 1 if $a = w^x \bmod n$.
2. Return 0 otherwise.

Figure 3·4: CLRSAB Algorithms

lator, since those can be chosen prior to seeing any accumulator or witness values.[7] This holds under the *strong RSA assumption* (Bari and Pfitzmann, 1997).

Assumption 1 (Strong RSA). *For any probabilistic polynomial-time adversary $\mathcal{A}$,*

$$\Pr[p, q \leftarrow \{\lambda\text{-}bit\ safe\ primes\}; n = pq; t \leftarrow \mathbb{Z}_n^*; (r, e) \leftarrow \mathcal{A}(n, t) : t = r^e \bmod n \wedge e \text{ is prime}] = \nu(\lambda)$$

For some negligible function ν.

Theorem 4. *The* CLRSAB *accumulator with a domain D consisting of odd primes is non-adaptively sound under the strong RSA assumption.*

Proof. In Figure 3·5, we reduce the non-adaptive soundness of the CLRSAB accumulator to the strong RSA assumption. The reduction $\mathcal{R}$ takes in an RSA integer n and

[7] We are not certain whether CLRSAB is also adaptively sound. Proving that it is or is not is an open problem. It is adaptively sound when the when a polynomial-size subset of D is used as the domain; however, this is a very limiting restriction.

a random value $t \in \mathbb{Z}_n^*$, and returns r, e such that $t = r^e \bmod n$. $\mathcal{R}$ leverages an adversary $\mathcal{A}$ which can break the non-adaptive soundness of the CLRSAB accumulator; that is, after making addition (Add) and deletion (Del) queries on elements chosen before seeing the initial state of the accumulator, $\mathcal{A}$ can produce an odd prime x and a witness w such that $a = w^x \bmod n$, and x is not in the accumulator.

$\mathcal{R}$ must be able to answer two types of queries from $\mathcal{A}$: Add queries on the non-adaptively chosen elements, and Del queries on the same elements. Let q_{Add} be an upper bound on the number of Add queries, and q_{Del} be an upper bound on the number of Del queries $\mathcal{A}$ can make. During the setup phase, having received the elements $x_1, \ldots, x_{q_{\mathsf{Add}}}$ from $\mathcal{A}$, the reduction $\mathcal{R}$ creates an accumulator for which it can answer Add and Del queries on elements $x_1, \ldots, x_{q_{\mathsf{Add}}}$. It does so by starting with $a = t^2 \bmod n$, and raising a to the power of the elements. By raising a to the power of $x_j^{q_{\mathsf{Del}}}$, $\mathcal{R}$ creates an accumulator value for which it is able to answer Del and Add queries on x_j even if $\mathcal{A}$ spends all of its Del queries on that one element. However, if $\mathcal{A}$ forges a witness w for x_j (after having added and deleted it fewer than q_{Del} times), the reduction won't be able to use w to break the strong RSA assumption, since it already knows w! For that reason, $\mathcal{R}$ guesses a "target" element x_j from among $x_1, \ldots, x_{q_{\mathsf{Add}}}$, and the number e_j of times that x_j will be added and deleted before the forgery (which can be anywhere from 0 to q_{Del}), and only raises t to the power of $x_j^{e_j}$, not $x_j^{q_{\mathsf{Del}}}$. Figure 3·5 shows the details of how the reduction picks an accumulator value based on $x_1, \ldots, x_{q_{\mathsf{Add}}}$, how it answers Add and Del queries, and how it then uses the output of $\mathcal{A}$ to break the strong RSA assumption.

This reduction succeeds as long as:

1. During the query phase, $\mathcal{R}$ does not output FAIL. $\mathcal{R}$ does not output FAIL if the target exponent e_i was chosen correctly, which happens with probability $\frac{1}{q_{\mathsf{Del}}+1}$.
2. During the output phase, $\mathcal{R}$ does not output FAIL. If $\mathcal{A}$ outputs a witness for an element $x_i \in \{x_1, \ldots, x_{q_{\mathsf{Add}}}\}$, $\mathcal{R}$ does not output FAIL as long as:
   (a) $\mathcal{R}$ makes x the "target" prime (that is, $j = i$). This happens with probability $\frac{1}{q_{\mathsf{Add}}}$.
   (b) $\mathcal{R}$ correctly chooses the target exponent e_i for x. This happens with probability $\frac{1}{q_{\mathsf{Del}}+1}$. However, this is already accounted for in item 1.
3. $\mathcal{A}$ succeeds in breaking the security of the CLRSAB accumulator, which we assume happens with non-negligible probability ϵ.

As long as $\mathcal{R}$ does not output FAIL, $\mathcal{A}$ sees the same transcript it would when interacting with a real accumulator manager. The probability of the reduction $\mathcal{R}$

$\mathsf{Setup}(n, t, q_{\mathsf{Add}}, q_{\mathsf{Del}})$:

1. Let $x_1, \ldots, x_{q_{\mathsf{Add}}}$ be the distinct odd primes provided by the adversary $\mathcal{A}$.
2. Let $a = t^2$ (so as to make a a quadratic residue).
3. Let $e_c = 2$ be the current exponent linking t to a. (So, $a = t^{e_c} \bmod n$ is an invariant.)
4. Pick a random index $j \leftarrow \{1, \ldots, q_{\mathsf{Add}}\}$.
5. For $i \in [1, \ldots, q_{\mathsf{Add}}]$:
   (a) If $i = j$: pick a random $e_i \leftarrow \{0, \ldots, q_{\mathsf{Del}}\}$.
   (b) Else: $e_i = q_{\mathsf{Del}}$.
   (c) Let $a = a^{(x_i^{e_i})} \bmod n$.
   (d) Let $e_c = e_c x_i^{e_i}$.
6. Return $a = e_i^{e_c} \bmod n$ to the adversary $\mathcal{A}$.

$\mathsf{Add}(x_i)$:

1. If $e_i = 0$: FAIL.
2. Let $w = t^{e_c/x_i} \bmod n$. (Note that e_c must be divisible by x_i, since e_c has a factor of $x_i^{e_i}$, and $e_i > 0$.)
3. Return w.

$\mathsf{Del}(x_i)$:

1. If $e_i = 0$: FAIL.
2. Let $e_c = e_c/x_i$. (Note that e_c must be divisible by x_i, since e_c has a factor of $x_i^{e_i}$, and $e_i > 0$.)
3. Let $e_i = e_i - 1$.
4. Let $a = t^{e_c} \bmod n$.
5. Let $\mathsf{upmsg} = (a, x_i)$.
6. Return (a, upmsg).

$\mathsf{Output}(e, w)$:

1. Check that $a = w^e \bmod n$. If not, FAIL.
2. If $e = x_i$ for $i \in \{1, \ldots, q_{\mathsf{Add}}\}$ and ($i \neq j$ or $e_i > 0$): FAIL.
3. We know that e and e_c must be relatively prime; e_c has no factors outside of $x_1, \ldots, x_{q_{\mathsf{Add}}}$, and those factors have powers e_i. So, $\mathcal{R}$ can compute Bezout coefficients α, β such that $\alpha e + \beta e_c = 1$.
4. Let $r = t^\alpha w^\beta \bmod n$.
   (Let $y = t^{e^{-1} \bmod p'q'} \bmod n$; equivalently, $y^e \bmod n = t$. Since $w^e \bmod n = a$ and $t^{e_c} \bmod n = (y^e)^{e_c} \bmod n = a$, it follows that $w = y^{e_c} \bmod n$. So, $r = t^\alpha w^\beta = (y^e)^\alpha (y^{e_c})^\beta = y^1 = y$.)
5. Return (r, e).

Figure 3·5: Reduction From the Non-Adaptive Soundness of CLRSAB to the Strong RSA Assumption

succeeding is $\frac{1}{q_{\mathsf{Add}}} \frac{1}{q_{\mathsf{Del}}+1} \epsilon$, which is non-negligible.

□

Other Approaches for Expanding the Domain and Getting Adaptive Soundness for CLRSAB The domain D of CLRSAB consists of odd primes. Such a limited domain is not a problem for our main application, because the Braavos accumulator manager can choose a (psuedo)random prime r when a new element is added to the accumulator for the first time, as described in Construction H. In fact, Construction H can be viewed as one approach to expanding the domain of CLRSAB and obtaining adaptive security for it. Here we briefly mention other approaches. Let D' be the

desired domain. Let f be a mapping from D' to λ-bit odd primes. To add $x \in D'$ to the accumulator, add $\mathsf{f}(x)$ instead. We can obtain adaptive soundness in the following ways:

- We can model f as a random oracle (the proof is straightforward).
- We can avoid the random oracle by making a different strong assumption instead: namely, the assumption that f is collision-resistant, and the very strong "adaptive strong-RSA assumption". Informally, the adaptive strong-RSA assumption states that even given an oracle that can take roots modulo n, it is difficult to find new roots whose power is relatively prime to those of the roots produced by the oracle.
- We can get somewhat better assumptions by having f be a randomized mapping, and include the randomness R as part of the witness. Then, assuming that for every two elements x_1 and x_2, the distributions $\mathsf{f}(x_1; R)$ and $\mathsf{f}(x_2; R)$ (over random choices of R) are statistically close, we can use the technique from (Gennaro et al., 1999). To do so, we need to assume that the strong RSA assumption (Assumption 1) also holds in a model where there exists an oracle $\mathcal{O}$ that on input x, p returns a random R such that $\mathsf{f}(R; x) = p$.
- Alternatively, we can use the strong-RSA assumption without modification if f is a trapdoor hash function, following the technique of (Shamir and Tauman, 2001).

All of these approaches require f that maps to primes. A way to build such f is described in (Cachin et al., 1999, Section 3.2) (see also (Micali et al., 1999, Section 7)).

3.3.2 Adding Zero Knowledge to Braavos

So far, we have only discussed the functionality of accumulators, ignoring potential privacy concerns. There typically exist three primary privacy goals in the context

of accumulators: hiding the membership (or non-membership) witness, hiding the element whose membership (or non-membership) is being demonstrated as well as the witness, and hiding all information about the accumulated set (Ghosh et al., 2016). For our application of anonymous credential revocation (discussed further in (Baldimtsi et al., 2017)), we mostly care about zero knowledge proofs of member knowledge, which hide not only the witness, but the member element itself.

The $\mathsf{Braavos}$ and $\mathsf{Braavos}'$ accumulators support efficient zero-knowledge proofs of member knowledge. They do this in the exact same way; in the rest of this section, we only discuss $\mathsf{Braavos}$. Given that $\mathsf{Braavos}$ is composed of two accumulators $\mathsf{ACC_A}$ and $\mathsf{ACC_{NA}}$, in order for a witness holder to produce a zero-knowledge proof of member knowledge in $\mathsf{Braavos}$, she would have to produce a conjunction of proofs of member knowledge in both $\mathsf{ACC_A}$ and $\mathsf{ACC_{NA}}$ and a proof that those members have the correct relationship. More concretely, she would have to compute the following zero-knowledge proof (described using Camenisch-Stadler (Camenisch and Stadler, 1997) notation):

$$
\begin{aligned}
&\mathsf{ZKP}[(x, r, \mathsf{ACC_A}.w, \mathsf{ACC_{NA}}.w) : \\
&\qquad \wedge\, \mathsf{ACC_A}.\mathsf{VerMem}(\mathsf{ACC_A}.a, (x, r), \mathsf{ACC_A}.w) \\
&\qquad \wedge\, \mathsf{ACC_{NA}}.\mathsf{VerMem}(\mathsf{ACC_{NA}}.a, r, \mathsf{ACC_{NA}}.w) \\
&\quad](\mathsf{ACC_{NA}}.a, \mathsf{ACC_A}.a)
\end{aligned}
$$

Where $\mathsf{ACC_A}$ is the signature scheme $\mathsf{SIG}_{CL} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ due to Camenisch and Lysyanskaya (Camenisch and Lysyanskaya, 2003), and $\mathsf{ACC_{NA}}$ is the CLRSAB accumulator.

For integration into larger systems, it might be important to be able to link the witnesses used in the proof to other statements, while still keeping the elements and witnesses private. To this end, commitments to the witnesses can be used. Let

$Com = (\mathsf{Commit}, \mathsf{Verify})$ be a commitment scheme; to integrate commitments into the zero knowledge proof, a witness holder computes commitments to the membership witnesses $\mathsf{ACC_A}.w$ and $\mathsf{ACC_{NA}}.w$: $(c_1, o_1) = Com.\mathsf{Commit}(\mathsf{ACC_A}.w)$ and $(c_2, o_2) = Com.\mathsf{Commit}(\mathsf{ACC_{NA}}.w)$, where o_1 and o_2 are decommitment values. The proof is then enhanced, as follows:

$$
\begin{aligned}
\mathsf{ZKP}[&(x, r, \mathsf{ACC_A}.w, \mathsf{ACC_{NA}}.w, o_1, o_2) : \\
&Com.\mathsf{Verify}(c_1, \mathsf{ACC_A}.w, o_1) \\
&\wedge Com.\mathsf{Verify}(c_2, \mathsf{ACC_{NA}}.w, o_2) \\
&\wedge \mathsf{ACC_A}.\mathsf{VerMem}(\mathsf{ACC_{NA}}.a, (x, r), \mathsf{ACC_A}.w) \\
&\wedge \mathsf{ACC_{NA}}.\mathsf{VerMem}(\mathsf{ACC_{NA}}.a, r, \mathsf{ACC_{NA}}.w) \\
&](\mathsf{ACC_{NA}}.a, \mathsf{ACC_A}.a, c_1, c_2)
\end{aligned}
$$

For concrete descriptions of the individual clauses of this proof using the commitment scheme due to Fujisaki and Okamoto (Fujisaki and Okamoto, 1997), please refer to Fujisaki and Okamoto (Fujisaki and Okamoto, 1997) and Camenisch and Lysyanskaya (Camenisch and Lysyanskaya, 2003; Camenisch and Lysyanskaya, 2002).

3.4 BraavosB: Another Communication-Optimal Adaptively Sound Dynamic Accumulator

In this section, we introduce the BraavosB accumulator, which is an instantiation of construction B from Figure 3·1. BraavosB is a dynamic positive accumulator derived from a positive accumulator and a dynamic negative accumulator. The positive accumulator is used to accumulate added elements, and the negative accumulator is used to accumulate deleted elements. A proof of membership in BraavosB consists of a proof of membership in the positive accumulator (that is, a proof that the element

in question has been added) and a proof of non-membership in the negative accumulator (that is, a proof that the element in question has not been deleted). In Figures 3·6, 3·7 and 3·8, we provide a detailed description of the construction B/ BraavosB algorithms (in terms of the two accumulators of which it is composed, the positive accumulator $\mathsf{ACC_P}$ and the negative dynamic accumulator $\mathsf{ACC_N}$).

Just like Braavos, we aim for BraavosB to be communication optimal, and to support of zero knowledge. It might seem strange to be building a dynamic accumulator out of a negative accumulator which is already dynamic; however, using the dynamic accumulator to accumulate deleted elements instead of added elements is the key to saving on communication.

Our choice of underlying positive accumulator $\mathsf{ACC_P}$ in BraavosB is any existentially unforgeable digital signature scheme (as long as it supports zero knowledge proofs of knowledge of a signature on a committed value). Note that though construction B has a MemWitUpOnAdd algorithm, this algorithm is not used by BraavosB, since digital signatures do not require membership updates.

The challenge that remains is instantiating $\mathsf{ACC_N}$; that is, building a negative dynamic accumulator that supports efficient zero knowledge proofs and the efficient generation of non-membership witnesses. The Merkle tree accumulator (Camacho et al., 2008) does not fit the criteria, because it does not support efficient zero-knowledge proofs. The RSA (Benaloh and de Mare, 1994; Camenisch and Lysyanskaya, 2002; Li et al., 2007a) and Bilinear Map (Nguyen, 2005; Damgård and Triandopoulos, 2008; Au et al., 2009) accumulators do not fit these criteria either, because for each of them, generating a non-membership witness takes time linear in the number of accumulated elements. If the number of accumulated elements is large, and the demand for non-membership witnesses is high, this can be prohibitive; ideally, non-membership

witnesses should be generated in constant time. To fill this need, we construct the range-RSA accumulator, described in Appendix 3.4.1. The range-RSA accumulator is the core technical piece of the BraavosB accumulator.

$\mathsf{Gen}(1^\lambda, \emptyset)$:
1. $(\mathsf{ACC_P}.a, \mathsf{ACC_P}.m, \mathsf{ACC_P}.sk) \leftarrow \mathsf{ACC_P}.\mathsf{Gen}(1^\lambda, \emptyset)$.
2. $(\mathsf{ACC_N}.a, \mathsf{ACC_P}.m, \mathsf{ACC_N}.sk) \leftarrow \mathsf{ACC_N}.\mathsf{Gen}(1^\lambda, \emptyset)$.
3. Let $sk = (\mathsf{ACC_P}.sk, \mathsf{ACC_N}.sk)$.
4. Let $a = (\mathsf{ACC_P}.a, \mathsf{ACC_N}.a)$.
5. Let $m = (\mathsf{ACC_P}.m, \mathsf{ACC_N}.m)$.
6. Return $(sk,\ a,\ m)$.

$\mathsf{Add}(sk, a, m, x)$:
1. $(\mathsf{ACC_P}.a,\ \mathsf{ACC_P}.m,\ \mathsf{ACC_P}.w,\ \mathsf{ACC_P}.\mathsf{upmsg}) \leftarrow \mathsf{ACC_P}.\mathsf{Add}(\mathsf{ACC_P}.sk,\ \mathsf{ACC_P}.a,\ \mathsf{ACC_P}.m,\ x)$.
2. $(\mathsf{ACC_N}.a,\ \mathsf{ACC_N}.m,\ \mathsf{ACC_N}.\mathsf{upmsg}) \leftarrow \mathsf{ACC_N}.\mathsf{Del}(\mathsf{ACC_N}.sk,\ \mathsf{ACC_N}.a,\ \mathsf{ACC_N}.m,\ x)$. (This should do nothing if x is not in $\mathsf{ACC_N}$ already.)
3. $\mathsf{ACC_N}.u \leftarrow \mathsf{ACC_N}.\mathsf{NonMemWitCreate}(\mathsf{ACC_N}.sk,\ \mathsf{ACC_N}.a,\ \mathsf{ACC_N}.m,\ x)$.
4. Let $w = (\mathsf{ACC_P}.w,\ \mathsf{ACC_N}.u)$.
5. Let $\mathsf{upmsg} = (\mathsf{ACC_P}.\mathsf{upmsg},\ \mathsf{ACC_N}.\mathsf{upmsg})$.
6. Return $(a,\ m,\ w,\ \mathsf{upmsg})$.

$\mathsf{Del}(sk, a, m, x)$:
1. $(\mathsf{ACC_N}.a,\ \mathsf{ACC_N}.m,\ \mathsf{ACC_N}.\mathsf{upmsg}) \leftarrow \mathsf{ACC_N}.\mathsf{Add}(\mathsf{ACC_N}.sk,\ \mathsf{ACC_N}.a,\ \mathsf{ACC_N}.m,\ x)$.
2. Return $(a, m, \mathsf{ACC_N}.\mathsf{upmsg})$.

$\mathsf{WitCreate}(sk, a, m, x, (\mathsf{upmsg}_1, \ldots, \mathsf{upmsg}_t))$:
1. $\mathsf{ACC_P}.w \leftarrow \mathsf{ACC_P}.\mathsf{WitCreate}((\mathsf{ACC_P}.sk,\ \mathsf{ACC_P}.a,\ \mathsf{ACC_P}.m,\ x,\ (\mathsf{ACC_P}.\mathsf{upmsg}_1, \ldots, \mathsf{ACC_P}.\mathsf{upmsg}_t))$.
2. $\mathsf{ACC_N}.u \leftarrow \mathsf{ACC_N}.\mathsf{NonMemWitCreate}((\mathsf{ACC_N}.sk,\ \mathsf{ACC_N}.a,\ \mathsf{ACC_N}.m,\ x,\ (\mathsf{ACC_N}.\mathsf{upmsg}_1, \ldots, \mathsf{ACC_N}.\mathsf{upmsg}_t))$.
3. Return $w = (\mathsf{ACC_P}.w, \mathsf{ACC_N}.u)$.

Figure 3·6: Construction B Accumulator Manager Algorithms

$\mathsf{MemWitUpOnAdd}(a, x, w, \mathsf{upmsg})$:
1. Parse $(\mathsf{ACC_P}.\mathsf{upmsg}, \mathsf{ACC_N}.\mathsf{upmsg}) = \mathsf{upmsg}$.
2. Parse $(\mathsf{ACC_P}.w, \mathsf{ACC_N}.u) = w$.
3. $\mathsf{ACC_N}.w \leftarrow \mathsf{ACC_N}.\mathsf{MemWitUpOnDel}(x,\ \mathsf{ACC_N}.u,\ \mathsf{ACC_N}.\mathsf{upmsg})$.
4. $\mathsf{ACC_P}.w \leftarrow \mathsf{ACC_P}.\mathsf{MemWitUpOnAdd}(x,\ \mathsf{ACC_P}.w,\ \mathsf{ACC_P}.\mathsf{upmsg})$.
5. Return $w = (\mathsf{ACC_P}.w, \mathsf{ACC_N}.u)$.

$\mathsf{MemWitUpOnDel}(a, x, w, \mathsf{upmsg})$:
1. Parse $(\mathsf{ACC_P}.w, \mathsf{ACC_N}.u) = w$.
2. $\mathsf{ACC_N}.u \leftarrow \mathsf{ACC_N}.\mathsf{NonMemWitUpOnAdd}(x,\ \mathsf{ACC_N}.u,\ \mathsf{upmsg})$.
3. Return $w = (\mathsf{ACC_P}.w, \mathsf{ACC_N}.u)$.

The witness holder can run BatchMemWitUpOnDel immediately before producing a proof.

Figure 3·7: Construction B Witness Holder Algorithms

3.4.1 Range-RSA: A Dynamic Negative Accumulator

In this section, we present the range-RSA accumulator. This accumulator is a modified version of the RSA accumulator. Like the Merkle tree accumulator of Camacho

$\mathsf{VerMem}(a, x, w)$:

1. Parse $(\mathsf{ACC_P}.a, \mathsf{ACC_N}.a) = a$.
2. Parse $(\mathsf{ACC_P}.w, \mathsf{ACC_N}.u) = w$.
3. Let $b_1 \leftarrow \mathsf{ACC_P}.\mathsf{VerMem}(\mathsf{ACC_P}.a, x, \mathsf{ACC_P}.w)$
4. Let $b_2 \leftarrow \mathsf{ACC_N}.\mathsf{VerNonMem}(\mathsf{ACC_N}.a, x, \mathsf{ACC_N}.u)$
5. Return 1 if $b_1 = b_2 = 1$, and return 0 otherwise.

Figure 3·8: Construction B Third Party Algorithms

et. al (Camacho et al., 2008), the range-RSA accumulator is based on ranges; it accumulates ranges in a positive RSA accumulator (Camenisch and Lysyanskaya, 2002). All elements belonging to such a range are considered to be *non-members*; so, a proof of non-membership of x in the range-RSA accumulator would just be a proof that some range $(\mathsf{low}, \mathsf{high})$ is in the underlying positive RSA accumulator, and that $\mathsf{low} \leq x \leq \mathsf{high}$. [8]

A range-RSA accumulator can be instantiated empty; the Gen algorithm then creates an empty positive RSA accumulator, and adds $(\mathsf{low}, \mathsf{high})$ to it (where low is smaller than the smallest supported element and high is the highest supported element). Informally, when an element x is added to the range-RSA accumulator, the range containing x is deleted from the underlying positive RSA accumulator. To replace that range, at most two new ranges are added, covering all of the other elements in the deleted range. When an element x is deleted from the range-RSA accumulator, at most two ranges containing x's direct neighbors are deleted from the underlying positive RSA accumulator. A new range, which covers x together with the deleted ranges, is added. The accumulator manager stores all of the range membership

[8]Note that range-RSA can be made universal by using open ranges instead of closed ones. A proof of non-membership of x in the range-RSA accumulator would be a proof that some range $(\mathsf{low}, \mathsf{high})$ is in the underlying positive RSA accumulator, and $\mathsf{low} < x < \mathsf{high}$. Then, the range-RSA accumulator would support proofs of membership as well as proofs of non-membership. A proof of membership of x in the range-RSA accumulator would be a proof that some range $(\mathsf{low}, \mathsf{high})$ is in the underlying positive RSA accumulator, with $x = \mathsf{low}$ or $x = \mathsf{high}$ or both. However, we do not make this simple modification in this paper, because the zero-knowledge proofs described in Section 3.4.4 are more efficient for closed ranges. We do not discuss proofs of membership in the range-RSA accumulator any further, as we only use range-RSA to prove non-membership.

witnesses. Each range membership witness functions as the non-membership witness for all of the elements in that range; so, producing a non-membership witness is a simple matter of a look-up. In Section 3.4.2, we spell out all of the details of range-RSA accumulator algorithms.

What remains is the question of how one accumulates ranges in the underlying positive RSA accumulator. This accumulator requires that all accumulated elements (in our case, ranges) be mapped to prime numbers using some canonical function h. We choose a function h that is particularly well-suited for use with efficient zero-knowledge proofs. We define $\mathsf{h}(\mathsf{low}, \mathsf{high})$ to choose an integer suffix such that $p = \mathsf{low}||\mathsf{high}||\mathsf{suffix}$ is a prime number, where $||$ denotes concatenation, and each of $\mathsf{low}, \mathsf{high}$ and suffix use a fixed number l of bits. (Assuming that prime numbers are dense, for a sufficiently large l such a suffix will always exist.) Our choice of h allows $\mathsf{h}(\mathsf{low}, \mathsf{high})$ to be expressed using arithmetic operations: $\mathsf{h}(\mathsf{low}, \mathsf{high}) = 2^{2l}\mathsf{low} + 2^{l}\mathsf{high} + \mathsf{suffix}$.

Notice that the range-RSA accumulator can only accumulate elements x such that $0 \leq x \leq 2^l - 1$. This can be avoided in one of two ways: by allowing l to depend on the range $(\mathsf{low}, \mathsf{high})$ in question and encoding l as part of the h output, or by using a collision-resistant hash function to map all elements to l-bit binary strings. For the rest of this paper, we only consider the accumulation of elements x such that $0 \leq x \leq 2^l - 1$, since that is sufficient for the anonymous revocation application.

Theorem 5. *The range-RSA accumulator is a adaptively sound negative dynamic accumulator under the strong RSA assumption.*

Sketch. In order to break the adaptive soundness of the range-RSA accumulator, an adversary would need to compute a non-membership witness u for an element x that *is* actually in the accumulator. This is equivalent to computing a membership witness w for $r = \mathsf{h}(\mathsf{low}, \mathsf{high})$ such that $\mathsf{low} \leq x \leq \mathsf{high}$ in the underlying positive RSA accumulator. No such r is actually in the underlying positive RSA accumulator, so this would require breaking the security of the positive RSA accumulator, which is hard under the strong RSA assumption. □

Properties other than security (such as completeness) are self-evident.

3.4.2 Range-RSA Accumulator Algorithms

In this section, we describe the algorithms of the range-RSA accumulator introduced in Section 3.4.1. Recall that $\mathsf{h}(\mathsf{low}, \mathsf{high})$ is a function that can be applied to ranges to get a prime integer of the form $p = \mathsf{low}||\mathsf{high}||\mathsf{suffix}$, where $||$ denotes concatenation, and each of $\mathsf{low}, \mathsf{high}$ and suffix have a fixed number l of bits assigned to them. suffix can be any value which makes p prime.

Let $\mathsf{RSA.Gen}$, $\mathsf{RSA.Add}$, $\mathsf{RSA.MemWitUpOnAdd}$, $\mathsf{RSA.Del}$, $\mathsf{RSA.MemWitUpOnDel}$ and

$\mathsf{RSA.VerMem}$ be the protocols the dynamic RSA accumulator (Camenisch and Lysyanskaya, 2002). The protocols of the range-RSA accumulator are described in Figures 3·9, 3·10 and 3·11.

3.4.3 BraavosB Soundness

The BraavosB accumulator uses Camenisch-Lysyanskaya (CL) signatures (Camenisch and Lysyanskaya, 2003) as the underlying positive accumulator ACC_P, and the range-RSA accumulator as the underlying dynamic negative accumulator ACC_N. CL signatures are existentially unforgeable under the strong RSA assumption. Recall that the range-RSA accumulator is secure under the same assumption. By Theorem 2, this implies that the BraavosB accumulator is an adaptively sound positive dynamic accumulator under the strong RSA assumption.

Properties other than adaptive soundness (such as completeness) are self-evident.

3.4.4 Adding Zero Knowledge to BraavosB

Zero-Knowledge Proofs of Non-Member Knowledge in Range-RSA The range-RSA accumulator supports efficient zero-knowledge proofs of non-member knowl-

$\mathsf{Gen}(1^\lambda)$:

1. $(\mathsf{RSA}.sk, \mathsf{RSA}.a) \leftarrow \mathsf{RSA.Gen}(1^\lambda, \emptyset)$
2. Let $r = \mathsf{h}(\mathsf{low}, \mathsf{high})$, where low denotes 0, and high denotes 2^l.
3. $(\mathsf{RSA}.a, \mathsf{RSA}.w, \mathsf{RSA.upmsg}) \leftarrow \mathsf{RSA.Add}(\mathsf{RSA}.a, r)$
4. Let $sk = \mathsf{RSA}.sk$.
5. Let $a = \mathsf{RSA}.a$.
6. Let $m = [((\mathsf{low}, \mathsf{high}), r, \mathsf{RSA}.w)]$.
7. Return $(sk,\ a,\ m)$.

$\mathsf{NonMemWitCreate}(x, m)$:

1. Find the entry $((\mathsf{low}, \mathsf{high}), r, \mathsf{RSA}.w)$ in m such that $\mathsf{low} \leq x \leq \mathsf{high}$.
2. If no such element exists, then x is a member of the accumulator: return $\perp$.
3. Otherwise, return $u = (r, \mathsf{RSA}.w)$.

$\mathsf{Add}(sk, a, m, x)$ (executed by the accumulator manager):

1. Let $= \leftarrow []$.
2. Find $((\mathsf{low}, \mathsf{high}), r, w)$ in m such that $\mathsf{low} \leq x \leq \mathsf{high}$. If no such element exists, then x is already a member and no work remains to be done. Otherwise, continue.
3. Remove $((\mathsf{low}, \mathsf{high}), r, w)$ from m.
4. $(\mathsf{RSA}.a, \mathsf{RSA}.m, \mathsf{RSA.upmsg}) \leftarrow \mathsf{RSA.Del}(\mathsf{RSA}.sk, \mathsf{RSA}.a, r)$.
5. Add $\mathsf{RSA.upmsg}$ to upmsg.
6. If $x \neq \mathsf{low}$:
   (a) Let $r^l = \mathsf{h}(\mathsf{low}, x-1)$.
   (b) $(\mathsf{RSA}.a, \mathsf{RSA}.w^l, \mathsf{RSA.upmsg}^l) \leftarrow \mathsf{RSA.Add}(\mathsf{RSA}.a, r^l)$.
   (c) Add $\mathsf{RSA.upmsg}^l$, $\mathsf{RSA}.w^l$ and r^l to upmsg.
   (d) Add $((\mathsf{low}, x-1), r^l, \mathsf{RSA}.w^l)$ to m.
7. If $x \neq \mathsf{high}$:
   (a) Let $r^h = \mathsf{h}(x+1, \mathsf{high})$.
   (b) $(\mathsf{RSA}.a, \mathsf{RSA}.w^h, \mathsf{RSA.upmsg}^h) \leftarrow \mathsf{RSA.Add}(\mathsf{RSA}.a, r^h)$.
   (c) Add $\mathsf{RSA.upmsg}^h$, $\mathsf{RSA}.w^h$ and r^{high} to upmsg.
   (d) Add $((x+1, \mathsf{high}), r^h, \mathsf{RSA}.w^h)$ to m.
8. Let $a = \mathsf{RSA}.a$.
9. Return (a, m, upmsg).

$\mathsf{Del}(sk, a, m, x)$ (executed by the accumulator manager):

1. If there exists $((\mathsf{low}, \mathsf{high}), r, w)$ in m such that $\mathsf{low} \leq x \leq \mathsf{high}$, then x is already a non-member: return $\perp$. Otherwise, continue.
2. Let $\mathsf{upmsg} = []$.
3. Let $\mathsf{low} = x - 1$ and $\mathsf{high} = x + 1$.
4. If there exists $((l, \mathsf{low}),\ r^l,\ \mathsf{RSA}.w^l)$ for some l in m:
   (a) $(\mathsf{RSA}.a, \mathsf{RSA}.m, \mathsf{RSA.upmsg}^l) \leftarrow \mathsf{RSA.Del}(\mathsf{RSA}.sk, \mathsf{RSA}.a, r^l)$.
   (b) Add $\mathsf{RSA.upmsg}^l$ to upmsg.
   (c) Delete $((l, \mathsf{low}), r^l, \mathsf{RSA}.w^l)$ from m.
   (d) Let $\mathsf{low} = l$.
5. If there exists $((\mathsf{high}, h),\ r^h,\ \mathsf{RSA}.w^h)$ for some h in m:
   (a) $(\mathsf{RSA}.a, \mathsf{RSA}.m, \mathsf{RSA.upmsg}^h) \leftarrow \mathsf{RSA.Del}(\mathsf{RSA}.sk, \mathsf{RSA}.a, r^h)$.
   (b) Add $\mathsf{RSA.upmsg}^h$ to upmsg.
   (c) Delete $((\mathsf{high}, h), r^h, \mathsf{RSA}.w^h)$ from m.
   (d) Let $\mathsf{high} = h$.
6. $r = \mathsf{h}(\mathsf{low}, \mathsf{high})$.
7. $(\mathsf{RSA}.a, \mathsf{RSA}.m, \mathsf{RSA}.w, \mathsf{RSA.upmsg}) \leftarrow \mathsf{RSA.Add}(\mathsf{RSA}.sk, \mathsf{RSA}.a, r)$.
8. Add r, $\mathsf{RSA}.w$ and $\mathsf{RSA.upmsg}$ to upmsg.
9. Add $((\mathsf{low}, \mathsf{high}), r, \mathsf{RSA}.w)$ to m.
10. Let $a = \mathsf{RSA}.a$.
11. Return (a, m, upmsg).

Figure 3·9: Range-RSA Accumulator Manager Algorithms

edge. To prove knowledge of a non-member, a witness holder proves knowledge of values $(x, w, r, \mathsf{low}, \mathsf{high}, \mathsf{suffix})$ such that

$\mathsf{NonMemWitUpOnAdd}(x, u, \mathsf{upmsg})$ (executed by the witness holder upon receipt of upmsg):

1. Parse $(r, \mathsf{RSA}.w) = u$.
2. Parse $(\mathsf{RSA.upmsg}, r^l, \mathsf{RSA}.w^l, \mathsf{RSA.upmsg}^l, r^h, \mathsf{RSA}.w^h, \mathsf{RSA.upmsg}^h) = \mathsf{upmsg}$.
3. If $\mathsf{RSA.upmsg}^l \neq \perp$:
   (a) If x is in the range denoted by r^l: return $(r^l, \mathsf{RSA}.w^l)$.
   (b) Otherwise: $\mathsf{RSA}.w = \mathsf{RSA.MemWitUpOnAdd}(x, \mathsf{RSA}.w, \mathsf{RSA.upmsg}^l)$.
4. If $\mathsf{RSA.upmsg}^h \neq \perp$:
   (a) If x is in the range denoted by r^h: return $(r^h, \mathsf{RSA}.w^h)$.
   (b) Otherwise: $\mathsf{RSA}.w \leftarrow \mathsf{RSA.MemWitUpOnAdd}(x, \mathsf{RSA}.w, \mathsf{RSA.upmsg}^h)$.
5. If $\mathsf{RSA.upmsg} \neq \perp$: $\mathsf{RSA}.w \leftarrow \mathsf{RSA.MemWitUpOnDel}(x, \mathsf{RSA}.w, \mathsf{RSA.upmsg})$
6. Return $u = (r, \mathsf{RSA}.w)$.

$\mathsf{NonMemWitUpOnDel}(a, x, u, \mathsf{upmsg})$ (executed by the witness holder upon receipt of upmsg):

1. Parse $(r, \mathsf{RSA}.w) = u$.
2. Parse $(\mathsf{RSA.upmsg}^l, \mathsf{RSA.upmsg}^h, r', \mathsf{RSA}.w', \mathsf{RSA.upmsg}) = \mathsf{upmsg}$.
3. If x is in the range denoted by r':
   (a) Return $(r', \mathsf{RSA}.w')$
4. Otherwise:
   (a) If $\mathsf{RSA.upmsg}^l \neq \perp$: $\mathsf{RSA}.w \leftarrow \mathsf{RSA.MemWitUpOnDel}(x, \mathsf{RSA}.w, \mathsf{RSA.upmsg}^l)$.
   (b) If $\mathsf{RSA.upmsg}^h \neq \perp$: $\mathsf{RSA}.w \leftarrow \mathsf{RSA.MemWitUpOnDel}(x, \mathsf{RSA}.w, \mathsf{RSA.upmsg}^h)$.
   (c) If $\mathsf{RSA.upmsg} \neq \perp$: $\mathsf{RSA}.w \leftarrow \mathsf{RSA.MemWitUpOnAdd}(x, \mathsf{RSA}.w, \mathsf{RSA.upmsg})$.
   (d) Return $u = (r, \mathsf{RSA}.w)$.

Figure 3·10: Range-RSA Witness Holder Algorithms

$\mathsf{VerNonMem}(a_t, x, u)$ (executed by any third party):

1. Parse $w = (r, \mathsf{RSA}.w)$.
2. $b_1 \leftarrow \mathsf{RSA.VerMem}(\mathsf{RSA}.a, r, \mathsf{RSA}.w)$
3. $b_2 = 1$ if x is in the range represented by r, and $b_2 = 0$ otherwise.
4. Return 1 if $b_1 = b_2 = 1$, and return 0 otherwise.

Figure 3·11: Range-RSA Third Party Algorithms

1. $\mathsf{low} \leq x \leq \mathsf{high}$,
2. $r = \mathsf{low} * 2^{2l} + \mathsf{high} * 2^l + \mathsf{suffix}$, and
3. $w^r \equiv a \bmod n$ (where n is the appropriate RSA integer).

Given that the witness holder (also referred to as the prover) wants to keep all of x, low and high secret, she will first commit to all of $x, w, r, \mathsf{low}, \mathsf{high}$ and suffix, and then use those commitments to prove statements about the underlying values. A suitable commitment scheme is the Fujisaki-Okamoto (FO) commitment scheme (Fujisaki and Okamoto, 1997).

For all the statements that the witness holder needs to prove, there exist standard techniques in the literature. These techniques, together with their conjunctions, come

from a standard zero knowledge proof toolbox:

- proofs of knowledge of a committed value (i.e. knowledge of discrete logarithm representation modulo a composite (Fujisaki and Okamoto, 1997)),
- proofs of equality of committed values (i.e. proof of knowledge of equality of discrete logarithms modulo a composite (or two different composites) (Camenisch and Michels, 1999b)), and
- proofs that a committed value is the product of two other committed values (Camenisch and Michels, 1999a).

All of the above-mentioned proofs are sound under the strong RSA assumption.

More specifically, to prove item 1 in the list above, the witness holder will use a range proof (Boudot, 2000; Lipmaa, 2003). A range proof involves showing that $x - \mathsf{low} \geq 0$ and that $\mathsf{high} - x \geq 0$. To do so, one can use the observation that any non-negative number can be represented as a sum of four squares. The prover would have to find these four squares for each of $x - \mathsf{low}$ and $\mathsf{high} - x$, commit to them, and use commitment equality and product proofs to demonstrate that each of $x - \mathsf{low}$ and $\mathsf{high} - x$ is, in fact, a sum of four squares.

To prove item 2, the witness holder will again have to use commitments to all of the elements in question: r, low, high and suffix. She will then use the homomorphic properties of FO commitments to obtain a single commitment to $\mathsf{low} * 2^{2l} + \mathsf{high} * 2^{l} + \mathsf{suffix}$, and use a commitment equality proof to show that the resulting commitment is to r.

Finally, to prove item 3, the witness holder will show that a committed value has been accumulated. Range-RSA uses the CL-RSA accumulator (Camenisch and Lysyanskaya, 2002) as an underlying building block, so this proof will be done as described by Camenisch and Lysyanskaya (Camenisch and Lysyanskaya, 2002).

Note that most of these proofs require interaction between the prover and the

verifier. One could apply the Fiat-Shamir heuristic (Fiat and Shamir, 1987) to obtain a non-interactive zero-knowledge proof, but this would require assuming random oracles.

Zero-Knowledge Proofs of Member Knowledge in $\mathsf{BraavosB}$ To prove member knowledge for $\mathsf{BraavosB}$ in zero knowledge, a witness holder would have to compute the following zero-knowledge proof (described using Camenisch-Stadler (Camenisch and Stadler, 1997) notation):

$$\begin{aligned} &\mathsf{ZKP}[(x, r, \mathsf{ACC_A}.w, \mathsf{ACC_{NA}}.w) : \\ &\quad \wedge \mathsf{ACC_P}.\mathsf{VerMem}(\mathsf{ACC_P}.a, x, \mathsf{ACC_P}.w) \\ &\quad \wedge \mathsf{ACC_N}.\mathsf{VerNonMem}(\mathsf{ACC_N}.a, x, \mathsf{ACC_N}.u) \\ &\ ](\mathsf{ACC_{NA}}.a, \mathsf{ACC_A}.a) \end{aligned}$$

Where $\mathsf{ACC_P}$ is the signature scheme $\mathsf{SIG}_{CL} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ due to Camenisch and Lysyanskaya (Camenisch and Lysyanskaya, 2003), and $\mathsf{ACC_N}$ is the range-RSA accumulator.

If integration into a larger system (where it is important to be able to link the witnesses used in the proof to other statements) is desired, the witness holder computes commitments to the membership witness $\mathsf{ACC_P}.w$ and non-membership witness $\mathsf{ACC_N}.u$: $(c_1, o_1) = Com.\mathsf{Commit}(\mathsf{ACC_P}.w)$ and $(c_2, o_2) = Com.\mathsf{Commit}(\mathsf{ACC_N}.u)$, where o_1 and o_2 are decommitment values. The proof is then enhanced, as follows:

$$
\begin{aligned}
\mathsf{ZKP}[(&x, \mathsf{ACC_P}.w, \mathsf{ACC_N}.u, o_1, o_2) : \\
&Com.\mathsf{Verify}(c_1, \mathsf{ACC_P}.w, o_1) \\
&\wedge Com.\mathsf{Verify}(c_2, \mathsf{ACC_N}.u, o_2) \\
&\wedge \mathsf{ACC_P}.\mathsf{VerMem}(\mathsf{ACC_P}.a, x, \mathsf{ACC_P}.w) \\
&\wedge \mathsf{ACC_N}.\mathsf{VerNonMem}(\mathsf{ACC_N}.a, x, \mathsf{ACC_N}.u) \\
&](\mathsf{ACC_P}.a, \mathsf{ACC_N}.a, c_1, c_2)
\end{aligned}
$$

For concrete descriptions of the individual clauses of this proof, please refer to Fujisaki and Okamoto (Fujisaki and Okamoto, 1997) and Camenisch and Lysyanskaya (Camenisch and Lysyanskaya, 2003; Camenisch and Lysyanskaya, 2002).

3.5 Comparison with Other Constructions

The Braavos, Braavos′ and BraavosB accumulators are positive, dynamic accumulators with efficient (constant-time) membership witness generation, and no membership witness updates upon element additions — only upon element deletions. In particular, for a fixed security parameter λ, these accumulators achieve the total communication lower bound shown by Camacho (Camacho, 2009). (Total communication refers to the sum of the sizes of all upmsg messages sent by the accumulator manager to the witness holders after $|A|$ additions and $|D|$ deletions.) In Section 3.6, we prove that adding universality would necessarily degrade the total communication of Braavos.

Accumulator	Sigs	RSA	BM	Merkle	DistAccs	range-RSA [9]	BraavosB	CLRSAB	Braavos	Braavos′
Protocol Runtimes										
Update(Op = Add, ...)	1	1	1	$\log a$	$\log a$	1	1	1	1	1
Update(Op = Del, ...)	–	1	1	$\log a$	$\log a$ (with additional inputs)[10]	1	1	1	1	1
WitCreate(stts = in, ...)	1	1 with sk, n without	n	$\log a$	–[11]	–	–	–	–	–
WitCreate(stts = out, ...)	–	n	n	$\log a$	–	1	–	–	–	–
WitUp(stts = in, ...) upon addition	0	1	1	$\log a$	$\log a$	–	0	0	0	0
WitUp(stts = in, ...) upon deletion	–	1	1	$\log a$	$\log a$	–	1	1	1	1
WitUp(stts = out, ...) upon addition	–	1	1	$\log a$	–	1	–	–	–	–
WitUp(stts = out, ...) upon deletion	–	1	1	$\log a$	–	1	–	–	–	–
VerStatus(stts = in, ...)	1	1	1	$\log a$	$\log a$	–	1	1	1	1
VerStatus(stts = out, ...)	–	1	1	$\log a$	–	1	–	–	–	–
Storage										
Accumulator size	1	1	1	1	$\log a$	1	1	1	1	1
Witness size	1	1	1	$\log a$	$\log a$	1	1	1	1	1
Manager storage ($\|m\|$)	1	n	n	a	$\log a$	n	d	1	1	$O(a)$
Communication										
Total comm. to Verifier [12]	0	$a+d$	$a+d$	$(a+d)\log a$	$(a+d)\log a$	$a+d$	d	d	d	d
Total comm. to Witness Holder	0	$a+d$	$a+d$	$(a+d)\log a$	$(d+\log a)\log a$	$a+d$	d	d	d	d
Other Properties										
Additive?	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Subtractive?		✓	✓	✓	✓	✓	✓	✓	✓	✓
Positive?	✓	✓	✓	✓	✓		✓	✓	✓	✓
Negative?		✓	✓	✓		✓				
Strong?		[13]		✓	✓					
Efficient ZKPs?	✓	✓	✓			✓	✓	✓	✓	✓
Adaptively sound?	✓	✓	✓	✓	✓	✓	✓		✓	✓
Add-Delete unlinkable?	–						✓	✓	✓	✓
Hiding Update-Message?	✓									✓
Fully distributed?	✓	✓	✓		✓			✓	✓	
Old accumulator compatible?	✓				✓					

Figure 3·12: Accumulator construction comparison. This figure describes various accumulators and their protocol runtimes, storage requirements, and properties. We let a denote the number of elements added to the accumulator, d denote the number of elements deleted from the accumulator, and n denote the total number of member elements in the accumulator. (Note that n is $a-d$.) The DistAccs accumulator (Chapter 4) is the first strong accumulator with witness update frequency sublinear in $a+d$. The Braavos, Braavos′ and BraavosB (Chapter 3) accumulators are the first adaptively sound dynamic (additive and subtractive) accumulators to have the optimal total communication of $O(d)$. CLRSAB and Range-RSA are building blocks used for Braavos, Braavos′ and BraavosB. Sigs represents any digital signature scheme. The RSA Construction is due to (Benaloh and de Mare, 1994; Camenisch and Lysyanskaya, 2002; Li et al., 2007a). The BM (bilinear map) construction is due to (Nguyen, 2005; Damgård and Triandopoulos, 2008; Au et al., 2009). The Merkle tree construction is due to (Camacho et al., 2008). Big-O notation is omitted from this table in the interest of brevity.

In Figure 3·12, we compare Braavos, Braavos′ (described in Section 3.3), and BraavosB (described in Section 3.4) to prior constructions in terms of the properties introduced in Chapter 2. (DistAccs, described in Chapter 4, is also included in the comparison.) We compare them to digital signatures, and to the three other primary lines of work on accumulators: the RSA construction (Benaloh and de Mare, 1994; Camenisch and Lysyanskaya, 2002; Li et al., 2007a), the bilinear map constructions (Nguyen, 2005; Damgård and Triandopoulos, 2008; Au et al., 2009), and the Merkle tree constructions (Camacho et al., 2008). In our comparison we also include two building blocks: the CLRSAB accumulator (used in Braavos and Braavos′, and described in Section 3.3.1), and the range-RSA accumulator (used in BraavosB, and described in Section 3.4.1).[15]

Though Figure 3·12 includes some of the most well known accumulator constructions to compare with Braavos, Braavos′ and BraavosB, we would like to note that there exists a large number of other dynamic accumulator constructions in the literature (Li et al., 2007a; Damgård and Triandopoulos, 2008; Camenisch et al., 2009; Au et al., 2009; Catalano and Fiore, 2013; Derler et al., 2015). To the best of our knowledge, these constructions do not achieve the efficiency we aim for.

[10]Here range-RSA is presented as a negative accumulator, because that is how it is used in the BraavosB accumulator. However, range-RSA can be easily modified to be universal.

[11]Refer to Section 4.6 for details.

[12]Though DistAccs is a positive accumulator, due to its distributed nature it does not support witness creation other than at the time of element addition.

[13]a and d refer to the number of elements added and deleted *after* the addition of the element whose witness updates are being discussed.

[14]Sander (Sander, 1999) shows a way to make the RSA accumulator strong by choosing the RSA modulus in such a way that its factorization is never revealed.

[15]Our comparison only considers secure accumulators in the standard model, excluding random-oracle-enhanced constructions.

3.6 Appendix A: Lower Bound on Total Communication in Negative Accumulators

Very importantly, note that unlike prior schemes in the standard model, in Braavos and BraavosB no witness updates need to be performed when new elements are added to the accumulator. We achieve the lower bound given by Camacho (Camacho, 2009) which states that $|D|$ deletions requires the total size of update messages upmsg to be of size $\Omega(|D|)$ (we are ignoring the implicit factor of $\log |S|$ used in their proof). We prove that while achieving this lower bound for dynamic accumulators, we cannot also support universality.

Theorem 6. *In a negative (or universal) accumulator, $|A|$ additions require the total size of update messages to be of size $\Omega(|A|)$.*

We prove this theorem in the style of Camacho (Camacho, 2009).

Proof. Assume a witness holder has l non-membership witnesses. The accumulator manager then adds a set of elements A to the accumulator. The witness holder must be able to determine which of the elements for which he holds non-membership witnesses have been added to the accumulator, simply by bringing his non-membership witnesses up to date and determining which of them are still valid. This must be true even if the witness holder holds non-membership witnesses for a superset of A. Thus, the update messages must specify all of A. Specifying a set A requires at least $\log\left(\binom{l}{|A|}\right) \geq |A| \log \frac{l}{|A|}$ bits of information. (The factor of $\log \frac{l}{|A|}$ can be ignored, as it is implicit in the size of the elements.) □

Chapter 4

Efficient Asynchronous Accumulators for Distributed PKI

The contents of this section are a collaboration with Leonid Reyzin (Reyzin and Yakoubov, 2016).

4.1 Introduction

One significant problem with accumulators in the context of distributed applications is that all existing strong constructions require that membership witnesses be updated every time a new element is added into the accumulator. If elements are added to the accumulator at a high rate, having to perform work linear in the number of new elements in order to retain the ability to prove membership can be prohibitively expensive.

In this work, we introduce a new strong accumulator construction which requires only a logarithmic amount of work (in the number of subsequent element additions) in order to keep a witness up to date. Unlike any prior construction, our accumulator construction also supports the verification of an up-to-date witness against an outdated accumulator, enabling verification by parties who are offline and without access to the most recent accumulator. Our construction is made even more well suited for distributed applications by the fact that it does not require any additional storage for the execution of accumulator updates. Section 4.3 describes our construction in detail, and provides comparisons to prior constructions.

Application: Distributed State The original distributed applications proposed by Benaloh and DeMare (Benaloh and de Mare, 1994) involved a canonical common state, but did not specify how to maintain it. Public append-only bulletin boards, such as the ones implemented by bitcoin (Nakamoto, 2008) and its alternatives (alt-coins, such as namecoin (Namecoin, norg)), provide a place for this common state. Bitcoin and altcoins implement this public bulletin board by means of block chains; in bitcoin they are used primarily as transaction ledgers, while altcoins extend their use to public storage of arbitrary data.

Altcoins such as namecoin can be used for storing identity information in a publicly accessible way. For instance, they can be used to store (IP address, domain) pairs, enabling DNS authentication (Slepak, 2013). They can also be used to store (identity id, public key pk) pairs, providing a distributed alternative to certificate authorities for public key infrastructure (PKI) (Yakoubov et al., 2014).

Elaborating on the PKI example, when a user Bob registers a public key pk_{Bob}, he adds the pair ("Bob", pk_{Bob}) to the bulletin board. When Alice needs to verify Bob's public key, she could look through the bulletin board to find this pair. However, when executed naively, this procedure would require Alice to read the entire bulletin board—i.e., a linear amount of data. Bob can save Alice some work by sending her a pointer to the bulletin board location where ("Bob", pk_{Bob}) is posted; however, that would still require that Alice *have access* to a linear amount of data during verification. What if Alice doesn't have access to the bulletin board at the time of verification at all, or wants to reduce latency by avoiding on-line access to the bulletin board during verification?

Our accumulator construction can be used in this setting to free Alice from the need for on-line random access to the bulletin board (Yakoubov et al., 2014) (see also (Garman et al., 2014) for a similar use of accumulators). It allows her to instead

simply download a small amount of data from the end of the bulletin board at predetermined (perhaps infrequent) intervals. The accumulator would contain all of the (id, pk) pairs on the bulletin board, with responsibility for the witnesses distributed among the interested individuals. When Bob posts ("Bob", pk_{Bob}) to the bulletin board, he also adds ("Bob", pk_{Bob}) to the accumulator, and stores his witness w_{Bob}. He posts the updated accumulator to the bulletin board, and since our accumulator construction is trapdoor-free and deterministic, the validity of the new accumulator can be checked by all parties simply by re-adding ("Bob", pk_{Bob}) to the old accumulator. Details of how such posts are monitored and validated can be found in (Yakoubov et al., 2014).

Then, when Alice wants to verify that pk_{Bob} is indeed the public key belonging to Bob, all she needs is w_{Bob} and a locally cached accumulator value. As long as Bob's bulletin board post pre-dates Alice's locally cached accumulator value, Alice can use that accumulator value and w_{Bob} to verify that ("Bob", pk_{Bob}) has been posted to the bulletin board. She does not need to refer to any of the new bulletin board contents, because in our scheme (as opposed to other accumulator schemes), an up-to-date witness can be used for verification even against an outdated accumulator (as long as the addition of the element in question pre-dates the accumulator).

Our construction also reduces the work for Bob, as compared to other accumulator constructions. In a typical accumulator construction, Bob needs to update w_{Bob} every time a new (id, pk) pair is added to the accumulator. However, in a large-scale PKI, the number of entries on the bulletin board and the frequency of element additions can be high. Thus, it is vital to spare Bob the need to be continuously updating his witness. Our accumulator reduces Bob's burden: Bob needs to update his witness only a logarithmic number of times. Moreover, Bob can update his witness on-demand—for instance, when he needs to prove membership—by looking at a logarithmic number

of bulletin board entries (see Section 4.5 for details).[1]

4.2 Definitions

In this chapter, we introduce the following three additional accumulator properties that are particularly relevant when an accumulator is used to maintain distributed state.

Full Distribution We consider an accumulator to be *fully distributed* if there is no party (including the accumulator manager, if one exists) which must store an amount of information that is linear or super-linear in the number of elements in the accumulator. That is, the parameter m (if it exists) must be sub-linear in size. (Note that all other parameters are already assumed to be sub-linear.)

Low Update Frequency We consider an accumulator to have a *low update frequency* if the frequency with which a witness for element x needs to be updated is sub-linear in the number of elements which are added after x.

Old Accumulator Compatibility We consider an accumulator to be *old accumulator compatible* if up-to-date witnesses w_t^x can be verified even against an outdated accumulator $a_{t'}$ where $t' < t$, as long as x was added to the accumulator before t'. Note that this does not compromise the soundness property of the accumulator, because if x was not a member of the accumulator at t', w_t^x does not verify with $a_{t'}$. Old accumulator compatibility allows the verifier to be offline and out of sync with the latest accumulator state.

[1] The question of whether accumulators updates can be batched, as in our scheme, was first posed by Fazio and Nicolosi (Fazio and Nicolosi, 2003) in the context of dynamic accumulators (i.e., accumulators that support deletions as described in Chapter 2). It was answered in the negative by Camacho (Camacho and Hevia, 2010), but only in the context of deletions, and only in the centralized case (when all witnesses are updated by the same entity).

4.3 Building Distributed Accumulators

There are several known accumulator constructions, including the RSA construction (Benaloh and de Mare, 1994; Camenisch and Lysyanskaya, 2002; Li et al., 2007a), the Bilinear Map construction (Nguyen, 2005; Damgård and Triandopoulos, 2008; Au et al., 2009), and the Merkle tree construction (Camacho et al., 2008). Their properties are described in Figure 3·12. None of these constructions have low update frequency or old-accumulator compatibility. The construction given in (Camacho et al., 2008), which is similar to ours in that both are based on Merkle trees, is made more complicated and somewhat less efficient by the fact that it is designed it to be universal. We present a different Merkle tree construction which, unlike the construction given in (Camacho et al., 2008), is fully distributed, old-accumulator compatible and saves on update frequency, but is not universal.

4.4 Construction

Let n be the number of elements in our accumulator, and let h be a collision-resistant hash function. (When h is applied to pairs or elements, the pair is encoded in such a way that it can never be confused with a single element x – e.g., a pair is prefaced with a 1, and a single element with a 0.)

The accumulator maintains a list of $D = \lceil \log(n+1) \rceil$ elements $r_0, \ldots, r_D$ (as opposed to just one Merkle tree root). The element r_i is the root of a complete Merkle tree with 2^i leaves if and only if the ith least significant bit of the binary expansion of n is 1. Otherwise, $r_i = \perp$. A witness w^x for x is the authenticating path for x in the Merkle tree that contains x. That is, $w^x = ((z_0, \mathsf{dir}_0), \ldots, (z_d, \mathsf{dir}_{d-1}))$, where each z_i is in the range of the hash function h, and each dir is either right or left. These are the (right / left) sibling elements of all of the nodes along the path from element x to the accumulator root of depth d. An illustration of an accumulator a

and a witness w is given in Figure 4·1.

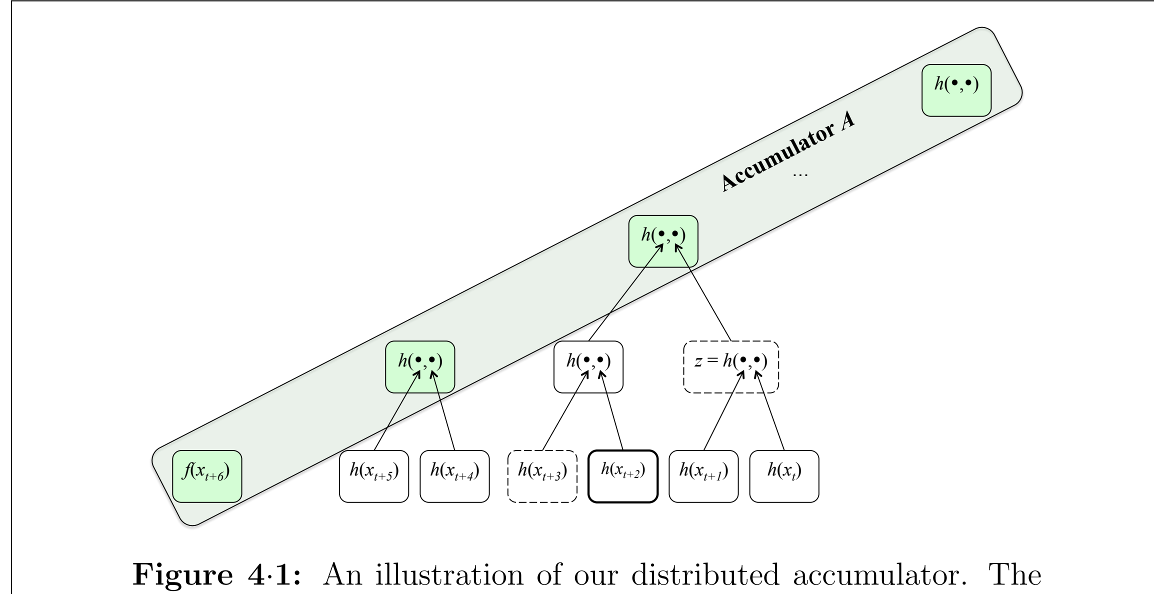


Figure 4·1: An illustration of our distributed accumulator. The accumulator itself is shaded; the unshaded elements are elements of the Merkle trees which are not actually a part of the accumulator. The elements with dashed outlines belong to the authenticating path for x_{t+2} (which itself has a bold outline). So, the witness for x_{t+2} would be $w^{x_{t+2}} = ((\mathsf{h}(x_{t+3}), \mathsf{left}), (z, \mathsf{right}))$.

Verification is done by using the authenticating path w^x and the element x in question to recompute the Merkle tree root and check that it indeed matches the accumulator element r_d, where d is the length of w^x. In more detail, this is done by recomputing the *ancestors* of the element x using the authenticating path w^x as described in Algorithm 1, where the ancestors are the nodes along the path from x to its root, as defined by x and by elements in w^x. If the accumulator is up to date, the last ancestor should correspond to the appropriate accumulator element r_d. If the accumulator is outdated but still contains x, one of the recomputed ancestors should still correspond to one of the accumulator elements. Verification is described in full detail in Algorithm 5 of Section 4.7.

Element addition is done by merging Merkle trees to create deeper ones. Specifi-

cally, when the nth element x is added to $a = [r_0, \ldots, r_D]$, if $r_0 = \perp$, we set $r_0 = \mathsf{h}(x)$. If, however, $r_0 \neq \perp$, we "carry" exactly as we would in a binary counter: we create a depth-one Merkle tree root $z = \mathsf{h}(\mathsf{h}(x), r_0)$, set $r_0 = \perp$, and try our luck with r_1. If $r_1 = \perp$, we can set $r_1 = z$. If $r_1 \neq \perp$, we must continue merging Merkle trees and "carrying" further up the chain. Element addition is described in full detail in Algorithm 3 of Section 4.7.

Membership witness updates need to be performed only when the root of the Merkle tree containing the element in question is merged, or "carried", during a subsequent element addition. This occurs at most D times. Membership updates use the update message $\mathsf{upmsg}_{t+1} = (y, w^y_{t+1})$ (where y is the element being added and w^y_{t+1} is the witness generated for y) in order to bring the witness w^x_t for the element x up to date, as described in Algorithm 4 of Section 4.7.

Properties This construction is trivially correct. It is sound as long as h is collision resistant. Soundness can be proven using the classical technique for Merkle trees: if an adversary $\mathcal{A}$ can find a witness for an element that has not been added to the accumulator, then $\mathcal{A}$ can be used to find a collision for h.

This construction is strong, since every operation is deterministic and publicly verifiable. It is also fully distributed; all storage requirements are logarithmic in the number of elements. No auxiliary storage m (as described in Section 4.2) is necessary for accumulator updates.

Section 4.5 discusses the membership witness update frequency of the construction; Section 4.6 discusses how the construction can be modified to support a limited notion of dynamism.

4.5 Infrequent Membership Witness Updates

As highlighted in Section 4.4, this accumulator scheme requires that the witness for a given element x be updated at most $D = \lceil \log(n+1) \rceil$ times, where n is the number of elements added to the accumulator after x. One might observe that having to *check* whether the witness needs updating each time a new element addition occurs renders this point moot, since this check itself must be done a linear number of times. However, we can get around this by giving our witness holders the ability to "go back in time" to observe past accumulator updates. If they can ignore updates when they occur, and go back to the relevant ones when they need to bring their witness up to date (e.g. at when they need to show it to a verifying third party), they can avoid looking at the irrelevant ones altogether.

"Going back in time" is possible in the public bulletin board setting of many distributed applications. Recall the application from Section 4.1, in which our accumulator is maintained as part of a public bulletin board. The bulletin board is append-only, so it contains a history of all of the accumulator states. Along with these states, we will include the update message, and a counter indicating how many additions have taken place to date. Additionally, we will include pointers to a selection of other accumulator states, so as to allow the bulletin board user to move amongst them efficiently. The pointers from accumulator state t would be to accumulator states $t - 2^i$ for all i such that $0 < 2^i < t$ (somewhat similarly to what is done in a skip-list). These pointers can be constructed in logarithmic time: there is a logarithmic number of them, and each of them can be found in constant time by using the previous one, since $t - 2^i = t - 2^{i-1} - 2^{i-1}$. Note that storing these pointers is not a problem, since we are already storing a logarithmic amount of data in the form of the accumulator and witness.

Our witness holder can then ignore update messages altogether, performing no

checks or work at all. Instead, he updates his witness only when he needs to produce a proof. When this happens, he checks the counter of the most recently posted accumulator state. The counter alone is sufficient to deduce whether his witness needs updating. If his witness does not need updating, he has merely performed a small additional constant amount of work for the verification at hand. If, as happens a logarithmic number of times, his witness does need updating, the pointers and counters allow him to locate in logarithmic time the (at most logarithmic number of) bulletin board entries he needs to access in order to bring his witness up to date, as described in Algorithm 9 of Section 4.7. Thus, the total work performed by our witness holder will remain logarithmic in the number of future element additions.

4.6 Limited Dynamism

We can make our accumulator construction dynamic by giving the accumulator manager auxiliary storage m consisting of the leaves of the Merkle trees (i.e., the set of elements in the accumulator). Then, to perform a deletion Del, the manager replaces the leaf in the tree corresponding to x with $\perp$, updates the ancestors of this leaf, and broadcasts the updated ancestors of $\perp$ as the update message upmsg. To perform a witness update (upon receipt of upmsg), each witness holder whose value x is in the same Merkle tree replaces one node on its path (namely, the child node of the lowest common ancestor of the deleted value and x) with the corresponding value from upmsg.

This modification degrades the space efficiency of the manager by adding auxiliary linear storage on top of the very short (logarithmic) accumulator, thus compromising full distribution. (We note that this extra storage can be avoided if the witness holder, or perhaps several other cooperating witness holders, can provide the necessary portions of the Merkle tree to the manager when needed. However, this would only

truly work if withdrawing an element from the accumulator was a voluntary act—for instance, this would not work in the application of credential revocation.) This modification will also degrade the low update frequency property, and old accumulator compatibility.

To keep both full distribution and low update frequency, we can limit deletions to newer elements (e.g. an element can only be deleted within a constant number of turns of being added), since newer elements are in the small trees. While this appears to be limiting, it should be noted that in many applications, deletions of older elements may be avoided altogether by wrapping "time to live stamps" or "expiration dates" into the elements themselves.

4.7 Appendix 1: Algorithms

In this section, we give the pseudocode for all of the algorithms used in our accumulator scheme. A Python implementation of these algorithms is available upon request.

4.7.1 Accumulator Algorithms

In this section, we give the pseudocode for the basic accumulator algorithms, such as Gen (Algorithm 2), Add (Algorithm 3), $\mathsf{MemWitUpOnAdd}$ (Algorithm 4) and VerMem (Algorithm 5). (Note that usually, in the context of dynamic accumulators, we denote Add as $\mathsf{Update}(\mathsf{Op} = \mathsf{Add}, \dots)$, $\mathsf{MemWitUpOnAdd}$ as $\mathsf{WitUp}(\mathsf{stts} = \mathsf{in}, \mathsf{Op} = \mathsf{Add}, \dots)$, and VerMem as $\mathsf{VerStatus}(\mathsf{stts} = \mathsf{in}, \dots)$. However, here we use different notation in the interest of bervity.)

Recall that h is a hash function.

Algorithm 1 GetAncestors: a helper function for MemWitUpOnAdd (Algorithm 4) and VerMem (Algorithm 5).

Require: p, x

1: $c = \mathsf{h}(x)$
2: $\overline{p} = [c]$
3: **for** (z, dir) in p **do**
4:     **if** dir = right **then**
5:         $c = \mathsf{h}(c||z)$
6:     **else if** dir = left **then**
7:         $c = \mathsf{h}(z||c)$
8:     **end if**
9:     append c to $\overline{p}$
10: **end for**
11: **return** $\overline{p}$

Algorithm 2 Gen

Require: 1^λ

1: **return** $a_0 = \perp$

Algorithm 3 Add

Require: a_t, x

1: $a_{t+1} = a_t$ (the new accumulator starts out as a copy of the old one)
2: $w^x_{t+1} = []$ (the witness starts out as an empty list)
3: $d = 0$ (the depth of the witness starts out as 0)
4: $z = \mathsf{h}(x)$
5: **while** $a_{t+1}[d] \neq \perp$ **do**
6:     **if** the length of $a_{t+1} < d + 2$ **then**
7:         append $\perp$ to a_{t+1}
8:     **end if**
9:     $z = \mathsf{h}(a_{t+1}[d]||z)$
10:     append $(a_{t+1}[d], \mathsf{left})$ to w^x_{t+1}
11:     $a_{t+1}[d] = \perp$
12:     $d = d + 1$
13: **end while**
14: $a_{t+1}[d] = z$
15: **return** $a_{t+1}, w^x_{t+1}, \mathsf{upmsg} = (x, w^x_{t+1})$

Algorithm 4 MemWitUpOnAdd

Require: y, w^y_{t+1}, w^x_t

1: let d^x_t be the length of w^x_t
2: let d^y_{t+1} be the length of w^y_{t+1}
3: **if** $d^y_{t+1} < d^x_t$ **then**
4: **return** w^x_t (the witness has not changed)
5: **else**
6: $d^x_{t+1} = d^y_{t+1}$
7: $w^x_{t+1} = w^x_t$ (the new authenticating path starts out as a copy of the old one)
8: $\overline{w}^y_{t+1} = \mathsf{GetAncestors}(w^y_{t+1}, y)$
9: append $(\overline{w}^y_{t+1}[d^x_t], \mathsf{right})$ to w^x_{t+1}
10: append $w^y_{t+1}[d^x_t + 1, \ldots]$ to w^x_{t+1}
11: **return** w^x_{t+1}
12: **end if**

Algorithm 5 VerMem

Require: a_t, x, w^x

1: $\overline{p} = \mathsf{GetAncestors}(w^x, x)$
2: **if** a_t and r have any elements in common (computable in linear time) **then**
3: **return** TRUE
4: **else**
5: **return** FALSE
6: **end if**

4.7.2 Batch Witness Updates

In this section, we give the pseudocode for the algorithms which allow our witness holder to avoid reading to every update message, and instead do only a logarithmic amount of work upon every verification in order to bring the witness up to date. In the following algorithms, we assume the existence of a public append-only random access bulletinboard. bulletinboard[ptr] gives the ptrth entry of bulletinboard. However, since bulletinboard may be used for things other than accumulator entries, the ptrth entry of bulletinboard is not guaranteed to correspond to the ptrth accumulator update. Instead, we let $t' = \mathsf{bulletinboard}[\mathsf{ptr}].t$ denote the timestep t' such that the ptrth entry of bulletinboard corresponds to the t'th accumulator update. GetPointers (Algorithm 6) and GetPointer (Algorithm 7) are helper algorithms for creating the pointers and using them to move amongst the entries of bulletinboard which are relevant to the accumulator.

Let i be the number of irrelevant entries on bulletinboard after the last relevant entry, and let n be the number of elements which have been added to the accumulator. GetPointers (Algorithm 6), which finds the pointer to include in a new bulletin board entry, takes $O(i)+O(\log(n))$ time. (The $O(i)$ is present because GetPointers finds the newest accumulator bulletinboard entry by iterating over the entries of bulletinboard backwards.) Similarly, GetPointer (Algorithm 7), which finds a pointer to a desired bulletin board entry given another pointer at which to start, takes $O(\log(n))$ time.

BatchMemWitUpOnAdd (Algorithm 9), the batch witness update algorithm itself, also takes $O(\log(n))$ time. BatchMemWitUpOnAdd reverses the list of relevant indices before finding the pointers to them for reasons of efficiency. This way, the total number pointers followed is $O(\log(n))$, and not $O(\log(n)^2)$. The list of relevant pointers is then reversed again, so as to perform the actual membership witness updates in order.

Algorithm 6 GetPointers: finds all the pointers needed for a new accumulator update bulletin board entry. If the accumulator update happens at timestep t, the pointers should be to all accumulator updates at timesteps $t - 2^i$ for i such that $0 < 2^i < t$. This is a helper function for BatchMemWitUpOnAdd (Algorithm 9).

Require: the bulletin board bulletinboard

1: ptrs = []
2: find the last occurring addition entry $(\mathsf{lt}, a_{\mathsf{lt}}, y, w_{\mathsf{lt}}^{y})$ on bulletinboard.
3: **if** one does not exist **then**
4:    **return** ptrs
5: **end if**
6: let lptr be the pointer to the last entry
7: append lptr to ptrs
8: exp = 1
9: stuck = FALSE
10: **while** not stuck **do**
11:    lptr = ptrs[−1] (the last element of ptrs)
12:    let numPointersAtLastPointer be the number of pointers stored at bulletinboard[lptr]
13:    **if** numPointersAtLastPointer < exp **then**
14:       stuck = TRUE
15:    **else**
16:       ptr = bulletinboard[lptr].ptrs[exp − 1]
17:       append ptr to ptrs
18:    **end if**
19:    exp = exp + 1
20: **end while**
21: **return** ptrs

Algorithm 7 GetPointer: finds a pointer to the bulletin board entry corresponding to the t^*th accumulator update. This is a helper function for BatchMemWitUpOnAdd (Algorithm 9).

Require: the bulletin board bulletinboard, the timestep t^*, and a pointer ptr to a bulletin board entry corresponding to $t' \geq t^*$

1: t' = bulletinboard[ptr].t
2: **while** $t' \neq t^*$ **do**
3:    difference = $t' - t^*$
4:    let exp be the largest exponent such that 2^{exp} is smaller than difference
5:    ptr = bulletinboard[ptr].ptrs[exp]
6:    t' = bulletinboard[ptr].t
7: **end while**
8: **return** ptr

Algorithm 8 GetUpdateTimeSteps: finds the timesteps at which a witness needs to be updated. This is a helper function for BatchMemWitUpOnAdd (Algorithm 9).

Require: the bulletin board bulletinboard, the timestep lupt at which the last witness update occurred, the timestep lt at which the last accumulator update occurred, and the depth d of the witness in question

1: relevantTimeSteps = []
2: power = 2^d
3: t = lupt + power
4: **while** $t \leq$ lt **do**
5:    append t to relevantTimeSteps
6:    **while** $t \mod$ power $\times 2 = 0$ **do**
7:       power = power $* 2$
8:    **end while**
9:    $t = t +$ power
10: **end while**
11: **return** relevantTimeSteps

Algorithm 9 BatchMemWitUpOnAdd

Require: the bulletin board bulletinboard, the witness w^x, and the timestep lupt at which w^x was last updated

1: let d be the length of w^x
2: find the last occurring addition entry (lt, a_{lt}, upmsg $= (y, w^y_{\mathsf{lt}})$) on bulletinboard, and let ptr be the pointer to this entry
3: relevantTimeSteps = GetUpdateTimeSteps(lupt, lt, d)
4: reverse the order of relevantTimeSteps
5: relevantPointers = []
6: **for** $t \in$ relevantTimeSteps **do**
7:    ptr = GetPointer(bulletinboard, t, ptr)
8:    append ptr to relevantPointers
9: **end for**
10: reverse the order of relevantPointers
11: **for** ptr $\in$ relevantPointers **do**
12:    get $(t, a_t,$ upmsg $= (y, w^y_t))$ using ptr from bulletinboard
13:    w^x = MemWitUpOnAdd(y, w^y_t, w^x)
14: **end for**
15: **return** w^x

Chapter 5

Stronger Notions and Constructions for Multi-Designated Verifier Signatures

The contents of this section are a collaboration with Ivan Damgård, Helene Haagh, Rebekah Mercer, Anca Nitulescu, and Claudio Orlandi1 (Damgård et al., 2019). My primary contribution is the construction based on generic primitives.

5.1 Introduction

Encrypted and authenticated messaging has experienced widespread adoption in recent years, due to the attractive combination of properties offered by, for example, the Signal protocol (Marlinspike, 2013). With so many conversations happening over the internet, there is a growing need for protocols offering security to conversation participants. Encryption can be used to guarantee privacy of message contents, but authenticating messages while maintaining the properties of an in person conversation is more involved. There are two properties of in person conversations related to authenticity that we wish to emulate in the context of digital conversations:

- *Unforgeability*, meaning that the receiver should be convinced that the message actually came from the sender in question, and
- *Off-the-record* or *deniability*, meaning that the receiver cannot later prove to a third party that the message came from the sender.

Off-the-record (OTR) messaging offers a solution to this in the two-party case, enabling authentication of messages such that participants can convincingly deny

having made certain statements, or even having taken part in the conversation at all (Borisov et al., 2004). The protocol deals with encrypted messages accompanied by a message authentication code (MAC) constructed with a shared key. MACs work well in two-party conversations, because for parties S(ender) and R(eciever) with a shared secret key, a MAC attests 'this message comes from S or R'. MACs provide unforgeability, since a party R receiving a message authenticated with such a MAC knows that if this MAC verifies, the message came from S. MACs provide off-the-record (deniable) communication as R cannot convince a third party that a message and MAC originally came from S (since R could have produced it just as easily). More generally, tools that provide unforgeable, off-the-record two-party communication are known as *Designated Verifier Signatures* (DVSs, proposed by (Jakobsson et al., 1996) and (Chaum, 1996)).

When there are multiple recipients, for example in group messaging, the situation becomes more complicated. DVSs have been extended to the multiparty setting under the name of *Multi-Designated Verifier Signatures (MDVSs)* (we give a number of references in Figure 5·1). One might hope that these schemes would work for off-the record group messaging; however, it turns out that existing MDVS definitions and schemes do not have the properties one would naturally ask for. In the following section, we give a motivating example illustrating which properties we should actually ask from an MDVS scheme, and we explain how existing schemes fall short of providing them.

5.1.1 A Motivating Example for MDVS

Imagine a government official Sophia who wants to blow the whistle on some corrupt government activity; e.g., perhaps her colleague, $\mathcal{A}$aron, accepted a bribe. She wants to send a message describing this corruption to Robert, Rachel and Rebekah, who are all Reporters at national newspapers.

Naturally, Sophia wants the Reporters to be convinced that she is the true sender of the message. Otherwise, they would have no reason to believe — or print — the story.

Goal 1 (Unforgeability). *It is vital that each of the Reporters be able to authenticate that the message came from Sophia.*

In order to achieve unforgeability, Sophia produces a signature σ using an MDVS scheme, and attaches it to her message. (In such a scheme, each sender has a private signing key and each recipient has a private verification key.) However, blowing the whistle and reporting on $\mathcal{A}$aron's corrupt activity could put Sophia in danger. If any of Robert, Rachel or Rebekah could use σ demonstrate to $\mathcal{A}$aron that Sophia blew the whistle on him, she could lose her position, or face other grave consequences.

Goal 2 (Source-Hiding / Off-the-Record). *It is vital that the Reporters be unable to prove to an outsider ($\mathcal{A}$aron) that the message came from Sophia.*

One way to guarantee that the Reporters cannot link Sophia to the message is to require that the Reporters can *simulate* a signature σ themselves. Then, if they try to implicate Sophia by showing σ to $\mathcal{A}$aron, he would have no reason to believe them; as far as he is concerned, the Reporters could have produced σ to try to frame Sophia.

All previous constructions only support off-the-record in the limited sense that *all* of the Reporters must collaborate in order to produce a simulated signature.[1] However, this is insufficient. Suppose, for instance, that $\mathcal{A}$aron knows Rachel was undercover — and thus unreachable — for the entire time between the bribery taking place, and Robert and Rebekah bringing σ to $\mathcal{A}$aron. Then he would conclude that Rachel could not have collaborated in simulating σ, and so it must be genuine. Even

[1] One previous work (Tian, 2012) achieves off-the-record when $\mathcal{C}$ consists of a single verifier. However, in this construction a simulated signature created by a malicious verifier will look like a real signature for all other designated verifers, violating unforgeability.

with the off-the-record definition used in prior works, it is still possible that some subset of the Reporters would be able to implicate Sophia in the eyes of $\mathcal{A}$aron. We therefore need a stronger off-the-record defintion.

Contribution 1 (Off-the-record For Any Subset)**.** *We give a stronger definition of the off-the-record property, where* any *subset of Reporters must be able to simulate a signature. A simulation looks like a genuine signature to an outsider, even given the verification keys of the subset that produced it (as well as a number of other signatures that are guaranteed to be genuine).*

Under our stronger definition, no set of Reporters is able to use σ to provably tie Sophia to the message *even if $\mathcal{A}$aron has side information about communication amongst the Reporters* as well as guaranteed-to-be-genuine signatures.

Remark 5. *(The Tension Between Off-The-Record and Unforgeability) Note that, if Rachel did not participate in Robert and Rebekah's signature simulation (e.g. if she was undercover at the time), she will later be able to distinguish the simulation from a real signature produced by Sophia. Otherwise, Robert and Rebekah would have succeeded in producing a forgery that fools Rachel.*

This means that under a sufficiently strong model of attack, we cannot have unforgeability and off-the-record at the same time. Namely, suppose $\mathcal{A}$aron first gets a signature σ from Robert and Rebekah, while preventing them from communicating with Rachel. Then he coerces Rachel into giving him her secret verification key. By the unforgeability property, he can use this key to tell if σ is a simulation. (Note that $\mathcal{A}$aron will be able to tell whether Rachel gives him her true verification key, since he may have other signatures from Sophia that he knows are genuine that he can use to test it. So, she has no choice but to hand over her real verification key.)

Given this observation, we choose to explore the model where the secret keys of all coerced/corrupted verifiers (but not honest ones) can be used to simulate a signature, as this is the strongest model of attack in which both unforgeability and off-the-record can be achieved. As we shall see, even in this model, achieving both properties requires highly non-trivial constructions and implies a lower bound on the size of signatures.

Finally, let us fast forward to the moment when Robert, Rachel and Rebekah receive Sophia's message. They want to print this high-profile story as soon as possible, but of course they want to be sure they won't make themselves look foolish by

printing the story if their colleagues — the other well-respected Reporters listed as recipients — don't believe it actually came from Sophia. The concern here is that Sophia could be dishonest and her actual goal could be to discredit the Reporters. Hence we need another property — consistency, or designated verifier transferability.

Goal 3 (Consistency / Designated Verifier Transferability)**.** *It is desirable that, even if Sophia is malicious, if one of the Reporters can authenticate that the message came from Sophia, all of them can.*

Contribution 2. *We provide the first formal definition of consistency.*

Now that we have covered the basic storyline, let us consider a few possible plot-twists. First, what if $\mathcal{A}$aron is tapping the wires connecting the government building to the outside world? Then he will see Sophia's message — together with her signature σ — as she sends it to the Reporters. In such a situation, we would want the signature σ not to give Sophia— or the Reporters— away.

Goal 4 (Privacy of Identities)**.** *It is desirable that σ shouldn't reveal Sophia's or the Reporters' identities* [2]*. When only the signer's — Sophia's — identity is hidden, this property is called* privacy of signer identity (PSI)*.*

Next, what if, at the time at which Sophia has the opportunity to send out her message, she cannot look up Rebekah's public key securely — perhaps because Rebekah has not yet set up an account on the secure messaging system Sophia uses? Then, it would be ideal for Sophia to need nothing other than Rebekah's identity (and some global public parameters) in order to include her as a designated verifier. Rebekah would then be able to get the appropriate key from a trusted authority such

[2] Note that privacy of identities is related to — but very different from — off-the-record. Neither of these definitions is strictly stronger than the other. *Privacy of identities* is weaker in that it assumes that none of the Reporters help in identifying Sophia as the sender, while *off-the-record* makes no such assumptions. However, *privacy of identities* is stronger in that it requires that σ alone reveal nothing about Sophia's identity to anyone other than the Reporters; *off-the-record* allows such leakage, as long as it is not provable. .

as the International Press Institute[3] (having proved that she is, in fact, Rebekah), and would be able to use that key to verify Sophia's signature.

Goal 5 (Verifier-Identity-Based (VIB) Signing)**.** *It is desirable that Sophia should only need the Reporters' identities, not their public keys, in order to produce her designated verifier signature.*

Contribution 3. *We give the first three constructions that achieve unforgeability, off-the-record with any-subset simulation, and consistency. One of them additionally achieves privacy of identities and verifier-identity-based signing.*

The third construction, which additionally achieves privacy of identities and verifier-identity-based signing, may, at first glance, seem strictly better; however, the price it pays is two-fold. It uses functional encryption (which requires strong computational assumptions), and it requires an involved trusted setup in which a master secret is used to derive verifier keys. Note that such a trusted setup is clearly necessary in order to achieve verifier-identity-based signing.

In contrast, our first two constructions can be instantiated either in the random oracle model, or with a common reference string — in both cases avoiding the need for a master secret key. They use only standard primitives such as pseudorandom functions, pseudorandom generators, key agreement and NIZKs. The first construction uses these primitives in a black-box way; the second construction uses specific instances of these primitives, for concrete efficiency.

In the following subsections, we give an overview of previous work and then discuss our results in more detail.

5.1.2 Flavors of Multi-Designated Verifier Signatures

There are many ways to define MDVS and its properties. Figure 5·1 summarizes the approaches taken by prior work, compared to our own.

[3]This trusted authority can also be distributed; perhaps the master secret is secret-shared across several different institutions, who must collaborate in order to produce a secret verification key.

There are several different flavors of verification. In some MDVS schemes, even a single designated verifier cannot link a signature to the signer; the designated verifiers need to work together in order to verify a signature. Thus, we have two notions of verification: *local* verification and *cooperative* verification (where *all* designated verifiers need to cooperate in order to verify the signature).

Recall that the off-the-record property states that an outsider cannot determine whether a given signature was created by the signer or simulated by the designated verifiers. We have three flavors of such simulateability: *one* designated verifier (out of n) can by himself simulate a signature (as done by (Tian, 2012))[4], *all* designated verifiers need to collude in order to simulate a signature (all other works on MDVS), or *any subset* of the designated verifiers can simulate a signature (this paper). Of course, the simulated signature should remain indistinguishable from a real one even in the presence of the secrets held by the simulating parties.

There is also the standard security property of signature schemes, which is *unforgeability*; no one (except the signer) should be able to construct a signature that any verifier will accept as a valid signature from that signer. There are two flavors of unforgeability. The first is *weak* unforgeability, where designated verifiers can forge, but others cannot. The second is *strong* unforgeability, where a designated verifier can distinguish between real signatures and signatures simulated by other verifiers; that is, even other designated verifiers cannot fool a verifier into accepting a simulated signature.[5]

---

[4]If *only* one designated verifier can simulate a signature, it must be distinguishable from a real signature by other verifiers (by the strong unforgeability property). Two colluding verifiers would be able to prove to an outsider that a given signature is not a simulation by showing that it verifiers for both of them. So, any-subset simulation gives strictly stronger off-the-record guarantees than one-verifier simulation.

[5]Note that when all designated verifiers are needed for the simulation, then a designated verifier will be able to distinguish a simulation from a real signature based on whether he participated in the simulation of the signature. However, if this is the only way he can distinguish, then the signature scheme has *weak* unforgeability, since the simulated signature is still a valid forgery.

Schemes	PSI	Verification	Simulation	Unforgeability	Signature Size
(Jakobsson et al., 1996; Laguillaumie and Vergnaud, 2004; Li et al., 2007b) (Chow, 2008; Vergnaud, 2006; Zhang et al., 2012)	No	Local	All	Weak	$O(1)$
Our work, from standard primitives	No	Local	Any subset $\mathcal{C}$	Strong	$O(\lvert\mathcal{D}\rvert)$
(Ng et al., 2005; Chow, 2006)	Yes	All	All	Weak	$O(\lvert\mathcal{D}\rvert)$
(Ming and Wang, 2008; Seo et al., 2008; Chang, 2011)	Yes	All	All	Weak	$O(1)$
(Shailaja et al., 2006; Laguillaumie and Vergnaud, 2007; Vergnaud, 2006; Zhang et al., 2012)	Yes	Local	All	Weak	$O(\lvert\mathcal{D}\rvert)$
(Tian, 2012)	Yes	Local	One	Weak	$O(1)$
Our work, from FE	Yes	Local	Any subset $\mathcal{C}$ of size up to t	Strong	$O(t)$

Figure 5·1: Existing MDVS constructions and their properties. Let $\mathcal{D}$ be the set of designated verifiers, and $t \leq |\mathcal{D}|$ be an upper bound on the set of colluding designated verifiers $\mathcal{C} \subseteq \mathcal{D}$.

5.1.3 Our Contributions

We propose formal definitions of all the relevant security properties of MDVS in the strongest flavor, including the definition of off-the-record with any-subset simulation. We also give the first formal (game based) definition of consistency, where a corrupt signer can collude with some of the designated verifiers to create an inconsistent signature.

We then give several different constructions of MDVS that achieve these properties, including local verification, off-the-record with any-subset simulation, and strong unforgeability. Our constructions, and the tools they require, are mapped out in Figure 5·2. In particular, these are the first constructions that combine any-subset simulation and with strong unforgeability, as described in Figure 5·1. We get these results at the expense of signature sizes that are larger than in some of the earlier constructions. However, this is unavoidable, as shown in Theorem 7 below.

Theorem 7. *Any MDVS with any-subset simulation and strong unforgeability must have signature size* $\Omega(|\mathcal{D}|)$.

Remark 6. *It may seem from the table that our functional encryption based scheme*

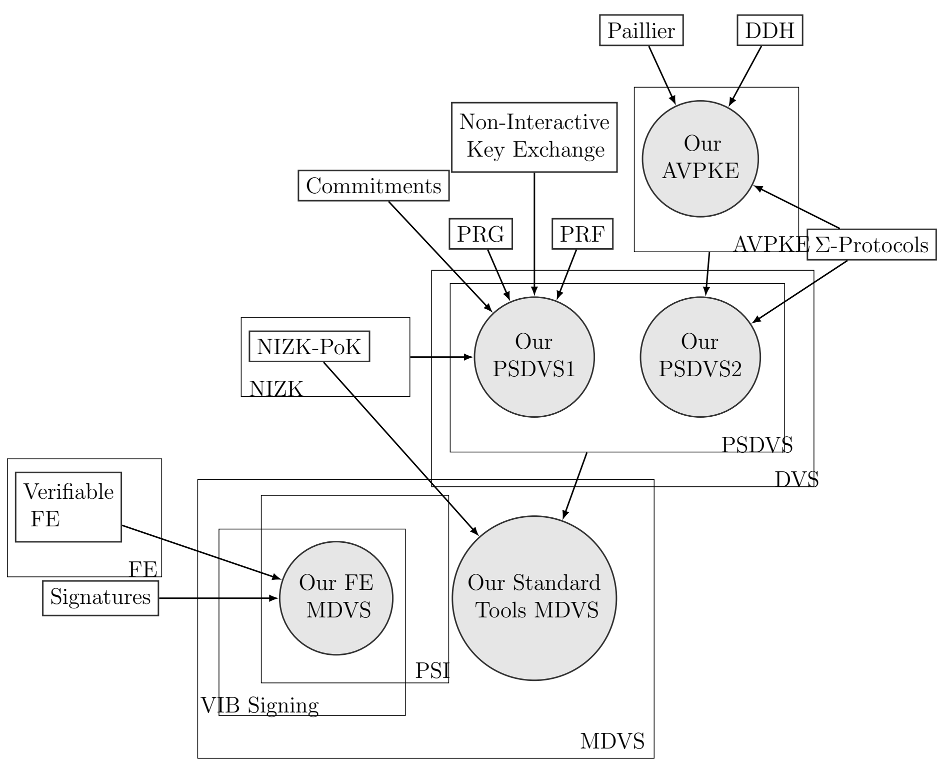


Figure 5·2: Our MDVS Constructions and Building Blocks

contradicts the theorem, but this is not the case. It can be instantiated such that signatures can be simulated by collusions up to a certain maximal size t, and then signatures will be of size $\Omega(|\mathcal{C}|)$. However, if we want any *subset to be able to simulate, the signature size is $\Omega(|\mathcal{D}|)$, in accordance with the theorem.*

Proof. Imagine that we give all the verifiers' keys to a sender and a receiver; the sender can now encode an arbitrary subset $\mathcal{C} \subseteq \mathcal{D}$ by letting $\mathcal{C}$ construct a simulated signature σ on some default message, and sending it to the receiver. The receiver can infer $\mathcal{C}$ from σ: by strong unforgeability, all verifiers' keys outside $\mathcal{C}$ will reject σ, whereas keys in $\mathcal{C}$ will accept, since we require the simulation to look convincing even given the secret keys in $\mathcal{C}$. It follows that σ must consist of enough bits to determine $\mathcal{C}$, which is $\log_2(2^{|\mathcal{D}|}) = |\mathcal{D}|$. □ □

Why First Ideas Fail

Using MACs Black-box usage of a standard MAC scheme cannot help us combine unforgeability with consistency.[6] There are two straightforward ways to use a standard MAC scheme in this context: sharing a MAC key among the entire group, and sharing MAC keys pairwise. Sharing a single key does not provide the desired notion of unforgeability, since any member of the group can forge messages from any other member. Sharing keys pairwise does not provide the desired notion of consistency. If recipients R_1 and R_2 are the chosen recipients of a message, and R_1 receives a message he accepts as coming from S, he cannot be sure that R_2 would also accept that message: If S is corrupt, he could include a valid MAC for R_1 and an invalid MAC for R_2.

Using Proofs of Knowledge A standard technique for making designated verifier signatures for a single verifier is to start from an interactive protocol that proves knowledge of either the signer's or the verifier's secret key, and turn this into a signature scheme using the Fiat-Shamir paradigm. It may seem natural to try to

[6]Note that our construction from standard primitives does make use of MAC schemes; however, it does so in a complex, non-black-box way.

build an MDVS from this. However, it turns out to be challenging to achieve strong unforgeability using this technique; a signature cannot consist of a proof of knowledge of the signer's or one of the verifiers' secret keys, since any verifier will be able to convince other verifiers to accept a signature that did not come from the signer. For the same reason, a signature cannot consist of a proof of knowledge of the signer's secret key or some subset of the verifiers' secret keys.

MDVS from Standard Primitives

Our first class of MDVS constructions is based only on standard primitives. With one exception specified below, all of these constructions can be instantiated in the random oracle model with no trusted setup. (Without random oracles, we would need to set up a common reference string.)

The idea is that the signer creates a DVS signature for each verifier individually, and then proves the consistency of those signatures.[7]. To support such proofs, we define a new primitive called Publicly Simulatable Designated Verifier Signatures (PSDVS) in Section 5.3.1, which is a single-verifier DVS equipped with extra properties. We then show, in Section 5.3.2, that a PSDVS together with a non-interactive zero knowledge proof of knowledge (NIZK-PoK) imply an MDVS for any number of signers and verifiers. Finally, we give some constructions of PSDVS. Our first PS-DVS construction (in Section 5.3.3) uses only generic tools, namely psedudorandom functions, non-interactive key exchange (such as Diffie-Hellman), and non-interactive zero-knowledge proofs of knowledge. Our second PSDVS construction (in Section 5.3.4) aims at better concrete efficiency. It is based on DDH, strong RSA and Paillier encryption, is secure in the random oracle model, and requires a constant number of exponentiations for all operations. This scheme requires the trusted generation of an

[7]Simply proving that all of the signatures verify would violate the off-the-record property; instead, the signer proves that either all of the signatures are real, or they are all simulated, as described in Section 5.3

RSA modulus so that the factorization remains unknown. We also sketch a variant that requires no trusted setup, is secure in the random oracle model, and only requires (a variant of) the DDH assumption. However, this version requires double discrete log proofs, and therefore requires a non-constant number of exponentiations.

In order to support one of our constructions in which the signer sends an encrypted MAC key, we introduce a new tool we call Authenticated and Verifiable Encryption (AVPKE), which may be of independent interest. This is a variant of Paillier encryption with built-in authentication, and as such it is related to the known primitive "signcryption" (Zheng, 1997). However, our AVPKE scheme has the additional property that we can give efficient zero-knowledge proofs involving the encrypted message, using the algebraic properties of Paillier encryption.

To sign in our PSDVS schemes, the signer and verifiers first must establish a shared symmetric key k. In some cases they can do this non-interactively, using their secret and public keys, while in other cases the signer must send an encrypted key alongside the signature. After this, the signer sends a MAC on the message under key k; this MAC is based on a pseudorandom function.

MDVS from Functional Encryption.

Our last construction is based on Verifiable Functional Encryption (VFE). It has the advantages of additionally meeting the privacy of identities and verifier-identity-based signing properties. Additionally, it can be set up to have smaller signatures if we are willing to make a stronger assumption on the number of colluding verifiers. Namely, the signature size is $O(t)$, where t is the size of the largest number of colluding verifiers we want to tolerate. The downsides are that, with current state of the art, VFE requires non-standard computational assumptions. We also need a trusted setup for generating keys; however, this is unavoidable if we wish to achieve verifier-identity-based signing.

Remark 7. *If we are going to put a bound on the size of a collusion, it may seem we can use bounded collusion FE, which can be realized from standard assumptions (Gorbunov et al., 2012; Ananth and Vaikuntanathan, 2019), and then there is no need for our other constructions from standard primitives. However, this is not true. Bounded collusion FE requires us to fix the bound on collusion size at key generation time; a bound that may later turn out to be too small. Additionally, ciphertext sizes in bounded collusion FE depend on the bound; thus, choosing a large bound to make sure we can handle the application implies a cost in efficiency. The MDVS signature sizes would depend on some upper bound on number of corrupt parties in the* system, *as opposed to on the number of recipients for the signature in question, which may be orders of magnitude smaller.*

In a nutshell, the idea behind the functional encryption based construction is to do the proof of knowledge of one of the relevant secret keys "inside the ciphertext". In a little more detail, the idea is to encrypt a list of t *standard* signatures, where t is the maximal size of collusion we want to protect against (that is, $t \geq |\mathcal{C}|$), and the MDVS signature will simply be this ciphertext. To sign, the signer will generate their own standard signature σ_{S} on the message, and then encrypt a list a signatures consisting of σ_{S} followed by $t-1$ dummy values. To verify a signature, a verifier V gets a functional decryption key that will look at the list of signatures inside the ciphertext and output accept or reject. It will accept if the list contains a valid signature from S or a valid signature from V. Now, if a corrupt set of verifiers $\mathcal{C}$ wants to simulate a signature, they will all sign the message and encrypt the list of these signatures. By security of the encryption scheme, this looks like a real signature, and will indeed verify under all verification keys belonging to verifiers in $\mathcal{C}$. However, no honest verifier will accept it as a signature from S, so we have strong unforgeability.

5.2 Multi-Designated Verifier Signatures

MDVS Algorithms

A *multi-designated verifier signature* (MDVS) scheme is defined by the following probabilistic polynomial-time algorithms:

$\mathsf{Setup}(1^\lambda) \to (pp, msk)$**:** On input the security parameter $\lambda \in \mathbb{N}$, outputs public parameters pp and the master secret key msk.

$\mathsf{SignKeyGen}(pp, msk) \to (spk, ssk)$**:** On input the public parameter pp and the master secret key msk, outputs the public key spk and secret key ssk for a signer.

$\mathsf{VerKeyGen}(pp, msk) \to (vpk, vsk)$**:** On input the public parameter pp and the master secret key msk, outputs the public key vpk and secret key vsk for a verifier.

$\mathsf{Sign}(pp, ssk_i, \{vpk_j\}_{j \in \mathcal{D}}, m) \to \sigma$**:** On input the public parameters pp, a secret signing key ssk_i, the public keys of the designated verifiers $\{vpk_j\}_{j \in \mathcal{D}}$, and a message m, outputs a signature σ.

$\mathsf{Verify}(pp, spk_i, vsk_j, \{vpk_{j'}\}_{j' \in \mathcal{D}}, m, \sigma) \to d$**:** On input the public parameters pp, a public verification key spk_i, a secret key vsk_j of a verifier such that $j \in \mathcal{D}$, the public keys of the designated verifiers $\{vpk_j\}_{j \in \mathcal{D}}$, a message m, and a signature σ, outputs a boolean decision d: $d = 1$ (accept) or $d = 0$ (reject).

$\mathsf{Sim}(pp, spk_i, \{vpk_j\}_{j \in \mathcal{D}}, \{vsk_j\}_{j \in \mathcal{C}}, m) \to \sigma'$**:** On input public parameters pp, a public verification key spk_i, the public keys of the designated verifiers $\{vpk_j\}_{j \in \mathcal{D}}$, the secret keys of the corrupt designated verifiers $\{vsk_j\}_{j \in \mathcal{C}}$, and a message m, outputs a simulated signature σ'.

The different algorithms take many different inputs, which are not all needed for all of our constructions. For instance, the constructions based on standard primitives (Section 5.3) do not need a master secret key; they allow key pairs for signers and verifiers to be generated locally. Additionally, some of our constructions do not use the signers' and verifiers' public keys in all of the algorithms in which they appear as

inputs above. Thus, to simplify the notation we exclude these inputs in later sections whenever they are not needed.

MDVS Properties

Let σ be a signature from signer i on message m and designated for verifiers $\mathcal{D}$. We ask for the following (informal) properties:

Correctness: All verifiers $j \in \mathcal{D}$ are able to verify an honestly generated signature σ.

Consistency: If there exists one verifier $j \in \mathcal{D}$ that accepts the signature σ, then all other designated verifiers (i.e. all $j' \in \mathcal{D} \setminus \{j\}$) also accept the signature.

Unforgeability: An adversary without knowledge of the secret key ssk_i for signer i cannot create a signature σ' that is accepted by any designated verifier as a signature from signer i.

Off-The-Record: Given a signature σ, any malicious subset of the designated verifiers $\mathcal{C} \subseteq \mathcal{D}$ cannot convince any outsider that σ is a signature from signer i (i.e. the malicious set could have simulated the signature themselves).

(Optionally) Privacy of Identities: Any outsider (without colluding with any designated verifiers) cannot determine the identity of the signer and/or the identities of the designated verifiers.

(Optionally) Verifier-Identity-Based Signing: The signer should be able to produce a signature for a set of designated verifiers without requiring any information about them apart from their identities. In other words, we should have $vpk_j = j$ for a verifier with identity j.

Throughout our formal definitions we use the following six oracles:

Signer Key Generation Oracle: $\mathcal{O}_{SKG}(i)$

1. If a signer key generation query has previously been performed for i, look up and return the previously generated key.

2. Otherwise, output and store $(spk_i, ssk_i) \leftarrow \mathsf{SignKeyGen}(pp, msk)$.

Verifier Key Generation Oracle: $\mathcal{O}_{VKG}(j)$

1. If a verifier key generation query has previously been performed for j, look up and return the previously generated key.
2. Otherwise, output and store $(vpk_j, vsk_j) \leftarrow \mathsf{VerKeyGen}(pp, msk)$.

Public Signer Key Generation Oracle: $\mathcal{O}_{SPK}(i)$

1. $(spk_i, ssk_i) \leftarrow \mathcal{O}_{SKG}(i)$.
2. Output spk_i.

Public Verifier Key Generation Oracle: $\mathcal{O}_{VPK}(j)$

1. $(vpk_j, vsk_j) \leftarrow \mathcal{O}_{VKG}(j)$.
2. Output vpk_j.

Signing Oracle: $\mathcal{O}_S(i, \mathcal{D}, m)$

1. $(spk_i, ssk_i) \leftarrow \mathcal{O}_{SKG}(i)$.
2. For all $j \in \mathcal{D}$: $vpk_j \leftarrow \mathcal{O}_{VPK}(j)$.
3. Output $\sigma \leftarrow \mathsf{Sign}(pp, ssk_i, \{vpk_j\}_{j \in \mathcal{D}}, m)$.

Verification Oracle: $\mathcal{O}_V(i, j, \mathcal{D}, m, \sigma)$

1. $spk_i \leftarrow \mathcal{O}_{SPK}(i)$.
2. $(vpk_j, vsk_j) \leftarrow \mathcal{O}_{VKG}(j)$.
3. Output $d \leftarrow \mathsf{Verify}(pp, spk_i, vsk_j, \{vpk_{j'}\}_{j' \in \mathcal{D}}, m, \sigma)$.

Definition 7 (Correctness)**.** *Let $\lambda \in \mathbb{N}$ be the security parameter, and let* $\mathsf{MDVS} = (\mathsf{Setup}, \mathsf{SignKeyGen}, \mathsf{VerKeyGen}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{Sim})$ *be an MDVS scheme.* MDVS *is* correct *if for all signer identities i, messages m, verifier identity sets $\mathcal{D}$ and $j \in \mathcal{D}$, it holds that*

$$\mathsf{Pr}\left[\mathsf{Verify}(pp, spk_i, vsk_j, \{vpk_{j'}\}_{j' \in \mathcal{D}}, m, \sigma) \neq 1\right] = 0,$$

where the inputs to Verify *are generated as follows:*

- $(pp, msk) \leftarrow \mathsf{Setup}(1^\lambda)$*;*
- $(spk_i, ssk_i) \leftarrow \mathsf{SignKeyGen}(pp, msk, i)$*;*

- $(vpk_j, vsk_j) \leftarrow \mathsf{VerKeyGen}(pp, msk, j)$ *for* $j \in \mathcal{D}$*;*
- $\sigma \leftarrow \mathsf{Sign}(pp, ssk_i, \{vpk_j\}_{j\in\mathcal{D}}, m)$*.*

In Def 7, we require that all the designated verifiers can verify the signature, without considering what happens for parties that are not designated verifiers (i.e. parties who should not be able to verify the signature). Parties that are not designated verifiers are accounted for by the off-the-record property.

Definition 8 (Consistency)**.** *Let* $\lambda \in \mathbb{N}$ *be the security parameter, and let* $\mathsf{MDVS} = (\mathsf{Setup}, \mathsf{SignKeyGen}, \mathsf{VerKeyGen}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{Sim})$ *be an MDVS scheme. Consider the following game between a challenger and an adversary* $\mathcal{A}$*:*

$$\mathsf{Game}^{con}_{\mathsf{MDVS},\mathcal{A}}(\lambda)$$

1. $(pp, msk) \leftarrow \mathsf{Setup}(1^\lambda)$
2. $(m^*, i^*, \mathcal{D}^*, \sigma^*) \leftarrow \mathcal{A}^{\mathcal{O}_{SKG}, \mathcal{O}_{VKG}, \mathcal{O}_{SPK}, \mathcal{O}_{VPK}, \mathcal{O}_V}(pp)$

We say that $\mathcal{A}$ *wins the game if there exist verifiers* $j_0, j_1 \in \mathcal{D}^*$ *such that:*

$$\mathsf{Verify}(pp, spk_{i^*}, vsk_{j_0}, \{vpk_{j'}\}_{j'\in\mathcal{D}^*}, m^*, \sigma^*) = 0,$$
$$\mathsf{Verify}(pp, spk_{i^*}, vsk_{j_1}, \{vpk_{j'}\}_{j'\in\mathcal{D}^*}, m^*, \sigma^*) = 1,$$

where all keys are the honestly generated outputs of the key generation oracles, and $\mathcal{O}_{VKG}$ *is never queried on* j_0 *or* j_1*.*
MDVS *is* consistent *if, for all PPT adversaries* $\mathcal{A}$*,*

$$\mathsf{Adv}^{con}_{\mathsf{MDVS},\mathcal{A}}(\lambda) = \Pr\left[\mathcal{A} \textit{ wins } \mathsf{Game}^{con}_{\mathsf{MDVS},\mathcal{A}}(\lambda)\right] \leq \nu(\lambda).$$

Def 8 states that even a valid signer (i.e. someone who knows a secret signing key) cannot create an inconsistent signature that will be accepted by some designated verifiers and rejected by others. By the correctness property, an honestly generated signature is accepted by all designated verifiers. By design, corrupt designated verifiers can construct an inconsistent signature, since some verifiers will accept it (i.e. those verifiers that created it), while the remaining honest designated verifiers will reject the simulated signature. Thus, we need to ask for $j \neq j_0, j_1$ for all queries j to the oracle $\mathcal{O}_{VKG}$.

Definition 9 (Existential Unforgeability)**.** *Let* $\lambda \in \mathbb{N}$ *be the security parameter, and let* $\mathsf{MDVS} = (\mathsf{Setup}, \mathsf{SignKeyGen}, \mathsf{VerKeyGen}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{Sim})$ *be an MDVS scheme. Consider the following game between a challenger and an adversary* $\mathcal{A}$*:*

$$\mathsf{Game}^{euf}_{\mathsf{MDVS},\mathcal{A}}(\lambda)$$

1. $(pp, msk) \leftarrow \mathsf{Setup}(1^\lambda)$
2. $(m^*, i^*, \mathcal{D}^*, \sigma^*) \leftarrow \mathcal{A}^{\mathcal{O}_{SKG}, \mathcal{O}_{VKG}, \mathcal{O}_{SPK}, \mathcal{O}_{VPK}, \mathcal{O}_S}(pp)$

We say that $\mathcal{A}$ *wins the game if we have all of the following:*

- *for all queries* i *to oracle* $\mathcal{O}_{SKG}$*, it holds that* $i^* \neq i$*;*
- *for all queries* $(i, \mathcal{D}, m)$ *to oracle* $\mathcal{O}_S$ *that result in signature* σ*, it holds that* $(i^*, \mathcal{D}^*, m^*) \neq (i, \mathcal{D}, m)$*;*
- *there exists a verifier* $j' \in \mathcal{D}^*$ *such that for all queries* j *to oracle* $\mathcal{O}_{VKG}$*, it holds that* $j' \neq j$ *and*

$$\mathsf{Verify}(pp, spk_{i^*}, vsk_{j'}, \{vpk_{j''}\}_{j'' \in \mathcal{D}^*}, m^*, \sigma^*) = 1,$$

where all keys are honestly generated outputs of the key generation oracles.

MDVS *is* existentially unforgeable *if, for all PPT adversaries* $\mathcal{A}$*,*

$$\mathsf{Adv}^{euf}_{\mathsf{MDVS},\mathcal{A}}(\lambda) = \Pr\left[\mathcal{A} \text{ wins } \mathsf{Game}^{euf}_{\mathsf{MDVS},\mathcal{A}}(\lambda)\right] \leq \nu(\lambda).$$

Def 9 states that an adversary cannot create a signature that any honest verifier will accept as coming from a signer whose secret signing key the adversary does not know. The adversary will always get the public keys of the involved parties, i.e. signer with identity i^* and the designated verifiers $\mathcal{D}$, through the key generation oracles. He is also allowed to obtain the secret keys of every party except the signer i^* and at least one designated verifier. The reason why we need at least one honest verifier is that corrupt verifiers can create a simulated signature that will look like a real signature with respect to their own verifier secret keys. However, this simulation will be rejected by any honest designated verifier, i.e. the simulation will be a valid forgery for the corrupt verifiers, but not for the honest verifiers.

Definition 10 (Off-The-Record)**.** *Let* $\lambda \in \mathbb{N}$ *be the security parameter, let* $\mathsf{MDVS} = (\mathsf{Setup}, \mathsf{SignKeyGen}, \mathsf{VerKeyGen}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{Sim})$ *be an MDVS scheme, and let* t *be an*

upper bound on the number of verifiers an adversary $\mathcal{A}$ can corrupt. Consider the following game between a challenger and an adversary $\mathcal{A}$, where all keys are honestly generated outputs of the key generation oracles:

$$\mathsf{Game}^{otr}_{\mathsf{MDVS},\mathcal{SIM},\mathcal{A}}(\lambda)$$

1. $(pp, msk) \leftarrow \mathsf{Setup}(1^\lambda)$
2. $(i^*, \mathcal{D}^*, m^*, \mathcal{C}^*) \leftarrow \mathcal{A}^{\mathcal{O}_{SKG},\mathcal{O}_{VKG},\mathcal{O}_{SPK},\mathcal{O}_{VPK},\mathcal{O}_S,\mathcal{O}_V}(pp)$
3. $b \leftarrow \{0,1\}$
4. $\sigma_0 \leftarrow \mathsf{Sign}(pp, ssk_{i^*}, \{vpk_j\}_{j\in\mathcal{D}^*}, m^*)$
5. $\sigma_1 \leftarrow \mathsf{Sim}(pp, spk_{i^*}, \{vpk_j\}_{j\in\mathcal{D}^*}, \{vsk_j\}_{j\in\mathcal{C}^*}, m^*)$
6. $b' \leftarrow \mathcal{A}^{\mathcal{O}_{SKG},\mathcal{O}_{VKG},\mathcal{O}_{SPK},\mathcal{O}_{VPK},\mathcal{O}_S,\mathcal{O}_V}(\sigma_b)$

We say that $\mathcal{A}$ wins the game if $b' = b$, and all of the following hold:

- $|\mathcal{C}^*| \leq t$ *and* $\mathcal{C}^* \subseteq \mathcal{D}^*$*;*
- *for all queries i to oracle $\mathcal{O}_{SKG}$ it holds that $i^* \neq i$;*
- *for all queries j to oracle $\mathcal{O}_{VKG}$ it holds that $j \notin \mathcal{D}^*\backslash\mathcal{C}^*$;*
- *for all queries $(i, j, \mathcal{D}, m, \sigma)$ to $\mathcal{O}_V$ it holds that $\sigma_b \neq \sigma$.*

We say that an MDVS scheme is t-off-the-record *if, for all PPT adversaries $\mathcal{A}$,*

$$\mathsf{Adv}^{otr}_{\mathsf{MDVS},\mathcal{SIM},\mathcal{A}}(\lambda) = \Pr\left[\mathcal{A} \text{ wins } \mathsf{Game}^{otr}_{\mathsf{MDVS},\mathcal{SIM},\mathcal{A}}(\lambda)\right] - \frac{1}{2} \leq \nu(\lambda).$$

If a scheme supports $t = |\mathcal{D}|$, we say that it is off-the-record*.*

Def 10 states that any adversary that corrupts a subset (of size t) of the designated verifiers $\mathcal{C}^*$ cannot determine whether the received signature was created by real signer i^* or simulated by the corrupt verifiers $\mathcal{C}^*$. The adversary is not allowed to see the secret keys for the designated verifiers that are in $\mathcal{D}^*\backslash\mathcal{C}^*$. If the adversary was allowed to get secret keys of additional parties in $\mathcal{D}^*$ (which are not in $\mathcal{C}^*$), then he would be able to distinguish trivially, since any honest designated verifiers (i.e. any $j \in \mathcal{D}^*\backslash\mathcal{C}^*$) can distinguish simulated signatures from real signatures (from the unforgeability property).

Definition 11 (Privacy of Identities)**.** *Let $\lambda \in \mathbb{N}$ be the security parameter, and let* $\mathsf{MDVS} = (\mathsf{Setup}, \mathsf{SignKeyGen}, \mathsf{VerKeyGen}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{Sim})$ *be an MDVS scheme. Consider the following game between a challenger and an adversary $\mathcal{A}$, where all keys are the honestly generated outputs of the key generation oracles:*

$$\mathsf{Game}^{pri}_{\mathsf{MDVS},\mathcal{A}}(\lambda)$$

1. $(pp, msk) \leftarrow \mathsf{Setup}(1^\lambda)$
2. $(m^*, i_0, i_1, \mathcal{D}_0, \mathcal{D}_1) \leftarrow \mathcal{A}^{\mathcal{O}_{SKG}, \mathcal{O}_{VKG}, \mathcal{O}_{SPK}, \mathcal{O}_{VPK}, \mathcal{O}_S, \mathcal{O}_V}(pp)$
3. $b \leftarrow \{0,1\}$
4. $\sigma^* \leftarrow \mathsf{Sign}(pp, ssk_{i_b}, \{vpk_j\}_{j \in \mathcal{D}_b}, m^*)$
5. $b' \leftarrow \mathcal{A}^{\mathcal{O}_{SKG}, \mathcal{O}_{VKG}, \mathcal{O}_{SPK}, \mathcal{O}_{VPK}, \mathcal{O}_S, \mathcal{O}_V}(\sigma^*)$

We say that $\mathcal{A}$ wins the game if $b = b'$, and all of the following hold:

- $|\mathcal{D}_0| = |\mathcal{D}_1|$;
- *for all queries i to $\mathcal{O}_{SKG}$, it holds that $i \notin \{i_0, i_1\}$;*
- *for all gueries j to $\mathcal{O}_{VKG}$, it holds that $j \notin \mathcal{D}_0 \cup \mathcal{D}_1$;*
- *for all queries $(i, j, \mathcal{D}, m, \sigma)$ to $\mathcal{O}_V$, it holds that $\sigma^* \neq \sigma$.*

MDVS *has* privacy of identities *if, for all PPT adversaries $\mathcal{A}$,*

$$\mathsf{Adv}^{pri}_{\mathsf{MDVS},\mathcal{A}}(\lambda) = \Pr\left[\mathcal{A} \textit{ wins } \mathsf{Game}^{pri}_{\mathsf{MDVS},\mathcal{A}}(\lambda)\right] - \frac{1}{2} \leq \nu(\lambda).$$

We say that MDVS *has additional properties as follows:*

- *privacy of the signer's identity (PSI) if we make the restriction that $\mathcal{D}_0 = \mathcal{D}_1$;*
- *privacy of the designated verifiers' identities (PVI) if we make the restriction that $i_0 = i_1$.*

Def 11 states that an adversary cannot distinguish between signatures from two different signers (PSI) if he does not know the secret key of any of the signers or designated verifiers (as designated verifiers *are* allowed to identify the signer). Furthermore, it should not help him to see other signatures that he knows are from the signers in question.

In addition, if we vary the verifier sets ($\mathcal{D}_0 \neq \mathcal{D}_1$), then the MDVS scheme has privacy of designated verifier's identities (PVI), which means that any outsider without knowledge of any secret keys cannot distinguish between signatures meant for different verifiers.

Definition 12 (Verifier-Identity-Based Signing)**.** *We say that an MDVS scheme has* verifier-identity-based signing *if for honestly generated verifier keys (vsk_j, vpk_j) for verifier with identity j, we have $vpk_j = j$.*

Note that, in order to achieve verifier-identity-based signing, verifier key generation must require a master secret key msk. Otherwise, any outsider would be able to generate a verification key for verifier j, and use it to verify signatures meant only for that verifier.

Relation to Previous Definitions Our definition of MDVS is consistent with previous work in this area, but with some differences. Our MDVS syntax closely follows the one introduced by (Laguillaumie and Vergnaud, 2004), but we allow for a master secret key in the case where the keys are generated by a trusted party (like in our construction based on functional encryption). Our security definitions are adapted from those in (Laguillaumie and Vergnaud, 2004; Zhang et al., 2012) to capture the flexibility introduced by allowing any subset of designated verifiers to simulate a signature, thus providing better deniability properties. Finally, we formalize consistency as an additional and desirable requirement.

5.3 Standard Primitive-Based MDVS Constructions

In this section we show how to create an MDVS scheme that uses only standard primitives, such as key exchange, commitments, pseudorandom functions and generators, and non-interactive zero knowledge proofs.

On a high level, one way to build an MDVS is for the signer to use a separate DVS with each verifier; the MDVS signature would then consist of a vector of individual DVS signatures. This gives us almost everything we need — the remaining issue is consistency. Each verifier can verify one of the DVS signatures, but is not convinced that all of the other verifiers will come to the same conclusion.

A solution to this consistency issue is to include as part of the MDVS signature a zero knowledge proof that all of the DVS signatures verify. However, this introduces a new issue with off-the-record. Now, a colluding set of verifiers will not be able to

simulate a signature unless *all* of the verifiers collude. In order to produce such a convincing zero knowledge proof as part of the signature, they would need to forge signatures for the other verifiers in the underlying DVS scheme, which they should not be able to do.

So, instead of using a zero knowledge proof of knowledge that all of the DVS signatures verify, we use a proof that *either* all of the DVS signatures verify, *or they are all simulated*. Then, a corrupt set of verifiers can simulate all of the underlying DVS signatures — with the caveat that the signatures they simulate for themselves should be convincing simulations even in the presence of their secret keys — and, instead of proving that all of the signatures verify, they prove that all of the signatures are simulations.

In order to support such proofs, in Section 5.3.1 we introduce a new primitive called a Provably Simulatable DVS (PSDVS). Then, in Section 5.3.2 we show how to compose PSDVS instances into an MDVS. In Section 5.3.3 we build a PSDVS out of generic standard primitives. In Section 5.3.4 we build a more efficient PSDVS out of concrete instantiations of those primitives.

5.3.1 New Primitive: Provably Simulatable Designated-Verifier Signatures (PSDVS)

Designated Verifier Signatures (DVS) have a simulation algorithm Sim which is used to satisfy the off-the-record property (the single-verifier equivalent of Def 10). Given the signer's public key, the verifier's secret key and a message m, $\mathcal{SIM}$ should return a signature which is indistinguishable from a real signature. A *Provably Simulatable DVS (PSDVS)*, in addition to correctness and existential unforgeability, must have two notions of transcript simulation: *public* simulation and *verifier* simulation. Furthermore, for each of these, it should be possible to produce a zero knowledge proof that the signature produced is a simulation. Additionally, it should be possible

to similarly prove that a real signature is, in fact, real. This makes a PSDVS well suited for use in an MDVS which uses a zero knowledge proof of knowledge to enforce consistency.[8]

More formally, a PSDVS consists of the standard DVS algorithms Setup, $\mathsf{SignKeyGen}, \mathsf{VerKeyGen}, \mathsf{Sign}, \mathsf{Verify}$, as well as five additional algorithms: $\mathsf{RealSigVal}$ to validate real signatures, and $\mathsf{PubSigSim}$, $\mathsf{PubSigVal}$, $\mathsf{VerSigSim}$ and $\mathsf{VerSigVal}$ to simulate signatures and to validate such simulations.

Definition 13. *A PSDVS must satisfy the standard notions of correctness and existential unforgeability. Additionally, it should satisfy* PubSigSim indstinguishability *(Def 14),* PubSigSim correctness *(Def 15),* PubSigSim soundness *(Def 16),* VerSigSim indstinguishability *(Def 17),* VerSigSim correctness *(Def 18),* VerSigSim soundness *(Def 19),* provable signing correctness *(Def 20), and* provable signing soundness *(Def 21).*

Provable Public Simulation

As in *PSI* (Def 11), anyone should be able to produce a signature that is indistinguishable from a real signature. Additionally, the party simulating the signature should be able to produce a proof that this is *not* a real signature. This proof will be incorporated into the MDVS proof of consistency; the colluding verifiers, when producing a simulation, need to prove that all underlying PSDVS signatures are real, or that they are all fake.

In other words, we require two additional algorithms, as follows:

1. $\mathsf{PubSigSim}(pp, spk, vpk, m) \rightarrow (\sigma, \pi)$
2. $\mathsf{PubSigVal}(pp, spk, vpk, m, \sigma, \pi) \rightarrow d \in \{0, 1\}$

The colluding verifiers will produce a public simulation in the underlying PSDVS for verifiers outside their coalition, and use $\mathsf{PubSigSim}$ to prove that this simulation is

[8]While these additional properties allow the composition of PSDVS into an MDVS, they are not useful when PSDVS is used on its own.

not a real signature. π will not be explicitly included in the proof of "the underlying PSDVS signatures are all real or all fake," of course, as it would give away the fact that all underlying signatures are fake, as opposed to all being real; rather, it will be wrapped in a larger zero knowledge proof.

Definition 14 (PubSigSim Indistinguishability)**.** *We say that the PSDVS has* PubSigSim Indistinguishability *if* PubSigSim *produces a signature σ that is indistinguishable from real. More formally, an adversary should not be able to win the following game with probability non-negligibly more than half:*

$$\mathsf{Game}^{PubSigSim\text{-}Ind}_{\mathsf{PVDVS},\mathcal{A}}(\lambda)$$

1. $pp \leftarrow \mathsf{Setup}(1^\lambda)$
2. $(spk, ssk) \leftarrow \mathsf{SignKeyGen}(pp)$
3. $(vpk, vsk) \leftarrow \mathsf{VerKeyGen}(pp)$
4. $m^* \leftarrow \mathcal{A}^{\mathcal{O}_S,\mathcal{O}_V}(spk, vpk)$
5. $b \leftarrow \{0,1\}$
6. $\sigma_0 \leftarrow \mathsf{Sign}(pp, ssk, vpk, m^*)$
7. $(\sigma_1, \pi) \leftarrow \mathsf{PubSigSim}(pp, spk, vpk, m^*)$
8. $b' \leftarrow \mathcal{A}^{\mathcal{O}_S,\mathcal{O}_V}(pp, spk, vpk, m^*, \sigma_b)$

We say that $\mathcal{A}$ wins the PubSigSim-Ind *game if $b = b'$ and for all queries (m, σ) to $\mathcal{O}_V$, it holds that $(m, \sigma) \neq (m^*, \sigma_b)$.*

Definition 15 (PubSigSim Correctness)**.** *We say that the PSDVS has* PubSigSim Correctness *if for all $pp \leftarrow \mathsf{Setup}(1^\lambda)$; $(spk, ssk) \leftarrow \mathsf{SignKeyGen}(pp)$; $(vpk, vsk) \leftarrow \mathsf{VerKeyGen}(pp)$; $m \in \{0,1\}^*$; $(\sigma, \pi) \leftarrow \mathsf{PubSigSim}(pp, spk, vpk, m)$;*

$$\Pr[\mathsf{PubSigVal}(pp, spk, vpk, m, \sigma, \pi) = 1] = 1.$$

Definition 16 (PubSigSim Soundness)**.** *We say that the PSDVS has* PubSigSim Soundness *if it is hard to construct a signature σ which is accepted by the verifier algorithm and at the same time can be proven to be a simulated signature. More formally, an adversary should not be able to win the following game with non-negligible probability:*

$$\mathsf{Game}^{PubSigSim\text{-}Sound}_{\mathsf{PVDVS},\mathcal{A}}(\lambda)$$

1. $pp \leftarrow \mathsf{Setup}(1^\lambda)$
2. $(spk, ssk) \leftarrow \mathsf{SignKeyGen}(pp)$
3. $(vpk, vsk) \leftarrow \mathsf{VerKeyGen}(pp)$
4. $(m^*, \sigma^*, \pi^*) \leftarrow \mathcal{A}(pp, ssk, spk, vpk)$

We say that $\mathcal{A}$ wins the PubSigSim-Sound *game if* $\mathsf{Verify}(pp, vsk, m^*, \sigma^*) = 1$ *and* $\mathsf{PubSigVal}(pp, spk, vpk, m^*, \sigma^*, \pi^*) = 1$.

Provable Verifier Simulation

As in *off-the-record* (Def 10), a verifier should be able to produce a signature that is indistinguishable from a real signature, *even given its secret key.* Additionally, the verifier should be able to produce a proof that the signature is *not* a real signature (that is, that the verifier, and not the signer, produced it). This proof will be incorporated into the MDVS proof of consistency.

In other words, we require two additional algorithms, as follows:

1. $\mathsf{VerSigSim}(pp, spk, vpk, vsk, m) \to (\sigma, \pi)$
2. $\mathsf{VerSigVal}(pp, spk, vpk, m, \sigma, \pi) \to d \in \{0, 1\}$

The colluding verifiers will produce a verifier simulation in the underlying PSDVS for verifiers *inside* their coalition, and use VerSigSim to prove that this simulation is not a real signature.

Definition 17 (VerSigSim Indistinguishability)**.** *We say that the PSDVS has* VerSigSim Indistinguishability *if* VerSigSim *produces a signature* σ *that is indistinguishable from real. More formally, an adversary should not be able to win the following game with probability non-negligibly more than half:*

$\mathsf{Game}^{VerSigSim\text{-}Ind}_{\mathsf{PVDVS},\mathcal{A}}(\lambda)$

1. $pp \leftarrow \mathsf{Setup}(1^\lambda)$
2. $(spk, ssk) \leftarrow \mathsf{SignKeyGen}(pp)$
3. $(vpk, vsk) \leftarrow \mathsf{VerKeyGen}(pp)$
4. $m^* \leftarrow \mathcal{A}^{\mathcal{O}_S}(pp, spk, vpk, vsk)$
5. $b \leftarrow_\$ \{0, 1\}$
6. $\sigma_0 \leftarrow \mathsf{Sign}(pp, ssk, vpk, m^*)$
7. $(\sigma_1, \pi) \leftarrow \mathsf{VerSigSim}(pp, spk, vsk, m^*)$
8. $b' \leftarrow \mathcal{A}^{\mathcal{O}_S}(pp, spk, vpk, vsk, m^*, \sigma_b)$

We say that $\mathcal{A}$ *wins the* VerSigSim-Ind *game if* $b = b'$.

Definition 18 (VerSigSim Correctness)**.** *We say that the PSDVS has* VerSigSim Correctness *if for all* $pp \leftarrow \mathsf{Setup}(1^\lambda)$, $(spk, ssk) \leftarrow \mathsf{SignKeyGen}(pp)$, $(vpk, vsk) \leftarrow \mathsf{VerKeyGen}(pp)$, $m \in \{0, 1\}^*$, $(\sigma, \pi) \leftarrow \mathsf{VerSigSim}(pp, spk, vpk, vsk, m)$,

$$\Pr[\mathsf{VerSigVal}(pp, spk, vpk, m, \sigma, \pi) = 1] = 1.$$

Definition 19 (VerSigSim Soundness)**.** *We say that the PSDVS has* VerSigSim Soundness *if the signer is not able to produce σ and π that pass the validation check* $\mathsf{VerSigVal}$*, i.e. π is a proof that σ was not produced by the signer. More formally, an adversary should not be able to win the following game with non-negligible probability:*

$$\mathsf{Game}^{VerSigSim\text{-}Sound}_{\mathsf{PVDVS},\mathcal{A}}(\lambda)$$

1. $pp \leftarrow \mathsf{Setup}(1^\lambda)$
2. $(spk, ssk) \leftarrow \mathsf{SignKeyGen}(pp)$
3. $(vpk, vsk) \leftarrow \mathsf{VerKeyGen}(pp)$
4. $(m^*, \sigma^*, \pi^*) \leftarrow \mathcal{A}(pp, ssk, spk, vpk)$

$\mathcal{A}$ *wins the* VerSigSim-Sound *game if* $\mathsf{VerSigVal}(pp, spk, vpk, m^*, \sigma^*, \pi^*) = 1.$

Provable Signing

Lastly, we require a provable variant of signing, so that the signer is able to produce a proof that a signature is real. In other words, we require the signing algorithm $\mathsf{Sign}(pp, spk, ssk, vpk, m) \rightarrow (\sigma, \pi)$ to output π as well. We also require one additional validation algorithm, as follows:

$$\mathsf{RealSigVal}(pp, spk, vpk, m, \sigma, \pi) \rightarrow d \in \{0, 1\}$$

Definition 20 (Provable Signing Correctness)**.** *We say that the PSDVS has* Provable Signing Correctness *if* $\forall pp \leftarrow \mathsf{Setup}(1^\lambda)$, $(spk, ssk) \leftarrow \mathsf{SignKeyGen}(pp)$, $(vpk, vsk) \leftarrow \mathsf{VerKeyGen}(pp)$, $m \in \{0,1\}^*$, $(\sigma, \pi) \leftarrow \mathsf{Sign}(pp, spk, ssk, vpk, m)$,

$$\Pr[\mathsf{RealSigVal}(pp, spk, vpk, m, \sigma, \pi) = 1] = 1.$$

Definition 21 (Provable Signing Soundness)**.** *We say that the PSDVS has* Provable Signing Soundness *if the proof of correctness π produced by* Sign *does not verify unless σ verifies. More formally, an adversary should not be able to win the following game with non-negligible probability:*

$$\mathsf{Game}^{Sign\text{-}Sound}_{\mathsf{PVDVS},\mathcal{A}}(\lambda)$$

1. $pp \leftarrow \mathsf{Setup}(1^\lambda)$
2. $(spk, ssk) \leftarrow \mathsf{SignKeyGen}(pp)$
3. $(vpk, vsk) \leftarrow \mathsf{VerKeyGen}(pp)$

4. $(m^*, \sigma^*, \pi^*) \leftarrow \mathcal{A}(pp, ssk, spk, vpk)$

We say that $\mathcal{A}$ *wins the* Sign-Sound *game if* $\mathsf{RealSigVal}(pp, spk, vpk, m^*, \sigma^*, \pi^*) = 1$ *and* $\mathsf{Verify}(pp, spk, vsk, m^*, \sigma^*) = 0$.

Note that none of these proofs π are parts of the signature. If included in the signature, such proofs would allow an adversary to distinguish a simulation from a real signature.

5.3.2 Standard Primitive-Based MDVS Construction

Given a PSDVS, as defined in Section 5.3.1, we can build an MDVS. The transformation is straightforward: the signer uses the PSDVS to sign a message for each verifier, and proves consistency using a non-interactive zero knowledge proof of knowledge. The proof of consistency will claim that either all of the PSDVS signatures verify, or all of them are simulated. Const 1 describes this transformation.

Construction 1. *Let* $\mathsf{PSDVS} = (\mathsf{Setup}, \mathsf{SignKeyGen}, \mathsf{VerKeyGen}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{RealSigVal}, \mathsf{PubSigSim}, \mathsf{PubSigVal}, \mathsf{VerSigSim}, \mathsf{VerSigVal})$ *be a provably simulatable designated verifier signature scheme, and* $\mathsf{NIZK\text{-}PoK} = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *be a non-interactive zero knowledge proof of knowledge system and* $\mathcal{R}_{cons}$ *a relation that we will define later in the protocol.*

$\mathsf{Setup}(1^\lambda)$**:**

1. $crs \leftarrow \mathsf{NIZK\text{-}PoK.Setup}(1^\lambda, \mathcal{R}_{cons})$.
2. $\mathsf{PSDVS}.pp \leftarrow \mathsf{PSDVS.Setup}(1^\lambda)$.

Output $(crs, \mathsf{PSDVS}.pp)$ as the public parameters pp.

$\mathsf{SignKeyGen}(pp)$**:** $(spk_i, ssk_i) \leftarrow \mathsf{PSDVS.SignKeyGen}(\mathsf{PSDVS}.pp)$.

Output (spk_i, ssk_i) as signer i's public/secret key pair.

$\mathsf{VerKeyGen}(pp)$**:** $(vpk_j, vsk_j) \leftarrow \mathsf{PSDVS.VerKeyGen}(\mathsf{PSDVS}.pp)$.

Output (vpk_j, vsk_j) as verifier j's public/secret key pair.

$\mathsf{Sign}(pp, ssk_i, \{vpk_j\}_{j \in \mathcal{D}}, m)$**:**

1. For every verifier $j \in \mathcal{D}$, compute a signature and proof of signature validity as $(\sigma_j, \pi_j) \leftarrow \mathsf{PSDVS.Sign}(\mathsf{PSDVS}.pp, ssk_i, vpk_j, m)$.
2. Create a proof π of consistency, i.e a proof of knowledge of $\{\pi_j\}_{j\in\mathcal{D}}$ such that either all signatures are real (as demonstrated by $\{\pi_j\}_{j\in\mathcal{D}}$), or all signatures are fake (as could be demonstrated by the proofs produced by $\mathsf{PSDVS.PubSigSim}$ or $\mathsf{PSDVS.VerSigSim}$).

   More formally, for this $\mathsf{NIZK\text{-}PoK}$ we define a relation for a statement $u = (\mathsf{PSDVS}.pp, spk_i, \{vpk_j\}_{j\in\mathcal{D}}, \{\sigma_j\}_{j\in\mathcal{D}})$ and the witness $w = \{\pi_j\}_{j\in\mathcal{D}}$:

$$\begin{aligned}\mathcal{R}_{cons} = \Big\{ & u = (\mathsf{PSDVS}.pp, spk_i, \{vpk_j\}_{j\in\mathcal{D}}, \{\sigma_j\}_{j\in\mathcal{D}}), w = \{\pi_j\}_{j\in\mathcal{D}} : \\ & \Big(\bigwedge_{j\in\mathcal{D}} \mathsf{PSDVS.RealSigVal}(\mathsf{PSDVS}.pp, spk_i, vpk_j, m, \sigma_j, \pi_j) = 1\Big) \bigvee \\ & \Big(\bigwedge_{j\in\mathcal{D}} \big(\mathsf{PSDVS.VerSigVal}(\mathsf{PSDVS}.pp, spk_i, vpk_j, m, \sigma_j, \pi_j) = 1 \vee \\ & \mathsf{PSDVS.PubSigVal}(\mathsf{PSDVS}.pp, spk_i, vpk_j, m, \sigma_j, \pi_j) = 1\big)\Big)\Big\} \end{aligned} \tag{5.1}$$

   Let $\pi \leftarrow \mathsf{NIZK\text{-}PoK.Prove}(crs, u = (\mathsf{PSDVS}.pp, spk_i, \{vpk_j\}_{j\in\mathcal{D}}, \{\sigma_j\}_{j\in\mathcal{D}}), w = \{\pi_j\}_{j\in\mathcal{D}})$.
3. $\sigma = (\{\sigma_j\}_{j\in\mathcal{D}}, \pi)$.

Output σ as the signature.

$\mathsf{Verify}(pp, spk_i, vsk_j, m, \sigma = (\{\sigma_j\}_{j\in\mathcal{D}}, \pi))$**:**

1. Let $d_\pi \leftarrow \mathsf{NIZK\text{-}PoK.Verify}(crs, u = (\mathsf{PSDVS}.pp, spk_i, \{vpk_j\}_{j\in\mathcal{D}}, \{\sigma_j\}_{j\in\mathcal{D}}), \pi)$.
2. Let $d \leftarrow \mathsf{PSDVS.Verify}(\mathsf{PSDVS}.pp, spk_i, vsk_j, m, \sigma_j) \wedge d_\pi$.

Output d as the verification decision.

$\mathsf{Sim}(pp, spk_i, \{vpk_j\}_{j\in\mathcal{D}}, \{vsk_j\}_{j\in\mathcal{C}}, m)$**:**

1. For $j \in \mathcal{D} \cap \mathcal{C}$: $(\sigma_j, \pi_j) \leftarrow \mathsf{VerSigSim}(\mathsf{PSDVS}.pp, spk_i, vpk_j, vsk_j, m)$.
2. For $j \in \mathcal{D} \backslash \mathcal{C}$: $(\sigma_j, \pi_j) \leftarrow \mathsf{PubSigSim}(\mathsf{PSDVS}.pp, spk_i, vpk_j, m)$.

3. Use these signatures and proofs to produce the NIZK π for relation $\mathcal{R}_{cons}$.
4. $\sigma = (\{\sigma_j\}_{j\in\mathcal{D}}, \pi)$.

Output σ as the signature.

Theorem 8. *Assume* PSDVS *is a secure provably simulatable designated verifier signature scheme and* NIZK-PoK *is a secure non-interactive zero knowledge proof of knowledge system. Then Const 1 is a correct and secure MDVS scheme (without privacy of identities (Def 11)).*

Proof. Correctness is apparent by inspection. We show consistency, unforgeability and off-the-record separately.

Claim 1. *Const 1 is consistent, as per Def 8.*

Assume that Const 1 is inconsistent; then there exists an adversary $\mathcal{A}$ that can produce a message m^* and signature $\sigma^* = (\{\sigma_j^*\}_{j\in\mathcal{D}^*}, \pi^*)$ such that σ^* verifies for some, but not all, of the intended recipients $\mathcal{D}^*$. We can then use $\mathcal{A}$ to create another adversary $\mathcal{B}$ that can break either the security of the underlying PSDVS, or the security of the NIZK-PoK.

$\mathcal{B}$ receives pp, ssk, spk, vpk regardless of whether it's playing the *PubSigSim-Sound,*
VerSigSim-Sound or *Sign-Sound* games.

It randomly chooses identities to assign the given signer and verifier keys to; it generates the other signer and verifier keys honestly. It answers signing oracle queries honestly, since in all these cases it has the signer secret key. It answers key generation keys honestly as well, unless asked for ssk or the secret key corresponding to vpk; then, it aborts. However, since at least one signer and one verifier secret key must remain unqueried by $\mathcal{A}$, the probability of an abort is not overwhelming. Eventually, it gets $\mathcal{D}^*$ and $(m^*, \sigma^* = (\{\sigma_j^*\}_{j\in\mathcal{D}}, \pi^*))$ from $\mathcal{A}$; by assumption, with non-negligible probability, σ^* is inconsistent.

Assume without loss of generality that $j_0, j_1 \in \mathcal{D}^*$, $\mathsf{Verify}(pp, vsk_{j_0}, m^*, \sigma^*) = 0$, and $\mathsf{Verify}(pp, vsk_{j_1}, m^*, \sigma^*) = 1$. Assume, also without loss of generality, that NIZK-PoK.Verify is deterministic; then, the decision d_π regarding the validity of the zero knowledge proof of knowledge π^* must be the same for verifiers V_{j_0} and V_{j_1}, and so it must be that $d_\pi = 1$. It follows that in order for $\mathsf{Verify}(pp, vsk_{j_0}, m^*, \sigma^*) = 0$ we need $\mathsf{PSDVS.Verify}(\mathsf{PSDVS}.pp, vsk_{j_0}, m^*, \sigma_{j_0}^*) = 0$, and in order for $\mathsf{Verify}(pp, vsk_{j_1}, m^*, \sigma^*)$

$= 1$ we need $\mathsf{PSDVS.Verify}(\mathsf{PSDVS}.pp, vsk_{j_1}, m^*, \sigma^*_{j_1}) = 1$. Then, if $d_\pi = 1$, either π^* violates the soundness of NIZK-PoK, or one of the following must be true:

1. $\mathsf{PSDVS.RealSigVal}(\mathsf{PSDVS}.pp, spk, vpk_{j_0}, m^*, \sigma^*_{j_0}, \pi_{j_0}) = 1$. If this is the case, then $\mathcal{B}$ returns $(m^*, \sigma^*_{j_0}, \pi_{j_0})$ (the last of which is extractable from the knowledge soundness property of π^*) as a break of Provable Signing Soundness of the PSDVS, since we know that $\mathsf{PSDVS.Verify}(\mathsf{PSDVS}.pp, vsk_{j_0}, m^*, \sigma^*_{j_0}) = 0$.
2. $\mathsf{PSDVS.VerSigVal}(\mathsf{PSDVS}.pp, spk, vpk_{j_1}, m^*, \sigma^*_{j_1}, \pi_{j_1}) = 1$. If this is the case, then $\mathcal{B}$ returns $(m^*, \sigma^*_{j_1}, \pi_{j_1})$ (the last of which is extractable from the knowledge soundness property of π^*) as a break of VerSigSim Soundness, since the adversary was not given the secret key corresponding to vpk.
3. $\mathsf{PSDVS.PubSigVal}(\mathsf{PSDVS}.pp, spk, vpk_{j_1}, m^*, \sigma^*_{j_1}, \pi_{j_1}) = 1$. If this is the case, then $\mathcal{B}$ returns $(m^*, \sigma^*_{j_1}, \pi_{j_1})$ (the last of which is extractable from the knowledge soundness property of π^*) as a break of PubSigSim Soundness, since we know that
   $\mathsf{PSDVS.Verify}(\mathsf{PSDVS}.pp, vsk_{j_1}, m^*, \sigma^*_{j_1}) = 1$.

Claim 2. *Const 1 is existentially unforgeable, as per Def 9.*

This holds since any forgery of Const 1 either includes a forgery of the underlying PSDVS, or includes a fresh proof of knowledge on a statement for which the adversary does not have a witness, which is impossible by the knowledge soundness property of our NIZK proof of knowledge.

Claim 3. *Const 1 is off-the-record, as per Def 10.*

The simulation algorithm Sim, described above, is defined to use public and verifier simulation to produce the individual PSDVS signatures, and to prove that all PSDVS signatures are simulations instead of proving that they all verify.

Below we describe a sequence of games; in Game 0, it is impossible for the adversary to distinguish $b = 0$ from $b = 1$, since its view in the two cases are identically distributed. In each subsequent game, the advantage of the adversary is at most negligibly greater than in the previous one; in the final game, the adversary will find itself playing exactly the game described in Def 10.

Game 0**:** This gives $\mathcal{A}$ a real signature no matter what the value of b is.

$\mathcal{A}$ can have no advantage in this game.

Game 1**:** This game is the same as the previous game, except that if $b = 1$, the NIZK-PoK is simulated (and thus does not require the witnesses $\{\pi_j\}_{j \in \mathcal{D}^*}$).

If $\mathcal{A}$ distinguishes $b = 0$ from $b = 1$ with non-negligibly greater advantage than in the previous game, $\mathcal{B}$ can use $\mathcal{A}$ to break the zero-knowledge property of the NIZK-PoK.

Game $2.j$ **for** $j \in \mathcal{D}^* \cap \mathcal{C}^*$**:** This game is the same as the previous game, except that if $b = 1$, V_j's portion of the signature is replaced with a verifier simulation; that is, $(\sigma_j, \pi_j) \leftarrow \mathsf{VerSigSim}(pp, spk_{i^*}, vpk_j, vsk_j, m^*)$.

If $\mathcal{A}$ distinguishes $b = 0$ from $b = 1$ with non-negligibly greater advantage than in the previous game, $\mathcal{B}$ can use $\mathcal{A}$ to break the VerSigSim indistinguishability property.

Note that the statement the NIZK-PoK is proving no longer holds; however, the NIZK-PoK simulator must still produce a simulated NIZK-PoK that is indistinguishable from a real one, since otherwise the NIZK-PoK simulator would be useable to distinguish a signature produced by VerSigSim from a signature produced by Sign.

Game $3.j$ **for** $j \in \mathcal{D}^* \backslash \mathcal{C}^*$**:** This game is the same as the previous game, except that if $b = 1$, V_j's portion of the signature is replaced with a public simulation; that is, $(\sigma_j, \pi_j) \leftarrow \mathsf{PubSigSim}(pp, spk_{i^*}, vpk_j, m^*)$.

If $\mathcal{A}$ distinguishes $b = 0$ from $b = 1$ with non-negligibly greater advantage than in the previous game, $\mathcal{B}$ can use $\mathcal{A}$ to break the PubSigSim indistinguishability property.

Game 4**:** This game is the same as the previous game, except that if $b = 1$, we replace the simulated π with a real proof; since all the individual signatures are now simulated, such a valid proof can be computed again. Note that now, if $b = 1$, we are executing exactly the simulation procedure $\mathcal{SIM}$ described above, and thus this is exactly the game described in Def 10.

If $\mathcal{A}$ distinguishes $b = 0$ from $b = 1$ with non-negligibly greater advantage than in the previous game, $\mathcal{B}$ can use $\mathcal{A}$ to break the zero-knowledge property of the NIZK-PoK.

□

5.3.3 Standard Primitive-Based PSDVS Construction

We can build a PSDVS from a special message authentication code (MAC) which looks uniformly random without knowledge of the secret MAC key — such a MAC can be built from any pseudorandom function. A signature on a message m will

be a MAC on (m, t), where t is some random tag. Proving that the signature is real simply involves proving knowledge of a MAC key that is consistent with the MAC and some global public commitment to the MAC key. A public proof that the signature is simulated and does not verify would involve proving that the MAC was pseudorandomly generated. A verifier's proof that the signature is simulated would involve proving that the tag was generated in a way that only the verifier could use (e.g. from a PRF to which only the verifier knows the key).

Of course, this is not ideal, since MACs require knowledge of a shared key; in order to use MACs, we would need to set up shared keys between every possible pair of signer and verifier. However, we can get around this using non-interactive key exchange (NIKE). Each signer and verifier publishes a public key, and any pair of them can agree on a shared secret key by simply using their own secret key and the other's public key.

Const 2 describes this construction in more detail.

Construction 2. *Let:*

- $\mathsf{COMM} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open})$ *be a commitment scheme,*
- $\mathsf{PRF} = (\mathsf{KeyGen}, \mathsf{Compute})$ *be a length-preserving pseudorandom function,*
- PRG *be a length-doubling pseudorandom generator,*
- $\mathsf{NIZK} = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *be a non-interactive zero knowledge proof system, and*
- $\mathsf{NIKE} = (\mathsf{KeyGen}, \mathsf{KeyExtract}, \mathsf{KeyMatch})$ *be a non-interactive key exchange protocol.* $\mathsf{KeyMatch}$ *is an additional algorithm that checks if a public key and a secret key match.* $\mathsf{KeyMatch}$ *is not typically defined as a part of a* NIKE *scheme; however, such an algorithm always exists.*

We consider the public parameters for the underlying primitives $\mathsf{COMM}, \mathsf{PRF}, \mathsf{PRG}$, NIKE *together with the three common reference strings* (crs_1, crs_2, crs_3) *corresponding to the relations* $\mathcal{R}_1, \tilde{\mathcal{R}}_2, \tilde{\mathcal{R}}_3$ [9] *necessary to compute* NIZK *proofs as part of the public parameters of the PSDVS. Note that relations denoted* $\tilde{\mathcal{R}}$ *refer to statements of fake-ness, whereas relations denoted* $\mathcal{R}$ *refer to statements of real-ness.*

[9] The three relations will be defined later in the protocol description.

$\mathsf{Setup}(1^\lambda)$:

1. $crs_i \leftarrow \mathsf{NIZK.Setup}(1^\lambda, \mathcal{R}_i), i = 1, 2, 3.$
2. $ck \leftarrow \mathsf{COMM.Setup}(1^\lambda).$

Output $(\{crs_1, crs_2, crs_3\}, ck)$ as the public parameters pp.

$\mathsf{SignKeyGen}(pp)$:

1. $(\mathsf{NIKE}.pk_\mathsf{S}, \mathsf{NIKE}.sk_\mathsf{S}) \leftarrow \mathsf{NIKE.KeyGen}(1^\lambda).$
2. $ssk = \mathsf{NIKE}.sk_\mathsf{S}.$
3. $spk = \mathsf{NIKE}.pk_\mathsf{S}.$

Output ssk as the signer's secret key and spk as the signer's public key.

$\mathsf{VerKeyGen}(pp)$:

1. $(\mathsf{NIKE}.pk_\mathsf{V}, \mathsf{NIKE}.sk_\mathsf{V}) \leftarrow \mathsf{NIKE.KeyGen}(1^\lambda).$
2. $k_\mathsf{V} \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$. (Informally, this key will be used by the verifier to simulate signatures using $\mathsf{VerSigSim}$.)
3. Choose randomness (i.e. decommitment value) r_V at random.
4. $c_\mathsf{V} = \mathsf{COMM.Commit}(ck, k_\mathsf{V}; r_\mathsf{V})$. (Informally, this commitment will be used by the verifier to support its proofs of fake-ness.)
5. $vsk = (\mathsf{NIKE}.sk_\mathsf{V}, k_\mathsf{V}, r_\mathsf{V}).$
6. $vpk = (\mathsf{NIKE}.pk_\mathsf{V}, c_\mathsf{V}).$

Output vsk as the verifier's secret key and vpk as the verifier's public key.

$\mathsf{Sign}(pp, ssk = \mathsf{NIKE}.sk_\mathsf{S}, vpk = (\mathsf{NIKE}.pk_\mathsf{V}, c_\mathsf{V}), m)$:

The signer computes a shared key with the designated verifier and proceeds to sign the message m:

1. $k_{shared} = \mathsf{NIKE.KeyExtract}(\mathsf{NIKE}.sk_\mathsf{S}, \mathsf{NIKE}.pk_\mathsf{V})$. (Informally, this key will be used as a MAC key.)
2. Choose t at random.
3. $\sigma = (\sigma_1, \sigma_2) \leftarrow (t, \mathsf{PRF}_{k_{shared}}((m, t))).$

4. $\pi \leftarrow \mathsf{NIZK.Prove}(crs_1, u, w)$ where $u = ((\sigma_1, \sigma_2), \mathsf{NIKE}.pk_\mathsf{S}, \mathsf{NIKE}.pk_\mathsf{V}, m)$ and $w = (\mathsf{NIKE}.sk_\mathsf{S}, k_{shared}))$

   We define the relation $\mathcal{R}_1$ indexed by NIKE public parameters and PRF for a statement u and witness w:

$$\begin{aligned}\mathcal{R}_1 = \{(u = (\sigma_1, \sigma_2, \mathsf{NIKE}.pk_\mathsf{S}, \mathsf{NIKE}.pk_\mathsf{V}, m), w = (\mathsf{NIKE}.sk_\mathsf{S}, k_{shared})) :\\ \mathsf{KeyMatch}(\mathsf{NIKE}.pk_\mathsf{S}, \mathsf{NIKE}.sk_\mathsf{S}) = 1\\ \wedge\ k_{shared} = \mathsf{NIKE.KeyExtract}(\mathsf{NIKE}.sk_\mathsf{S}, \mathsf{NIKE}.pk_\mathsf{V})\\ \wedge\ \sigma_2 = \mathsf{PRF}_{k_{shared}}((m, \sigma_1))\}\end{aligned}$$

Output σ as the signature, and π as the proof of real-ness.

$\mathsf{Verify}(pp, spk = \mathsf{NIKE}.pk_\mathsf{S}, vsk = (\mathsf{NIKE}.sk_\mathsf{V}, k_\mathsf{V}, r_\mathsf{V}), m, \sigma = (\sigma_1, \sigma_2))$:

1. $k_{shared} = \mathsf{NIKE.KeyExtract}(\mathsf{NIKE}.sk_\mathsf{V}, \mathsf{NIKE}.pk_\mathsf{S})$. (Informally, this key will be used as a MAC key.)
2. If $\mathsf{PRF}_{k_{shared}}((m, \sigma_1)) = \sigma_2$, set $d = 1$. Otherwise, set $d = 0$.

Output d as the verification decision.

$\mathsf{RealSigVal}(pp, spk, vpk, m, \sigma, \pi)$:

Output $d \leftarrow \mathsf{NIZK.Verify}(crs_1, \sigma, \pi)$ as the validation decision.

$\mathsf{PubSigSim}(pp, m)$:

1. Choose a PRG seed s.
2. Choose σ_1 and σ_2 pseudorandomly by running PRG on s.
3. $\sigma \leftarrow (\sigma_1, \sigma_2)$.
4. Let $\pi \leftarrow \mathsf{NIZK.Prove}(crs_2, u = \sigma, w = s)$.

   We define the relation $\tilde{\mathcal{R}}_2$ indexed by the PRG for a statement $u = (\sigma = (\sigma_1, \sigma_2))$ and the witnesses $w = s$:

$$\tilde{\mathcal{R}}_2 = \{(u = \sigma; w = s) : u = \mathsf{PRG}(w)\} \tag{5.2}$$

Output σ as the simulated signature, and π as the proof of fake-ness.

$\mathsf{PubSigVal}(pp, spk, vpk, m, \sigma = (\sigma_1, \sigma_2), \pi)$**:**

Output $d \leftarrow \mathsf{NIZK.Verify}(crs_2, \sigma, \pi)$ as the validation decision.

$\mathsf{VerSigSim}(pp, spk = \mathsf{NIKE}.pk_\mathsf{S}, vpk = (\mathsf{NIKE}.pk_\mathsf{V}, c_\mathsf{V}), vsk = (\mathsf{NIKE}.sk_\mathsf{V}, k_\mathsf{V}, r_\mathsf{V}), m)$**:**

The verifier can fake a signature using its PRF key k_V.

1. $k_{shared} = \mathsf{NIKE.KeyExtract}(\mathsf{NIKE}.sk_\mathsf{V}, \mathsf{NIKE}.pk_\mathsf{S})$.
2. Choose r at random.
3. $t \leftarrow \mathsf{PRF}_{k_\mathsf{V}}(r)$.
4. $\sigma \leftarrow (t, \mathsf{PRF}_{k_{shared}}((m, t)))$.
5. Let $\pi \leftarrow \mathsf{NIZK.Prove}(crs_3, u = (c_\mathsf{V}, \sigma_1), w = (k_\mathsf{V}, r_\mathsf{V}, r))$.

   We define the relation $\tilde{\mathcal{R}}_3$ indexed by the NIKE public parameters and PRF for statements u and witnesses w:

$$\tilde{\mathcal{R}}_3 = \{(u = (c_\mathsf{V}, \sigma_1), w = (k_\mathsf{V}, r_\mathsf{V}, r)) : \\ k_\mathsf{V} = \mathsf{COMM.Open}(c_\mathsf{V}, r_\mathsf{V}) \wedge \ \sigma_1 = \mathsf{PRF}_{k_\mathsf{V}}(r)\}$$

Output σ as the simulated signature and π as the proof of fake-ness.

$\mathsf{VerSigVal}(pp, spk, vpk, m, \sigma, \pi)$**:**

Output $d \leftarrow \mathsf{NIZK.Verify}(crs_3, (c_\mathsf{V}, \sigma_1), \pi)$ as the validation decision.

Theorem 9. *If the schemes* COMM, PRF, PRG, NIZK, NIKE *are secure, then Const 2 is a correct and secure PSDVS scheme as per Def 13.*

Proof. Correctness. It is straightforward to verify that any honestly generated signature will pass the verification test.

Existential Unforgeability. We can reduce the unforgeability of the PVDVS to the pseudorandomness of the underlying PRF. Suppose there exists a forger $\mathcal{A}$ having non-negligible advantage in winning the existential unforgeability game 9 for a single signer and a single verifier (see Def 9). We can use this forger to build a distinguisher $\mathcal{B}$ that is able to break the pseudo-randomness property of the PRF. $\mathcal{B}$ runs $\mathcal{A}$ and simulate the signing queries m of $\mathcal{A}$ by picking a random tag t and forwarding the

query (m, t) to its evaluation oracle that outputs either the evaluation $\mathsf{PRF}(m, t)$, either a truly random value. In the first case, the forger $\mathcal{A}$ has the same view as in the game 9. Given that $\mathcal{A}$ is able to forge a signature on a fresh message m^* in the first case, but not in the second, then this implies that the adversary $\mathcal{B}$ can distinguish between the two, breaking the pseudo-randomness of the PRF.

PubSigSim Indistinguishability (Def 14). By the security property of the PRG (indistinguishability from real randomness), the advantage of an adversary $\mathcal{A}$ in the game $\mathsf{Game}_{\mathsf{PVDVS},\mathcal{A}}^{PubSigSim\text{-}Sound}(\lambda)$ is bounded by the advantage of an adversary $\mathcal{B}$ in distinguishing between PRG and a truly random generator. Note that we can apply the property only for the first half of the signature, adversary $\mathcal{A}$ should not be able to distinguish between a random t and σ_1 generated as $(\sigma_1, \sigma_2) \leftarrow \mathsf{PRG}(s)$.

PubSigSim Correctness (Def 15). By the completeness of the NIZK scheme, any proof π generated honestly by running the PubSigSim, i.e $\pi \leftarrow \mathsf{NIZK.Prove}(crs_2, u = \sigma, w)$ will be validated by $\mathsf{PubSigVal}(pp, spk, vpk, m, \sigma, \pi)$ that runs NIZK.Verify $(crs_2, u = \sigma, \pi)$ algorithm.

PubSigSim Soundness (Def 16). Suppose there is an adversary $\mathcal{A}$ that wins the game of *PubSigSim-Sound* game with non negligible probability. Then, from an output (m^*, σ^*, π^*) of $\mathcal{A}(ssk, spk, vpk)$ we have from the soundness of the NIZK, since PubSigVal outputs 1, that there is a value s such that $\sigma^* = (\sigma_1^*, \sigma_2^*) \leftarrow \mathsf{PRG}(s)$ and also $\sigma^* = (\sigma_1^*, \mathsf{PRF}(m^*, \sigma_1^*))$ from the verification check of the signature $\mathsf{Verify}(pp, spk, vsk, m^*, \sigma^*) = 1$. This implies there exists a collision $\mathsf{PRG}(s) = (\sigma_1^*, \mathsf{PRF}(m^*, \sigma_1^*))$ breaking the pseudorandomness of the underlying primitives PRG and PRF. In the first case, given a signature $\sigma^* = (\sigma_1^*, \mathsf{PRF}(m^*, \sigma_1^*))$ that verifies, $\mathcal{A}$ should not be able to find a preimage s^* for σ^* with respect to the PRG with advantage significantly better than for a truly random function, without breaking the pseudorandomness of the PRG. Otherwise, from computing an output of the pseudorandom generator $\mathsf{PRG}(s) = (\sigma_1^*, \sigma_2^*)$, $\mathcal{A}$ should not be able to find a (fixed prefix) preimage (t, m^*) of PRF such that $t = \sigma_1^*$. This is indeed infeasable without breaking the pseudorandom property of the PRF.

VerSigSim Indistinguishability (Def 17). This follows from the pseudorandomness of our PRF. Remark that both a real signature and a verifier-simulated signature pass the verification test, the only difference is in how the tag t is generated, truly random, or as $t \leftarrow \mathsf{PRF}_{k_\mathsf{V}}(r)$.

VerSigSim Correctness (Def 18). As in the case of PubSigSim correctness, this holds by considering the completeness of the NIZK scheme, since an honnest proof

generated by $\mathsf{VerSigSim}$, will be validated by $\mathsf{VerSigVal}$ that simply runs $\mathsf{NIZK.Verify}$ algorithm.

VerSigSim Soundness (Def 19.) This follows from the properties of the underlying COMM and NIZK schemes. Consider an adversary $\mathcal{A}$ that is able to win the game of VerSigSim-Sound, meaning that it produces a couple (σ, π) validated by $\mathsf{VerSigVal}$. Then, if the NIZK scheme is assumed to be sound, the following should hold: $\mathcal{A}$ is able to compute an opening k_{V} of c_{V}, breaking the hidding of the commitment scheme COMM or $\mathcal{A}$ is able to find a preimage r for $\mathsf{PRF}_{k_{\mathsf{V}}}$, i.e $\sigma_1 = \mathsf{PRF}_{k_{\mathsf{V}}}(r)$ which breaks the pseudorandomness of the PRF.

Provable Signing Correctness (Def 20). As in the case of PubSigSim and VerSigSim, this follows by definition of $\mathsf{RealSigVal}$ and the completeness of the NIZK.

Provable Signing Soundness (Def 21). This holds if we assume soundness of the NIZK proof togheter with NIKE security properties and pseudorandomness of the PRF. Assuming that the NIZK is sound, then an adversary winning the game $\mathsf{RealSigVal}$, is able either to break the soundness of the NIKE scheme or to find a preimage of the PRF. Finding a preimage is infeasible, given the pseudorandomness property of the PRF.

□

5.3.4 DDH and Paillier-Based PSDVS Construction

The goal of this section is to construct a PSDVS scheme based on DDH and the security of Paillier encryption. The idea in the PSDVS construction is that the authenticator for a message m will be $\mathsf{H}(m,t)^k$ in a group G where t is a nonce, k is a key known to both parties and H is a hash function modeled as a random oracle. The construction requires that certain properties of the key can be proved in zero-knowledge, and we can do this efficiently using standard Σ-protocols because the key is in the exponent. However, naive use of this idea would mean that a sender needs to store a key for every verifier he talks to, and the set-up must generate correlated secret keys for the parties. To get around this, we will instead let the sender choose k on the fly and send it to the verifier, encrypted using a new variant of Paillier encryption. In the following subsection we describe and prove this new encryption

scheme, and then we specify the actual PSDVS construction. Paillier-style encryption comes in handy since its algebraic properties are useful in making our zero knowledge proofs efficient.

Paillier-based Authenticated and Verifiable Encryption

An authenticated and verifiable encryption scheme (AVPKE) involves a sender S and a receiver V. Such a scheme comes with the following polynomial time algorithms:

$\mathsf{Setup}(1^\lambda) \rightarrow pp$**:** A probabilistic algorithm for setup which outputs public parameters.

$\mathsf{KeyGen}_\mathsf{S}(pp) \rightarrow (sk_\mathsf{S}, pk_\mathsf{S})$**:** A probabilistic sender key generation algorithm.

$\mathsf{KeyGen}_\mathsf{V}(pp) \rightarrow (sk_\mathsf{V}, pk_\mathsf{V})$**:** A probabilistic receiver key generation algorithm.

$\mathsf{Enc}_{pp,sk_\mathsf{S},pk_\mathsf{V}}(k) \rightarrow CT$**:** A probabilistic encryption algorithm for message k.

$\mathsf{Dec}_{pp,sk_\mathsf{V},pk_\mathsf{S}}(CT) \rightarrow \{k, \perp\}$**:** A decryption algorithm that outputs either reject or a message.

We require, of course, that $\mathsf{Dec}_{pp,sk_\mathsf{V},pk_\mathsf{S}}(\mathsf{Enc}_{pp,sk_\mathsf{S},pk_\mathsf{V}}(k)) = k$ for all messages k.

Intuitively, the idea is that given only the receiver public key pk_V and his own secret key sk_S, the sender S can encrypt a message k in such a way that on receiving the ciphertext, V can check that k comes from S, no third party knows k and finally, the encryption is verifiable in that it allows S to efficiently prove in zero-knowledge that k satisfies certain properties.

To help understand our concrete construction of an AVPKE scheme, we recall that standard Paillier encryption of a message k under the public key n is defined as $(n+1)^k v^n \bmod n^2$ where $v \in_\mathsf{R} \mathbb{Z}_n^*$. In (Damgård and Jurik, 2003), it was suggested that first, v can be chosen as $\pm\hat{g}^s$ for a random s and a $\hat{g}$ of large order modulo n – or equivalently, a random number of Jacobi symbol 1 mod n. This is not a security problem, as the Jacobi symbol of v can be efficiently computed from the ciphertext anyway. Further, they suggested that v can be chosen similarly as in El-Gamal encryption, if the sender sends along a random power of $\hat{g}$. They also showed

that the resulting encryption scheme is still CPA secure under the same assumption. In this way all users can share the same modulus, which comes in very handy in our setting.

We add an authentication mechanism to this encryption scheme and get the following AVPKE scheme.

Construction 3. *Let:*

- Ggen *be a* Group Generator*, a probabilistic polynomial time algorithm which on input* 1^λ *outputs the description of a cyclic group* G *and a generator* g*, such that the order of* G *is a random* λ*-bit RSA modulus* n*, which is the product of so-called safe primes. (That is,* $n = pq$ *where* $p = 2p' + 1, q = 2q' + 1$ *and* p', q' *are also primes.) Finally, we need the algorithm to output an element* $\hat{g} \in \mathbb{Z}_n^*$ *of order* $p'q'$*.*
- $\mathsf{NIZK} = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *be a simulation-sound non-interactive zero knowledge proof system. In this section, we will use* Σ*-protocols made non-interactive using the Fiat-Shamir heuristic, so in this case* Setup *is empty and there is no common reference string.*

Ggen *can be constructed using standard techniques. For instance, first generate* n *using standard techniques, then repeatedly choose a small random number* r *until* $P = 2rn+1$ *is a prime. Let* g' *be a generator of* $\mathbb{Z}_P^*$*. Then let* G *be the subgroup of* $\mathbb{Z}_P^*$ *generated by* $g = g'^{2r} \bmod P$*.*[10] *Finally, to construct the element* $\hat{g}$*, let* $u \in_{\mathsf{R}} \mathbb{Z}_n$ *and set* $\hat{g} = u^2 \bmod n$*. Indeed, this is a random square, and since the subgroup of squares modulo* n *has only large prime factors in its order (*p' *and* q'*), a random element is a generator with overwhelming probability*[11]*.*

$\mathsf{Setup}(1^\lambda)$**:** *Run* Ggen *to generate a modulus* n *and* $\hat{g} \in \mathbb{Z}_n^*$ *as explained above. Output* $pp = (n, \hat{g})$*.*

$\mathsf{KeyGen}_\mathsf{S}(pp)$**:** *Pick* $sk_\mathsf{S} \in_\mathsf{R} \mathbb{Z}_n$*, and set* $pk_\mathsf{S} = \hat{g}^{sk_\mathsf{S}}$*. Output* $(sk_\mathsf{S}, pk_\mathsf{S})$*.*

$\mathsf{KeyGen}_\mathsf{V}(pp)$**:** *Pick* $\alpha_1, \alpha_2 \in_\mathsf{R} \mathbb{Z}_n$*, set* $sk_\mathsf{V} = (\alpha_1, \alpha_2)$*, and set* $pk_\mathsf{V} = (\beta_1, \beta_2) = (\hat{g}^{\alpha_1}, \hat{g}^{\alpha_2})$*.*

The public key values are statistically indistinguishable from random elements in the group generated by $\hat{g}$ *since* n *is a sufficiently good "approximation" to the order* $p'q'$ *of* $\hat{g}$*.*

---

[10] The group G will be more prominently used in the construction of the PSDVS scheme.

[11] This set-up need to keep the factorization of n secret. Hence, to avoid relying on a trusted party, the parties can use an interactive protocol to generate n securely, there are several quite efficient examples in the literature.

$\mathsf{Enc}_{pp,sk_\mathsf{S},pk_\mathsf{V}}(k; r, b_1, b_2)$:

1. *The randomness should have been picked as follows:* $r \in_\mathsf{R} \mathbb{Z}_n$ *and* $b_1, b_2 \in_\mathsf{R} \{0, 1\}$.
2. *Set* $CT_1 = (-1)^{b_1} \hat{g}^r \bmod n$.
3. *Set* $CT_2 = (n+1)^k ((-1)^{b_2} \beta_1^{sk_\mathsf{S}} \beta_2^r \bmod n)^n \bmod n^2$.
4. *Let* π_{valid} *be a non-interactive zero-knowledge proof of knowledge wherein given public data* $(n, \hat{g}, (CT_1, CT_2))$, *the prover shows knowledge of a witness* $w = (k, r, b_1, v)$ *such that* $CT_1 = (-1)^{b_1} \hat{g}^r$ *and* $CT_2 = (n+1)^k v^n \bmod n^2$. *An honest prover can use* $v = (-1)^{b_2} \beta_1^{sk_\mathsf{S}} \beta_2^r \bmod n$. *The factor* $(-1)^{b_1}$ *is only in the ciphertext for technical reasons: it allows* π_{valid} *to be efficient.*

Output $CT = (CT_1, CT_2, \pi_{valid})$.

$\mathsf{Dec}_{pp,sk_\mathsf{V}=(\alpha_1,\alpha_2),pk_\mathsf{S}}(CT = (CT_1, CT_2, \pi_{valid}))$:

1. *Check that* CT_1, CT_2 *have Jacobi symbol 1 modulo* n, *and check* π_{valid}. *Output reject if either check fails.*
2. *Let* $u = pk_\mathsf{S}^{\alpha_1} CT_1^{\alpha_2} \bmod n$ *and check that* $(CT_2 u^{-n})^n \bmod n^2 = \pm 1$. $\mathbb{Z}_{n^2}^*$ *contains a unique subgroup of order* n, *generated by* $n+1$. *So here we are verifying that – up to a sign difference –* $CT_2 u^{-n} \bmod n^2$ *is in the subgroup generated by* $n+1$. *If the check fails, output reject.*
3. *Otherwise, compute* k *such that* $(n+1)^k = \pm CT_2 u^{-n} \bmod n^2$.[12]

An AVPKE scheme should allow anyone to make "fake" ciphertexts that look indistinguishable from real encryptions, given only the system parameters. Furthermore, the receiver V should be able to use his own secret key sk_V and the public key pk_S of the sender to make ciphertexts with exactly the same distribution as real ones. This is indeed true for our scheme:

Fake Encryption: Let $r \in_\mathsf{R} \mathbb{Z}_n, b, b' \in_\mathsf{R} \{0, 1\}$ and $v \in \mathbb{Z}_n^*$ be a random square. Then,

$$\mathsf{Enc}_{pp,fake}(k; r, b, b', v) = ((-1)^b \hat{g}^r \bmod n, (n+1)^k ((-1)^{b'} v)^n \bmod n^2), \pi_{valid}$$

where π_{valid} is constructed following the NIZK prover algorithm.

[12] k can be computed using the standard "discrete log" algorithm from Paillier decryption.

V**'s Equivalent Encryption:**

$$\mathsf{Enc}_{pp,sk_{\mathsf{V}},pk_{\mathsf{S}}}(k;r,b_1,b_2) =$$

$$((-1)^{b_1}\hat{g}^r \bmod n,\ (n+1)^k(-1)^{b_2}(pk_{\mathsf{S}}^{\alpha_1}\hat{g}^{r\alpha_2} \bmod n)^n \bmod n^2),\ \pi_{valid}$$

where $r \in_{\mathsf{R}} \mathbb{Z}_n$, $b_1, b_2 \in_{\mathsf{R}} \{0,1\}$ and π_{valid} is constructed following the NIZK prover algorithm.

In the following, we will sometimes suppress the randomness from the notation and just write, e.g., $\mathsf{Enc}_{pp,sk_{\mathsf{S}},pk_{\mathsf{V}}}(k)$.

By simple inspection of the scheme it can be seen that:

Lemma 1. *For all k, $\mathsf{Dec}_{pp,sk_{\mathsf{V}},pk_{\mathsf{S}}}(\mathsf{Enc}_{pp,sk_{\mathsf{S}},pk_{\mathsf{V}}}(k)) = k$. Furthermore, encryption by S and by V returns the same ciphertexts: for all messages k and randomness r, b_1, b_2, we have $\mathsf{Enc}_{pp,sk_{\mathsf{S}},pk_{\mathsf{V}}}(k;r,b_1,b_2) = \mathsf{Enc}_{pp,sk_{\mathsf{V}},pk_{\mathsf{S}}}(k;r,b_1,b_2)$.*

Lemma 2. *If DDH in $\langle\hat{g}\rangle$ is hard, then $(k, \mathsf{Enc}_{pp,sk_{\mathsf{S}},pk_{\mathsf{V}}}(k;r,b_1,b_2))$ is computationally indistinguishable from $(k, \mathsf{Enc}_{pp,fake}(k;r',b,b',v))$ for any fixed message k and randomness $r, b_1, b_2, r', b, b', v$, as long as the discrete log of β_2 to the base $\hat{g}$ is unknown.*

Proof. This follows immediately from that fact that $(\hat{g}^r, \beta_2^r)$ is indistinguishable from $(\hat{g}^{r'}, v)$ by assumption. □

Definition 22. *Consider the following experiment for an AVPKE scheme and a probabilistic polynomial time adversary $\mathcal{A}$: Run the set-up and key generation and run $A^{\mathcal{O}_E,\mathcal{O}_D}(pp, pk_{\mathsf{V}}, pk_{\mathsf{S}})$. Here, $\mathcal{O}_E$ takes a message k as input and returns $\mathsf{Enc}_{pp,sk_{\mathsf{S}},pk_{\mathsf{V}}}(k)$, while $\mathcal{O}_D$ takes a ciphertext and returns the result of decrypting it under $pk_{\mathsf{S}}, sk_{\mathsf{V}}$ (which will be either reject or a message). $\mathcal{A}$ wins if it makes $\mathcal{O}_D$ accept a ciphertext that was not obtained from $\mathcal{O}_E$. The scheme is* authentic *if any PPT $\mathcal{A}$ wins with negligible probability.*

Lemma 3. *If the DDH problem in $\langle\hat{g}\rangle$ is hard, the AVPKE scheme defined above is authentic.*

Proof. Assume for contradiction that we have adversary $\mathcal{A}$ who wins the authenticity game with non-negligible probability. We now stepwise transform the game into an algorithm that will solve the computational Diffie-Hellman problem in $\langle\hat{g}\rangle$, which will

certainly contradict the assumption. Let Game 0 be the original authenticity game. In Game 1, we replace $\mathcal{O}_E$ by $\mathcal{O}_E^{fake}$ which on input k from $\mathcal{A}$ returns $\mathsf{Enc}_{pp,fake}(k)$. At the same time, we replace $\mathcal{O}_D$ by an alternative version $\mathcal{O}_D^{alt}$ that does not use the α_2 part of sk_{V}, namely instead of computing $u = pk_{\mathsf{S}}^{\alpha_1} c_1^{\alpha_2}$ it extracts r from the proof in the ciphertext and computes instead $u = pk_{\mathsf{S}}^{\alpha_1} \beta_2^r$ which is the same value (up to a sign) and hence leads to an equivalent decryption. In addition, if it gets an encryption $\mathsf{Enc}_{pp,fake}(k)$ produced by the fake encryption oracle, it always returns k. Since now the discrete log of β_2 base $\hat{g}$ is not used, we can use our assumption and Lemma 2 to conclude that $\mathcal{A}$'s winning probability in the new game is essentially the same. In Game 2, we guess the index j of the first call to $\mathcal{O}_D$ where $\mathcal{A}$ gets an accept for a forged ciphertext, which we can do with an inverse polynomial probability, since $\mathcal{A}$ is poly-time. If $\mathcal{A}$ does not win the game at the j'th call, we declare that it loses. Clearly $\mathcal{A}$ still wins Game 2 with non-negligible probability. In Game 3, we replace $\mathcal{O}_D^{alt}$ by $\mathcal{O}_D^{sim}$ which, up to (but not including) call j, responds to a ciphertext that was output by $\mathcal{O}_E^{fake}$ with the message that was encrypted and rejects anything else. Observe that in the event where $\mathcal{A}$ wins Game 2, $\mathcal{O}_D^{sim}$ simulates $\mathcal{O}_D^{alt}$ perfectly, so $\mathcal{A}$ wins Game 3 with the same probability as Game 2.

Now, observe that we can execute Game 3 up to step j and get $\mathcal{A}$'s ciphertext for the j'th call without access to the secret keys of S and V. We can therefore take group elements $\hat{g}, pk_{\mathsf{S}} = \hat{g}^{sk_{\mathsf{S}}}, \beta_1 = \hat{g}^{\alpha_1}$ as input to the CDH problem (where the goal is to find $\hat{g}^{sk_{\mathsf{S}}\alpha_1}$), and execute Game 3 with $\mathcal{A}$ on this input. Assume $\mathcal{A}$ wins, and let c^* be the ciphertext submitted in the j'th call to $\mathcal{O}_D^{sim}$. Recall that c^* is of form (c_1, c_2, π_{valid}). By simulation soundness of the NIZK used, we can extract the witness claimed in the proof, and so we get r and k, v such that $c_1 = \pm\hat{g}^r$ and $c_2 = (n+1)^k v^n \bmod n^2$.

We can assume that V's decryption algorithm would accept c^*, this implies that v must have Jacobi symbol 1. Also, since the algorithm computes $u = pk_{\mathsf{S}}^{\alpha_1} c_1^{\alpha_2} \bmod n$ and checks that $(c_2 u^{-n})^n \bmod n^2 = \pm 1$, we can assume this check is satisfied. Inserting the expression we have for c_2 we get

$$\pm 1 = (c_2 u^{-n})^n \bmod n^2 = ((n+1)^k v^n u^{-n})^n \bmod n^2 = ((vu^{-1} \bmod n)^n)^n \bmod n^2$$

Here, we have used the fact that $\mathbb{Z}_{n^2}^*$ is the direct product of a group G of order n and a group H of order $\phi(n)$ isomorphic to $\mathbb{Z}_n^*$ under the isomorphism $x \mapsto x^n \bmod n^2$. Since $(vu^{-1} \bmod n)^n \bmod n^2 \in H$ and raising to power n is a 1-1 mapping on H, it

follows that $vu^{-1} \bmod n = \pm 1$. Inserting the expression for u, we get

$$\pm v = u = pk_{\mathsf{S}}^{\alpha_1} c_1^{\alpha_2} = pk_{\mathsf{S}}^{\alpha_1} \hat{g}^{r\alpha_2} = \hat{g}^{sk_{\mathsf{S}}\alpha_1} \beta_2^r$$

In conclusion, we can flip a coin and submit plus or minus $v\beta_2^{-r} \bmod n$ as a solution to the CDH problem and this will be correct with half the probability with which $\mathcal{A}$ wins Game 3. □

We proceed to show that the AVPKE scheme hides the message encrypted even if adversary knows the secret key of the sender, and even if a decryption oracle is given. This is essentially standard CCA security.

Definition 23. *Consider the following experiment for an AVPKE scheme and a probabilistic polynomial time adversary $\mathcal{A}$: Run the set-up and key generation and run $\mathcal{A}^{\mathcal{O}_E}(pp, pk_{\mathsf{V}}, sk_{\mathsf{S}})$. Here, $\mathcal{O}_E$ takes two messages k_0, k_1 as input, selects a bit η at random and returns $CT^* = \mathsf{Enc}_{pp,sk_{\mathsf{S}},pk_{\mathsf{V}}}(k_\eta)$. $\mathcal{O}_D$ takes a ciphertext and returns the result of decrypting it under $pk_{\mathsf{S}}, sk_{\mathsf{V}}$ (which will be either reject or a message). $\mathcal{A}$ may submit anything other than CT^* to $\mathcal{O}_D$, and must output a bit η' at the end. It wins if $\eta' = \eta$. The scheme is* private *if any PPT $\mathcal{A}$ wins with negligible advantage over $\frac{1}{2}$.*

In the following we will use the assumption underlying the Paillier encryption scheme, sometimes known as the *composite degree residuosity assumption* (CDRA): a random element x in $\mathbb{Z}_{n^2}^*$ where $x \bmod n$ has Jacobi symbol 1 is computationally indistinguishable from $y^n \bmod n^2$ where $y \in \mathbb{Z}_n^*$ is random of Jacobi symbol 1[13].)

Lemma 4. *Assume that DDH in $\langle \hat{g} \rangle$ is hard and that CDRA holds. Then the AVPKE scheme satisfies Definition 23.*

Proof. Assume for contradiction that adversary $\mathcal{A}$ wins the game with probability non-negligibly larger than 1/2. Let Game 0 be the original game. We change the game stepwise to get new games that can be used to either solve DDH or break CDRA. In Game 1, we replace $\mathcal{O}_D$ by an alternative version $\mathcal{O}_D^{alt}$ that does not use the α_2 part of sk_{V}, namely instead of computing $u = pk_{\mathsf{S}}^{\alpha_1} c_1^{\alpha_2}$ it extracts r from

[13]The original CDRA assumption does not have the restriction to Jacobi symbol 1, but since the Jacobi symbol is easy to compute without the factors of n, the two versions are equivalent.

the proof in the ciphertext and computes instead $u = pk_{\mathsf{S}}^{\alpha_1}\beta_2^r$ which is the same value (up to a sign) and hence leads to an equivalent decryption. $\mathcal{A}$ wins Game 1 with exactly the same probability. In Game 2, we replace $\mathcal{O}_E$ by $\mathcal{O}_E^{sim}$, defined as follows: take $a, b \in \langle \hat{g} \rangle$ as input, where $a = \hat{g}^r, b = \beta_2^r$. let $\eta \in_{\mathsf{R}} \{0,1\}, r \in_{\mathsf{R}} \mathbb{Z}_n$ and return $(\pm a, (n+1)^{k_\eta}(\pm \beta_1^{sk_{\mathsf{S}}} b)^n \bmod n^2, \pi_{valid})$, where π_{valid} is a simulated proof. Except for the simulation, this is exactly Game 1, so $\mathcal{A}$ wins with essentially the same probability – note that by simulation soundness, the witness extraction used by $\mathcal{O}_D^{alt}$ still works. In Game 3, we make a, b be random group elements in $\langle \hat{g} \rangle$. Now, $\mathcal{A}$'s winning probability remains essentially the same since otherwise, we could use the difference between Game 2 and 3 to solve DDH. Now note that in Game 3 $(\pm \beta_1^{sk_{\mathsf{S}}} b)^n \bmod n^2$ is in fact a uniformly random element of form $y^n \bmod n^2$, where the Jacobi symbol of $y \bmod n$ is 1. In Game 4, we replace this by x chosen uniformly in $\mathbb{Z}_{n^2}^*$ subject to $x \bmod n$ having Jacobi symbol 1. In Game 4, c^* has no information on η, so here $\mathcal{A}$ wins with probability 1/2. This means that we can use the difference between Game 3 and 4 to break CDRA, and we have a contradiction. □

We say that an AVPKE scheme is *secure* if it is authentic, private, supports equivalent encryption by V and indistinguishable fake encryption.

The PSDVS Scheme

We now return to the promised PSDVS scheme.

Construction 4. *Let:*

- Ggen *be a* Group Generator*, a probabilistic polynomial time algorithm which on input* 1^λ *outputs* $G, g, n, \hat{g}$ *exactly as in the previous AVPKE construction.*
- H *be a hash function which we model as a random oracle. We assume it maps onto the group* G*.*
- $\mathsf{NIZK} = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ *be a simulation-sound non-interactive zero knowledge proof system. In this section, we will use* Σ*-protocols made non-interactive using the Fiat-Shamir heuristic, so in this case* Setup *is empty and there is no common reference string.*

$\mathsf{Setup}(1^\lambda)$**:** Let $(G, g, \hat{g}, n) \leftarrow \mathsf{Ggen}(1^\lambda)$ and let $h \in_{\mathsf{R}} G$. Set $pp = (G, g, \hat{g}, n, h)$. Return pp as the public parameters.

$\mathsf{SignKeyGen}(pp)$**:** Run key generation for the AVPKE scheme as defined above to get keys $ssk = sk_{\mathsf{S}}, spk = pk_{\mathsf{S}}$ for the signer S. Output ssk as the signer's secret key and spk as the signer's public key.

$\mathsf{VerKeyGen}(pp)$**:**

1. Run key generation for the AVPKE scheme as defined above to get keys $sk_{\mathsf{V}}, pk_{\mathsf{V}}$ for the verifier V. (These keys will be used to sign messages and verify signatures.)
2. Choose $k_{\mathsf{V}} \in_{\mathsf{R}} \mathbb{Z}_n$. (This key will be used by the verifier to simulate signatures using $\mathsf{VerSigSim}$.)
3. Choose $r_{\mathsf{V}} \in_{\mathsf{R}} \mathbb{Z}_n$ and let $c_{\mathsf{V}} = g^{k_{\mathsf{V}}} h^{r_{\mathsf{V}}}$. (This commitment will be used by the verifier to support its proofs of fake-ness.)
4. $vsk = (sk_{\mathsf{V}}, k_{\mathsf{V}}, r_{\mathsf{V}})$, $vpk = (pk_{\mathsf{V}}, c_{\mathsf{V}})$.

Output vsk as the verifier's secret key and vpk as the verifier's public key.

$\mathsf{Sign}(pp, ssk = sk_{\mathsf{S}}, pk_{\mathsf{V}}, m)$**:**

1. Choose $t \in_{\mathsf{R}} G, r \in_{\mathsf{R}} \mathbb{Z}_n, b_1, b_2 \in_{\mathsf{R}} \{0,1\}, s \in_{\mathsf{R}} \mathbb{Z}_n^*, k_s \in_{\mathsf{R}} \mathbb{Z}_n$.
2. Let $\sigma \leftarrow (t, \mathsf{H}(m,t)^{k_s}, \mathsf{Enc}_{pp, sk_{\mathsf{S}}, pk_{\mathsf{V}}}(k_s; r, b_1, b_2))$.
3. $\pi \leftarrow \mathsf{NIZK.Prove}(u = (\sigma = (\sigma_1, \sigma_2, \sigma_3), pk_V, pk_S, m), w = (sk_S, k_s, r, b_1, b_2))$ be a zero knowledge proof of knowledge of witness w such that:

$$\sigma_2 = \mathsf{H}(m, \sigma_1)^{k_s} \wedge \sigma_3 = \mathsf{Enc}_{pp, sk_{\mathsf{S}}, pk_{\mathsf{V}}}(k_s; r, b_1, b_2)$$

Output σ as the signature, and π as the proof of real-ness. Recall that the ciphertext by construction already contains a proof implying that the ciphertext contains a well defined plaintext. The role of the tag t in the signature is to let the verifier give a proof for his way to simulate a signature. This will become clear below.

$\mathsf{Verify}(pp, spk = pk_{\mathsf{S}}, vsk = (sk_{\mathsf{V}}, k_{\mathsf{V}}, r_{\mathsf{V}}), m, \sigma = (\sigma_1, \sigma_2, \sigma_3))$**:**

1. Decrypt σ_3 as $k_s = \mathsf{Dec}_{pp,sk_{\mathsf{V}},pk_{\mathsf{S}}}(\sigma_3)$. If this fails, set $d = 0$ and abort.
2. If $\sigma_2 = \mathsf{H}(m, \sigma_1)^{k_s}$, set $d = 1$. Otherwise, set $d = 0$.

Output d as the verification decision.

Informally, forging a signature is hard since the verifier rejects forged ciphertexts and valid ones hide the k_s value inside, by properties of the AVPKE scheme. The only other option is to reuse an existing k_s in a new signature, which is hard if CDH is hard in G. Namely, you have to raise a random group element (output by the random oracle) to the secret exponent k_s.

$\mathsf{PubSigSim}(pp, m)$:

1. Choose $k, k' \in_{\mathsf{R}} \mathbb{Z}_n$, such that $k \neq k'$.
2. Choose $t \in_{\mathsf{R}} G$, $r \in_{\mathsf{R}} \mathbb{Z}_n, b, b' \in_{\mathsf{R}} \{0, 1\}$, $v \in_{\mathsf{R}} \mathbb{Z}_n$, such that v has Jacobi symbol 1.
3. $\sigma \leftarrow (t, \mathsf{H}(m, t)^k, \mathsf{Enc}_{pp,fake}(k'; r, b, b', v))$.
4. Let $\pi \leftarrow \mathsf{NIZK.Prove}(u = \sigma = (\sigma_1, \sigma_2, \sigma_3), w = (k, k'))$ be a zero-knowledge proof of knowledge such that:

$$\sigma_2 = \mathsf{H}(m, \sigma_1)^k \wedge \sigma_3 = \mathsf{Enc}_{pp,fake}(k'; \cdot, \cdot, \cdot, \cdot) \wedge k \neq k'.$$

Output σ as the simulated signature, and π as the proof of fake-ness. The notation $\mathsf{Enc}_{pp,fake}(k'; \cdot, \cdot, \cdot, \cdot)$ means that the proof only has to establish that the plaintext inside the encryption is some value k' different from k.

Informally, a simulated signature looks like a real one since fake encryptions are indistinguishable from real ones and by privacy of the encryption scheme, one cannot decide efficiently if $k = k'$ or not. Clearly, a fake signature as defined is always rejected by the verifier.

$\mathsf{VerSigSim}(pp, spk = pk_{\mathsf{S}}, vpk = (pk_{\mathsf{V}}, c_{\mathsf{V}}), vsk = (sk_{\mathsf{V}}, k_{\mathsf{V}}, r_{\mathsf{V}}), m)$:

1. Choose $r_t \in_{\mathsf{R}} \mathbb{Z}_n$, $t = g^{k_{\mathsf{V}}} h^{r_t}$, $k_s \in_{\mathsf{R}} \mathbb{Z}_n$, $b_1, b_2 \in_{\mathsf{R}} \{0, 1\}$ and $v \in_{\mathsf{R}} \mathbb{Z}_n^*$ of Jacobi

symbol 1.

2. $\sigma \leftarrow (t, \mathsf{H}(m,t)^{k_s}, \mathsf{Enc}_{pp,sk_\mathsf{V},pk_\mathsf{S}}(k_s; r, b_1, b_2))$.

3. Let $\pi \leftarrow \mathsf{NIZK.Prove}(u = ((\sigma_1, \sigma_2, \sigma_3), c_\mathsf{V}, m), w = (k_\mathsf{V}, r_\mathsf{V}, r_t))$ be a zero-knowledge proof of knowledge of witness $w = (k_\mathsf{V}, r_\mathsf{V}, r_t)$ such that:

$$\sigma_1 = g^{k_\mathsf{V}} h^{r_t} \wedge c_\mathsf{V} = g^{k_\mathsf{V}} h^{r_\mathsf{V}}.$$

Output σ as the simulated signature and π as the proof of fake-ness.

Informally, the simulated signature has exactly the same distribution as a real signature, and cannot be distinguished even given the verifier's key: for every t there exists a r_t that the verifier could have used to generate it. Only the verifier can prove fake-ness since no one else knows k_V, and so giving a proof would, with overwhelming probability, require opening c_V to a value different from k_V, which is infeasible if discrete log is hard in G.

Theorem 10. *If the AVPKE scheme is secure, and under the DDH assumption, Const 4 is a secure PSDVS scheme.*

Remark 8. *To get a concrete instantiation of the PSDVS scheme, we need to instantiate the AVPKE scheme (as explained above) and also the NIZKs assumed in Const 4 (as explained in Section 5.5). This way, we get an instantiation in the random oracle model, secure under the strong RSA, the DDH and the CDRA assumptions.*

Proof. In this proof we refer to the definitions in Section 5.3.1.
Unforgeability. Suppose adversary $\mathcal{A}$ wins the unforgeability game (in presence of a signing and verification oracle), and let $\sigma^* = (\sigma_1^*, \sigma_2^*, \sigma_3^*)$ be the forged signature, and m^* the message in question. If σ_3^* is a ciphertext that was not output by the signing oracle, then $\mathcal{A}$ can be used to break authenticity of the AVPKE scheme, since V only accepts a signature if decryption of σ_3^* does not reject. So we may assume that σ_3^* is a valid ciphertext $\mathsf{Enc}_{pp,sk_\mathsf{S},pk_\mathsf{V}}(k)$ that was used in a genuine signature $\sigma = (t, \mathsf{H}(m,t)^k, \mathsf{Enc}_{pp,sk_\mathsf{S},pk_\mathsf{V}}(k))$. Since $\mathcal{A}$ wins, $m \neq m^*$, so we can assume that $\mathsf{H}(m,t)$ and $\mathsf{H}(m^*, \sigma_1^*)$ are independent random variables output by the random oracle, and furthermore since V accepts the forged signature, we have $\sigma_2^* = \mathsf{H}(m, t^*)^k$.

This means we can use $\mathcal{A}$ to solve CDH in $\langle \hat{g} \rangle$ which contradicts the assumption that DDH (and hence CDH) is hard: Given a random CDH challenge $\hat{h}, \hat{h}^a, \hat{h}^b$, we will guess which genuine signature and which calls to the random oracle will be involved in the forgery. We will program the random oracle so that $\mathsf{H}(t, m) = \hat{h}$ and set $\sigma_2 = \hat{h}^a$. This means that we are implicitly claiming that the exponent used in the signature is a. This does not match the encryption $\mathsf{Enc}_{pp,sk_\mathsf{S},pk_\mathsf{V}}(k)$, but we program the verification oracle to accept the signature anyway, and by privacy of the encryption scheme, the inconsistency does not affect $\mathcal{A}$'s behavior significantly. Finally, we set $\mathsf{H}(t^*, m^*) = \hat{h}^b$, and it is now clear that if $\mathcal{A}$ wins, we have $\sigma_2^* = (\hat{h}^b)^a = \hat{h}^{ab}$.

Provable Public Simulation. PubSigSim indistinguishability follows from privacy of the AVPKE scheme: if an adversary $\mathcal{A}$ wins the PubSigSim-Ind game, we can use $\mathcal{A}$ to construct an adversary that wins the privacy game. The adversary can use sk_S to emulate the signing oracle and the use the decryption oracle to emulate the (less powerful) verification oracle. PubSigSim correctness and PubSigSim soundness follow immediately from completeness and soundness of the NIZK used.

Provable Verifier Simulation. VerSigSim indistinguishability is clear, since the signature produced by VerSigSim has exactly the same distribution as regular signatures. VerSigSim correctness follows from completeness of the NIZK used. Finally, for VerSigSim soundness, assume that a corrupt S could produce a proof that VerSigVal would accept. By soundness of the NIZK used, this would mean that we could extract from the proof data such that both the tag t in the signature and c_V could be opened as commitments to the same value, say x. However, V already knows from the key generation how to open c_V to the value k_V. The adversary gets no information on k_V during the game because commitments are perfectly hiding, and so $x \neq k_\mathsf{V}$ with overwhelming probability. This means we can use the adversary to break the binding property of the commitment scheme.

Sign. It is easy to see that Provable Signing Correctness and Soundness follow immediately from completeness and soundness of the NZIK used in Sign. □

In Section 5.5 we list, for completeness, the standard Σ-protocols we need to instantiate our construction.

5.3.5 Sketch of a PSDVS Scheme Based on Prime Order Groups

In this section we sketch a variant of the scheme in Section 5.3.4 where we need only prime order groups and no trusted set-up. So we let G be a group of prime order p with generator g, where $p = 2q+1$ and q is also prime. We let H denote the subgroup of Z_p^* of order q. H is also the group of squares modulo p. We let h be a generator of H. These parameters can be generated in public and can be verified by anyone, so no trusted set-up with secret trapdoor is required.

We make public keys for S and V as follows: $pk_{\mathsf{S}} = h^{sk_{\mathsf{S}}}$, $pk_{\mathsf{V}} = h^{sk_{\mathsf{V}}}$, $sk_{\mathsf{S}}, sk_{\mathsf{V}} \in_{\mathsf{R}} \in Z_q$.

The parties can compute a shared key $k = h^{sk_{\mathsf{S}} sk_{\mathsf{V}}}$ from only the public keys, which is pseudorandom for anyone else if DDH in H is hard.

We let the signature on m be $\mathsf{H}(m,t)^k$, which can be verified by V in the obvious way.

To prove in ZK that a signature $\sigma = (\sigma_1, \sigma_2)$ is valid, one proves knowledge of sk_{S}, such that $g^{pk_{\mathsf{S}}} = g^{h^{sk_{\mathsf{S}}}}$ and $\sigma_2 = \mathsf{H}(m,\sigma_1)^k = \mathsf{H}(m,\sigma_1)^{pk_{\mathsf{V}}^{sk_{\mathsf{S}}}}$ share the same "level-2 " discrete log, which can be done using a standard protocol which, however, requires a number of exponentiations linear in the security parameter.

For $\mathsf{PubSigSim}$, one generates a fake signature by choosing a random element $e \in \mathbb{Z}_p^*$ and letting the simulated signature be $\mathsf{H}(m,t)^{-e^2 \bmod p}$. $-e^2 \bmod p$ is not a square modulo p and hence is not in H. It will therefore not be accepted, since the correct k is in H by construction. This is indistinguishable from a genuine signature if we make a nonstandard variant of the DDH assumption saying that g raised to a random square is indistinguishable from g raised to a random non-square. One can show in ZK that $\mathsf{H}(m,t)^{-e^2 \bmod p}$ has the right form by showing that the discrete log of its inverse is a square, which can be done with standard Σ protocols.

The case of $\mathsf{VerSigSim}$ is handled in exactly the same way as the previous con-

struction.

5.4 FE-based Construction

In this section, we present an MDVS scheme based on functional encryption. One disadvantage of this scheme is that it requires a trusted setup; secret verification keys must be derived from a master secret key. However, the accompanying advantage is that this scheme has verifier-identity-based signing; verifiers' public keys consist simply of their identity, allowing any signer to encrypt to any set of verifiers without needing to retrieve their keys from some PKI first.

At a high level, we are first given a digital signature scheme (DS) and a functional encryption scheme (FE). The keys of the signer with identity i are a secret DS signing key sk_i and corresponding public DS verification key vk_i. An MDVS signature CT is a FE ciphertext obtained by encrypting the plaintext that consists of the message m, the signer's DS verification key vk_i, a set of designated verifier identities $\mathcal{D}$, and the signer's DS signature σ on the message using the secret DS signing key sk_i. That is, $CT = \mathsf{FE.Enc}(pp, (m, vk_i, \mathcal{D}, \sigma))$. Verifier j's public key is simply their identity j (that is, $vpk_j = j$). Their secret key consists of a DS key pair (sk_j, vk_j), and an FE secret key dk_j. dk_j is the secret key for a function that checks whether j is among the specified designated verifiers, and then checks whether the DS signature σ inside the ciphertext CT is either a valid signature under the signer's verification key vk_i, or under the verifier's verification key vk_j. However, this basic scheme does not give us the off-the-record property; we therefore tweak it slightly, as we describe below.

From One to Many DS Signatures

In order to ensure that any subset of valid verifiers cannot convince an outsider of the origin of the MDVS signature, we need to replace the one DS signature in the ciphertext with a set of DS signatures. The reason is that, if only one signature is

contained in the ciphertext, any designated verifier can prove to an outsider that "it was either me or the signer that constructed the signature". If more than one verifier proves this about the same MDVS signature, then the signature must have come from the signer.

To prevent this kind of "intersection attack", we allow the ciphertext to contain a set Σ of DS signatures, and change the corresponding FE secret keys to check if there exists a DS signature in the set that either verifies under the signer's or the verifier's DS verification key. Now, an outsider will no longer be convinced that it was the signer who constructed the MDVS signature, since each of the colluding verifiers could have constructed a DS signature that verifies under their own verification key, and then encrypted this set together with the public verification key of the signer.

Achieving Consistency

In order to achieve consistency, we need security against malicious encryption in the underlying FE scheme. We need to ensure that any (possibly maliciously generated) ciphertext is consistent with one specific message across decryption with different functions. Otherwise, a malicious MDVS signer may be able to construct a ciphertext (i.e. a signature) that will be valid for one designated verifier but not valid for another, thereby breaking the consistency property. Security against a malicious encryption is a property of verifiable functional encryption (VFE), which was introduced by Badrinarayanan et. al (Badrinarayanan et al., 2016). However, it turns out that we do not need the full power of VFE, which also includes precautions against a malicious setup. Thus, we define a weaker notion of VFE, and substitute the standard FE scheme with this new scheme allowing us to achieve the MDVS consistency property.

In Section 5.4.1 we introduce ciphertext verifiable functional encryption, followed by our MDVS construction based on functional encryption which is presented in Section 5.4.2.

5.4.1 Functional Encryption

An FE scheme starts with an authority that generates the public parameters pp and a master secret key msk. Then the holder of the master secret key can generate a decryption key dk_f associated with some function f that belongs to some predefined function family. Anyone can generate an encryption CT of some value x, using only the public parameters, and the party that has been given the decryption key dk_f can decrypt the ciphertext CT to obtain $f(x)$.

The standard security properties of functional encryption consider only the case where an adversary holds a set of decryption keys $dk_{f_1}, \ldots, dk_{f_q}$, and wants to learn more than it is allowed to about some encrypted message. The security property says that given an encryption of x, the adversary should only learn $f_1(x), \ldots, f_q(x)$.

However, in some settings, we additionally need security against malicious encryption, and possibly a malicious key generation authority. To achieve this, Badrinarayanan et. al (Badrinarayanan et al., 2016) introduced verifiable functional encryption (VFE), which is an FE scheme extended with verification algorithms that check the validity of the ciphertexts and decryption keys.

We require security only against malicious encryption, allowing us to define a weaker notion of verifiability for functional encryption that handles malicious encryptors and decryptors, while assuming an honest authority.

Ciphertext Verifiable Functional Encryption

Let $\mathcal{F} = \mathcal{F}_\lambda$ be a function family, $\mathcal{M} = \mathcal{M}_\lambda$ the message space, and $\mathcal{Y} = \mathcal{Y}_\lambda$ the output space such that $\mathcal{F} : \mathcal{M} \to \mathcal{Y}$. Let $\mathcal{C} = \mathcal{C}_\lambda$ be the ciphertext space. Then, we define a *ciphertext verifiable functional encryption* (VFE) scheme for function family $\mathcal{F}$ by the following algorithms:

$\mathsf{Setup}(1^\lambda) \to (pp, msk)$**:** The PPT algorithm Setup, on input the security parameter

λ, outputs the public parameters pp and the master secret key msk.

$\mathsf{KeyGen}(msk, f) \to dk_f$: The PPT key generation algorithm KeyGen, on input the master secret key msk and a function $f \in \mathcal{F}$, outputs a secret key dk_f.

$\mathsf{Enc}(pp, m) \to CT$: The PPT encryption algorithm Enc, on input the public parameters pp and a message m, outputs a ciphertext CT.

$\mathsf{Dec}(dk_f, CT) \to y'$: The decryption algorithm Dec, on the decryption key dk_f and the ciphertext CT, outputs $y' \in \mathcal{Y} \cup \{\perp\}$.

$\mathsf{Verify}(pp, CT) \to d$: The public verification algorithm Verify, on input the public parameters pp and the ciphertext CT, outputs a boolean decision $d = 0$ (reject) or $d = 1$ (accept).

Properties

A functional encryption scheme must have the standard correctness (Def 24) and IND-CPA security (Def 25). A ciphertext verifiable scheme must additionally have ciphertext verifiability (Def 26).

Definition 24 (Correctness). *Let $\lambda \in \mathbb{N}$ be the security parameter, and let $\mathcal{F} : \mathcal{M} \to \mathcal{Y}$ be a function family. Let $\mathsf{VFE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be a FE scheme for function family $\mathcal{F}$. For all messages $m \in \mathcal{M}$, all functions $f \in \mathcal{F}$ we have*

$$\Pr\left[\mathsf{Dec}(\mathsf{KeyGen}(msk, f), \mathsf{Enc}(pp, m)) \neq f(m)\right] \leq \nu(\lambda),$$

where $(pp, msk) \leftarrow \mathsf{Setup}(1^\lambda)$, and the probability is taken over the random choices of all algorithms.

Definition 25 (IND-CPA Security). *Let $\lambda \in \mathbb{N}$ be the security parameter, and let $\mathcal{F} : \mathcal{M} \to \mathcal{Y}$ be a function family. Let $\mathsf{VFE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be a FE scheme for function family $\mathcal{F}$. Consider the following game between a challenger and an adversary $\mathcal{A}$:*

$\mathsf{Game}^{\mathsf{IND-CPA}}_{\mathsf{VFE},\mathcal{F},\mathcal{A}}(\lambda)$

1. $(pp, msk) \leftarrow \mathsf{Setup}(1^\lambda)$
2. $(m_0, m_1) \leftarrow \mathcal{A}^{\mathcal{O}_G}(pp)$

3. $b \leftarrow_\$ \{0,1\}$
4. $CT \leftarrow \mathsf{Enc}(pp, m_b)$
5. $b' \leftarrow \mathcal{A}^{\mathcal{O}_G}(CT)$

The key generation oracle is defined $\mathcal{O}_G(f_i) := \mathsf{KeyGen}(msk, f_i)$.

We say that $\mathcal{A}$ wins the IND-CPA game *if* $b = b'$, $|m_0| = |m_1|$, *and* $f_i(m_0) = f_i(m_1)$ *for all queries* $f_i \in \mathcal{F}$ *to oracle* $\mathcal{O}_G$. *We say a FE scheme satisfies the IND-CPA security property if, for all PPT* $\mathcal{A}$,

$$\mathsf{Adv}^{\mathsf{IND-CPA}}_{\mathsf{VFE},\mathcal{F},\mathcal{A}}(\lambda) = \Pr[\mathcal{A} \textit{ wins the game}] - \frac{1}{2} \leq \nu(\lambda).$$

We consider a *ciphertext verifiable functional encryption* scheme $\mathsf{VFE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Verify})$ for function family $\mathcal{F}$. In such a scheme, we also need security against a malicious encryptor. In Def 26 we simplify the verifiability of (Badrinarayanan et al., 2016) to consider only ciphertext verifiability, not .

Definition 26 (Ciphertext Verifiability)**.** *A scheme* VFE *for function family* $\mathcal{F}$ *is ciphertext verifiable, if, for all* $CT \in \{0,1\}^*$, *there exists* $x \in \mathcal{M}$ *such that for all* $f \in \mathcal{F}$ *and* $dk_f \leftarrow \mathsf{KeyGen}(msk, f)$, *if* $\mathsf{Verify}(pp, CT) = 1$, *then*

$$\Pr\left[\mathsf{Dec}(dk_f, CT) = f(x)\right] = 1,$$

where $(pp, msk) \leftarrow \mathsf{Setup}(1^\lambda)$.

Def 26 states that for all ciphertexts constructed by a malicious encryptor, it must hold that if the ciphertext CT passes the verification algorithm, then there exists a unique input x that can be associated with ciphertext CT, meaning that for all functions $f \in \mathcal{F}$ the decryption of CT will yield $f(x)$. As an example, consider the functions f_{even} and f_{odd} that determine whether an input natural number is even or odd. If the scheme has ciphertext verifiability, the adversary should not be able to create a ciphertext CT that passes the verification algorithm, and on decryption using these functions will yield a result that claims that the element encrypted in CT is both even and odd.

As discussed in (Goyal et al., 2015), an FE scheme satisfying these properties can be achieved by combining a standard FE scheme with a simulation-sound NIZK proof of knowledge to achieve security against malicious encryption.

5.4.2 The MDVS Construction

Construction 5. *Let* $\mathsf{SIGN} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *be a standard digital signature scheme and let* $\mathsf{VFE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Verify})$ *be a functional encryption scheme secure with ciphertext verifiability. Then we define a MDVS scheme* $\mathsf{FEMDVS} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{Sim})$ *as follows:*

$\mathsf{Setup}(1^\lambda)$**:** $(pp^{\mathsf{FE}}, msk^{\mathsf{FE}}) \leftarrow \mathsf{VFE.Setup}(1^\lambda)$.

Output public parameter $pp = pp^{\mathsf{FE}}$ and master secret key $msk = msk^{\mathsf{FE}}$.

$\mathsf{SignKeyGen}(i)$**:** $(sk_i, vk_i) \leftarrow \mathsf{SIGN.KeyGen}(1^\lambda)$.

Output the signer's secret key $ssk_i = sk_i$ and public key $spk_i = vk_i$.[14]

$\mathsf{VerKeyGen}(msk, j)$**:**

1. $vpk_j = j$,
2. $(sk_j, vk_j) \leftarrow \mathsf{SIGN.KeyGen}(1^\lambda)$,
3. $dk_j \leftarrow \mathsf{VFE.KeyGen}(msk^{\mathsf{FE}}, f_j)$, where f_j is defined as follows.

Function f_j

Input: $m, vk_i, \{vpk_{j'}\}_{j' \in \mathcal{D}}, \Sigma$;
Const: vpk_j, vk_j;
1. If $vpk_j \notin \{vpk_{j'}\}_{j' \in \mathcal{D}}$: output $\bot$;
2. If $\exists \sigma \in \Sigma$: $\mathsf{SIGN.Verify}(vk_i, m, \sigma) = 1$ OR $\mathsf{SIGN.Verify}(vk_j, m, \sigma) = 1$:
    output $(m, vk_i, \{vpk_{j'}\}_{j' \in \mathcal{D}})$;
3. Else: output $\bot$

Output the verifiers secret key $vsk_j = (sk_j, dk_j)$ and public key $vpk_j = j$.[15]

$\mathsf{Sign}(pp, ssk_i, \{vpk_j\}_{j \in \mathcal{D}}, m)$**:**

[14] We assume that the mapping $i \rightarrow (ssk_i, spk_i)$ is unique in the system. This can be achieved without loss of generality by pseudorandomly generating the randomness required in the key generation process from the identity i and the master secret key.

[15] We assume that the mapping $j \rightarrow (vsk_j, vpk_j)$ is unique in the system. This can be achieved wlog by pseudorandomly generating the randomness required in the key generation process from the identity j and the master secret key.

1. $\sigma \leftarrow \mathsf{SIGN.Sign}(sk_i, m)$.
2. Output $CT = \mathsf{VFE.Enc}(pp^{\mathsf{FE}}, (m, vk_i, \{vpk_j\}_{j\in\mathcal{D}}, \{\sigma, \perp, \cdots, \perp\}))$.

$\mathsf{Verify}(pp, spk_i, vsk_j, \{vpk_j\}_{j\in\mathcal{D}}, m, CT)$**:**

1. Check whether $\mathsf{VFE.Verify}(pp^{\mathsf{FE}}, CT) = 1$. If not, output 0.
2. Compute $(m', vk_i', \{vpk_j\}_{j\in\mathcal{D}'})\backslash\perp \leftarrow \mathsf{VFE.Dec}(dk_j, CT)$. If the output is $\perp$, output 0.
3. Check $m' = m$, $vk_i' = vk_i$ (with $spk_i = vk_i$), and $\mathcal{D}' = \mathcal{D}$. If all hold, output 1. Otherwise output 0.

$\mathsf{Sim}(pp, spk_i, \{vpk_j\}_{j\in\mathcal{D}}, \{vsk_j\}_{j\in\mathcal{C}}, m)$**:**

1. For each $j \in \mathcal{C}$, $vsk_j = (sk_j, dk_j)$.
2. Compute $\sigma_j \leftarrow \mathsf{SIGN.Sign}(sk_j, m^*)$.
3. Let $\Sigma = \{\sigma_j\}_{j\in\mathcal{C}^*}$, add default values to get the required size.
4. Output $CT = \mathsf{VFE.Enc}(pp^{\mathsf{FE}}, (m^*, spk_i, \{vpk_j\}_{j\in\mathcal{D}}, \Sigma))$.

Theorem 11. *Assume that* VFE *is an IND-CPA secure functional encryption scheme with ciphertext verifiability, and* SIGN *is an existential unforgeable digital signature scheme. Then Const 5 is a correct and secure MDVS scheme with privacy of identities and verifier-identity-based signing.*

Proof. Correctness: Follows directly from an inspection of the algorithms and the correctness of the functional encryption scheme.

Consistency: Assume that there exist an adversary that produces an inconsistent signature: $(i^*, \{vpk_j\}_{j\in\mathcal{D}^*}, m^*, CT^*)$, such that there exists $j_1, j_2 \in \mathcal{D}^*$ (for which the adversary does not have the corresponding secret keys) such that:

$$\mathsf{Verify}(spk_{i^*}, vsk_{j_1}, \{vpk_j\}_{j\in\mathcal{D}^*}, m^*, CT^*) = 1,$$
$$\mathsf{Verify}(spk_{i^*}, vsk_{j_2}, \{vpk_j\}_{j\in\mathcal{D}^*}, m^*, CT^*) = 0,$$

where $spk_{i^*} = vk_{i^*}$, $vsk_{j_1} = (sk_{j_1}, dk_{j_1})$, and $vsk_{j_2} = (sk_{j_2}, dk_{j_2})$.

Since the verification for j_1 yields 1, then both j_1 and j_2 will verify the ciphertext: $\mathsf{VFE.Verify}(pp^{\mathsf{FE}}, CT) = 1$, meaning that (thanks to ciphertext verifiability) there

exists a unique encrypted message $(m, vk_i, \{vpk_j\}_{j\in\mathcal{D}}, \sigma)$ in CT that is consistent across decryption with different functions.

Then j_1 will decrypt: $(m, vk_i, \{vpk_j\}_{j\in\mathcal{D}}) \leftarrow \mathsf{VFE.Dec}(dk_j, CT)$, which is equal to $(m^*, vk_{i^*}, \{vpk_j\}_{j\in\mathcal{D}^*})$. On the other hand j_2 will decrypt

$$(m', vk'_i, \{vpk_j\}_{j\in\mathcal{D}'}) \text{ or } \perp \leftarrow \mathsf{VFE.Dec}(dk_j, CT)$$

If the output was $\perp$, then either $j_2 \notin \mathcal{D}$ or there does not exists a valid signature in σ. Otherwise the output was $(m', vk'_i, \{vpk_j\}_{j\in\mathcal{D}'})$ which is different from $(m, vk_i, \{vpk_j\}_{j\in\mathcal{D}})$ in at least one component. Thus, the output of the decrypt of j_1 and j_2 is not consistent with a unique message, which contradicts the fact that $\mathsf{VFE.Verify}(pp^{\mathsf{FE}}, CT) = 1$ (i.e. there exists a unique encrypted message).

Thus, an adversary that violated the consistency of the MDVS scheme, violates the ciphertext verifiability of the FE scheme.

Existential Unforgeability: Assume that the adversary produces a valid forgery: $(i^*, \{vpk_j\}_{j\in\mathcal{D}^*}, m^*, CT^*)$, where CT^* is the signature (a FE ciphertext) on message m^*, from signer i^* designated to the verifiers in the set $\mathcal{D}^*$.

Then there exists a designated verifier $j \in \mathcal{D}^*$, who has not been corrupted by the adversary, such that the FE decryption will yield m', vk'_i and $\mathcal{D}'$, where $m' = m^*$, $vk'_i = vk_{i^*}$, and $\mathcal{D}' = \mathcal{D}^*$ (otherwise it would not be a valid forgery).

The only thing left in the FE ciphertext is the set of signatures σ. In order for CT^* to be a forgery, then there must exist a digital signature $\sigma \in \sigma$ such that

$$\mathsf{SIGN.Verify}(vk'_i, m', \sigma) = 1 \text{ OR } \mathsf{SIGN.Verify}(vk_j, m', \sigma) = 1.$$

This means that the adversary must create a digital signature forgery for either signer i or "signer" j, without knowing the corresponding secret signing keys. This contradicts the assumption that the digital signature satisfies existential unforgeability.

Privacy of Identities: The adversary receives a signature CT^*, which is an FE encryption of one of the two following messages:

1. $(m^*, vk_{i_0}, \{vpk_j\}_{j\in\mathcal{D}_0}, \{\mathsf{SIGN.Sign}(sk_{i_0}, m^*), \perp, \cdots, \perp\})$,
2. $(m^*, vk_{i_1}, \{vpk_j\}_{j\in\mathcal{D}_1}, \{\mathsf{SIGN.Sign}(sk_{i_1}, m^*), \perp, \cdots, \perp\})$.

In the PSI game, the adversary is not allowed to ask for verification keys for any of the designated verifiers in $\mathcal{D}_0$ or $\mathcal{D}_1$. This means that for all verification keys (i.e. FE

decryption keys) the adversary can ask for, we have that the underlying function f_j evaluated on the two messages will yield $\perp$, since j is not in any of the two sets of designated verifiers.

Thus, privacy of the identities (PSI and PVI) follows directly from the IND-CPA security of the functional encryption scheme.

Off-The-Record: The off-the-record property follows directly from the IND-CPA security of the functional encryption scheme. The messages in the real and simulated version have the same length, since we add default values to σ to ensure we have the same number of elements in both cases.

For all functions f_j that the adversary gets an FE decryption key for, we argue that evaluation on each of the two messages results in the same output. First we look at the decryption keys dk_j for $j \notin \mathcal{D}^*$ (i.e. not a designated verifier). In both the real and the simulated case the decryption will yield $\perp$, since j is not a designated verifier.

Next, we look at the decryption keys dk_j for $j \in \mathcal{D}^*$. In this case j must also be in the corruption set $\mathcal{C}^*$. Thus, in the real case the set σ contains σ_{i^*} a digital signature of message m^* under the signing key of signer i^*, and the decryption will yield $(m^*, vk_{i^*}, \{vpk_j\}_{j \in \mathcal{D}^*})$. In the simulated case the set σ contains σ_j a digital signature of message m^* under the signing key of party j. Thus, the decryption will again yield $(m^*, vk_{i^*}, \{vpk_j\}_{j \in \mathcal{D}^*})$, since function f_j does not differentiate whether it was the verification vk_{i^*} or vk_j that was used to verify the digital signature.

□

5.5 Appendix A: Instantiation of Non-Interactive ZK Proofs

We list here, for completeness, the standard Σ-protocols we need. We assume the same set-up as above, that is, a group G of order an RSA modulus n is given (a product of safe primes), as well as a generator g of G, and a generator $\hat{g}$ of the subgroup of squares mod n.

The protocols we list here are well-known or simple variations of know protocols. It is easy to show the standard completeness, soundness and honest verifier ZK properties, so we will not present these proofs, but recall some ideas where these may

be less well known. The protocols can be turned into non-interactive ZK proofs of knowledge in the standard way in the random oracle model using the Fiat-Shamir heuristic.

All proofs have negligible soundness error, so we always need only 1 iteration of each protocol.

A general technical remark: some of the protocols are usually designed for use in a group of prime order, while here we use them in a group of order n. The only difficulty this could lead to is that the proofs of soundness requires us to invert various non-zero numbers modulo the group order. This could in principle fail modulo n, but this would lead to finding a non-trivial factor of n (which is generated in a trusted manner as part of the setup) and so can only happen with negligible probability if factoring is hard, which we have to assume throughout anyway.

Protocols for the AVPKE scheme. Some number theoretic background first: because $n = pq = (2p' + 1)(2q' + 1)$ is a product of safe primes, the subgroup of squares in $\mathbb{Z}_n^*$ has order $p'q'$ and $\hat{g}$ is chosen to be a generator. It lies inside the subgroup of numbers of Jacobi symbol 1 which has order $2p'q'$ and is generated by $\hat{g}$ and -1.

The first protocol has as public input $\hat{g}, \hat{h} \in \mathbb{Z}_n^*$ and we assume $\hat{h}$ Jacobi symbol 1, this can be easily checked by the verifier. Recall that an honest prover knows r such that $\hat{h} = \hat{g}^r$. The protocol goes as follows:

Protocol Composite order discrete log.

1. P chooses $s \in_{\mathsf{R}} \{0,1\}^{3\lambda}$ and sends $a = \hat{g}^s$ to V (interpreting s as a binary number).
2. V sets $e \in_{\mathsf{R}} \mathbb{Z}_n$ and sends it to P.
3. P returns $z = s + er$ to V, and V checks that $\hat{g}^z = a\hat{h}^e \bmod n$.

This protocol is easily seen to be complete and statistical honest verifier zero-knowledge

(note that s is chosen to be exponentially larger than er so z is statistically close to a random 3λ bit number. Soundness is more tricky. *Assuming* that $\hat{h}$ is in the group generated by $\hat{g}$, then the protocol is a proof of knowledge of r, under the strong RSA assumption modulo n, as shown by Fujisaki and Okamoto (Fujisaki and Okamoto, 1997). Now, since $\hat{h}$ has Jacobi symbol 1, either $\hat{h}$ or $-\hat{h}$ is in the subgroup, so the protocol proves that P knows the discrete log of $\hat{h}$ or $-\hat{h}$.

For the second protocol we have public input $c = (n+1)^k v^n \bmod n^2$ and $\beta = \gamma^k$ for some $\gamma \in G$. The prover knows k and v. The protocol goes as follows:

Protocol Plaintext and discrete log knowledge.

1. P chooses $s \in_{\mathsf{R}} \mathbb{Z}_n, u \in_{\mathsf{R}} \mathbb{Z}_n^*$ and sends $a = (n+1)^s u^n \bmod n^2$, $\alpha = \gamma^s$ to V.
2. V sets $e \in_{\mathsf{R}} \mathbb{Z}_n$ and sends it to P.
3. P returns $z = s + er \bmod n, w = uv^e \bmod n$, and V checks that $(n+1)^z w^n = ac^e \bmod n^2$ and that $\gamma^z = \alpha\beta^e$.

It is trivial to prove this protocol complete, sound and honest-verifier zero-knowledge.

To to the proof π_{valid} in the AVPKE scheme, we run the Composite order discrete log and the Plaintext and discrete knowledge protocols where in the latter we omit γ, β, α.

Protocols for the PVDVS scheme. The first protocol works with the well-known Pedersen commitments, where a commitment to x with randomness r is of form $g^x h^r$ where $g, h \in G$. These commitments are perfectly hiding and computationally binding if discrete log in G is hard. The protocol shows that two Pedersen commitments contain the same value. The public input is $c_1 = g_1^x h_1^{r_1}, c_2 = g_2^x h_2^{r_2}$. The protocol works as follows:

Protocol Commitment equality.

1. P sets $y, s_1, s_2 \in_{\mathsf{R}} \mathbb{Z}_n$, and sends $a_1 = g_1^y h_1^{s_1}, a_2 = g_2^y h_2^{s_2}$ to V

2. V sets $e \in_{\mathsf{R}} \mathbb{Z}_n$ and sends to P.

3. P sends $z = y + ex \bmod n$ and $u_1 = s_1 + er_1 \bmod n, u_2 = s_2 + er_2 \bmod n$ to V.

4. V checks that $g_1^z h_1^{u_1} = a_1 c_1^e$ and that $g_2^z h_2^{u_2} = a_2 c_2^e$.

The NIZK in VerSigSim uses this protocol.

For the NIZK in PubSigSim, recall that we have $\mathsf{H}(m,t)^k$, $E_{fake}(k', r, b, b', v)$ as input, and we want to demonstrate that $k \neq k'$. The prover includes $\mathsf{H}(m,t)^{k'}$ in the proof, and runs the Plaintext and discrete log knowledge protocol to demonstrate that the exponent k' is also present in the encryption. For this, we consider only the last part of the ciphertext, which is a Paillier encryption of k', as this part already uniquely determines k'. The prover can use k' and $(-1)^{b'}v$ as witness. Finally, the verifier checks that $\mathsf{H}(m,t)^{k'} \neq \mathsf{H}(m,t)^k$. Soundness and completeness of this should be clear. For zero-knowledge, the simulator would produce $\mathsf{H}(m,t)^{k''}$ for random k'', and simulate the Plaintext and discrete log knowledge protocol. Of course the statement in question is now false, but by privacy of the encryption scheme, the simulated public data is indistinguishable from the case where the statement is true, so the simulator must still produce an indistinguishable transcript.

Finally, for the proof of real signature in Sign, we have to show that the ciphertext encrypting k is completely well formed so that V will accept, and that this exponent is used in the MAC. For this we use the following protocol. The public input is $\delta = \gamma^k$ for a publicly known $\gamma \in G$ (namely a hash-value), $pk_{\mathsf{S}} = \hat{g}^{sk_{\mathsf{S}}} \bmod n$ and

$$C = E_{sk_{\mathsf{S}}, pk_{\mathsf{V}}}(k, r, b_1, b_2) = ((-1)^{b_1} \hat{g}^r \bmod n,\ (n+1)^k ((-1)^{b_2} \beta_1^{sk_{\mathsf{S}}} \beta_2^r \bmod n)^n \bmod n^2),$$

for publicly known $\beta_1, \beta_2 \in Z_{n^2}^*$, which in our context come from V's public key pk_{V}. The prover's secret witness is $sk_{\mathsf{S}}, k, r, b_1, b_2$.

Protocol Valid signature.

1. P chooses $s \in \mathbb{Z}_n$, $x, y \in \{0,1\}^{3\lambda}$, $c_1, c_2 \in \{0,1\}$. He then sends to V: $a =$

$E_{sk_{\mathsf{S}}+x,pk_{\mathsf{V}}}(s, y, c_1, c_2)$, $\xi = \hat{g}^x \bmod n$ and $\omega = \gamma^s$.

2. V sets $e \in_{\mathsf{R}} \mathbb{Z}_n$ and sends it to P.
3. P returns $z_s = s + er \bmod n$, $z_x = x + esk_{\mathsf{S}}$, $z_y = y + er$, $d_1 = (c_1 + eb_1) \bmod 2$, $d_2 = (c_2 + eb_2) \bmod 2$. V checks that $E_{z_x,pk_{\mathsf{V}}}(z_s, z_y, d_1, d_2) = aC^e$, that $\hat{g}^{z_x} = \xi pk_{\mathsf{S}}^e$ and that $\gamma^{z_s} = \omega\delta^e$.

It is tedious but straightforward to check completeness. For (statistical) honest verifier ZK we use that x, y are both exponentially lager than esk_{S} and er, respectively. For soundness, assume we get acceptable answers $(z_s, z_x, z_y, d_1, d_2)$ and $(z_s', z_x', z_y', d_1', d_2')$ to challenges e, e'. This will imply that we get equations:

$$E_{z_x - z_x', pk_{\mathsf{V}}}(z_s - z_s', z_y - z_y', (d_1 - d_1') \bmod 2, (d_2 - d_2') \bmod 2) = C^{e-e'}$$

$$\hat{g}^{z_x - z_x'} = pk_{\mathsf{S}}^{e-e'}, \quad \gamma^{z_s - z_s'} = \delta^{e-e'}$$

Now, note that the protocol is in fact (implicitly) using the Composite order discrete log protocol to prove knowledge of the discrete logs of plus or minus the numbers $pk_{\mathsf{S}} = \hat{g}^{sk_{\mathsf{S}}} \bmod n$ and $(-1)^{b_1}\hat{g}^r \bmod n$, where the latter occurs inside the ciphertext.

The soundness proof for that protocol from (Fujisaki and Okamoto, 1997) argues if we get acceptable answers to two challenges e, e', then under the strong RSA assumption, it must be that $e - e'$ divides the difference between the answers. So under this assumption we get that $e - e'$ divides $z_x - z_x'$ and $z_y - z_y'$. It is now straightforward to extract a valid witness, essentially by dividing by $e - e'$ in the exponent on both sides.

Chapter 6

Fuzzy Password Authenticated Key Exchange

The contents of this section are a collaboration with Pierre-Alain Dupont, Julia Hesse, David Pointcheval, and Leonid Reyzin (Dupont et al., 2018). My primary contribution is the construction based on Yao's garbled circuits.

6.1 Introduction

Consider key agreement by two parties who start out knowing a common secret (which we refer to as "pass-string", a generalization of "password"). These parties may face several complications: (1) the pass-string may come from a non-uniform, low-entropy distribution, and (2) the two parties' copies of the pass-string may have some noise, and thus not match exactly. The use of such pass-strings for security has been extensively studied; examples include biometrics and other human-generated data (Daugman, 2004; Zviran and Haga, 1993; Brostoff and Sasse, 2000; Ellison et al., 2000; Mayrhofer and Gellersen, 2009; Monrose et al., 2002; Kolesnikov and Rackoff, 2008), physically unclonable functions (PUFs) (Pappu et al., 2002; Gassend et al., 2002; Tuyls et al., 2006; Suh and Devadas, 2007; Yu and Devadas, 2010), noisy channels (Wyner, 1975), quantum information (Bennett et al., 1988), and sensor readings of a common environment (Han et al., 2017; Han et al., 2018).

The Noiseless Case. When the starting secret is not noisy (i.e., the same for both parties), existing approaches work quite well. The case of low-entropy secrets is covered by *password-authenticated key exchange* (PAKE), in a long line of work the first formal models for which were introduced by Bellare *et al.* (Bellare et al., 2000) and Boyko *et al.* (Boyko et al., 2000). A PAKE protocol allows two parties to agree on a shared high-entropy key if and only if they hold the same short password. Even though the password may have low entropy, PAKE ensures that off-line dictionary attacks are impossible. Roughly speaking, an adversary has to participate in one on-line interaction for every attempted guess at the password. Because key agreement is not usually the final goal, PAKE protocols need to be securely composable with whatever protocols (such as authenticated encryption) use the output key. This composability has been achieved by universally composable (UC) PAKE defined by Canetti *et al.* (Canetti et al., 2005) and implemented in several follow-up works.

In the case of high-entropy secrets, off-line dictionary attacks are not a concern, which enables more efficient protocols. If the adversary is passive, randomness extractors (Nisan and Zuckerman, 1993) do the job. The case of active adversaries is covered by the literature on so-called robust extractors defined by Boyen *et al.* (Boyen et al., 2005) and, more generally, by many papers on privacy amplification protocols secure against active adversaries, starting with the work of Maurer (Maurer, 1997). Composability for these protocols is less studied; in particular, most protocols leak information about the pass-string itself, in which case reusing the pass-string over multiple protocol executions may present problems (Boyen, 2004) (with the exception of (Canetti et al., 2016)).

The Noisy Case. When the pass-string is noisy (i.e., the two parties have slightly different versions of it), this problem has been studied only for the case of high-entropy pass-strings. A long series of works on information-reconciliation protocols (started

by Bennett *et al.* (Bennett et al., 1988)) and their one-message variants called fuzzy extractors (defined by Dodis *et al.* (Dodis et al., 2008), further enhanced for active security starting by Renner *et al.* (Renner and Wolf, 2004)) achieves key agreement when the pass-string has a lot of entropy and not too much noise. Unfortunately, these approaches do not extend to the low-entropy setting and are not designed to prevent off-line dictionary attacks.

Constructions for the noisy case depend on the specific noise model. The case of binary Hamming distance — when the n pass-string bits held by the two parties are the same at all but δ locations — is the best studied. Most existing constructions require, at a minimum, that the pass-string should have at least δ bits of entropy. This requirement rules out using most kinds of biometric data as the pass-string—for example, estimates of entropy for iris scans (transformed into binary strings via wavelet transforms and projections) are considerably lower than the amount of errors that need to be tolerated (Blanton and Hudelson, 2009, Section 5). Even the PAKE-based construction of Boyen *et al.* (Boyen et al., 2005) suffers from the same problem.

One notable exception is the construction of Canetti *et al.* (Canetti et al., 2016), which does not have such a requirement, but places other stringent limitations on the probability distribution of pass-strings. In particular, because it is a one-message protocol, it cannot be secure against off-line dictionary attacks.

6.1.1 Our Contributions

We provide definitions and constant-round protocols for key agreement from noisy pass-strings that:

- Resist off-line dictionary attacks and thus can handle low-entropy pass-strings,
- Can handle a variety of noise types and have high error-tolerance, and
- Have well specified composition properties via the UC framework (Canetti, 2001).

Instead of imposing entropy requirements or other requirements on the distribution of pass-strings, our protocols are secure as long as the adversary cannot guess a pass-string value that is sufficiently close. There is no requirement, for example, that the amount of pass-string entropy is greater than the number of errors; in fact, one of our protocols is suitable for iris scans. Moreover, our protocols prevent off-line attacks, so each adversarial attempt to get close to the correct pass-string requires an on-line interaction by the adversary. Thus, for example, our protocols can be meaningfully run with pass-strings whose entropy is only 30 bits—something not possible with any prior protocols for the noisy case.

New Models. Our security model is in the Universal Composability (UC) Framework of Canetti (Canetti, 2001). The advantage of this framework is that it comes with a composability theorem that ensures that the protocol stays secure even when run in arbitrary environments, including arbitrary parallel executions. Composability is particularly important for key agreement protocols, because key agreement is rarely the ultimate goal. The agreed-upon key is typically used for some subsequent protocol—for example, to instantiate a secure channel. Further, this framework allows to us to give a definition that is indifferent to how the initial pass-strings are generated. We have no entropy requirements or constraints on the pass-string distribution; rather, security is guaranteed as long as the adversary's input to the protocol is far enough from the correct pass-string.

As a starting point, we use the definition of UC security for PAKE from Canetti *et al.* (Canetti et al., 2005). The PAKE ideal functionality is defined as follows: the secret pass-strings (called "passwords" in PAKE) of the two parties are the inputs to the functionality, and two random keys, which are equal if and only if the two inputs are equal, are the outputs. The main change we make to PAKE is enhancing the functionality to give equal keys even if the two inputs are not equal, as long as

they are close enough. We also relax the security requirement to allow one party to find out some information about the other party's input—perhaps even the entire input—if the two inputs are close. This relaxation makes sense in our application: if the two parties are honest, then the differences between their inputs are a problem rather than a feature, and we would not mind if the inputs were in fact the same. The benefit of this relaxation is that it permits us to construct more efficient protocols. (We also make a few other minor changes which will be described in Section 6.2.) We call our new UC functionality "Fuzzy Password-Authenticated Key Exchange" or fPAKE.

New Protocols. The only prior PAKE-based protocol for the noisy setting by Boyen *et al.* (Boyen et al., 2005), although more efficient than ours, does not satisfy our goal. In particular, it is not composable, because it reveals information about the secret pass-strings (we demonstrate this formally in Section Section 6.13. Because some information about the pass-strings is unconditionally revealed, high-entropy pass-strings are required. Thus, in order to realize our definition for arbitrary low-entropy pass-strings, we need to construct new protocols.

Realizing our fPAKE definition is easy using general two-party computation techniques for protocols with malicious adversaries and without authenticated channels (Barak et al., 2005). However, we develop protocols that are considerably more efficient: our definitional relaxation allows us to build protocols that achieve security against malicious adversaries but cost just a little more than the generic two-party computation protocols that achieve security only against honest-but-curious adversaries (i.e., adversaries who do not deviate from the protocol, but merely try to infer information they are not supposed to know).

Our first construction uses Yao's garbled circuits (Yao, 1986; Bellare et al., 2012) and oblivious transfer (see Chou and Orlandi (Chou and Orlandi, 2015) and references

therein). The use of these techniques is standard in two-party computation. However, by themselves they give protocols secure only against honest-but-curious adversaries. In order to prevent malicious behavior of the players, one usually applies the cut-and-choose technique (Lindell and Pinkas, 2011), which is quite costly: to achieve an error probability of $2^{-\lambda}$, the number of circuits that need to be garbled increases by a factor of λ, and the number of oblivious transfers that need to be performed increases by a factor of $\lambda/2$. We show that for our special case, to achieve malicious security, it suffices to repeat the honest-but-curious protocol twice (once in each direction), incurring only a factor of 2 overhead over the semi-honest case. [1] Mohassel *et al.* (Mohassel and Franklin, 2006) and Huang *et al.* (Huang et al., 2012) suggest a similar technique (known as "dual execution"), but at the cost of leaking a bit of the adversary's choice to the adversary. In contrast, our construction leaks nothing to the adversary at all (as long as the pass-strings are not close). This construction works regardless of what it means for the two inputs to be "close," as long as the question of closeness can be evaluated by an efficient circuit.

Our second construction is for the Hamming case: the two n-character pass-strings have low Hamming distance if not too many characters of one party's pass-string are different from the corresponding characters of the other's pass-string. The two parties execute a PAKE protocol for each position in the string, obtaining n values each that agree or disagree depending on whether the characters of the pass-string agree or disagree in the corresponding positions. It is important that at this stage, agreement or disagreement at individual positions remains unknown to everyone; we therefore make use of a special variant of PAKE which we call *implicit-only PAKE*

[1]Gasti *et al.* (Gasti et al., 2016) similarly use Yao's garbled circuits for continuous biometric user authentication on a smartphone. Our approach can eliminate the third party in their application, at the cost of requiring two garbled circuits instead of one. As far as we know, ours is the first use of garbled circuits in the two-party fully malicious setting without calling on an expensive transformation.

(we give a formal UC security definition of implicit-only PAKE and show that it is realized by the PAKE protocol of Bellovin and Merritt (Bellovin and Merritt, 1992) and Abdalla *et al.* (Abdalla et al., 2008)). This first step upgrades Hamming distance over a potentially small alphabet to Hamming distance over an exponentially large alphabet. We then secret-share the ultimate output key into n shares using a robust secret sharing scheme, and encrypt each share using the output of the corresponding PAKE protocol.

The second construction is more efficient than the first in the number of rounds, communication, and computation. However, it works only for Hamming distance. Moreover, it has an intrinsic gap between functionality and security: if the honest parties need to be within distance δ to agree, then the adversary may break security by guessing a secret within distance 2δ. See Figure 6·11 for a comparison between the two constructions.

The advantages of our protocols are similar to the advantages of universally composable PAKE: They provide composability, protection against off-line attacks, the ability to use low-entropy secret inputs, and handle any distribution of those inputs. And, of course, because we construct *fuzzy* PAKE, our protocols can handle noisy inputs—including many types of noisy inputs that could not be handled before. Our first protocol can handle any type of noise as long as the notion of "closeness" can be efficiently computed, whereas most prior work was for Hamming distance only. However, these advantages come at the price of efficiency. Our protocols require 2–5 rounds of interaction, as opposed to many single-message protocols in the literature (Dodis et al., 2012; Canetti et al., 2016; Woodage et al., 2017). They are also more computationally demanding than most existing protocols for the noisy case, requiring one public-key operation per input character. We emphasize, however, that our protocols are much less computationally demanding than the protocols based

on general two-party computation, as already discussed above, or general-purpose obfuscation, as discussed in (Bitansky et al., 2014, Section 4.3.4).

6.2 Security Model

We now present a security definition for fuzzy password-authenticated key exchange (fPAKE). We adapt the definition of PAKE from Canetti *et al.* (Canetti et al., 2005) to work for pass-strings (a generalization of "passwords") that are similar, but not necessarily equal. Our definition uses measures of the distance $d(\mathsf{pw}, \mathsf{pw}')$ between pass-strings $\mathsf{pw}, \mathsf{pw}' \in \mathbb{F}_p^n$. In Section 6.3.3 and Section 6.4, Hamming distance is used, but in the generic construction of Section 6.3, any (efficiently computable) other notion of distance can be used instead. We say that pw and pw′ are "similar enough" if $d(\mathsf{pw}, \mathsf{pw}') \leq \delta$ for a distance notion d and a threshold δ that is hard-coded into the functionality.

Parties first engage our functionality (described in Figure 6·1) by making `NewSession` queries, which include their pass-strings. Once both parties have made `NewSession` queries, the simulator can make `NewKey` queries on behalf of the parties, prompting the functionality to release an appropriate session key to the party in question. In an execution in which the adversary does not meddle, both session keys will be random: they will match if the pass-strings are "similar enough", and be independent otherwise.

Modeling Adversarial Capabilities To model the possibility of dictionary attacks, the functionality allows the adversary to make one pass-string guess against each player ($\mathcal{P}_0$ and $\mathcal{P}_1$). In the real world, if the adversary succeeds in guessing (a pass-string similar enough to) party $\mathcal{P}_i$'s pass-string, it can often choose (or at least bias) the session key computed by $\mathcal{P}_i$. To model this, the functionality then allows the adversary to set the session key for $\mathcal{P}_i$.

As usual in security notions for key exchange, the adversary also sets the session keys for corrupted players. In the definition of Canetti *et al.* (Canetti et al., 2005), the adversary additionally sets $\mathcal{P}_i$'s key if $\mathcal{P}_{1-i}$ is corrupted. However, contrary to the original definition, we do not allow the adversary to set $\mathcal{P}_i$'s key if $\mathcal{P}_{1-i}$ is corrupted but did not guess $\mathcal{P}_i$'s pass-string. We make this change in order to protect an honest $\mathcal{P}_i$ from, for instance, revealing sensitive information to an adversary who did not successfully guess her pass-string, but did corrupt her partner.

Roles There are two categories of fPAKE protocols: *symmetric protocols* in which the two parties execute the same code, and *asymmetric protocols* in which the two parties execute different code. Frequently in asymmetric protocols, one party can be seen as the "sender" who initiates the protocol, and the other can be seen as the "receiver" who responds.[2]

In our ideal functionality, each party includes a `role` tag in her `NewSession` query; one party should identify herself as the sender (denoted as `role` = `sender`), while the other should identify herself as the receiver (`role` = `receiver`). The functionality simply forwards these role tags to the simulator; the roles do not affect any of the functinality's decisions.

In the case of symmetric protocols, the `role` tags are unnecessary, since a sender and a receiver execute the same code. In the case of asymmetric protocols, the simulator needs the `role` tags in order to determine which code to execute. It might look strange that the functionality ignores these `role` tags once it forwards them to the simulator; it might seem that, in the case of an asymmetric protocol, the functionality should only proceed if one of the roles provided is `sender` and the other `receiver`. However, in such a situation, the simulator can trigger the desired behavior — an

[2]To reflect the fact that, even in symmetric protocols, one party likely requests that the other engage in key exchange with her, such a request message can be pre-pended to any symmetric protocol.

abort — simply by never issuing a `NewKey` query.[3]

Notes Another minor change we make is considering only two parties — $\mathcal{P}_0$ and $\mathcal{P}_1$ — in the functionality, instead of considering arbitrarily many parties and enforcing that only two of them engage the functionality. This is because universal composability takes care of ensuring that a two-party functionality remains secure in a multi-party world.

As in the definition of Canetti *et al.* (Canetti et al., 2005), we consider only static corruptions in the standard corruption model of Canetti (Canetti, 2001). Also as in their definition, we chose not to provide the players with confirmation that key agreement was successful. The players might obtain such confirmation from subsequent use of the key.

Leakage By default, in the fPAKE functionality the `TestPwd` interface provides the adversary with one bit of information — whether the pass-string guess was correct or not. This definition can be strengthened by providing the adversary with no information at all, as in implicit-only PAKE ($\mathcal{F}_{\mathsf{iPAKE}}$, Figure 6·8), or weakened by providing the adversary with extra information when the adversary's guess is close enough.

To capture the diversity of possibilities, we introduce a more general `TestPwd` interface, described in Figure 6·2. It includes three leakage functions that we will instantiate in different ways below—L_c if the guess is close enough to succeed, L_f if it is too far. Moreover, a third leakage function—L_m for medium distance—allows the adversary to get some information even if the adversary's guess is only somewhat close (closer than some parameter $\gamma \geq \delta$), but not close enough for successful key

[3] An asymmetric protocol where the parties do *not* abort when both are executing the same role's code (but the resulting keys are not distributed as they should be) cannot securely instantiate our functionality.

The functionality fPAKE is parameterized by a security parameter λ and tolerances $\delta \leq \gamma$. It interacts with an adversary $\mathcal{S}$ and two parties $\mathcal{P}_0$ and $\mathcal{P}_1$ via the following queries:

- **Upon receiving a query ($\texttt{NewSession}, sid, \mathsf{pw}_i, \texttt{role}$) from party $\mathcal{P}_i$,** where pw_i is a password and $\texttt{role} = \texttt{sender}$ implies that $\mathcal{P}_i$ wishes to initiate a key exchange, while $\texttt{role} = \texttt{receiver}$ implies that $\mathcal{P}_i$ wishes to respond:
  - Send ($\texttt{NewSession}, sid, \mathcal{P}_i, \texttt{role}$) to $\mathcal{S}$;
  - If one of the following is true, record $(\mathcal{P}_i, \mathsf{pw}_i)$ and mark this record $\texttt{fresh}$:
    * This is the first $\texttt{NewSession}$ query
    * This is the second $\texttt{NewSession}$ query and there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$
- **Upon receiving a query ($\texttt{TestPwd}, sid, \mathcal{P}_i, \mathsf{pw}'_i$) from the adversary $\mathcal{S}$:**
  If there is a $\texttt{fresh}$ record $(\mathcal{P}_i, \mathsf{pw}_i)$, then set $d \leftarrow d(\mathsf{pw}_i, \mathsf{pw}'_i)$ and do:
  - If $d \leq \delta$, mark the record $\texttt{compromised}$ and reply to $\mathcal{S}$ with "correct guess";
  - If $d > \delta$, mark the record $\texttt{interrupted}$ and reply to $\mathcal{S}$ with "wrong guess".
- **Upon receiving a query ($\texttt{NewKey}, sid, \mathcal{P}_i, \mathsf{sk}$) from the adversary $\mathcal{S}$:**
  If there is no record of the form $(\mathcal{P}_i, \mathsf{pw}_i)$, or if this is not the first $\texttt{NewKey}$ query for $\mathcal{P}_i$, then ignore this query. Otherwise:
  - If at least one of the following is true, then output (sid, sk) to player $\mathcal{P}_i$:
    * The record is $\texttt{compromised}$
    * $\mathcal{P}_i$ is corrupted
    * The record is $\texttt{fresh}$, $\mathcal{P}_{1-i}$ is corrupted, and there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $d(\mathsf{pw}_i, \mathsf{pw}_{1-i}) \leq \delta$
  - If this record is $\texttt{fresh}$, both parties are honest, there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $d(\mathsf{pw}_i, \mathsf{pw}_{1-i}) \leq \delta$, a key sk' was sent to $\mathcal{P}_{1-i}$, and $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ was $\texttt{fresh}$ at the time, then output (sid, sk') to $\mathcal{P}_i$;
  - In any other case, pick a new random key sk' of length λ and send (sid, sk') to $\mathcal{P}_i$.
  - Mark the record $(\mathcal{P}_i, \mathsf{pw}_i)$ as $\texttt{completed}$.

Figure 6·1: Ideal Functionality fPAKE

agreement. We thus decouple the distance needed for functionality from the (possibly larger) distance needed to guarantee security; the smaller the gap between these two distances, the better, of course.

Below, we list the specific leakage functions L_c, L_m and L_f that we consider in this work, in order of decreasing strength (or increasing leakage):

1. The strongest option is to provide no feedback at all to the adversary. We define fPAKE^N to be the functionality described in Figure 6·1, except that `TestPwd` is from Figure 6·2 with

$$L_c^N(\mathsf{pw}_i, \mathsf{pw}'_i) = L_m^N(\mathsf{pw}_i, \mathsf{pw}'_i) = L_f^N(\mathsf{pw}_i, \mathsf{pw}'_i) = \bot\,.$$

2. The basic functionality fPAKE, described in Figure 6·1, leaks the correctness of

- **Upon receiving a query** $(\texttt{TestPwd}, \boldsymbol{sid}, \boldsymbol{\mathcal{P}_i}, \mathbf{pw}'_i)$ **from the adversary** $\boldsymbol{\mathcal{S}}$**:**
  If there is a `fresh` record $(\mathcal{P}_i, \mathsf{pw}_i)$, then set $d \leftarrow d(\mathsf{pw}_i, \mathsf{pw}'_i)$ and do:
  - If $d \leq \delta$, mark the record `compromised` and reply to $\mathcal{S}$ with $L_c(\mathsf{pw}_i, \mathsf{pw}'_i)$;
  - If $\delta < d \leq \gamma$, mark the record `compromised` and reply to $\mathcal{S}$ with $L_m(\mathsf{pw}_i, \mathsf{pw}'_i)$;
  - If $\gamma < d$, mark the record `interrupted` and reply to $\mathcal{S}$ with $L_f(\mathsf{pw}_i, \mathsf{pw}'_i)$.

Figure 6·2: A Modified `TestPwd` Interface to Allow for Different Leakage

the adversary's guess. That is, in the language of Figure 6·2,

$$L_c(\mathsf{pw}_i, \mathsf{pw}'_i) = \text{“correct guess”},$$

$$\text{and} \qquad L_m(\mathsf{pw}_i, \mathsf{pw}'_i) = L_f(\mathsf{pw}_i, \mathsf{pw}'_i) = \text{“wrong guess”}.$$

The classical PAKE functionality from (Canetti et al., 2005) has such a leakage.

3. Assume the two pass-strings are strings of length n over some finite alphabet, with the jth character of the string pw denoted by $\mathsf{pw}[j]$. We define fPAKE^M to be the functionality described in Figure 6·1, except that `TestPwd` is from Figure 6·2, with L_c and L_m that leak the indices at which the guessed pass-string differs from the actual one when the guess is close enough (we will call this leakage the *mask* of the pass-strings). That is,

$$L_c^M(\mathsf{pw}_i, \mathsf{pw}'_i) = (\{j \text{ s.t. } \mathsf{pw}_i[j] = \mathsf{pw}'_i[j]\}, \text{“correct guess”}),$$

$$L_m^M(\mathsf{pw}_i, \mathsf{pw}'_i) = (\{j \text{ s.t. } \mathsf{pw}_i[j] = \mathsf{pw}'_i[j]\}, \text{“wrong guess”})$$

$$\text{and} \qquad L_f^M(\mathsf{pw}_i, \mathsf{pw}'_i) = \text{“wrong guess”}.$$

4. The weakest definition — or the strongest leakage — reveals the entire actual pass-string to the adversary if the pass-string guess is close enough. We define

fPAKE^P to be the functionality described in Figure 6·1, except that `TestPwd` is from Figure 6·2, with

$$L_c^P(\mathsf{pw}_i, \mathsf{pw}'_i) = L_m^P(\mathsf{pw}_i, \mathsf{pw}'_i) = \mathsf{pw}_i \text{ and } L_f^P(\mathsf{pw}_i, \mathsf{pw}'_i) = \text{“wrong guess”}.$$

Here, L_c^P and L_m^P do not need to include “correct guess” and “wrong guess”, respectively, because this is information that can be easily derived from pw_i itself.

The first two functionalities are the strongest, but there are no known constructions that realize them, other than through generic two-party computation secure against malicious adversaries, which is an inefficient solution. The last two functionalities, though weaker, still provide meaningful security, especially when $\gamma = \delta$. Intuitively, this is because strong leakage only occurs when an adversary guesses a “close” pass-string, which enables him to authenticate as though he knows the real pass-string anyway.

In Section 6.3, we present a construction satisfying fPAKE^P for any efficiently computable notion of distance, with $\gamma = \delta$ (which is the best possible). We present a construction for Hamming distance satisfying fPAKE^M in Section 6.4, with $\gamma = 2\delta$.

6.3 General Construction Using Garbled Circuits

In this section, we describe a protocol realizing fPAKE^P that uses Yao's garbled circuits (Yao, 1986). We briefly introduce this primitive in Sec. 6.3.1 and refer to Yakoubov (Yakoubov, 2017) for a more thorough introduction.

The Yao's garbled circuit-based fPAKE construction has two advantages:

1. It is more flexible than other approaches; any notion of distance that can be efficiently computed by a circuit can be used. In Section 6.3.3, we describe a suitable circuit for Hamming distance. The total size of this circuit is $O(n)$,

where n is the length of the pass-strings used. Edit distance is slightly less efficient, and uses a circuit whose total size is $O(n^2)$.

2. There is no gap between the distances required for functionality and security — that is, there is no leakage about the pass-strings used unless they are similar enough to agree on a key. In other words, $\delta = \gamma$.

Informally, the construction involves the garbled evaluation of a circuit that takes in two pass-strings as input, and computes whether their distance is less than δ. Because Yao's garbled circuits are only secure against semi-honest garblers, we cannot simply have one party do the garbling and the other party do the evaluation. A malicious garbler could provide a garbling of the wrong function — maybe even a constant function — which would result in successful key agreement even if the two pass-strings are very different. However, as suggested by Mohassel *et al.* (Mohassel and Franklin, 2006) and Huang *et al.* (Huang et al., 2012), since a malicious evaluator (unlike a malicious garbler) cannot compromise the computation, by performing the protocol twice with each party playing each role once, we can protect against malicious behavior. They call this the *dual execution* protocol.

The dual execution protocol has the downside of allowing the adversary to specify and receive a single additional bit of leakage. It is important to note that because of this, dual execution cannot directly be used to instantiate fPAKE, because a single bit of leakage can be too much when the entropy of the pass-strings is low to begin with — a few adversarial attempts will uncover the entire pass-string. Our construction is as efficient as that of Mohassel *et al.* and Huang *et al.*, while guaranteeing no leakage to a malicious adversary in the case that the pass-strings used are not close. We describe how we achieve this in Section 6.3.1.

Due to the symmetric layout of our construction, we skip all `role` tags in this section.

6.3.1 Building Blocks

In Section 6.3.1, we briefly review oblivious transfer. In Section 6.3.1, we review Yao's Garbled Circuits. In Section 6.3.1, we describe in more detail our take on the dual execution protocol, and how we avoid leakage to the adversary when the pass-strings used are dissimilar.

Oblivious Transfer (OT)

Informally, 1-out-of-2 Oblivious Transfer (see Chou and Orlandi (Chou and Orlandi, 2015) and citations therein) enables one party (the sender) to transfer exactly one of two secrets to another party (the receiver). The receiver chooses (by index 0 or 1) which secret she wants. The security of the OT protocol guarantees that the sender does not learn this choice bit, and the receiver does not learn anything about the other secret.

Yao's Garbled Circuits (YGC)

In this section, we give a brief introduction to Yao's garbled circuits (Yao, 1986). We refer to Yakoubov (Yakoubov, 2017) for a more detailed description, as well as a summary of some of the Yao's garbled circuits optimizations (Beaver et al., 1990; Kolesnikov and Schneider, 2008; Pinkas et al., 2009; Kolesnikov et al., 2014; Zahur et al., 2015; Ball et al., 2016). Informally, Yao's garbled circuits are an asymmetric secure two-party computation scheme. They enable two parties with sensitive inputs (in our case, pass-strings) to compute a joint function of their inputs (in our case, an augmented version of similarity) without revealing any additional information about their inputs. One party "garbles" the function they wish to evaluate, and the other evaluates it in its garbled form.

Below, we summarize the garbling scheme formalization of Bellare *et al.* (Bellare et al., 2012), which is a generalization of YGC.

Functionality. A garbling scheme $\mathcal{G}$ consists of a tuple of four polynomial-time algorithms $(\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$:

1. $\mathsf{Gb}(1^\lambda, f) \to (F, e, d)$. The garbling algorithm Gb takes in the security parameter λ and a circuit f, and returns a garbled circuit F, encoding information e, and decoding information d.
2. $\mathsf{En}(e, x) \to X$. The encoding algorithm En takes in the encoding information e and an input x, and returns a garbled input X.
3. $\mathsf{Ev}(F, X) \to Y$. The evaluation algorithm Ev takes in the garbled circuit F and the garbled input X, and returns a garbled output Y.
4. $\mathsf{De}(d, Y) \to y$. The decoding algorithm De takes in the decoding information d and the garbled output Y, and returns the plaintext output y.

A garbling scheme $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ is *projective* if the encoding information e consists of $2n$ *wire labels* (each of which is essentially a random string), where n is the number of input bits. Two wire labels are associated with each bit of the input; one wire label corresponds to the event of that bit being 0, and the other corresponds to the event of that bit being 1. The garbled input includes only the wire labels corresponding to the actual values of the input bits. In projective schemes, in order to give the evaluator the garbled input she needs for evaluation, the garbler can send her all of the wire labels corresponding to the garbler's input. The evaluator can then use OT to retrieve the wire labels corresponding to her own input.

Similarly, we call a garbling scheme *output-projective* if decoding information d consists of two labels for each output bit, one corresponding to each possible value of that bit. The garbling schemes used in this paper are both projective and output-projective.

Correctness. Informally, a garbling scheme $(\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ is *correct* if it always holds that $\mathsf{De}(d, \mathsf{Ev}(F, \mathsf{En}(e, x))) = f(x)$.

Security. Bellare *et al.* (Bellare et al., 2012) describe three security notions for garbling schemes: *obliviousness*, *privacy* and *authenticity.* Informally, a garbling scheme $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ is *oblivious* if a garbled function F and a garbled input X do not reveal anything about the input x. It is *private* if additionally knowing the decoding information d reveals the output y, but does not reveal anything more about the input x. It is *authentic* if an adversary, given F and X, cannot find a garbled output $Y' \neq \mathsf{Ev}(F, X)$ which decodes without error.

In Section 6.7, we define a new property of output-projective garbling schemes called *garbled output randomness.* Informally, it states that even given one of the output labels, the other should be indistinguishable from random.

Malicious Security: A New Take on Dual Execution with Privacy - Correctness Tradeoffs

While Yao's garbled circuits are naturally secure against a malicious evaluator, they have the drawback of being insecure against a malicious garbler. A garbler can "misgarble" the function, either replacing it with a different function entirely or causing an error to occur in an informative way (this is known as "selective failure").

Typically, malicious security is introduced to Yao's garbled circuits by using the cut-and-choose transformation (Lindell and Pinkas, 2015; Lindell, 2013; Huang et al., 2013). To achieve a $2^{-\lambda}$ probability of cheating without detection, the parties need to exchange λ garbled circuits (Lindell, 2013).[4] Some of the garbled circuits are "checked", and the rest of them are evaluated, their outputs checked against one another for consistency. Because of the factor of λ computational overhead, though, cut-and-choose is expensive, and too heavy a tool for fPAKE. Other, more efficient transformations such as LEGO (Nielsen and Orlandi, 2009) and authenticated garbling (Wang et al., 2017) exist as well, but those rely heavily on pre-processing, which

[4]There are techniques (Lindell and Riva, 2014) that improve this number in the amortized case when many computations are done — however, this does not fit our setting.

MF06 HKE12	Correct Output	Computed Output	Privacy
	1	1 OR ‘cheating’	1-bit leakage
	0	0 OR ‘cheating’	1-bit leakage

Our Protocol	Correct Output	Computed Output	Privacy
	1	1 OR 0	1-bit leakage
	0	0	full privacy

Figure 6·3: The Privacy-Correctness Tradeoff of Dual Execution Protocols for Boolean Functions. MF06 stands for (Mohassel and Franklin, 2006); HKE12 stands for (Huang et al., 2012).

cannot be used in fPAKE since it requires advance interaction between the parties.

Mohassel *et al.* (Mohassel and Franklin, 2006) and Huang *et al.* (Huang et al., 2012) suggest an efficient transformation known as “dual execution”: each party plays each role (garbler and evaluator) once, and then the two perform a comparison step on their outputs in a secure fashion. Dual execution incurs only a factor of 2 overhead over semi-honest garbled circuits. However, it does not achieve fully malicious security. It guarantees correctness, but reduces the privacy guarantee by allowing a malicious garbler to learn one bit of information of her choice. Specifically, if a malicious garbler garbles a wrong circuit, she can use the comparison step to learn one bit about the output of this wrong circuit on the other party’s input. This one extra bit of information could be crucially important, violating the privacy of the evaluator’s input in a significant way.

We introduce a tradeoff between correctness and privacy for boolean functions. For one of the two possible outputs (without loss of generality, ‘0’), we restore full privacy at the cost of correctness. The new privacy guarantee is that if the correct output is ‘0’, then a malicious adversary cannot learn anything beyond this output, but if the correct output is ‘1’, then she can learn a single bit of her choice. The new correctness guarantee is that a malicious adversary can cause the computation that should output ‘1’ to output ‘0’ instead, but not the other way around. Our privacy–correctness tradeoff is summarized in Figure 6·3.

The main idea of dual execution is to have the two parties independently evaluate one another's circuits, learn the output values, and compare the output labels using a secure comparison protocol. This comparison step is simply a check for malicious behavior; if comparison fails, then honest party $\mathcal{P}_i$ learns that $\mathcal{P}_{1-i}$ cheated. If the comparison step succeeded, $\mathcal{P}_{1-i}$ might still have cheated — and gleaned an extra bit of information — but $\mathcal{P}_i$ is assured that she has the correct output.

In our construction, however, the parties need not learn the output values before the comparison. Instead, the parties can compare output labels *assuming* an output of '1', and if the comparison fails, the output is determined to be '0'. More formally, let $d_0[0]$, $d_0[1]$ be the two output labels corresponding to $\mathcal{P}_0$'s garbled circuit, and $d_1[0]$, $d_1[1]$ be the two output labels corresponding to $\mathcal{P}_1$'s circuit. Let $Y_1 \in [d_1[0], d_1[1]]$ be the output label learned by $\mathcal{P}_0$ as a result of evaluation, and $Y_0 \in [d_0[0], d_0[1]]$ be the label learned by $\mathcal{P}_1$. The two parties securely compare $(d_0[1], Y_1)$ to $(Y_0, d_1[1])$; if the comparison succeeds, the output is "1".

Whereas in dual execution the comparison step is just a sanity check, here it determines the actual computation output. If the correct output is '1', a cheating $\mathcal{P}_{1-i}$ can still learn one bit of information by mis-garbling her circuit; depending on the output of the mis-garbled circuit, the comparison step will either succeed or fail. If the comparison fails, $\mathcal{P}_i$ will accept an incorrect output of '0', and never be aware that $\mathcal{P}_{1-i}$ cheated. If the correct output is '0', however, there is nothing $\mathcal{P}_{1-i}$ can do to cause the comparison step to succeed, since in order to do this, she would need to use the second output label $d_i[1]$ as an input. Since the true output was '0', and thus $Y_i = d_i[0]$, by the garbled output randomness property of the garbling scheme, $\mathcal{P}_{1-i}$ can't even distinguish $d_i[1]$ from random.

Our privacy–correctness tradeoff is perfect for fPAKE. If the parties' inputs are similar, learning a bit of information about each other's inputs is not problematic,

since arguably the small amount of noise in the inputs is a bug, not a feature. If the parties' inputs are not similar, however, we are guaranteed to have no leakage at all. We pay for the lack of leakage by allowing a malicious party to force an authentication failure even when authentication should succeed, which either party can do anyway simply by providing an incorrect input.

In Section 6.3.2, we describe our Yao's garbled circuit-based fPAKE protocol in detail. Note that in this protocol, we omit the final comparison step; instead, we use the output lables ($(d_0[1], Y_1)$ and $(Y_0, d_1[1])$) to compute the agreed-upon key directly (via XOR).

6.3.2 Construction

Building a fPAKE from YGC and OT is not straightforward, since all constructions of OT assume authenticated channels, and fPAKE (or PAKE) is designed with unauthenticated channels in mind. We therefore follow the framework of Canetti *et al.* (Canetti et al., 2012), who build a UC secure PAKE protocol using OT. We first build our protocol assuming authenticated channels, and then apply the generic transformation of Barak *et al.* (Barak et al., 2005) to adapt it to the unauthenticated channel setting. More formally, we proceed in three steps:

1. First, in Section 6.3.2, we define a randomized fuzzy equality-testing functionality $\mathcal{F}_{\mathsf{RFE}}$, which is analogous to the randomized equality-testing functionality of Canetti *et al.*
2. In Section 6.3.2, we build a protocol that securely realizes $\mathcal{F}_{\mathsf{RFE}}$ in the OT-hybrid model, assuming authenticated channels.
3. In Section 6.3.2, we apply the transformation of Barak *et al.* to our protocol. This results in a protocol that realizes the "split" version of functionality $\mathcal{F}^{P}_{\mathsf{RFE}}$, which we show to be enough to implement to fPAKE^{P}. Split functionalities, which were introduced by Barak *et al.*, adapt functionalities which assume

authenticated channels to an unauthenticated channels setting. The only additional ability an adversary has in a split functionality is the ability to execute the protocol separately with the participating parties.

The Randomized Fuzzy Equality Functionality

Figure 6·4 shows the randomized fuzzy equality functionality $\mathcal{F}^{P}_{\mathsf{RFE}}$, which is essentially what $\mathcal{F}^{P}_{\mathsf{fPAKE}}$ would look like assuming authenticated channels. The primary difference between $\mathcal{F}^{P}_{\mathsf{RFE}}$ and $\mathcal{F}^{P}_{\mathsf{fPAKE}}$ is that the only pass-string guesses allowed by $\mathcal{F}^{P}_{\mathsf{RFE}}$ are the ones actually used as protocol inputs; this limits the adversary to guessing by corrupting one of the participating parties, not through man in the middle attacks. Like in $\mathcal{F}^{P}_{\mathsf{fPAKE}}$, if a pass-string guess is "similar enough", the entire pass-string is leaked. This leakage could be replaced with any other leakage from Section 6.2; $\mathcal{F}_{\mathsf{RFE}}$ would leak the correctness of the guess, $\mathcal{F}^{M}_{\mathsf{RFE}}$ would leak which characters are the same between the two pass-strings, etc.

Note that, unlike the randomized equality functionality in the work of Canetti *et al.* (Canetti et al., 2012), $\mathcal{F}^{P}_{\mathsf{fPAKE}}$ has a `TestPwd` interface. This is because `NewKey` does not return the necessary leakage to an honest user. So, an interface enabling the adversary to retrieve additional information is necessary.

A Randomized Fuzzy Equality Protocol

In Figure 6·5 we introduce a protocol Π_{RFE} that securely realizes $\mathcal{F}^{P}_{\mathsf{RFE}}$ using Yao's garbled circuits. Garbled circuits are secure against a malicious evaluator, but only a semi-honest garbler; however, we obtain security against malicious adversaries by having each party play each role once, as describe in Section 6.3.1. In more detail, both parties $\mathcal{P}_i \in \{\mathcal{P}_0, \mathcal{P}_1\}$ proceed as follows:

1. $\mathcal{P}_i$ garbles the circuit f that takes in two pass-strings pw_0 and pw_1, and returns '1' if $d(\mathsf{pw}_0, \mathsf{pw}_1) \leq \delta$ and '0' otherwise. Section 6.3.3 describes how f can be

The functionality $\mathcal{F}_{\mathsf{RFE}}$ is parameterized by a security parameter λ and a tolerance δ. It interacts with an adversary $\mathcal{S}$ and two parties $\mathcal{P}_0$ and $\mathcal{P}_1$ via the following queries:

- **Upon receiving a query $(\mathtt{NewSession}, sid, \mathsf{pw}_i)$ from party $\mathcal{P}_i \in \{\mathcal{P}_0, \mathcal{P}_1\}$:**
  - Send $(\mathtt{NewSession}, sid, \mathcal{P}_i)$ to $\mathcal{S}$;
  - If this is the first NewSession query, or if this is the second NewSession query and there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$, then record $(\mathcal{P}_i, \mathsf{pw}_i)$.
- **Upon receiving a query $(\mathtt{TestPwd}, sid, \mathcal{P}_i)$ from the adversary $\mathcal{S}$, $\mathcal{P}_i \in \{\mathcal{P}_0, \mathcal{P}_1\}$:**
  If records of the form $(\mathcal{P}_0, \mathsf{pw}_0)$ and $(\mathcal{P}_1, \mathsf{pw}_1)$ do not exist, if $\mathcal{P}_{1-i}$ is not corrupted, or this is not the first TestPwd query for $\mathcal{P}_i$, ignore this query. Otherwise, if $d(\mathsf{pw}_0, \mathsf{pw}_1) \leq \delta$, send pw_i to the adversary $\mathcal{S}$.
- **Upon receiving a query $(\mathtt{NewKey}, sid, \mathcal{P}_i, \mathsf{sk})$ from the adversary $\mathcal{S}$, $\mathcal{P}_i \in \{\mathcal{P}_0, \mathcal{P}_1\}$:**
  If there are no records of the form $(\mathcal{P}_i, \mathsf{pw}_i)$ and $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$, or if this is not the first NewKey query for $\mathcal{P}_i$, then ignore this query. Otherwise:
  - If at least one of the following is true, then output (sid, sk) to party $\mathcal{P}_i$.
    * $\mathcal{P}_i$ is corrupted
    * $\mathcal{P}_{1-i}$ is corrupted and $d(\mathsf{pw}_0, \mathsf{pw}_1) \leq \delta$
  - If both parties are honest, $d(\mathsf{pw}_0, \mathsf{pw}_1) \leq \delta$, and a key k_{1-i} was sent to $\mathcal{P}_{1-i}$, then output (sid, k_{1-i}) to $\mathcal{P}_i$.
  - In any other case, pick a new random key k_i of length λ and send (sid, k_i) to $\mathcal{P}_i$.

Figure 6·4: Ideal Functionality $\mathcal{F}^{P}_{\mathsf{RFE}}$ for Randomized Fuzzy Equality

designed efficiently for Hamming distance. Instead of using the output of f ('0' or '1'), we will use the garbled output, also referred to as an *output label* in an output-projective garbling scheme. The possible output labels are two random strings — one corresponding to a '1' output (we call this label $\mathsf{k}_{i,correct} = d_i[1]$), and one corresponding to a '0' output (we call this label $\mathsf{k}_{i,wrong} = d_i[0]$).

2. $\mathcal{P}_i$ uses OT to retrieve the input labels from $\mathcal{P}_{1-i}$'s garbling that correspond to $\mathcal{P}_i$'s pass-string. (Similarly, $\mathcal{P}_i$ uses OT to send $\mathcal{P}_{1-i}$ the input labels from her own garbling that correspond to $\mathcal{P}_{1-i}$'s pass-string.)
3. $\mathcal{P}_i$ sends $\mathcal{P}_{1-i}$ her garbled circuit, together with the input labels from her gar-

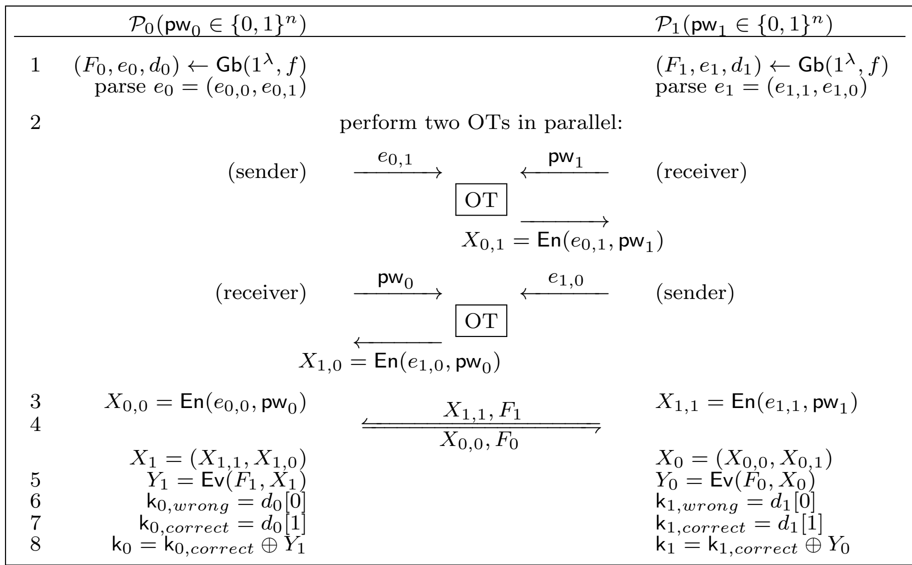


Figure 6·5: A Protocol Π_{RFE} Realizing $\mathcal{F}^{P}_{\mathsf{RFE}}$ using Yao's garbled circuits and an Ideal OT Functionality. If at any point an expected message fails to arrive (or arrives malformed), the parties output a random key. Subscripts are used to indicate who produced the object in question. If a double subscript is present, the second subscript indicates whose data the object is meant for use with. For instance, a double subscript $0,1$ denotes that the object was produced by party $\mathcal{P}_0$ for use with $\mathcal{P}_1$'s data; $e_{0,1}$ is encoding information produced by $\mathcal{P}_0$ to encode $\mathcal{P}_1$'s pass-string. Note that we abuse notation by encoding inputs to a single circuit separately; the input to $\mathcal{P}_0$'s circuit corresponding to pw_0 is encoded by $\mathcal{P}_0$ locally, and the input corresponding to pw_1 is encoded via OT. For any projective garbling scheme, this is not a problem.

bling that correspond to her own pass-string. After this step, $\mathcal{P}_i$ should have $\mathcal{P}_{1-i}$'s garbled circuit and a garbled input consisting of input labels corresponding to the bits of the two pass-strings.

4. $\mathcal{P}_i$ evaluates $\mathcal{P}_{1-i}$'s garbled circuit, and obtains an output label Y_{1-i} (where $Y_{1-i} \in \{\mathsf{k}_{1-i,correct}, \mathsf{k}_{1-i,wrong}\}$) .
5. $\mathcal{P}_i$ outputs $\mathsf{k}_i = \mathsf{k}_{i,correct} \oplus Y_{1-i}$.

The natural question to ask is why Π_{RFE} only realizes $\mathcal{F}^{P}_{\mathsf{RFE}}$, and not a stronger functionality with less leakage. We argue this assuming (without loss of generality) that $\mathcal{P}_1$ is corrupted. Π_{RFE} cannot realize a functionality that leaks less than the full

pass-string pw_0 to $\mathcal{P}_1$ if $d(\mathsf{pw}_0, \mathsf{pw}_1) \leq \delta$; intuitively, this is because if $\mathcal{P}_1$ knows a pass-string pw_1 such that $d(\mathsf{pw}_0, \mathsf{pw}_1) \leq \delta$, $\mathcal{P}_1$ can extract the actual pass-string pw_0, as follows. If $\mathcal{P}_1$ plays the role of OT receiver and garbled circuit evaluator honestly, $\mathcal{P}_0$ and $\mathcal{P}_1$ will agree on $\mathsf{k}_{0,correct}$. $\mathcal{P}_1$ can then mis-garble a circuit that returns $\mathsf{k}_{1,correct}$ if the first bit of pw_0 is 0, and $\mathsf{k}_{1,wrong}$ if the first bit of pw_0 is 1. By testing whether the resulting keys k_0 and k_1 match (which $\mathcal{P}_1$ can do in subsequent protocols where the key is used), $\mathcal{P}_1$ will be able to determine the actual first bit of pw_0. $\mathcal{P}_1$ can then repeat this for the second bit, and so on, extracting the entire pass-string pw_0. Of course, if $\mathcal{P}_1$ does *not* know a sufficiently close pw_1, $\mathcal{P}_1$ will not be able to perform these tests, because the keys will not match no matter what circuit $\mathcal{P}_1$ garbles.

More formally, if $\mathcal{P}_1$ knows a pass-string pw_1 such that $d(\mathsf{pw}_0, \mathsf{pw}_1) \leq \delta$ and carries out the mis-garbling attack described above, then in the real world, the keys produced by $\mathcal{P}_0$ and $\mathcal{P}_1$ either will or will not match based on some predicate p of $\mathcal{P}_1$'s choosing on the two pass-strings pw_0 and pw_1. Therefore, in the ideal world, the keys should also match or not match based on $p(\mathsf{pw}_0, \mathsf{pw}_1)$; otherwise, the environment will be able to distinguish between the two worlds. In order to make that happen, since the simulator does not know the predicate p in question, the simulator must be able to recover the entire pass-string pw_0 (given a sufficiently close pw_1) through the `TestPwd` interface.

Theorem 12. *If* $(\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ *is a projective, output-projective and garbled-output random secure garbling scheme, then protocol* Π_{RFE} *with authenticated channels in the* $\mathcal{F}_{\mathsf{OT}}$*-hybrid model securely realizes* $\mathcal{F}^P_{\mathsf{RFE}}$ *with respect to static corruptions for any threshold* δ*, as long as the pass-string space and notion of distance are such that for any pass-string* pw*, it is easy to compute another pass-string* pw' *such that* $d(\mathsf{pw}, \mathsf{pw}') > \delta$[5].

Sketch. For every efficient adversary $\mathcal{A}$, we describe a simulator $\mathcal{S}_{\mathsf{RFE}}$ such that no efficient environment can distinguish an execution with the real protocol Π_{RFE} and

[5]This is used in the argument of indistinguishability of Games $\mathbf{G}_2$ and $\mathbf{G}_3$ in Appendix 6.8.

$\mathcal{A}$ from an execution with the ideal functionality $\mathcal{F}_{\mathsf{RFE}}^{P}$ and $\mathcal{S}_{\mathsf{RFE}}$. $\mathcal{S}_{\mathsf{RFE}}$ is described in Figure 6·21. We prove indistinguishability in a series of hybrid steps. First, we introduce the ideal functionality as a dummy node. Next, we allow the functionality to choose the parties' keys, and we prove the indistinguishability of this step from the previous using the garbled output randomness property of our garbling scheme (Definition 30, Theorem 15). Next, we simulate an honest party's interaction with another honest party without using their pass-string, and prove the indistinguishability of this step from the previous using the obliviousness property of our garbling scheme. Finally, we simulate an honest party's interaction with a corrupted party without using the honest party's pass-string, and prove the indistinguishability of this step from the previous using the privacy property of our garbling scheme. □

We give a more formal proof of Theorem 12 in Section 6.8.

From Split Randomized Fuzzy Equality to fPAKE

The Randomized Fuzzy Equality (RFE) functionality $\mathcal{F}_{\mathsf{RFE}}^{P}$ assumes authenticated channels, which an fPAKE protocol cannot do. In order to adapt RFE to our setting, we use the split functionality transformation defined by Barak *et al.* (Barak et al., 2005). Barak *et al.* provide a generic transformation from protocols which require authenticated channels to protocols which do not. In the "transformed" protocol, an adversary can engage in two separate instances of the protocol with the sender and receiver, and they will not realize that they are not talking to one another. However, it does guarantee that the adversary cannot do anything beyond this attack. In other words, it provides "session authentication", meaning that each party is guaranteed to carry out the entire protocol with the same partner, but not "entity authentication", meaning that the identity of the partner is not guaranteed.

Barak *et al.* achieve this transformation in three steps. First, the parties generate signing and verification keys, and send one another their verification keys. Next, the parties sign the list of all keys they have received (which, in a two-party protocol, consists of only one key), sign that list, and send both list and signature to all other

The functionality $s\mathcal{F}^{P}_{\mathsf{RFE}}$ is parameterized by a security parameter λ. It interacts with an adversary $\mathcal{S}$ and two parties $\mathcal{P}_0$ and $\mathcal{P}_1$ via the following queries:

- **Initialization**
  - **Upon receiving a query** $(\texttt{Init}, sid)$ **from a party** $\mathcal{P}_i \in \{\mathcal{P}_0, \mathcal{P}_1\}$, send $(\texttt{Init}, sid, \mathcal{P}_i)$ to the adversary $\mathcal{S}$.
  - **Upon receiving a query** $(\texttt{Init}, sid, \mathcal{P}_i, H, sid_H)$ **from the adversary** $\mathcal{S}$:
    - Verify that $H \subseteq \{\mathcal{P}_0, \mathcal{P}_1\}$, that $\mathcal{P}_i \in H$, and that if a previous set H' was recorded, either (1) $H \cap H'$ contains only corrupted parties and $sid_H \neq sid_{H'}$, or (2) $H = H'$ and $sid_H = sid_{H'}$.
    - If verification fails, do nothing.
    - Otherwise, record the pair (H, sid_H) (if it was not already recorded), output $(\texttt{Init}, sid, sid_H)$ to $\mathcal{P}_i$, and locally initialize a new instance of the original RFE functionality $\mathcal{F}_{\mathsf{RFE}}$ denoted $H\mathcal{F}^{P}_{\mathsf{RFE}}$, letting the adversary play the role of $\{\mathcal{P}_0, \mathcal{P}_1\} - H$ in $H\mathcal{F}^{P}_{\mathsf{RFE}}$.
- **RFE**
  - **Upon receiving a query from a party** $\mathcal{P}_i \in \{\mathcal{P}_0, \mathcal{P}_1\}$**,** find the set H such that $\mathcal{P}_i \in H$, and forward the query to $H\mathcal{F}^{P}_{\mathsf{RFE}}$. Otherwise, ignore the query.
  - **Upon receiving a query from the adversary** $\mathcal{S}$ **on behalf of** $\mathcal{P}_i$ **corresponding to set** H**,** if $H\mathcal{F}^{P}_{\mathsf{RFE}}$ is initialized and $\mathcal{P}_i \notin H$, then forward the query to $H\mathcal{F}^{P}_{\mathsf{RFE}}$. Otherwise, ignore the query.

Figure 6·6: Functionality $s\mathcal{F}^{P}_{\mathsf{RFE}}$

parties. Finally, they verify all of the signatures they have received. After this process — called “link initialization” — has been completed, the parties use those public keys they have exchanged to authenticate subsequent communication.

We describe the Randomized Fuzzy Equality Split Functionality in Figure 6·6. It is simplified from Figure 1 in Barak *et al.* (Barak et al., 2005) because we only need to consider two parties and static corruptions.

It turns out that $s\mathcal{F}^{P}_{\mathsf{RFE}}$ is enough to realize $\mathcal{F}^{P}_{\mathsf{fPAKE}}$. In fact, the protocol Π_{RFE} with the split functionality transformation directly realizes $\mathcal{F}^{P}_{\mathsf{fPAKE}}$. In Section 6.9, we prove that this is the case.

6.3.3 An Efficient Circuit f for Hamming Distance

The Hamming distance of two pass-strings $\mathsf{pw}, \mathsf{pw}' \in \mathbb{F}_p^n$ is equal to the number of locations at which the two pass-strings have the same character. More formally,

$$d(\mathsf{pw}, \mathsf{pw}') := |\{j \mid \mathsf{pw}[j] \neq \mathsf{pw}'[j], j \in [n]\}|.$$

We design f for Hamming distance as follows:

1. First, f XORs corresponding (binary) pass-string characters, resulting in a list of bits indicating the (in)equality of those characters.
2. Then, f feeds those bits into a threshold gate, which returns 1 if at least $n-\delta$ of its inputs are 0, and returns 0 otherwise. f returns the output of that threshold gate, which is 1 if and only if at least $n-\delta$ pass-string characters match.

This circuit, illustrated in Figure 6·7, is very efficient to garble; it only requires n ciphertexts. Below, we briefly explain this garbling. Our explanation assumes familiarity with YGC literature (Yakoubov, 2017, and references therein). Briefly, garbled gadget labels (Ball et al., 2016) enable the evaluation of modular addition gates for free (there is no need to include any information in the garbled circuit to enable this addition). However, for a small modulus m, converting the output of that addition to a binary decision requires $m-1$ ciphertexts. We utilize garbled gadgets with a modulus of $n+1$ in our efficient garbling as follows:

1. The input wire labels encode 0 or 1 modulo $n+1$. However, instead of having those input wire labels encode the characters of the two pass-strings directly, they encode the outputs of the comparisons of corresponding characters. If the jth character of $\mathcal{P}_i$'s pass-string is 0, then $\mathcal{P}_i$ puts the 0 label first; however, if the jth character of $\mathcal{P}_i$'s pass-string is 1, then $\mathcal{P}_i$ flips the labels. Then, when $\mathcal{P}_{1-i}$ is using oblivious transfer to retrieve the label corresponding to her jth pass-string character, she will retrieve the 0 label if the two characters are equal,

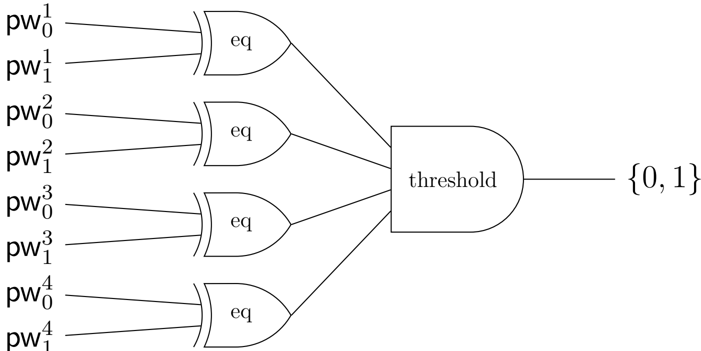


Figure 6·7: The f circuit

and the 1 label otherwise. (Note that this pre-processing on the garbler's side eliminates the need to send $X_{0,0}$ and $X_{1,1}$ in Figure 6·5.)

2. Compute a n-input threshold gate, as illustrated in Figure 6 of Yakoubov (Yakoubov, 2017). This gate returns 0 if the sum of the inputs is above a certain threshold (that is, if at least $n - \delta$ pass-string characters differ), and 1 otherwise. This will require n ciphertexts.

Thus, a garbling of f consists of n ciphertexts. Since fPAKE requires two such garbled circuits (Figure 6·5), $2n$ ciphertexts will be exchanged.

Larger Pass-string Characters. If larger pass-string characters are used, then Step 1 above needs to change to check (in)equality of the larger characters instead of bits. Step 2 will remain the same. There are several ways to perform an (in)equality check on characters in $\mathbb{F}_p$ for $p \geq 2$:

1. Represent each character in terms of bits. Step 1 will then consist of XORing corresponding bits, and taking an OR or the resulting XORs of each character to get negated equality. This will take an additional $n \log(p)$ ciphertexts for every pass-string character.
2. Use garbled gadget labels from the outset. We will require a larger OT (1-out-of-p instead of 1-out-of-2), but nothing else will change.

6.4 Specialized Construction For Hamming Distance

In Section 6.13, we show that it is not straightforward to build a secure fPAKE from primitives that are, by design, well-suited for correcting errors. However, PAKE protocols are appealingly efficient compared to the garbled circuits used in the prior construction. In this section, we ask whether the failed approach can be rescued in an efficient way, and we answer this question in the affirmative.

6.4.1 Building Blocks

Robust Secret Sharing

We recall the definition of a robust secret sharing scheme, slightly simplified for our purposes from Cramer *et al.* (Cramer et al., 2015). For a vector $c \in \mathbb{F}_q^n$ and a set $A \subseteq [n]$, we denote with c_A the projection $\mathbb{F}_q^n \to \mathbb{F}_q^{|A|}$, i.e., the sub-vector $(c_i)_{i \in A}$.

Definition 27. *Let $\lambda \in \mathbb{N}$, q a λ-bit prime, $\mathbb{F}_q$ a finite field and $n, t, m, r \in \mathbb{N}$ with $t < r \leq n$ and $m < r$. An $(n, t, r)_q$ robust secret sharing scheme (RSS) consists of two probabilistic algorithms $\mathsf{Share} : \mathbb{F}_q \to \mathbb{F}_q^n$ and $\mathsf{Reconstruct} : \mathbb{F}_q^n \to \mathbb{F}_q$ with the following properties:*

- t-privacy*: for any $s, s' \in \mathbb{F}_q, A \subset [n]$ with $|A| \leq t$, the projections c_A of $c \xleftarrow{\$} \mathsf{Share}(s)$ and c'_A of $c' \xleftarrow{\$} \mathsf{Share}(s')$ are identically distributed.*
- r-robustness*: for any $s \in \mathbb{F}_q, A \subset [n]$ with $|A| \geq r$, any c output by $\mathsf{Share}(s)$, and any $\tilde{c}$ such that $c_A = \tilde{c}_A$, it holds that $\mathsf{Reconstruct}(\tilde{c}) = s$.*

In other words, an $(n, t, r)_q$-RSS is able to reconstruct the shared secret even if the adversary tampered with up to $n - r$ shares, while each set of t shares is distributed independently of the shared secret s and thus reveals nothing about it. We note that we allow for a gap, i.e., $r \geq t + 1$. Schemes with $r > t + 1$ are called *ramp* RSS.

Definition 28 (Smoothness)**.** *We say that an $(n, t, r)_q$-RSS is*

- m-smooth *if for any $s \in \mathbb{F}_q, A \subset [n]$ with $|A| \leq m$, any c output by $\mathsf{Share}(s)$, and any $\tilde{c}$ such that $\tilde{c}_A = c_A$, $\tilde{c}_{\bar{A}} \xleftarrow{\$} \mathbb{F}_q^{n-|A|}$, for all PPT $\mathcal{A}$ it holds that*

$$|\Pr[1 \leftarrow \mathcal{A}(1^\lambda, \mathsf{Reconstruct}(\tilde{c}))] - \Pr[1 \leftarrow \mathcal{A}(1^\lambda, u)]|$$

is negligible in λ, where the probability is taken over the random coins of $\mathcal{A}$ and $\mathsf{Reconstruct}$ *and* $u \xleftarrow{\$} \mathbb{F}_q$.

- m-smooth on random secrets *if it is m-smooth for randomly chosen $s \xleftarrow{\$} \mathbb{F}_q$ and the probabilities are additionally taken over the coins consumed by this choice.*

Definition 29 (Strong t-privacy)**.** *We say that an $(n, t, r)_q$-RSS has* strong t-privacy*, if for any $s \in \mathbb{F}_q, A \subset [n]$ with $|A| \leq t$, the projection c_A of $c \xleftarrow{\$} \mathsf{Share}(s)$ is distributed uniformly randomly in $\mathbb{F}_q^{|A|}$.*

Note that strong t-privacy implies t-privacy. The opposite does not necessarily hold. (To see that the opposite might not hold, imagine a Share algorithm creating shares that start with "I'm a share!"). Also note that, in case of random errors occuring, as long as there are fewer then t undisturbed shares, a strong t-private scheme actually hides the locations (and with this also the number) of errors.

Linear Codes

A linear q-ary code of length n and rank k is a subspace C with dimension k of the vector space $\mathbb{F}_q^n$. The vectors in C are called codewords. The size of a code is the number of codewords it contains, and is thus equal to q^k. The weight of a word $w \in \mathbb{F}_q^n$ is the number of its non-zero components, and the distance between two words is the Hamming distance between them (equivalently, the weight of their difference). The minimal distance d of a linear code C is the minimum weight of its non-zero codewords, or equivalently, the minimum distance between any two distinct codewords.

A code for an alphabet of size q, of length n, rank k, and minimal distance d is called an $(n, k, d)_q$-code. Such a code can be used to detect up to $d - 1$ errors (because if a codeword is sent and fewer than $d - 1$ errors occur, it will not get transformed to another codeword), and correct up to $\lfloor (d - 1)/2 \rfloor$ errors (because for any received word, there is a unique codeword within distance $\lfloor (d-1)/2 \rfloor$). For linear codes, the encoding of a (row vector) word $W \in \mathbb{F}_q^k$ is performed by an algorithm

$C.\mathsf{Encode} : \mathbb{F}_q^k \to \mathbb{F}_q^n$, which is the multiplication of W by a so-called "generating matrix" $G \in \mathbb{F}_q^{k\times n}$ (which defines an injective linear map). This leads to a row-vector codeword $W \cdot G =: c \in C \subset \mathbb{F}_q^n$.

The Singleton bound states that for any linear code, it holds that $k+d \leq n+1$. A *maximum distance separable* (or MDS) code satisfies $k+d = n+1$. Since $d = n-k+1$, MDS codes are fully described by the parameters (q, n, k). Such an $(n, k)_q$-MDS code can correct up to $\lfloor (n-k)/2 \rfloor$ errors; it can detect if there are errors whenever there are no more than $n-k$ of them. For a matrix G generating an MDS code, any set of k columns of G are linearly independent.

For a thorough introduction to linear codes and proofs of all statements in this short overview we refer the reader to (Roth, 2006).

Observe that a linear code, due to the linearity of its encoding algorithm, is not a primitive designed to hide anything about the encoded message (e.g., a popular choice for the generating matrix is $G := (I_k|*)$ with I_k being the $k \times k$ identity matrix). However, we show in the following lemma how to turn an MDS code into a RSS scheme with additional smoothness guarantees.

Lemma 5. *Let C be a $(n+1, k)_q$-MDS code. We set L to be the last column of the generating matrix G of the code C and we denote by C' the $(n, k)_q$-MDS code whose generating matrix G' is G without the last column. Let further algorithm* Decode *of the MDS code C' be of the following form:*

1. *On input a word $c \in \mathbb{F}_q^n$,* Decode *chooses $D \subseteq [n]$ with $|D| = k$.*
2. *Let G'_D denote the matrix obtained from G' by eliminating all columns with indices not in D.* Decode *now outputs $c_D \cdot G'^{-1}_D$.*

Let Share *and* Reconstruct *work as follows:*

- Share(s) *for $s \in \mathbb{F}_q$ first chooses a random row vector $W \in \mathbb{F}_q^k$ such that $W \cdot L = s$, and outputs $c \leftarrow C'.\mathsf{Encode}(W)$ (equivalently, we can say that* Share(s) *chooses a uniformly random codeword of C whose last coordinate is s, and outputs the first n coordinates as c).*
- Reconstruct(w) *for $w \in \mathbb{F}_q^n$ first runs $C'.\mathsf{Decode}(w)$. If it gets a vector W', then output $s = W' \cdot L$, otherwise output $s \stackrel{\$}{\leftarrow} \mathbb{F}_q$.*

Then Share *and* $\mathsf{Reconstruct}$ *form a* t*-smooth, strongly* t*-private* $(n,t,r)_q$*-RSS for* $t = k-1$ *and* $r = \lceil (n+k)/2 \rceil$ *that is* $(r-1)$*-smooth on random secrets.*

Proof. Let us consider all required properties from Definitions 27, 28 and 29.

- strong t-privacy: Assume $|A| = t$ (privacy for smaller A will follow immediately by adding arbitrary coordinates to it to get to size t). Let $J = A \cup \{n+1\}$; note that $|J| = t+1 = k$. Note that for the code C, any k coordinates of a codeword determine uniquely the input to Encode that produces this codeword (otherwise, there would be two codewords that agreed on k elements and thus had distance $n-k+1$, which is less than the minimum distance of C). Therefore, the mapping given by $\mathsf{Encode}_J : \mathbb{F}_q^k \to \mathbb{F}_q^{|J|}$ is bijective; thus coordinates in J are uniform when the input to Encode is uniform. The algorithm Share chooses the input to Encode uniformly subject to fixing the coordinate $n+1$ of the output. Therefore, the remaining coordinates (i.e., the coordinates in A) are uniform.
- r-robustness: Note that C has minimum distance $n-k+2$, and therefore C' has minimum distance $n-k+1$ (because dropping one coordinate reduces the distance by at most 1). Therefore, C' can correct $\lfloor (n-k)/2 \rfloor = n-r$ errors. Since $c_A = \tilde{c}_A$ and $|A| \geq r$, there are at most $n-r$ errors in $\tilde{c}$, so the call to $C'.\mathsf{Decode}(\tilde{c})$ made by $\mathsf{Reconstruct}(\tilde{c})$ will output $\tilde{W} = W$. Then $\mathsf{Reconstruct}(\tilde{c})$ will output $s = \tilde{W} \cdot L = W \cdot L$.
- t-smoothness: to prove this, we show that disturbing one share uniformly random already randomizes the output of $\mathsf{Reconstruct}$. Let D denote the set chosen by Decode. Since every codeword is uniquely determined by k elements, the mappings $f_i : \mathbb{F}_q \to \mathbb{F}_q$, $x \mapsto \mathsf{Encode}(G_D'^{-1}(c))_{n+1}$ with $c \leftarrow \mathbb{F}_q^k, c_i = x$ are bijective for all $i \in [k]$. Since $t = k-1$, $\tilde{c}_D$ contains at least one entry that is chosen uniformly at random and thus the claim follows from the fact that the output of $\mathsf{Reconstruct}$ is computed as $\mathsf{Encode}(G_D'^{-1}(\cdot))$.
- $(r-1)$-smoothness on random secrets: first, it holds that $r-1 > k$ and thus $\tilde{c}$ contains more than k undisturbed shares. We distinguish two cases. Either D chosen by Decode contains only undisturbed shares (i.e., $D \subseteq A$), then $\mathsf{Reconstruct}$ will output s which is distributed uniformly random in $\mathbb{F}_q$. Else, $D \nsubseteq A$. In this case, at least one element of $\tilde{c}_D$ is distributed uniformly random and the randomness of the output of $\mathsf{Reconstruct}$ follows as in the proof of t-smoothness.

□

Note that the Shamir's secret sharing scheme is exactly the above construction with Reed-Solomon codes (McEliece and Sarwate, 1981). Further, we are not aware of any decoding algorithm for linear MDS codes that cannot be adopted to comply with our restrictions on Decode.

Implicit-Only PAKE

PAKE protocols can have two types of authentication: implicit authentication, where at the end of the protocol the two parties share the same key if they used the same pass-string and random independent keys otherwise; or explicit authentication where, in addition, they actually know which of the two situations they are in. A PAKE protocol that only achieves implicit authentication can provide explicit authentication by adding key-confirmation flows (Bellare et al., 2000).

The standard PAKE functionality $\mathcal{F}_{\mathsf{pwKE}}$ from (Canetti et al., 2005) (see Figure 6·17) is designed with explicit authentication in mind, or at least considers that success or failure will later be detected by the adversary when he will try to use the key. Thus, it reveals to the adversary whether a pass-string guess attempt was successful or not. However, some applications could require a PAKE that does not provide any feedback, and so does not reveal the situation before the keys are actually used. Observe that, regarding honest players, $\mathcal{F}_{\mathsf{pwKE}}$ already features implicit authentication since the players do not learn anything but their own session key.

Definition of implicit-only PAKE. We introduce a new notion called implicit-only PAKE or iPAKE (see Figure 6·8). The iPAKE ideal functionality is designed to implement implicit authentication with respect to the adversary as well as honest players, by not providing the adversary with any feedback in response to a password guess. Of course, in many cases, the parties and the adversary can later check whether their session keys match or not, and so whether the pass-strings were the same or

The functionality $\mathcal{F}_{\mathsf{iPAKE}}$ is parameterized by a security parameter λ. It interacts with an adversary $\mathcal{S}$ and the (dummy) parties $\mathcal{P}_0$ and $\mathcal{P}_1$ via the following queries:

- **Upon receiving a query $(\texttt{NewSession}, sid, \mathsf{pw}_i, \texttt{role})$ from party $\mathcal{P}_i$:**
  - Send $(\texttt{NewSession}, sid, \mathcal{P}_i, \texttt{role})$ to $\mathcal{S}$;
  - If one of the following is true, record $(\mathcal{P}_i, \mathsf{pw}_i)$ and mark this record `fresh`:
    * This is the first `NewSession` query
    * This is the second `NewSession` query and there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$
- **Upon receiving a query $(\texttt{TestPwd}, sid, \mathcal{P}_i, \mathsf{pw}'_i)$ from $\mathcal{S}$:**
  If there is a `fresh` record $(\mathcal{P}_i, \mathsf{pw}_i)$, then:
  - If $\mathsf{pw}_i = \mathsf{pw}'_i$, mark the record `compromised`;
  - If $\mathsf{pw}_i \neq \mathsf{pw}'_i$, mark the record `interrupted`.
- **Upon receiving a query $(\texttt{NewKey}, sid, \mathcal{P}_i, \mathsf{sk})$ from $\mathcal{S}$, where $|\mathsf{sk}| = \lambda$:**
  If there is no record of the form $(\mathcal{P}_i, \mathsf{pw}_i)$, or if this is not the first `NewKey` query for $\mathcal{P}_i$, then ignore this query. Otherwise:
  - If at least one of the following is true, then output (sid, sk) to player $\mathcal{P}_i$:
    * The record is `compromised`
    * $\mathcal{P}_i$ is corrupted
    * The record is `fresh`, $\mathcal{P}_{1-i}$ is corrupted, and there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $\mathsf{pw}_i = \mathsf{pw}_{1-i}$
  - If this record is `fresh`, both parties are honest, there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $\mathsf{pw}_i = \mathsf{pw}_{1-i}$, a key sk' was sent to $\mathcal{P}_{1-i}$, and $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ was `fresh` at the time, then output (sid, sk') to $\mathcal{P}_i$;
  - In any other case, pick a new random key sk' of length λ and send (sid, sk') to $\mathcal{P}_i$.
  - Mark the record $(\mathcal{P}_i, \mathsf{pw}_i)$ as `completed`.

Figure 6·8: Functionality $\mathcal{F}_{\mathsf{iPAKE}}$

not. We stress that this is not leakage from the PAKE protocol itself, but from the environment.

In addition to the changes we already make to the $\mathcal{F}_{\mathsf{pwKE}}$ ideal functionality in our fPAKE functionality description, we make one more change in our iPAKE functionality. In response to a `TestPwd` query, the functionality silently updates the internal state of the record (from `fresh` to either `compromised` or `interrupted`), meaning that the query outcome is not given to the adversary $\mathcal{S}$. Without going into details, it is clear that the simulation of an honest party is hard if the simulator $\mathcal{S}$ does not know whether the password it extracted from the corrupt party is correct or not. However, the simulator gets help from the functionality, which sets honest parties' keys appropriately.

We further alter this functionality to allow for public labels, as shown in Section 6.11, Figure 6·26. The resulting functionality $\mathcal{F}_{\ell\text{-iPAKE}}$ idealizes what we call *labeled implicit-only PAKE* (or ℓ-iPAKE for short), resembling the notion of labeled public key encryption as formalized in (Shoup, 2001). In a nutshell, labels are public au-

$A(\mathsf{pw} \in \mathbb{F}_p)$		$B(\mathsf{pw}' \in \mathbb{F}_p)$
$x \xleftarrow{\$} \mathbb{F}_P,\ \ell \leftarrow \mathcal{L},\ X \leftarrow g^x$		
$X^* \leftarrow \mathcal{E}_{\mathsf{pw}\|\|\ell}(X)$	$\xrightarrow{\ell, X^*}$	$y \xleftarrow{\$} \mathbb{F}_P,\ \ell' \leftarrow \mathcal{L},\ Y \leftarrow g^y$
	$\xleftarrow{\ell', Y^*}$	$Y^* \leftarrow \mathcal{E}_{\mathsf{pw}'\|\|\ell'}(Y)$
$Z \leftarrow \mathcal{D}_{\mathsf{pw}\|\|\ell'}(Y^*)^x$		$Z' \leftarrow \mathcal{D}_{\mathsf{pw}'\|\|\ell}(X^*)^y$
$k \leftarrow H(X^*, Y^*, Z)$		$k' \leftarrow H(X^*, Y^*, Z')$
output (ℓ', k)		output (ℓ, k')

Figure 6·9: Protocol EKE2, in a group $\mathbb{G} = \langle g \rangle$ of prime order P, with a hash function $H : \mathbb{G}^3 \to \{0,1\}^k$ and a symmetric cipher $\mathcal{E}, \mathcal{D}$ onto $\mathbb{G}$ for keys in $\mathbb{F}_p \times \mathcal{L}$, where $\mathcal{L}$ is the label space.

thenticated strings that are chosen by each user individually for each execution of the protocol. Authentication here means that tampering with the label can be efficiently detected. Such labels can be used to, e.g., distribute public information such as public keys reliably over unauthenticated channels.

A UC-Secure ℓ-iPAKE Protocol. In the seminal paper by Bellovin and Merritt (Bellovin and Merritt, 1992), the Encrypted Key Exchange protocol (EKE) is proposed, which is essentially a Diffie-Hellman (Diffie and Hellman, 1976) key exchange. The two flows of the protocol are encrypted using the pass-string as the encryption key with an appropriate symmetric encryption scheme. The EKE protocol has been further formalized by Bellare *et al.* (Bellare et al., 2000) under the name EKE2. We present its labeled variant in Figure 6·9. The idea of appending the label to the symmetric key is taken from (Abdalla et al., 2008). We prove security of this protocol in the $\mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{IC}}, \mathcal{F}_{\mathsf{CRS}}$-hybrid model. That is, we use an ideal random oracle functionality $\mathcal{F}_{\mathsf{RO}}$ to model the hash function, and ideal cipher functionality $\mathcal{F}_{\mathsf{IC}}$ to model the encryption scheme and assume a publicly available common reference string modeled by $\mathcal{F}_{\mathsf{CRS}}$. Formal definitions of these functionalities are given in Section 6.6.

Theorem 13. *If the CDH assumption holds in $\mathbb{G}$, the protocol EKE2 depicted in Figure 6·9 securely realizes $\mathcal{F}_{\ell\text{-iPAKE}}$ in the $\mathcal{F}_{RO}, \mathcal{F}_{IC}, \mathcal{F}_{CRS}$-hybrid model with respect to*

static corruptions.

We note that this result is not surprising, given that other variants of EKE2 have already been proven to UC-emulate $\mathcal{F}_{\mathsf{pwKE}}$. Intuitively, a protocol with only two flows not depending on each other does not leak the outcome to the adversary via the transcript, which explains why EKE2 is implicit-only. Hashing of the transcript keeps the adversary from biasing the key unless he knows the correct pass-string or breaks the ideal cipher. For completeness, we include the full proof in Section 6.11.

6.4.2 Construction

We show how to combine an RSS with a signature scheme and an ℓ-iPAKE to obtain an fPAKE. The high-level idea is to fix the issue that arose in the protocol from Section 6.13 due to pass-strings being used as one-time pads. Instead, we first expand the pass-string characters to keys — which we refer to as "character keys" — with large entropy using ℓ-iPAKE. The resulting character keys are then used as one-time pads for shares of a chosen output session key. We also apply known techniques from the literature, i.e., adding signatures and labels to prevent man-in-the-middle attacks. Our full protocol is depicted in Figure 6·10. It works as follows:

1. In the first phase, the two parties aim at enhancing their pass-strings to a vector of high-entropy character keys. For this, the pass-strings are viewed as vectors of characters. The parties repeatedly execute a PAKE on each of these characters separately. The PAKE will ensure that the character key vectors held by the two parties match in all positions where their pass-strings matched, and are independent in all other positions.
2. In the second phase of the protocol one party, the sender, will pick the final session key uniformly at random and send it in such a way that it reaches the

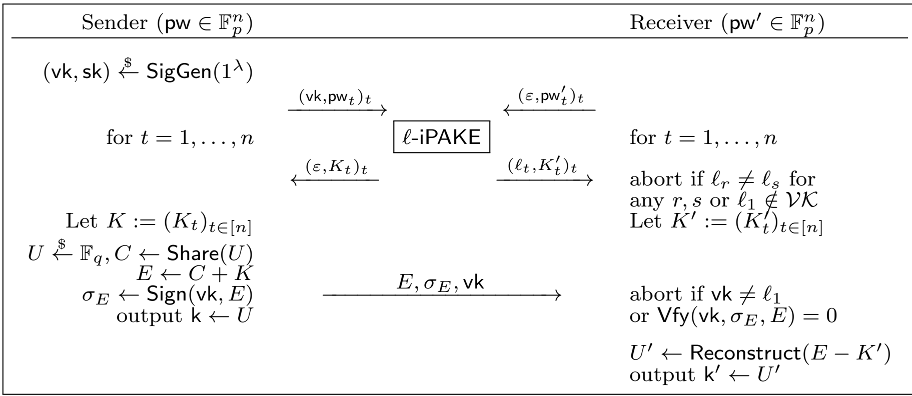


Figure 6·10: Protocol $\mathsf{fPAKE_{RSS}}$ where $q \approx 2^\lambda$ is a prime number and $+$ denotes the group operation in $\mathbb{F}_q^n$. ε denotes the empty string. $(\mathsf{Share}, \mathsf{Reconstruct})$ is a Robust Secret Sharing scheme with $\mathsf{Share} : \mathbb{F}_q \to \mathbb{F}_q^n$, and $(\mathsf{SigGen} \to \mathcal{VK} \times \mathcal{SK}, \mathsf{Sign}, \mathsf{Vfy})$ is a signature scheme. The parties repeatedly execute a labeled implicit-only PAKE protocol with label space $\mathcal{VK}$ and key space $\mathbb{F}_q$, which takes inputs from $\mathcal{VK} \times \mathbb{F}_p$. If at any point an expected message fails to arrive (or arrives malformed), the parties output a random key.

other party only if enough of the character keys match. This is done by applying an RSS to the key, and sending it to the other party using the character key vector as a one time pad. The robustness property of the RSS ensures that a few non-matching password digits do not prevent the receiver from recovering the sender's key.

When using the MDS code–based RSS which is described in Lemma 5, the one-time pad encryption of the shares (which form a codeword) can be viewed as the code-offset construction for information reconciliation (aka secure sketch) (Juels and Wattenberg, 1999; Dodis et al., 2004) applied to the character key vectors. We present our construction in terms of RSS, but we could instead present this construction using information reconciliation. The syndrome construction of secure sketches 5 can also be used here instead of the code-offset construction.

6.4.3 Security of $\mathsf{fPAKE}_{\mathsf{RSS}}$

We show that our protocol realizes functionality $\mathcal{F}^M_{\mathsf{fPAKE}}$ in the $\mathcal{F}_{\ell\text{-}\mathsf{iPAKE}}$-hybrid model. In a nutshell, the idea is to simulate without the pass-strings by adjusting the character keys outputted by $\mathcal{F}_{\ell\text{-}\mathsf{iPAKE}}$ to the mask of the pass-strings, which is leaked by $\mathcal{F}^M_{\mathsf{fPAKE}}$.

Theorem 14. *If* $\mathsf{RSS} := (\mathsf{Share} : \mathbb{F}_q \to \mathbb{F}_q^n, \mathsf{Reconstruct} : \mathbb{F}_q^n \to \mathbb{F}_q)$ *is a strongly t-private, t-smooth $(n, t, r)_q$-RSS that in addition is $(r-1)$-smooth on random secrets, and* $(\mathsf{SigGen}, \mathsf{Sign}, \mathsf{Vfy})$ *is an EUF-CMA secure one-time signature scheme, protocol* fPAKE_{RSS} *securely realizes* $\mathcal{F}^M_{\mathsf{fPAKE}}$ *with $\gamma = n - t - 1$ and $\delta = n - r$ in the* $\mathcal{F}_{\ell\text{-}\mathsf{iPAKE}}$*-hybrid model with respect to static corruptions.*

In particular, if we wish key agreement to succeed as long as there are fewer than δ errors, we instantiate RSS using the construction of Lemma 5 based on a $(n+1, k)_q$ MDS code, with $k = n - 2\delta$. This will give $r = \lceil (n+k)/2 \rceil = n - \delta$, so δ will be equal to $n - r$, as required. It will also give $\gamma = n - t - 1 = 2\delta$.

We thus obtain the following corollary:

Corollary 2. *For any δ and $\gamma = 2\delta$, given an $(n+1, k)_q$-MDS code for $k = n - 2\delta$ (with minimal distance $d = n - k + 2$) and an EUF-CMA secure one-time signature scheme, protocol* fPAKE_{RSS} *securely realizes* $\mathcal{F}^M_{\mathsf{fPAKE}}$ *in the* $\mathcal{F}_{\ell\text{-}\mathsf{iPAKE}}$*-hybrid model with respect to static corruptions.*

Proof sketch of Theorem 14. We start with the real execution of the protocol and indistinguishably switch to an ideal execution with dummy parties relaying their inputs to and obtaining their outputs from $\mathcal{F}^M_{\mathsf{fPAKE}}$. To preserve the view of the distinguisher (the environment $\mathcal{Z}$), a simulator $\mathcal{S}$ plays the role of the real world adversary by controlling the communication between $\mathcal{F}^M_{\mathsf{fPAKE}}$ and $\mathcal{Z}$. During the proof, we build $\mathcal{F}^M_{\mathsf{fPAKE}}$ and $\mathcal{S}$ by subsequently randomizing pass-strings (since the final simulation has to work without them) and session keys (since $\mathcal{F}^M_{\mathsf{fPAKE}}$ hands out random session keys

in certain cases). We have to tackle the following difficulties, which we will describe in terms of attacks.

- Passive attack: in this attack, $\mathcal{Z}$ picks two pass-strings and then observes the transcript and outputs of the protocol, without having access to any internal state of the parties. We show that $\mathcal{Z}$ cannot distinguish between transcript and outputs that were either produced using $\mathcal{Z}$'s pass-strings or random pass-strings. Regarding the outputs, we argue that even in the real execution the session keys were chosen uniformly at random (with $\mathcal{Z}$ not knowing the coins consumed by this choice) as long as the RSS either reconstructs the uniformly random secret correctly or outputs a different uniformly random value. Depending on the distance of the passwords this is guaranteed either by robustness or smoothness for random secrets, both of which are properties of the RSS. Regarding the transcript, randomization is straightforward using properties of the one-time pad.
- Man-in-the-middle attack: in this attack, $\mathcal{Z}$ injects a malicious message into a session of two honest parties. There are several ways to secure protocols that have to run in unauthenticated channels and are prone to this attack. Basically, all of them introduce methods to bind messages together to prevent the adversary from injecting malicious messages. To do this, we need the *labeled* version of our iPAKE and a one-time signature scheme[6]. Unless $\mathcal{Z}$ is able to break a one-time-signature scheme, this attack always results in an abort.
- Honest-but-curious/Active attack: in this attack, $\mathcal{Z}$ corrupts one of the parties. The simulator will get help from $\mathcal{F}^{M}_{\mathsf{fPAKE}}$ by issuing a TestPwd query, which will inform him whether the passwords used by both parties are close and, if so, in

[6]Instead of labels and one-time signature, one could just sign all the messages, as would be done using the split-functionality (Barak et al., 2005), but this would be less efficient. This trade-off, with labels, is especially useful when we use a PAKE that admits adding labels basically for free, as it is the case with the special PAKE protocol we use.

which positions they match (i.e., their mask).

- If the sender is honest, we show how to use this information to simulate the transcript. Note that knowledge of the mask is necessary since, due to corruption, $\mathcal{Z}$ can now actually decrypt the one-time pad and thus the transcript reveals the positions of the errors in the pass-strings, which are of course already known to $\mathcal{Z}$. If the simulator does not learn a mask, then the passwords are too far away and it follows from the strong privacy of the RSS that real and simulated transcript are indistinguishable from $\mathcal{Z}$'s view. Note that here, since $\mathcal{Z}$ is aware of the non-matching pass-string positions, it is crucial that strong privacy guarantees that the locations of all errors are hidden from $\mathcal{Z}$.
- If the receiver is honest and $\mathcal{Z}$ injects a malicious message on behalf of the sender, the simulator uses the mask to compute the output of the honest receiver. If no mask is obtained then again the pass-strings are too far away from each other, and the smoothness property of the RSS (for arbitrarily chosen secrets) says that the receiver's output can be simulated by chosing it uniformly at random.

One interesting subtlety that arises is the usage of the iPAKE. Observe that the UC security notion for a regular PAKE as defined by Canetti *et al.* (Canetti et al., 2005) and recalled in Section 6.6 provides an interface to the adversary to test a pass-string once and learn whether it is right or wrong. Using this notion, our simulator would have to answer to such queries from $\mathcal{Z}$. Since this is not possible without $\mathcal{F}^{M}_{\mathsf{fPAKE}}$ leaking the mask all the time, it is crucial to use the iPAKE variant that we introduce in Section 6.11. Using this stronger notion, the adversary is still allowed one pass-string guess which may affect the output, but the adversary learns nothing more about the outcome of his guess than he can infer from whatever access he has to the

outputs alone. Since our protocol uses the outputs of the PAKE as one-time pad keys, it is intuitively clear that by preventing $\mathcal{Z}$ from getting additional leakage about these keys, we protect the secrets of honest parties.

6.4.4 Further Discussion: Removing Modeling Assumptions

All of the assumptions in our protocol come from the realization of the $\mathcal{F}_{\ell\text{-iPAKE}}$ functionality. The ℓ-iPAKE protocol from section 6.11 requires a random oracle, an ideal cipher and a CRS. We note that we can remove everything except for the CRS by, e.g., taking the PAKE protocol introduced by Katz and Vaikuntanathan (Katz and Vaikuntanathan, 2011). This protocol also securely realizes our $\mathcal{F}_{\ell\text{-iPAKE}}$ functionality[7]. However, it is more costly than our ℓ-iPAKE protocol since both messages each contain one non-interactive zero knowledge proof.

Since fPAKE implies a regular PAKE (simply set $\delta = 0$), Canetti *et al.* (Canetti et al., 2005) give strong evidence that we cannot hope to realize $\mathcal{F}_{\text{fPAKE}}$ without a CRS.

6.5 Comparison of fPAKE Protocols

In this section, we give a brief comparison of our fPAKE protocols. First, in Figure 6·11, we describe the assumptions necessary for the two constructions, and the security parameters that they can achieve.

Then, in Figure 6·12, we describe the efficiency of the constructions when concrete

[7] In a nutshell, their protocol is implicit-only for the same reason as the ℓ-iPAKE protocol we use here: there are only two flows that do not depend on each other, so the transcript cannot reveal the outcome of a guess unless it reveals the pass-string to anyone. Regarding the session keys, usage of a hash function takes care of randomizing the session key in case of a failed dictionary attack. Furthermore, the protocol already implements labels. A little more detailed, looking at the proof by Katz and Vaikuntanathan (Katz and Vaikuntanathan, 2011), the simulator does not make use of the answer of `TestPwd` to simulate any messages. Regarding the session key that an honest player receives in an corrupted session, they are chosen to be random in the simulation (in Expt_3). Letting this happen already in the functionality makes the simulation independent of the answer of `TestPwd` also regarding the computation of the session keys.

	Assumptions	Threshold δ	Gap $\gamma - \delta$
$\mathsf{fPAKE_{RSS}}$	UC-secure ℓ-iPAKE	$< n/2$	δ
$\mathsf{fPAKE_{YGC}}$	(1) UC-secure OT (2) projective, output-projective and garbled-output random secure garbling scheme	Any	None

Figure 6·11: Assumptions, Distance Thresholds and Functionality/Security Gaps achieved by the two schemes. $\mathsf{fPAKE_{RSS}}$ is the construction in Figure 6·10. $\mathsf{fPAKE_{YGC}}$ is the construction in Figure 6·5 with the split functionality transformation of Barak *et al.* (Barak et al., 2005).

primitives (OT / ℓ-iPAKE) are used to instantiate them. $\mathsf{fPAKE_{RSS}}$ is instantiated as the construction in Figure 6·10 with the ℓ-iPAKE in Figure 6·9 and an RSS. $\mathsf{fPAKE_{YGC}}$ is instantiated as the construction in Figure 6·5 with the UC-secure oblivious transfer protocol of Chou and Orlandi (Chou and Orlandi, 2015) described in Figure 6·25, with the garbling scheme of Bal *et al.* (Ball et al., 2016), and with the split functionality transformation of Barak *et al.* (Barak et al., 2005). Though $\mathsf{fPAKE_{YGC}}$ can handle any efficiently computable notion of distance, Figure 6·12 assumes that both constructions use Hamming distance (and that, specifically, $\mathsf{fPAKE_{YGC}}$ uses the circuit described in Figure 6·7). We describe efficiency in terms of sub-operations (per-party, not in aggregate).

Note that these concrete primitives each have their own set of required assumptions. Specifically, the ℓ-iPAKE in Figure 6·9 requires a random oracle (RO, described in Figure 6·14), ideal cipher (, described in Figure 6·15) and common reference string (CRS, described in Figure 6·13). The oblivious transfer protocol in Figure 6·25 requires a random oracle. The garbling scheme of Bal *et al.* (Ball et al., 2016) requires a mixed modulus circular correlation robust hash function, which is a weakening of the random oracle assumption.

For $\mathsf{fPAKE_{RSS}}$, the factor of n arises from the n times EKE2 is executed. For

$\mathsf{fPAKE}_{\mathsf{YGC}}$, the factor of n comes from the garbled circuit. Additionally, in $\mathsf{fPAKE}_{\mathsf{YGC}}$, three communication flows come from OT. The last of these is combined with sending the garbled circuits. Two additional flows of communication come from the split functionality transformation. The need for signatures also arises from the split functionality transformation.

		Output Key Format	# (Bidirectional) Communication Flows	# Exponentiations	# Hashes	# Encryptions	# Decryptions	# Share	# Reconstruct	# SigKeyGens	# Signs	# SigVerifies
$\mathsf{fPAKE}_{\mathsf{RSS}}$	sender	$\mathbb{F}_q$	2	$2n$	n	n	n	1	0	1	1	0
	receiver							0	1	0	0	1
$\mathsf{fPAKE}_{\mathsf{YGC}}$		$\{0,1\}^\lambda$	5	$3n+2$	$4n+7$	$2n$	n	–	–	1	5	5

Figure 6·12: Efficiency (in Terms of Sub-Operations) of the Two Constructions. Here, by "bidirectional communication flow" we mean two flows, one in each direction, which do not depend on one another. $\mathsf{fPAKE}_{\mathsf{RSS}}$ is the construction in Figure 6·10 instantiated with the ℓ-iPAKE in Figure 6·9. The first $\mathsf{fPAKE}_{\mathsf{RSS}}$ row describes the sender's efficiency; the second row describes the receiver's efficiency. $\mathsf{fPAKE}_{\mathsf{YGC}}$ is the construction in Figure 6·5 instantiated with the UC-secure oblivious transfer protocol of Chou and Orlandi (Chou and Orlandi, 2015) described in Figure 6·25, the garbling scheme of Bal *et al.* (Ball et al., 2016), and with the split functionality transformation of Barak *et al.* (Barak et al., 2005). $\mathsf{fPAKE}_{\mathsf{YGC}}$ is described in a single row, since it is a symmetric protocol.

Efficiency Optimizations to $\mathsf{fPAKE}_{\mathsf{YGC}}$ We can make several small efficiency improvements to the $\mathsf{fPAKE}_{\mathsf{YGC}}$ construction which are not reflected in Figure 6·12. First, instead of using the split functionality transformation of Barak *et al.* (Barak et al., 2005), we can use the split split functionality of Camenisch *et al.* (Camenisch et al., 2010). It uses a split key exchange functionality to establish symmetric keys, and then uses those to symmetrically encrypt and authenticate each flow. While this does not save any communication rounds, it does reduce the number of public key operations needed. Second, if the pass-strings are more than λ bits long (where λ

is the security parameter), OT extensions that are secure against malicious adversaries (Afshar et al., 2015) can be used. If the pass-strings are fewer than λ bits long, then nothing is to be gained from using OT extensions, since OT extensions require λ "base OTs". However, if the pass-strings are longer — say, if they are some biometric measurement that is thousands of bits long — then OT extensions would save on the number of public key operations, at the cost of an extra round of communication.

6.6 Appendix A: Ideal UC Functionalities

COMMON REFERENCE STRING. The Common Reference String (CRS) functionality was already defined in (Canetti, 2007). We recall it in Figure 6·13 for completeness. Note that we do not let $\mathcal{F}_{\mathsf{CRS}}$ check whether a party is allowed to obtain the CRS — it is assumed public.

The functionality $\mathcal{F}_{\mathsf{CRS}}^{\mathcal{D}}$ is parameterized with a distribution $\mathcal{D}$ and proceeds as follows:

- Upon receiving $(sid, \texttt{crs})$:
  - If there is no value r recorded, then choose and record a value $r \xleftarrow{\$} \mathcal{D}$.
  - Reply with (sid, r).

Figure 6·13: Functionality $\mathcal{F}_{\mathsf{CRS}}$

RANDOM ORACLES. The Random Oracle (RO) functionality was already defined by Hofheinz and Müller-Quade in (Hofheinz and Müller-Quade, 2004). We recall it in Figure 6·14 for completeness. It is clear that the random oracle model UC-emulates this functionality.

IDEAL CIPHER. An ideal cipher (Bellare et al., 2000) is a block cipher that takes a plaintext or a ciphertext as input. We describe the ideal cipher functionality $\mathcal{F}_{\mathsf{IC}}$ in Figure 6·15, in the same vein as the above random oracle functionality. It is clear that the ideal cipher model UC-emulates this functionality. Note that this functionality

The functionality $\mathcal{F}_{\mathsf{RO}}$ proceeds as follows, running on security parameter k, with a set of (dummy) parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ and an adversary $\mathcal{S}$:

- $\mathcal{F}_{\mathsf{RO}}$ keeps a list L (which is initially empty) of pairs of bit strings.
- Upon receiving a value (sid, m) (with $m \in \{0,1\}^*$) from some party $\mathcal{P}_i$ or from $\mathcal{S}$, do:
  - If there is a pair $(m, \tilde{h})$ for some $\tilde{h} \in \{0,1\}^k$ in the list L, set $h := \tilde{h}$.
  - If there is no such pair, choose uniformly $h \in \{0,1\}^k$ and store the pair $(m, h) \in L$.

  Once h is set, reply to the activating machine (i.e., either $\mathcal{P}_i$ or $\mathcal{S}$) with (sid, h).

Figure 6·14: Functionality $\mathcal{F}_{\mathsf{RO}}$

characterizes a perfectly random permutation for each key by ensuring injectivity for each query simulation: to this aim, it uses a list L and projections M_{sk} and C_{sk}, that are global, independently of the sid.

Oblivious Transfer. The Oblivious Transfer (OT) functionality was defined by Canetti *et al.* (Canetti et al., 2002). We recall it in Figure 6·16.

Password-Authenticated Key Exchange. The initial PAKE functionality $\mathcal{F}_{\mathsf{pwKE}}$ has been defined by Canetti *et al.* (Canetti et al., 2005). We recall it in Figure 6·17. We stress that this functionality immediately leaks the result of the `TestPwd`-query, which models explicit authentication; when the adversary tries a password, it learns whether the guess was correct or not.

In our paper, we start from this functionality to derive the basic functionality fPAKE, in Section 6.2, after a few changes:

- we consider only two parties — $\mathcal{P}_0$ and $\mathcal{P}_1$ —, which is enough since universal composability takes care of ensuring that a two-party functionality remains secure in a multi-party world;
- we do not allow the adversary to set $\mathcal{P}_i$'s key if $\mathcal{P}_{1-i}$ is corrupted but did not guess $\mathcal{P}_i$'s password. We make this change in order to protect an honest $\mathcal{P}_i$ from, for instance, revealing sensitive information to an adversary who did not

The functionality $\mathcal{F}_{\mathsf{IC}}$ takes as input the security parameter k, and interacts with an adversary $\mathcal{S}$ and with a set of (dummy) parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ by means of these queries:

- $\mathcal{F}_{\mathsf{IC}}$ keeps a (initially empty) list L containing 3−tuples of bit strings and two (initially empty) sets C_{sk} and M_{sk} for every sk. (The sets are not created until sk is first used, thus avoiding the need to instantiate exponentially many sets.)
- **Upon receiving a query $(sid, \mathcal{E}, \mathsf{sk}, m)$ (with $m \in \{0,1\}^k$) from some party $\mathcal{P}_i$ or $\mathcal{S}$, do:**
  - If there is a 3−tuple $(\mathsf{sk}, m, \tilde{c})$ for some $\tilde{c} \in \{0,1\}^k$ in the list L, set $c := \tilde{c}$.
  - If there is no such record, choose uniformly $c \in \{0,1\}^k \backslash C_{\mathsf{sk}}$ which is the set consisting of ciphertexts not already used with sk. Next, it stores the 3−tuple $(\mathsf{sk}, m, c) \in L$ and sets both $M_{\mathsf{sk}} \leftarrow M_{\mathsf{sk}} \cup \{m\}$ and $C_{\mathsf{sk}} \leftarrow C_{\mathsf{sk}} \cup \{c\}$.

  Once c is set, reply to the activating machine with (sid, c).
- **Upon receiving a query $(sid, \mathcal{D}, \mathsf{sk}, c)$ (with $c \in \{0,1\}^k$) from some party $\mathcal{P}_i$ or $\mathcal{S}$, do:**
  - If there is a 3−tuple $(\mathsf{sk}, \tilde{m}, c)$ for some $\tilde{m} \in \{0,1\}^k$ in L, set $m := \tilde{m}$.
  - If there is no such record, choose uniformly $m \in \{0,1\}^k \backslash M_{\mathsf{sk}}$ which is the set consisting of plaintexts not already used with sk. Next, it stores the 3−tuple $(\mathsf{sk}, m, c) \in L$ and sets both $M_{\mathsf{sk}} \leftarrow M_{\mathsf{sk}} \cup \{m\}$ and $C_{\mathsf{sk}} \leftarrow C_{\mathsf{sk}} \cup \{c\}$.

  Once m is set, reply to the activating machine with (sid, m).

Figure 6·15: Functionality $\mathcal{F}_{\mathsf{IC}}$

The functionality $\mathcal{F}_{\mathsf{OT}}$ is parameterized by a security parameter λ. It interacts with an adversary $\mathcal{S}$ and the players $\mathcal{S}$ (the sender) and $\mathcal{R}$ (the receiver) via the following queries:

- **Upon receiving a query $(\texttt{Send}, sid, x_0, x_1)$ from $\mathcal{S}$**, where $x_0, x_1 \in \{0,1\}^\lambda$, record the tuple (x_0, x_1).
- **Upon receiving a query $(\texttt{Receive}, sid, i)$ from $\mathcal{R}$:**
  If there is a record (x_0, x_1), then send (sid, x_i) to $\mathcal{R}$ and sid to $\mathcal{S}$, and halt. Otherwise, ignore the query.

Figure 6·16: Functionality $\mathcal{F}_{\mathsf{OT}}$

The functionality $\mathcal{F}_{\mathsf{pwKE}}$ is parameterized by a security parameter k. It interacts with an adversary $\mathcal{S}$ and a set of (dummy) parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ via the following queries:

- **Upon receiving a query $(\texttt{NewSession}, sid, \mathcal{P}_i, \mathcal{P}_j, \mathsf{pw}, \texttt{role})$ from party $\mathcal{P}_i$:**
  - Send $(\texttt{NewSession}, sid, \mathcal{P}_i, \mathcal{P}_j, \texttt{role})$ to $\mathcal{S}$;
  - If one of the following is true, record $(\mathcal{P}_i, \mathcal{P}_j, \mathsf{pw})$ and mark this record `fresh`:
    * This is the first `NewSession` query
    * This is the second `NewSession` query and there is a record $(\mathcal{P}_j, \mathcal{P}_i, \mathsf{pw}')$
- **Upon receiving a query $(\texttt{TestPwd}, sid, \mathcal{P}_i, \mathsf{pw}')$ from $\mathcal{S}$:**
  If there is a `fresh` record $(\mathcal{P}_i, \mathcal{P}_j, \mathsf{pw})$, then do:
  - If $\mathsf{pw} = \mathsf{pw}'$, mark the record `compromised`, and reply to $\mathcal{S}$ with *correct guess*;
  - If $\mathsf{pw} \neq \mathsf{pw}'$, mark the record `interrupted`, and reply to $\mathcal{S}$ with *wrong guess*.
- **Upon receiving a query $(\texttt{NewKey}, sid, \mathcal{P}_i, \mathsf{sk})$ from the $\mathcal{S}$, where $|\mathsf{sk}| = k$:**
  If there is a record of the form $(\mathcal{P}_i, \mathcal{P}_j, \mathsf{pw})$, and this is the first `NewKey` query for $\mathcal{P}_i$, then:
  - If this record is `compromised`, or either $\mathcal{P}_i$ or $\mathcal{P}_j$ is corrupted, then output (sid, sk) to player $\mathcal{P}_i$;
  - If this record is `fresh`, there is a record $(\mathcal{P}_j, \mathcal{P}_i, \mathsf{pw}')$ with $\mathsf{pw} = \mathsf{pw}'$, a key sk' was sent to $\mathcal{P}_j$, and $(\mathcal{P}_j, \mathcal{P}_i, \mathsf{pw})$ was `fresh` at the time, then output (sid, sk') to $\mathcal{P}_i$;
  - In any other case, pick a new random key sk' of length k and send (sid, sk') to $\mathcal{P}_i$.

  Either way, mark the record $(\mathcal{P}_i, \mathcal{P}_j, \mathsf{pw})$ as `completed`.

Figure 6·17: Functionality $\mathcal{F}_{\mathsf{pwKE}}$

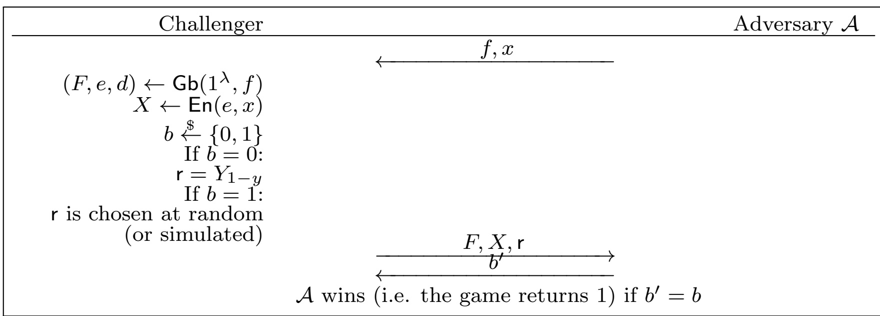


Figure 6·18: The $\mathsf{GOutRand}_{\mathcal{G}}^{\mathcal{A}}(1^\lambda)$ Game, where $y = f(x) \in \{0,1\}$, Y_y is the corresponding garbled output (or output label), and Y_{1-y} is the other output label.

successfully guess her password, but did corrupt her partner.

6.7 Appendix B: Garbled Output Randomness: A New Yao's Garbled Circuit Definition

We refer to Yakoubov (Yakoubov, 2017) for a gentle introduction to Yao's Garbled Circuits. Note that the authenticity property implies that, in an output-projective garbling scheme, if the output is a single bit, the second output label and the second token of d are hard for the evaluator to guess (no probabilistic polynomial-time adversary can guess it with non-negligible probability). However, for our fPAKE construction (Section 6.3), we require a stronger property: not only should the second output label be hard to guess, but it should be *indistinguishable from random*. We call this *garbled-output randomness.*

Define the adversary $\mathcal{A}$'s advantage in the the garbled-output randomness game (Figure 6·18) as

$$\mathsf{GOutRandAdv}_{\mathcal{G}}(1^\lambda, \mathcal{A}) = \left| \Pr[\mathsf{GOutRand}_{\mathcal{G}}^{\mathcal{A}}(1^\lambda) = 1] - \frac{1}{2} \right|.$$

Definition 30. *An output-projective binary output garbling scheme* $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$ *is* garbled-output random *if for all sufficiently large security parameters* λ*, for*

any polynomial time adversary $\mathcal{A}$,

$$\mathsf{GOutRandAdv}_{\mathcal{G}}(1^{\lambda}, \mathcal{A}) = \nu.$$

In order to achieve this, we modify the scheme of Bal *et al.* (Ball et al., 2016) to put the output wire label through the hash function H one more time; the two labels will thus be $Y_0 = H(\mathsf{finaloutput}, W^0_{output})$ and $Y_1 = H(\mathsf{finaloutput}, W^0_{output} \oplus R)$, where W^0_{output} and $W^0_{output} \oplus R$ were the labels in the scheme of Bal *et al.*

Theorem 15. *This modified scheme is garbled-output random (Definition 30) when the key derivation function H is mixed-modulus circular correlation robust (Definition 1 of Bal* et al. *(Ball et al., 2016)).*

Proof Sketch. Definition 30 requires the indistinguishability of (real garbled circuit and inputs, real second output label) and (real garbled circuit and inputs, random second output label); in shorthand, we want to show that $(real, real) \sim (real, random)$, where $\sim$ denotes computational indistinguishability.

We use the setting (simulated garbled circuit and inputs, random second output label) — $(simulated, random)$ for short — as a hybrid step. We show that $(real, real) \sim (simulated, random)$. We do this by having the adversary $\mathcal{A}$ (modeled after the one in Choi *et al.* (Choi et al., 2012)) compute the second output label $\mathcal{O}(\mathsf{finaloutput}, 2, 2, W^b_{output}, 1, 0)$ (using the mixed-modulus circular correlation robustness oracle) and send it to the obliviousness adversary $\mathcal{B}$ along with the garbled circuit and garbled inputs. If the oracle is random, $\mathcal{B}$ will see $(simulated, random)$. Otherwise, $\mathcal{B}$ will see $(real, real)$. If $\mathcal{B}$ can distinguish between those two, then $\mathcal{A}$ can use that to break mixed-modulus circular correlation robustness. Hence, $(real, real) \sim (simulated, random)$.

Because the garbling scheme is oblivious, we know that $(simulated, random) \sim (real, random)$, since we can always add a random value to the adversary's view in the obliviousness game.

Now that we have $(real, real) \sim (simulated, random)$ and $(simulated, random) \sim (real, random)$, we can conclude that $(real, real) \sim (real, random)$.

□

6.8 Appendix C: Proof of Theorem 12

We proceed in a series of games, where no probabilistic polynomial-time environment can distinguish the view of the adversary $\mathcal{A}$ in each game from that in the previous game. We start with the real execution of the protocol and end with the ideal execution. Figure 6·19 summarizes the changes made in each game.

Game $\mathbf{G}_0$: Real

This is the real execution of Π_{RFE} where the environment $\mathcal{Z}$ runs the protocol (described in Figure 6·5) with parties $\mathcal{P}_0$ and $\mathcal{P}_1$, both having access to an ideal OT functionality $\mathcal{F}_{\mathsf{OT}}$, and an adversary $\mathcal{A}$ that, w.l.o.g., can be assumed to be the dummy adversary as shown in (Canetti, 2001, section 4.4.1).

Game $\mathbf{G}_1$: Adding Ideal Layout

This is the real game, but with dummy party and ideal functionality nodes thrown in and all previously existing nodes (except the environment) grouped into one machine, called the simulator ($\mathcal{S}_{\mathsf{RFE}}$, or $\mathcal{S}$ for short). Please refer to Figure 6·20 for the differences between $\mathbf{G}_0$ and $\mathbf{G}_1$.

Game $\mathbf{G}_2$: Adding $\mathcal{F}$'s Record-Keeping and `TestPwd` Interface

Modifications to $\mathcal{F}$: We now allow $\mathcal{F}$ to do all of the record-keeping described in Figure 6·4.

$\mathcal{F}$ still forwards `NewSession` queries from the dummy parties in their entirety (including the pass-string) to $\mathcal{S}_{\mathsf{RFE}}$, but also records them. Since this is a matter of internal record-keeping only, this does not affect $\mathcal{A}$'s view.

Modifications to $\mathcal{S}_{RFE}$: $\mathcal{S}_{\mathsf{RFE}}$ creates `NewKey` queries for $\mathcal{F}$ from whatever output the simulated parties produce. In this game, $\mathcal{F}$ still simply forwards the keys it

	Game	Functionality $\mathcal{F}$			$\mathcal{S}_{\mathsf{RFE}}$	Property Used
		`NewSession`	`TestPwd`	`NewKey`		
	Game $\mathbf{G}_0$	N/A	N/A	N/A	N/A	
	Game $\mathbf{G}_1$	forwards inputs to $\mathcal{S}_{\mathsf{RFE}}$		forwards outputs to dummy parties	runs protocol for honest parties	
	Game $\mathbf{G}_2$	records inputs			creates `NewKey` queries from party outputs	
both parties honest	Game $\mathbf{G}_3$			chooses keys for both parties when $d(\mathsf{pw}_0, \mathsf{pw}_1) \leq \delta$		garbled output randomness
	Game $\mathbf{G}_4$			chooses key for $\mathcal{P}_0$ when $d(\mathsf{pw}_0, \mathsf{pw}_1) > \delta$		garbled output randomness
	Game $\mathbf{G}_5$			chooses keys for both parties when $d(\mathsf{pw}_0, \mathsf{pw}_1) > \delta$		garbled output randomness
	Game $\mathbf{G}_6$				simulates F_0, X_0	obliviousness
	Game $\mathbf{G}_7$	does not forward $\mathsf{pw}_0, \mathsf{pw}_1$			simulates F_1, X_1	obliviousness
$\mathcal{P}_i$ honest, $\mathcal{P}_{1-i}$ corrupt	Game $\mathbf{G}_8$	replaces the malicious `NewSession` input with the one given by $\mathcal{S}_{\mathsf{RFE}}$			extracts malicious pw'_{1-i} from OT, and tells $\mathcal{F}$ to replace the malicious `NewSession` input with pw'_{1-i}	
	Game $\mathbf{G}_9$			chooses key for $\mathcal{P}_i$ when $d(\mathsf{pw}_0, \mathsf{pw}_1) > \delta$ (now fully implemented)		garbled output randomness
	Game $\mathbf{G}_{10}$				simulates F_i, X_i using $d(\mathsf{pw}_i, \mathsf{pw}'_{1-i})$	privacy
	Game $\mathbf{G}_{11}$	does not forward pw_i	fully implemented		makes `TestPwd` query to set pw'_i	

Figure 6·19: A Summary of the Sequence of Games in the Proof of Theorem 12

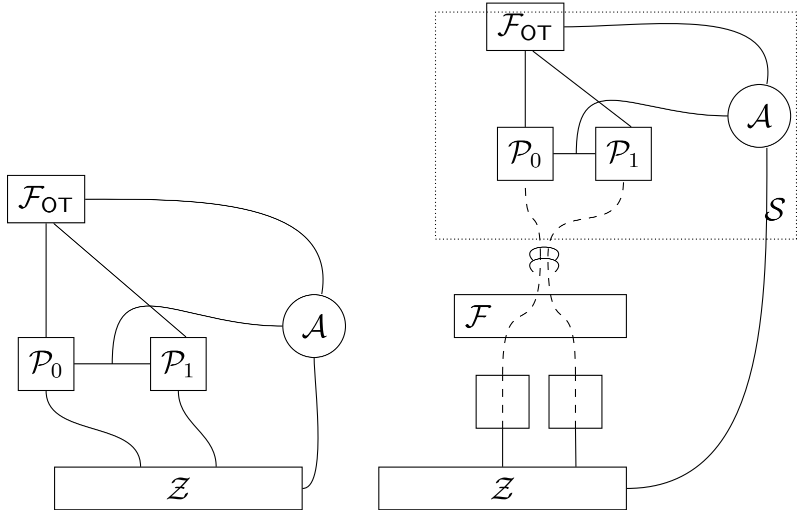


Figure 6·20: Transition from game $\mathbf{G}_0$ (left) to game $\mathbf{G}_1$ (right), showing a setting where both parties are honest.

is given to the dummy parties without modifying them, so this does not affect $\mathcal{A}$'s view.

Game $\mathbf{G}_3$: Allowing $\mathcal{F}$ to Choose Keys For Two Honest Parties With Close Pass-strings

Modifications to $\mathcal{F}$: We now allow $\mathcal{F}$ to follow the instructions in Figure 6·4 to choose the key when $\mathcal{P}_0$ and $\mathcal{P}_1$ are both honest, and $d(\mathsf{pw}_0, \mathsf{pw}_1) \leq \delta$.

We can use any environment who can distinguish this game from Game $\mathbf{G}_2$ to build an adversary $\mathcal{B}$ that can break the garbled output randomness property (Definition 30, Theorem 15) of our garbling scheme.

Since both parties are honest, in order to use the environment as a distinguisher, the adversary needs to give the environment a transcript of the parties' interactions $(F_0, X_{0,0}, F_1, X_{1,1})$ as well as the parties' output keys. Because we are in the OT hybrid model, the environment sees neither inputs to the OT nor its outputs.

Our adversary $\mathcal{B}$ executes $\mathcal{S}_{\mathsf{RFE}}$'s simulation of $\mathcal{P}_0$ with the $\mathcal{F}$ of Game $\mathbf{G}_2$ with

some modifications. First, it finds a pass-string pw such that $d(\mathsf{pw}_0, \mathsf{pw}) > \delta$. (Note that in order for this reduction to work, such a pass-string must be efficiently computable, which it is by the assumption in the statement of Theorem 12.) Instead of running Gb, $\mathcal{B}$ queries the garbled output randomness challenger on $(f, (\mathsf{pw}_0, \mathsf{pw}))$ to obtain (F_0, X_0, r). Let $X_0 = (X_{0,0}, X_{0,1})$. Note that F_0 and $X_{0,0}$ are generated by the challenger exactly as they would be by $\mathcal{S}_{\mathsf{RFE}}$, so those values do not change. ($X_{0,1}$ is different, since pw is different from pw_1, but $X_{0,1}$ is not visible to the environment.) Since the adversary used a pass-string pw that is dissimilar to pw_1, it uses the value r — corresponding to the output label *not* returned by $\mathsf{Ev}(F_0, X_0)$ — as $\mathsf{k}_{0,correct}$. ($\mathsf{Ev}(F_0, X_0)$ would give $\mathsf{k}_{0,wrong}$.) If $b = 0$, the challenger will return the actual $\mathsf{k}_{0,correct}$ as r, and the environment's view will be that of Game $\mathbf{G}_2$. If $b = 1$, the challenger will give a random value as r. If r is truly random, then so is $\mathsf{r} \oplus Y_1$; so, the environment's view will be that of Game $\mathbf{G}_3$. (Note that we do not change the way in which $\mathcal{P}_1$ generates F_1, X_1, but we do set $\mathcal{P}_1$'s output key to be the same as $\mathcal{P}_0$'s. This will be the key that an honest execution of the protocol would produce if $b = 0$, and random otherwise.) The adversary $\mathcal{B}$ then returns the environment's guess as b'. The advantage of $\mathcal{B}$ in the garbled output randomness game will be exactly the same as that of the environment in distinguishing between Game $\mathbf{G}_2$ and Game $\mathbf{G}_3$.

Game $\mathbf{G}_4$: Allowing $\mathcal{F}$ to Choose Keys For One of Two Honest Parties With Dissimilar Pass-strings

Modifications to $\mathcal{F}$: We now allow $\mathcal{F}$ to follow the instructions in Figure 6·4 to choose the key for $\mathcal{P}_0$ when $\mathcal{P}_0$ and $\mathcal{P}_1$ are both honest, and $d(\mathsf{pw}_0, \mathsf{pw}_1) > \delta$.

We can use any environment who can distinguish this game from Game $\mathbf{G}_3$ to build an adversary $\mathcal{B}$ that can break the garbled output randomness property

(Definition 30, Theorem 15) of our garbling scheme.

Our adversary $\mathcal{B}$ executes $\mathcal{S}_{\mathsf{RFE}}$'s simulation of $\mathcal{P}_0$ with the $\mathcal{F}$ of Game $\mathbf{G}_3$ with some modifications. Instead of running Gb, $\mathcal{B}$ queries the garbled output randomness challenger on $(f, (\mathsf{pw}_0, \mathsf{pw}_1))$ to obtain (F_0, X_0, r). Note that F_0 and X_0 are generated by the challenger exactly as they would be by $\mathcal{S}_{\mathsf{RFE}}$, so these values do not change. If $b = 0$, the challenger will return the actual $\mathsf{k}_{0,correct}$ as r, and the environment's view will be that of Game $\mathbf{G}_3$. If $b = 1$, the challenger will give a random value as r. If r is truly random, then so is $\mathsf{r} \oplus Y_1$; so, the environment's view will be that of Game $\mathbf{G}_4$. The adversary $\mathcal{B}$ then returns the environment's guess as b'. The advantage of $\mathcal{B}$ in the garbled output randomness game will be exactly the same as that of the environment in distinguishing between Game $\mathbf{G}_3$ and Game $\mathbf{G}_4$.

Game $\mathbf{G}_5$: Allowing $\mathcal{F}$ to Choose Keys For Both Honest Parties With Dissimilar Pass-strings

Modifications to $\mathcal{F}$: We now allow $\mathcal{F}$ to follow the instructions in Figure 6·4 to choose the key for $\mathcal{P}_1$ as well as for $\mathcal{P}_0$ when $\mathcal{P}_0$ and $\mathcal{P}_1$ are both honest, and $d(\mathsf{pw}_0, \mathsf{pw}_1) > \delta$.

We can use any environment who can distinguish this game from Game $\mathbf{G}_4$ to build an adversary that can break the garbled output randomness property (Definition 30, Theorem 15) of our garbling scheme, exactly as we did in the reduction above.

Game $\mathbf{G}_6$: Simulating F, X for One of Two Honest Parties

Modifications to $\mathcal{S}_{\mathsf{RFE}}$: Consider the case when both $\mathcal{P}_0$ and $\mathcal{P}_1$ are honest. In this game, the simulator replaces $\mathcal{P}_0$'s garbled circuit and input with simulated ones. $\mathcal{S}_{\mathsf{RFE}}$ does not need to simulate anything relating to the OT, since the

environment cannot observe OT functionality inputs or outputs if both participating parties are honest. $\mathcal{S}_{\mathsf{RFE}}$ uses the obliviousness simulator to generate F_0, X_0 (while continuing to generate F_1, X_1 honestly), and sends the garbled circuits and the appropriate parts of the garbled inputs between the parties. $\mathcal{S}_{\mathsf{RFE}}$ outputs $\perp$ bot as both parties' keys, since the outputs don't matter - $\mathcal{F}$ takes care of outputting appropriate keys as of a few games ago (Game $\mathbf{G}_3$ if $d(\mathsf{pw}_0, \mathsf{pw}_1) \leq \delta$, and Games $\mathbf{G}_4$,$\mathbf{G}_5$ otherwise), so this change is not observable by the environment.

We can use any environment who can distinguish this game from Game $\mathbf{G}_5$ to build an adversary $\mathcal{B}$ that can break the obliviousness property of our garbling scheme. $\mathcal{B}$ executes $\mathcal{S}_{\mathsf{RFE}}$'s simulation of $\mathcal{P}_0$ as in Game $\mathbf{G}_5$, but instead of generating (F_0, X_0) and (F_1, X_1) according to the protocol, it queries the obliviousness challenger on $(f, (\mathsf{pw}_0, \mathsf{pw}_1))$ to obtain (F_0, X_0). If $b = 0$, the challenger will return actual (F_i, X_i) values, and the environment's view we will be that of Game $\mathbf{G}_5$. If $b = 1$, the challenger will return simulated values, and the environment's view will be that of this game. The adversary $\mathcal{B}$ then returns the environment's guess as b'. The advantage of $\mathcal{B}$ in the obliviousness game will be exactly the same as that of the environment in distinguishing between Game $\mathbf{G}_5$ and Game $\mathbf{G}_6$.

Game $\mathbf{G}_7$: Removing Pass-string Forwarding Always

Modifications to $\mathcal{F}$: We now modify $\mathcal{F}$ to forward only $(\texttt{NewSession}, sid, \mathcal{P}_i)$ to $\mathcal{S}_{\mathsf{RFE}}$ (omitting the pass-string pw_i) for $\mathcal{P}_i \in \{\mathcal{P}_0, \mathcal{P}_1\}$ when $\mathcal{P}_0$ and $\mathcal{P}_1$ are both honest.

Modifications to $\mathcal{S}_{\mathsf{RFE}}$: In Game $\mathbf{G}_6$, $\mathcal{S}_{\mathsf{RFE}}$ started simulating $\mathcal{P}_0$'s messages without using knowledge of pw_0. $\mathcal{S}_{\mathsf{RFE}}$ now also simulates $\mathcal{P}_1$'s messages by using the obliviousness simulator to generate the garbled circuit and input F_1, X_1.

We can use any environment who can distinguish this game from Game $\mathbf{G}_6$ to build an adversary $\mathcal{B}$ that can break the obliviousness property of our garbling scheme. $\mathcal{B}$ executes $\mathcal{S}_{\mathsf{RFE}}$'s simulation of $\mathcal{P}_1$ as in Game $\mathbf{G}_6$, but instead of generating (F_1, X_1) according to the protocol, it queries the obliviousness challenger on $(f, (\mathsf{pw}_1, \mathsf{pw}_0))$ to obtain (F_1, X_1). If $b = 0$, the challenger will return actual (F_1, X_1) values, and the environment's view we will be that of Game $\mathbf{G}_6$. If $b = 1$, the challenger will return simulated values, and the environment's view will be that of this game. The adversary $\mathcal{B}$ then returns the environment's guess as b'. The advantage of $\mathcal{B}$ in the obliviousness game will be exactly the same as that of the environment in distinguishing between Game $\mathbf{G}_6$ and Game $\mathbf{G}_7$.

Game $\mathbf{G}_8$: Setting the Malicious Input as in the "Standard Corruption Model"

Modifications to $\mathcal{S}_{\mathsf{RFE}}$: In this game, $\mathcal{S}_{\mathsf{RFE}}$ sets a corrupt party $\mathcal{P}_{1-i}$'s `NewSession` input according to the standard corruption model (Canetti, 2001). It does so as soon as it sees pw'_{1-i}, when it is given as an input to the ideal OT functionality. Since $\mathcal{F}$ does not currently use pw_{1-i}, this does not affect the environment's view.

Game $\mathbf{G}_9$: Allowing $\mathcal{F}$ to Choose the Key For An Honest Party With a Pass-string Dissimilar to Its Corrupt Partners'

Modifications to $\mathcal{F}$: We now allow $\mathcal{F}$ to follow the instructions in Figure 6·4 to choose all keys. Note that the only remaining scenario this affects is the one where only one party $\mathcal{P}_i \in \{\mathcal{P}_0, \mathcal{P}_1\}$ is honest, and $d(\mathsf{pw}_0, \mathsf{pw}_1) > \delta$. If both parties are corrupt, or if only one party is corrupt and $d(\mathsf{pw}_0, \mathsf{pw}_1) \leq \delta$, $\mathcal{F}$ still simply forwards the output key.

We can use any environment who can distinguish this game from Game $\mathbf{G}_8$ to

build an adversary $\mathcal{B}$ that can break the garbled output randomness property (Definition 30, Theorem 15) of our garbling scheme. Our adversary $\mathcal{B}$ executes $\mathcal{S}_{\mathsf{RFE}}$'s simulation of $\mathcal{P}_0$ with the $\mathcal{F}$ of Game $\mathbf{G}_8$ with some modifications. Instead of running Gb first, it waits to see $\mathcal{P}_{1-i}$'s input pw'_{1-i} to the ideal OT functionality, and then queries the garbled output randomness challenger on $(f, (\mathsf{pw}_i, \mathsf{pw}'_{1-i}))$ to obtain (F_i, X_i, r). Note that F_i and X_i are generated by the challenger exactly as they would be by $\mathcal{S}_{\mathsf{RFE}}$, so those values do not change. The adversary then uses r as $\mathsf{k}_{i,correct}$. If $b = 0$, the challenger will return the actual $\mathsf{k}_{i,correct}$ as r, and the environment's view will be that of Game $\mathbf{G}_8$. If $b = 1$, the challenger will give a random value as r. If r is truly random, then so is $\mathsf{r} \oplus Y_{1-i}$; so, the environment's view will be that of Game $\mathbf{G}_9$. The adversary $\mathcal{B}$ then returns the environment's guess as b'. The advantage of $\mathcal{B}$ in the garbled output randomness game will be exactly the same as that of the environment in distinguishing between Game $\mathbf{G}_8$ and Game $\mathbf{G}_9$.

Game $\mathbf{G}_{10}$: Simulating Garbled Circuit and Inputs For An Honest Party With a Corrupt Partner

Modifications to $\mathcal{S}_{\mathsf{RFE}}$: In this game, $\mathcal{S}_{\mathsf{RFE}}$ simulates F_i and X_i when $\mathcal{P}_i$ is honest and $\mathcal{P}_{1-i}$ is corrupt.

In more detail, $\mathcal{S}_{\mathsf{RFE}}$ proceeds as follows on behalf of $\mathcal{P}_i$:

- $\mathcal{S}_{\mathsf{RFE}}$ postpones step 1.
- In step 2, $\mathcal{S}_{\mathsf{RFE}}$:
  - Plays an OT sender as follows:
    * Waits for $\mathcal{P}_{1-i}$ to provide their select bits to the OT. As a result, $\mathcal{S}_{\mathsf{RFE}}$ learns the pass-string used by $\mathcal{P}_{1-i}$, pw'_{1-i}.
    * If $d(\mathsf{pw}_i, \mathsf{pw}'_{1-i}) \leq \delta$ sets $y = 1$, and sets $y = 0$ otherwise.
    * Uses the privacy simulator for the garbling scheme to generate

$(F_i, X_i, d_i) \leftarrow \mathcal{SIM}(1^\lambda, f, y)$.

  * Parses $(X_{i,i}, X_{i,1-i}) \leftarrow X_i$.
  * Sends $X_{i,1-i}$ to $\mathcal{P}_{1-i}$ as the OT output.

 – Plays an OT receiver honestly.

- $\mathcal{S}_{\mathsf{RFE}}$ follows the instructions in Figure 6·5 for steps 3-8.

We can use any environment who can distinguish this game from Game $\mathbf{G}_9$ to build an adversary $\mathcal{B}$ that can break the privacy property of our garbling scheme. Our adversary $\mathcal{B}$ executes $\mathcal{S}_{\mathsf{RFE}}$'s simulation of $\mathcal{P}_0$ with some modifications. Instead of running the privacy simulator $\mathcal{SIM}$, it queries the privacy challenger on $(f, (\mathsf{pw}_i, \mathsf{pw}'_{1-i}))$ to obtain (F_i, X_i, d_i). If $b = 0$, the challenger will return actual (F_i, X_i, d_i) values, and the environment's view will be that of Game $\mathbf{G}_9$; if $b = 1$, the challenger return simulated values, and the environment's view will be that of this game. The adversary $\mathcal{B}$ then returns the environment's guess as b'. The advantage of $\mathcal{B}$ in the privacy game will be exactly the same as that of the environment in distinguishing between Game $\mathbf{G}_9$ and Game $\mathbf{G}_{10}$.

Game $\mathbf{G}_{11}$: Removing Pass-string Forwarding To An Honest Party With a Corrupt Partner

Modifications to $\mathcal{F}$:

- If $\mathcal{P}_i$ is honest and $\mathcal{P}_{1-i}$ is corrupt, then, upon receiving a `NewSession` query, $\mathcal{F}$ forwards only (`NewSession`, $sid, \mathcal{P}_i$) to $\mathcal{S}_{\mathsf{RFE}}$ (omitting the pass-string pw_i).
- $\mathcal{F}$ now processes `TestPwd` queries (which were not issued in any prior game) according to the instructions in Figure 6·4. Given a (`TestPwd`, $sid, \mathcal{P}_i$) query ($\mathcal{P}_i \in \{\mathcal{P}_0, \mathcal{P}_1\}$), if $d(\mathsf{pw}_0, \mathsf{pw}_1) \leq \delta$, $\mathcal{F}$ sends pw_i to $\mathcal{S}_{\mathsf{RFE}}$.

Modifications to $\mathcal{S}_{\mathsf{RFE}}$: Now that $\mathcal{S}_{\mathsf{RFE}}$ does not know $\mathcal{P}_i$'s pass-string, it must

simulate the honest party's messages without that knowledge.

In more detail, $\mathcal{S}_{\mathsf{RFE}}$ proceeds as follows on behalf of $\mathcal{P}_i$:

- $\mathcal{S}_{\mathsf{RFE}}$ postpones step 1.
- In step 2, $\mathcal{S}_{\mathsf{RFE}}$:
  - Plays an OT sender as follows:
    * As in Game $\mathbf{G}_{10}$, waits for $\mathcal{P}_{1-i}$ to provide their select bits to the OT. As a result, $\mathcal{S}_{\mathsf{RFE}}$ learns the pass-string used by $\mathcal{P}_{1-i}$, pw'_{1-i}.
    * Makes a $(\mathtt{TestPwd}, sid, \mathcal{P}_i)$ query to $\mathcal{F}$. If $d(\mathsf{pw}_i, \mathsf{pw}'_{1-i}) \leq \delta$ (that is, the adversary has approximately guessed $\mathcal{P}_i$'s pass-string), $\mathcal{S}_{\mathsf{RFE}}$ learns pw_i, and sets $\mathsf{pw}'_i = \mathsf{pw}_i$. Otherwise, it sets pw'_i at random such that $d(\mathsf{pw}'_i, \mathsf{pw}'_{1-i}) > \delta$, and uses pw'_i in place of pw_i in the rest of the simulation.
    * As in Game $\mathbf{G}_{10}$, If $d(\mathsf{pw}_i, \mathsf{pw}'_{1-i}) \leq \delta$ sets $y = 1$, and sets $y = 0$ otherwise.
    * As in Game $\mathbf{G}_{10}$, uses the privacy simulator for the garbling scheme to generate $(F_i, X_i, d_i) \leftarrow \mathcal{SIM}(1^\lambda, f, y)$.
    * As in Game $\mathbf{G}_{10}$, parses $(X_{i,i}, X_{i,1-i}) \leftarrow X_i$.
    * As in Game $\mathbf{G}_{10}$, sends $X_{i,1-i}$ to $\mathcal{P}_{1-i}$ as the OT output.
  - Plays an OT receiver honestly with pw'_i.
- $\mathcal{S}_{\mathsf{RFE}}$ follows the instructions in Figure 6·5 for steps 3-8.

Nothing could have changed from the point of view of $\mathcal{Z}$. If $d(\mathsf{pw}_i, \mathsf{pw}'_{1-i}) \leq \delta$, this game is identical to Game $\mathbf{G}_{10}$. If $d(\mathsf{pw}_i, \mathsf{pw}'_{1-i}) > \delta$, a random pw'_i is used instead of pw_i. However, pw'_i only affects the OT execution with $\mathcal{P}_i$ as receiver, where $\mathcal{P}_{1-i}$ does not receive any outputs anyway. In this case, $\mathcal{P}_i$'s output key gets set randomly by $\mathcal{F}$ as of Game $\mathbf{G}_9$, so that does not change.

In Figure 6·21, we show the simulator $\mathcal{S}_{\mathsf{RFE}}$ for Π_{RFE}.

- Upon receiving ($\texttt{NewSession}, sid, \mathcal{P}_i$) from $\mathcal{F}^P_{\mathsf{RFE}}$ for honest party $\mathcal{P}_i$ when $\mathcal{P}_{1-i}$ is also honest, $\mathcal{S}_{\mathsf{RFE}}$ generates F_i, X_{1-i} using the obliviousness simulator for the garbling scheme, and sends those to $\mathcal{P}_{1-i}$.
- Upon receiving ($\texttt{NewSession}, sid, \mathcal{P}_i$) from $\mathcal{F}^P_{\mathsf{RFE}}$ for honest party $\mathcal{P}_i$ when $\mathcal{P}_{1-i}$ is corrupt, $\mathcal{S}_{\mathsf{RFE}}$ does the following:
  - $\mathcal{S}_{\mathsf{RFE}}$ postpones step 1.
  - In step 2, $\mathcal{S}_{\mathsf{RFE}}$:
    - Plays an OT sender as follows:
      - Waits for $\mathcal{P}_{1-i}$ to provide their select bits to the OT. As a result, $\mathcal{S}_{\mathsf{RFE}}$ learns the pass-string used by $\mathcal{P}_{1-i}$, pw'_{1-i}.
      - Makes a ($\texttt{TestPwd}, sid, \mathcal{P}_i$) query to $\mathcal{F}$. If $d(\mathsf{pw}_i, \mathsf{pw}'_{1-i}) \leq \delta$ (that is, the adversary has approximately guessed $\mathcal{P}_i$'s pass-string), $\mathcal{S}_{\mathsf{RFE}}$ learns pw_i, and sets $\mathsf{pw}'_i = \mathsf{pw}_i$. Otherwise, it sets pw'_i at random such that $d(\mathsf{pw}'_i, \mathsf{pw}'_{1-i}) > \delta$, and uses pw'_i in place of pw_i in the rest of the simulation.
      - If $d(\mathsf{pw}_i, \mathsf{pw}'_{1-i}) \leq \delta$ sets $y = 1$, and sets $y = 0$ otherwise.
      - Uses the privacy simulator for the garbling scheme to generate $(F_i, X_i, d_i) \leftarrow \mathcal{SIM}(1^\lambda, f, y)$.
      - Parses $(X_{i,i}, X_{i,1-i}) \leftarrow X_i$.
      - Sends $X_{i,1-i}$ to $\mathcal{P}_{1-i}$ as the OT output.
    - Plays an OT receiver honestly with pw'_i.
  - $\mathcal{S}_{\mathsf{RFE}}$ follows the instructions in Figure 6·5 for steps 3-8 with pw'_i.

Additionally, $\mathcal{S}_{\mathsf{RFE}}$ forwards all other instructions from $\mathcal{Z}$ to $\mathcal{A}$ and reports all output of $\mathcal{A}$ towards $\mathcal{Z}$. Instructions of corrupting a player are only obeyed if they are received before the protocol started, i.e., before $\mathcal{S}$ received any $\texttt{NewSession}$ query from $\mathcal{F}^P_{\mathsf{RFE}}$.

Figure 6·21: Simulator $\mathcal{S}_{\mathsf{RFE}}$ for Π_{RFE}

6.9 Appendix D: Proof that $s\mathcal{F}^P_{\mathsf{RFE}}$ is Enough to Realize $\mathcal{F}^P_{\mathsf{fPAKE}}$

In Figure 6·22, we describe a protocol $\mathsf{fPAKE}_{\mathsf{YGC}}$ which trivially realizes $\mathcal{F}^P_{\mathsf{fPAKE}}$ in the $s\mathcal{F}^P_{\mathsf{RFE}}$-hybrid model.

Upon receiving the input (sid, pw_i), $\mathcal{P}_i \in \{\mathcal{P}_0, \mathcal{P}_1\}$ does the following:

- Sends $(\texttt{Init}, sid)$ to $s\mathcal{F}^P_{\mathsf{RFE}}$;
- Sends $(\texttt{NewSession}, sid, \mathsf{pw}_i)$ to $s\mathcal{F}^P_{\mathsf{RFE}}$;
- Waits for k from $s\mathcal{F}^P_{\mathsf{RFE}}$, and
- Outputs k.

Figure 6·22: Protocol $\mathsf{fPAKE}_{\mathsf{YGC}}$ realizing $\mathcal{F}^P_{\mathsf{fPAKE}}$ in the $s\mathcal{F}^P_{\mathsf{RFE}}$-hybrid model.

Theorem 16. *Protocol $\mathsf{fPAKE}_{\mathsf{YGC}}$ realizes $\mathcal{F}^P_{\mathsf{fPAKE}}$ in the $s\mathcal{F}^P_{\mathsf{RFE}}$-hybrid model.*

Proof. For every efficient adversary $\mathcal{A}$, we describe a simulator $\mathcal{S}_{\mathsf{fPAKE}}$ in Figure 6·23 such that no efficient environment can distinguish an execution with the real protocol $\mathsf{fPAKE}_{\mathsf{YGC}}$ and $\mathcal{A}$ from an execution with the ideal functionality $\mathcal{F}^P_{\mathsf{fPAKE}}$ and $\mathcal{S}_{\mathsf{fPAKE}}$. Since the environment does not get any information about the honest parties except their output, all the simulator needs to do is respond to queries to $s\mathcal{F}^P_{\mathsf{RFE}}$. Since the

honest party does nothing except query the ideal functionality $s\mathcal{F}^{P}_{\mathsf{RFE}}$, and its output gets replaced by values chosen by $\mathcal{F}^{P}_{\mathsf{fPAKE}}$, there is nothing to simulate.

$\mathcal{S}_{\mathsf{fPAKE}}$ responds to queries to $s\mathcal{F}^{P}_{\mathsf{RFE}}$ as follows:

- Upon getting $(\texttt{Init}, sid)$ from $\mathcal{A}$ on behalf of corrupt party $\mathcal{P}_{1-i} \in \{\mathcal{P}_0, \mathcal{P}_1\}$, $\mathcal{S}_{\mathsf{fPAKE}}$ does nothing.
- Upon getting $(\texttt{Init}, sid, \mathcal{P}_i, H, sid_H)$ from $\mathcal{A}$, $\mathcal{S}_{\mathsf{fPAKE}}$ does nothing.
- Upon getting $(\texttt{NewSession}, sid, \mathsf{pw}_i)$ from $\mathcal{A}$ on behalf of honest party $\mathcal{P}_i \in \{\mathcal{P}_0, \mathcal{P}_1\}$, $\mathcal{S}_{\mathsf{fPAKE}}$ does nothing.
- Upon getting $(\texttt{NewSession}, sid, \mathsf{pw}'_{1-i})$ from $\mathcal{A}$ on behalf of corrupt party $\mathcal{P}_{1-i} \in \{\mathcal{P}_0, \mathcal{P}_1\}$, $\mathcal{S}_{\mathsf{fPAKE}}$:
  - Records pw'_{1-i};
  - Sends $(\texttt{TestPwd}, sid, \mathcal{P}_i, \mathsf{pw}'_{1-i})$ to $\mathcal{F}^{P}_{\mathsf{fPAKE}}$;
  - If $d(\mathsf{pw}_i, \mathsf{pw}'_{1-i}) \leq \delta$, $\mathcal{S}_{\mathsf{fPAKE}}$ learns pw_i.
- Upon getting a $(\texttt{TestPwd}, sid, \mathcal{P}_i)$ query from $\mathcal{A}$, $\mathcal{S}_{\mathsf{fPAKE}}$ responds with the output of the $\texttt{TestPwd}$ query above.
- Upon getting a $(\texttt{NewKey}, sid, \mathcal{P}_i, \mathsf{k}_i)$ query from $\mathcal{A}$, if $\mathcal{P}_i$ is corrupt, $\mathcal{S}_{\mathsf{fPAKE}}$ outputs k_i to $\mathcal{P}_i$. In any case, $\mathcal{S}_{\mathsf{fPAKE}}$ forwards $(\texttt{NewKey}, sid, \mathcal{P}_i, \mathsf{k}_i)$ to $\mathcal{F}^{P}_{\mathsf{fPAKE}}$.

Additionally, $\mathcal{S}_{\mathsf{fPAKE}}$ forwards all other instructions from $\mathcal{Z}$ to $\mathcal{A}$ and reports all output of $\mathcal{A}$ towards $\mathcal{Z}$. Instructions of corrupting a player are only obeyed if they are received before the protocol started, i.e., before $\mathcal{S}$ received any $\texttt{NewKey}$ query from $\mathcal{F}^{P}_{\mathsf{fPAKE}}$.

Figure 6·23: Simulator $\mathcal{S}_{\mathsf{fPAKE}}$ for $\mathsf{fPAKE}_{\mathsf{YGC}}$.

All that remains to show is that the values produced by $\mathcal{F}^{P}_{\mathsf{fPAKE}}$ and by $s\mathcal{F}^{P}_{\mathsf{RFE}}$ are identically distributed. We describe the outputs of $\mathcal{F}^{P}_{\mathsf{fPAKE}}$ and $s\mathcal{F}^{P}_{\mathsf{RFE}}$ in Figure 6·24. The table enumerates all possible cases in the functionalities. Cases in $s\mathcal{F}^{P}_{\mathsf{RFE}}$ are described in terms of the distances between pass-strings: between the honest parties' pass-strings if no man in the middle attack occurred, and between adversarial and honest pass-strings if it did. (If the adversary engaged one party but not the other, only one of the distances is filled in.) Cases in $\mathcal{F}^{P}_{\mathsf{fPAKE}}$ are described in terms of record markings ("fresh", "compromised" or "interrupted"). There is a one-to-one mapping between the cases in $s\mathcal{F}^{P}_{\mathsf{RFE}}$ and $\mathcal{F}^{P}_{\mathsf{fPAKE}}$ such that the outputs for the parties, whether they are honest or corrupt, are identically distributed. Those outputs are described in the last three columns of the table, as tuples of values the first of which is output to $\mathcal{P}_0$, and the second of which is output to $\mathcal{P}_1$. a, b are adversarially chosen values (which may or may not be distinct). r, s are independent, uniformly random values.

□

$s\mathcal{F}^P_{\mathsf{RFE}}$			$\mathcal{F}^P_{\mathsf{fPAKE}}$		outputs in both $s\mathcal{F}^P_{\mathsf{RFE}}$ and $\mathcal{F}^P_{\mathsf{fPAKE}}$		
distance between $\mathcal{P}_0$'s pass-string and $\mathcal{P}_1$'s pass-string $d(\mathsf{pw}_0, \mathsf{pw}_1)$, if MITM didn't happen	distance between $\mathcal{P}_0$'s pass-string and the adversary's $d(\mathsf{pw}_0, \mathsf{pw}'_1)$, if MITM happened	distance between $\mathcal{P}_1$'s pass-string and the adversary's $d(\mathsf{pw}'_0, \mathsf{pw}_1)$, if MITM happened	$\mathcal{P}_0$'s record	$\mathcal{P}_1$'s record	$\mathcal{P}_0$ and $\mathcal{P}_1$ honest	$\mathcal{P}_0$ honest, $\mathcal{P}_1$ corrupt	$\mathcal{P}_0$ and $\mathcal{P}_1$ corrupt
close ($\leq \delta$)			fresh	fresh	r, r	a, b	a, b
far ($> \delta$)			fresh	fresh	r, s	r, b	a, b
	close ($\leq \delta$)	close ($\leq \delta$)	compromised	compromised	a, b	a, b	a, b
	close ($\leq \delta$)	far ($> \delta$)	compromised	interrupted	a, s	a, b	a, b
	close ($\leq \delta$)		compromised	fresh	a, s	a, b	a, b
	far ($> \delta$)	close ($\leq \delta$)	interrupted	compromised	r, b	r, b	a, b
	far ($> \delta$)	far ($> \delta$)	interrupted	interrupted	r, s	r, b	a, b
	far ($> \delta$)		interrupted	fresh	r, s	r, b	a, b
		close ($\leq \delta$)	fresh	compromised	r, b	r, b	a, b
		far ($> \delta$)	fresh	interrupted	r, s	r, b	a, b

Figure 6·24: Output tables for $\mathcal{F}^P_{\mathsf{fPAKE}}$ and $s\mathcal{F}^P_{\mathsf{RFE}}$. r, s represent random outputs, while a, b represent adversarially chosen outputs.

The transformation of Barak *et al.* proceeds in two steps. First, links are initialized:

1. Each party generates a signing and verification key pair, and sends the verification key to its partner.
2. Each party then signs the key it receives and sends the signature back.
3. Each party verifies the signature it receives on its own verification key using the verification key it received; if the signature does not verify, it aborts.

Second, the parties run the protocol exactly as they would over authenticated channels, signing each message with their signing key, and verifying each signature they receive.

Applying this transformation adds (1) two rounds of communication, and (2) a hash operation and a signature operation for each message, assuming the hash-and-sign paradigm is used.

6.10 Appendix E: A Concrete OT

In this section, we recall a concrete UC-secure oblivious transfer protocol due to Chou and Orlandi (Chou and Orlandi, 2015). While they consider the general case of 1-out-of-n transfer, we only consider $n = 2$. Say m 1-out-of-2 OTs are performed. The

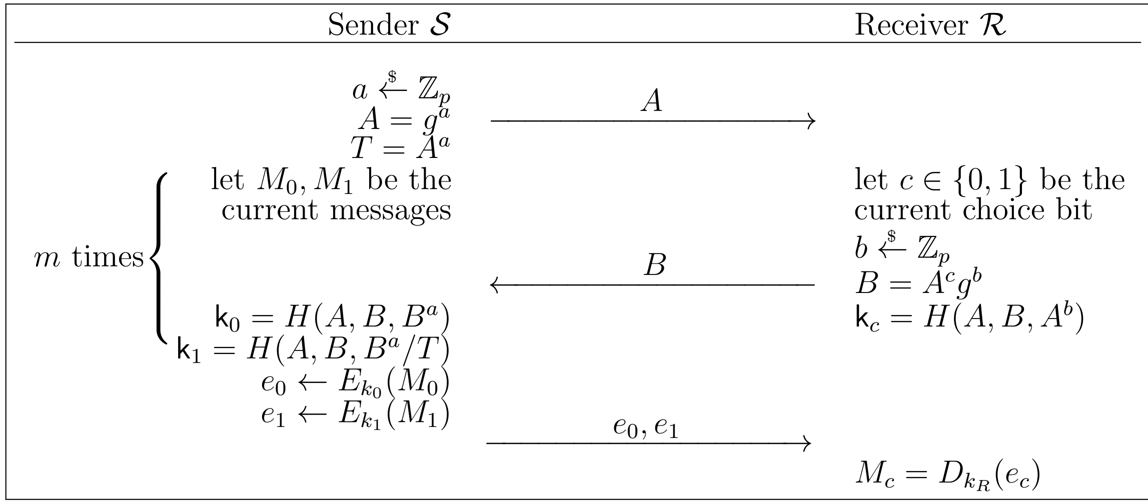


Figure 6·25: A Concrete OT (Chou and Orlandi, 2015) to be used in Π_{RFE}

protocol requires the sender to compute $m + 2$ exponentiations, and the receiver to compute $2m$ exponentiations, for a total of $3m + 2$ exponentiations. Figure 6·25 shows a summary of the protocol. Note that this construction does require a random oracle.

6.11 Appendix F: Proof of Theorem 13

We proceed in a series of games, starting with the real execution of the protocol and ending up with the ideal execution, with a simulator. To abbreviate notation, we skip all `role` tags used by $\mathcal{F}_{\ell\text{-iPAKE}}$ since they are not needed due to the symmetric layout of the protocol. For convenience, we refer to a query $(\texttt{NewKey}, sid, \mathcal{P}_i, \ell', \mathsf{k}_i)$ from the adversary $\mathcal{S}$ as *due* when:

- $\mathcal{P}_i$ is honest
- there is a `fresh` record of the form $(\mathcal{P}_i, \mathsf{pw}_i)$ in $\Lambda_{\mathcal{P}}$
- this is the first `NewKey` query for $\mathcal{P}_i$
- there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ in $\Lambda_{\mathcal{P}}$ with $\mathsf{pw}_i = \mathsf{pw}_{1-i}$ and $\mathcal{P}_{1-i}$ is honest
- a key k_{1-i} was sent to the other party, and $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ was `fresh` at the time.

The functionality $\mathcal{F}_{\ell\text{-iPAKE}}$ is parameterized by a security parameter λ and makes use of two initially empty lists $\Lambda_{\mathcal{P}}$ and $\Lambda_{\mathcal{L}}$, storing pass-strings and labels, respectively. It interacts with an adversary $\mathcal{S}$ and the (dummy) parties $\mathcal{P}_0$ and $\mathcal{P}_1$ via the following queries:

- **Upon receiving a query $(\texttt{NewSession}, sid, \mathsf{pw}_i, \texttt{role}, \ell)$ from party $\mathcal{P}_i$:**
  - Send $(\texttt{NewSession}, sid, \mathcal{P}_i, \texttt{role}, \ell)$ to $\mathcal{S}$;
  - If one of the following is true, record $(\mathcal{P}_i, \mathsf{pw}_i)$ in $\Lambda_{\mathcal{P}}$ and mark this record `fresh`, and record $(\mathcal{P}_i, \ell)$ in $\Lambda_{\mathcal{L}}$unless there already exists a record $(\mathcal{P}_i, \cdot)$ in $\Lambda_{\mathcal{L}}$.
    - This is the first `NewSession` query
    - This is the second `NewSession` query and there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$
- **Upon receiving a query $(\texttt{TestPwd}, sid, \mathcal{P}_i, \mathsf{pw}'_i, \ell')$ from $\mathcal{S}$:**
  If there is a `fresh` record $(\mathcal{P}_i, \mathsf{pw}_i)$ in $\Lambda_{\mathcal{P}}$, then do:
  - If $\mathsf{pw}_i = \mathsf{pw}'_i$, mark the record `compromised`; else mark it `interrupted`;
  - Record $(\mathcal{P}_{1-i}, \ell')$ in $\Lambda_{\mathcal{L}}$, possibly overwriting any existing record $(\mathcal{P}_{1-i}, \cdot)$.
- **Upon receiving a query $(\texttt{NewKey}, sid, \mathcal{P}_i, \mathsf{sk})$ from $\mathcal{S}$, where $|\mathsf{sk}| = \lambda$:**
  If there is a record $(\mathcal{P}_{1-i}, \ell)$ in $\Lambda_{\mathcal{L}}$, extract ℓ from it; otherwise set $\ell \leftarrow \perp$.
  If there is a record of the form $(\mathcal{P}_i, \mathsf{pw}_i)$ in $\Lambda_{\mathcal{P}}$, and this is the first `NewKey` query for $\mathcal{P}_i$, then:
  - If at least one of the following is true, then output (sid, ℓ, sk) to player $\mathcal{P}_i$:
    - The record is `compromised`
    - $\mathcal{P}_i$ is corrupted
    - The record is `fresh`, $\mathcal{P}_{1-i}$ is corrupted, and there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $\mathsf{pw}_{1-i} = \mathsf{pw}_i$
  - If this record is `fresh`, both parties are honest, there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $\mathsf{pw}_{1-i} = \mathsf{pw}_i$, a key sk' was sent to $\mathcal{P}_{1-i}$, and $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ was `fresh` at the time, then output $(sid, \ell, \mathsf{sk}')$ to $\mathcal{P}_i$;
  - In any other case, pick a new random key sk' of length λ and send $(sid, \ell, \mathsf{sk}')$ to $\mathcal{P}_i$.

  No matter what, mark the record $(\mathcal{P}_i, \mathsf{pw}_i)$ as `completed`.

Figure 6·26: Functionality $\mathcal{F}_{\ell\text{-iPAKE}}$

The parties $\mathcal{P}_0$ and $\mathcal{P}_1$ are running with $\mathcal{F}_{\mathsf{CRS}}, \mathcal{F}_{\mathsf{RO}}$ and $\mathcal{F}_{\mathsf{IC}}$.
Protocol Steps:

1. When a party $\mathcal{P}_i$, $i \in \{0,1\}$, receives an input $(\texttt{NewSession}, sid, \mathcal{P}_i, \mathsf{pw}_i, \ell)$ from $\mathcal{Z}$, it does the following:
   - chooses $x \xleftarrow{\$} \mathbb{F}_q$
   - sends $(sid, \texttt{crs})$ to $\mathcal{F}_{\mathsf{CRS}}$ and receives $(sid, (g, q))$ back
   - sends $(sid, \mathcal{E}, \mathsf{pw}_i || \ell, g^x)$ to $\mathcal{F}_{\mathsf{IC}}$ and receives (sid, X^*) back
   - sends (sid, ℓ, X^*) to $\mathcal{P}_{1-i}$ and waits for an answer
2. When $\mathcal{P}_i$, who already obtained an input $(\texttt{NewSession}, sid, \mathcal{P}_i, \mathsf{pw}_i, \ell)$ and thus holds $(x, (g, q), X^*)$, receives a message (sid, ℓ', Y^*) from $\mathcal{P}_{1-i}$, it
   - sends $(sid, \mathcal{D}, \mathsf{pw}_i || \ell, Y^*)$ to $\mathcal{F}_{\mathsf{IC}}$ and receives (sid, X') back
   - sends (sid, X^*, Y^*, X'^y) to $\mathcal{F}_{\mathsf{RO}}$ and receives (sid, k_i) back
   - outputs $(sid, \ell', \mathsf{k}_i)$ towards $\mathcal{Z}$ and terminates the session.

Figure 6·27: A UC Execution of EKE2. We skip the `role` tags since they are not needed due to the symmetric layout of the protocol.

Game $\mathbf{G}_0$: The real protocol execution. This is the real execution where the environment $\mathcal{Z}$ runs the EKE2 protocol (see Figure 6·27) with parties $\mathcal{P}_0$ and $\mathcal{P}_1$, both having access to ideal CRS, RO, and IC functionalities, and an adversary $\mathcal{A}$ that, w.l.o.g., is assumed to be the dummy adversary as shown in (Canetti, 2001, section 4.4.1).

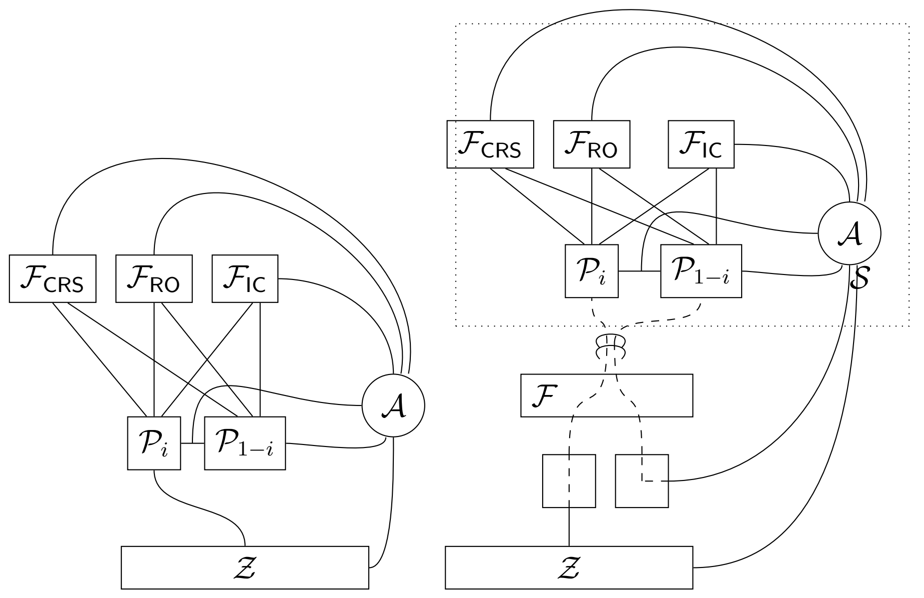


Figure 6·28: Transition from game $\mathbf{G}_0$ (left) to game $\mathbf{G}_1$ (right), showing a setting where $\mathcal{P}_{1-i}$ is corrupted.

Game $\mathbf{G}_1$: Modeling the ideal layout. We first regroup and create new machines, similar to Game 1 in the proof of Theorem 14. The new machine $\mathcal{S}$ executes the code of the CRS, RO and IC functionalities as depicted in Figures 6·13, 6·14, and 6·15.

Game $\mathbf{G}_2$: Simulating the ideal functionalities. We modify simulation of $\mathcal{F}_{\mathsf{RO}}$ and $\mathcal{F}_{\mathsf{IC}}$ as follows. We let $\mathcal{S}$ implement Figure 6·15 by maintaining a list Λ_{IC} with entries of the form $(k, m, \alpha, \mathcal{E}|\mathcal{D}, c)$. $\mathcal{S}$ handles encryption and decryption queries as follows:

- Upon receiving $(sid, \mathcal{E}, \mathsf{k}, m)$ (for shortness of notation, we will also write $\mathcal{E}_{\mathsf{k}}(m)$ for this query), if $\mathsf{k} \notin \mathbb{F}_p$ or $m \notin \mathbb{G}$ then abort. Else, if there is an entry $(\mathsf{k}, m, *, *, c)$ in Λ_{IC}, $\mathcal{S}$ replies with (sid, c). Else, $\mathcal{S}$ chooses $c \stackrel{\$}{\leftarrow} \mathbb{G} \setminus \{1\}$. If there already is a record $(*, *, *, *, *, c)$ in Λ_{IC}, $\mathcal{S}$ aborts. Else, $\mathcal{S}$ adds $(\mathsf{k}, m, \perp, \mathcal{E}, c)$ to Λ_{IC} and replies with (sid, c).
- Upon receiving $(sid, \mathcal{D}, \mathsf{k}, c)$ (or $\mathcal{D}_{\mathsf{k}}(c)$, for short), if $\mathsf{k} \notin \mathbb{F}_p$ or $c \notin \mathbb{G}$ then abort. Else, if there is an entry $(\mathsf{k}, m, *, *, c)$ in Λ_{IC}, $\mathcal{S}$ replies with (sid, m). Else, $\mathcal{S}$ chooses $\alpha \leftarrow \mathbb{F}_q^*$. If there already is a record $(*, *, g^\alpha, *, *, *)$ in Λ_{IC}, $\mathcal{S}$ aborts. Else, $\mathcal{S}$ adds $(\mathsf{k}, g^\alpha, \alpha, \mathcal{D}, c)$ to Λ_{IC} and replies with (sid, g^α).

Similarly, let Λ_{RO} denote the list that $\mathcal{S}$ maintains upon implementing Figure 6·14, containing entries of the form (m, h). We let $\mathcal{S}$ handle queries to $\mathcal{F}_{\mathsf{RO}}$ as follows:

- Upon receiving $H(m)$, if $m \notin \{0,1\}^* \times \{0,1\}^* \times \mathbb{G}^3$, then abort. Else, if there is an entry (m, h) in Λ_{RO}, $\mathcal{S}$ replies with (sid, h). Else, $\mathcal{S}$ chooses $h \stackrel{\$}{\leftarrow} \{0,1\}^k$. If there already is a record $(*, *, h)$ in Λ_{RO}, $\mathcal{S}$ aborts. Else, $\mathcal{S}$ adds (m, h) to Λ_{RO} and replies with (sid, h).

These modifications later allow $\mathcal{S}$ to extract unique inputs from values obtained from the two functionalities. Especially, note that Λ_{IC} will never contain $(*, \mathsf{k}_i, *, *, \mathcal{E}, c)$, $(*, \mathsf{k}_{1-i}, *, *, \mathcal{E}, c)$ with $\mathsf{k}_i \neq \mathsf{k}_{1-i}$. The entry α serves $\mathcal{S}$ as a trapdoor for solving discrete-log type problems.

Since q is greater than 2^λ, if the oracles are only queried a polynomial number of times, the birthday problem states that game $\mathbf{G}_1$ and game $\mathbf{G}_2$ are indistinguishable with probability overwhelming in λ.

Game $\mathbf{G}_3$: Building $\mathcal{F}_{\mathsf{iPAKE}}$. In this game, we start modeling $\mathcal{F}_{\ell\text{-}\mathsf{iPAKE}}$. First, we let $\mathcal{F}$ maintain two initially empty lists: $\Lambda_{\mathcal{P}}$, a list of tuples of the form

$(\mathcal{P}_i, \mathsf{pw}_i)$ and $\Lambda_{\mathcal{L}}$, a list of tuples of the form $(\mathcal{P}_i, \ell)$. Upon receiving a query $(\texttt{NewSession}, sid, \mathsf{pw}_i, \ell)$ from (dummy) party $\mathcal{P}_i$, if this is the first NewSession query, or if this is the second NewSession query and there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i}, \ell')$, then $\mathcal{F}$ records $(\mathcal{P}_i, \mathsf{pw}_i)$ in $\Lambda_{\mathcal{P}}$ and marks this record as fresh. If $\Lambda_{\mathcal{L}}$ does not contain any record $(\mathcal{P}_i, \cdot)$ so far, $\mathcal{F}$ also records $(\mathcal{P}_i, \ell)$ in $\Lambda_{\mathcal{L}}$. Then, $\mathcal{F}$ relays the query $(\texttt{NewSession}, sid, \mathsf{pw}_i, \ell)$ to $\mathcal{S}$. Now that $\mathcal{F}$ knows about pass-strings and labels, we can add a TestPwd interface to $\mathcal{F}$ as described in Figure 6·8. We let $\mathcal{S}$ parse outputs $(sid, \ell', \mathsf{k}_i)$ towards $\mathcal{F}$ to be of the form $(\texttt{NewKey}, sid, \mathcal{P}_i, \ell', \mathsf{k}_i)$ by adding the NewKey tag and the name of the party who produced the output. Additionally, we let $\mathcal{F}$ translate this back to $(sid, \ell', \mathsf{k}_i)$ and send it to $\mathcal{Z}$ via the dummy party $\mathcal{P}_i$, marking the corresponding record as completed.

None of these modifications changes the output towards $\mathcal{Z}$ compared to the previous game $\mathbf{G}_2$.

Game $\mathbf{G}_4$: $\mathcal{F}$ generates a random session key for an honest, interrupted session. Upon receiving a query $(\texttt{NewKey}, sid, \mathcal{P}_i, \mathsf{k}_i)$ from $\mathcal{S}$, if $\mathcal{P}_i$ is not corrupted and there is a record of the form $(\mathcal{P}_i, \mathsf{pw}_i)$ that is marked as interrupted, and this is the first NewKey query for $\mathcal{P}_i$, we let $\mathcal{F}$ choose a random session key k^* of length λ. Additionally, $\mathcal{F}$ derives the label as follows: if there is a record $(\mathcal{P}_{1-i}, \ell^*)$ in $\Lambda_{\mathcal{L}}$, extract ℓ^* from it; otherwise, set $\ell^* \leftarrow \perp$. Then, $\mathcal{F}$ outputs $(sid, \ell^*, \mathsf{k}^*)$ to P.

If there is no such interrupted record, $\mathcal{F}$ continues to relay k_i and ℓ'.

Since the simulators described in game $\mathbf{G}_3$ and game $\mathbf{G}_4$ do not make use of the TestPwd interface, none of the records of $\mathcal{F}$ are marked as interrupted and thus the output towards $\mathcal{Z}$ is equally distributed in both games.

Game $\mathbf{G}_5$: $\mathcal{S}$ handles dictionary attacks against the client $\mathcal{P}_0$ using the

$\texttt{TestPwd}$ **interface.** In this game, we will only change the simulation. First note that the client, the initiator of the protocol, is intended to send the first message, and we call him $\mathcal{P}_0$. If both $\mathcal{P}_0$ and $\mathcal{P}_1$ are honest, $\mathcal{P}_0$ obtained input and $\mathcal{Z}$ advises $\mathcal{A}$ to substitute (sid, ℓ', Y^*) with $(sid, \ell_{\mathcal{Z}}, Y^*_{\mathcal{Z}})$, or if $\mathcal{P}_1$ is corrupted and produces $(sid, \ell_{\mathcal{Z}}, Y^*_{\mathcal{Z}})$ as first flow, then $\mathcal{S}$ will proceed simulation of $\mathcal{P}_0$ using $\ell_{\mathcal{Z}}$ and $Y^*_{\mathcal{Z}}$. In this situation, we modify $\mathcal{S}$ as follows: upon receiving $(sid, \ell_{\mathcal{Z}}, Y^*_{\mathcal{Z}})$, if there is an entry $(\mathsf{pw}_{\mathcal{Z}}||\hat{\ell}, *, *, \mathcal{E}, Y^*_{\mathcal{Z}})$ for any $\hat{\ell} \in \mathcal{L}$ in Λ_{IC}[8], the simulator asks a $\texttt{TestPwd}$ query $(\texttt{TestPwd}, sid, \mathcal{P}_0, \mathsf{pw}_{\mathcal{Z}}, \ell_{\mathcal{Z}})$ to $\mathcal{F}$. $\mathcal{S}$ then proceeds the simulation using $\mathsf{pw}_{\mathcal{Z}}$ and $\ell_{\mathcal{Z}}$ instead of pw_0 and ℓ'[9]. If there is no entry $(\mathsf{pw}_{\mathcal{Z}}||\hat{\ell}, *, *, \mathcal{E}, Y^*_{\mathcal{Z}})$ in Λ_{IC}, $\mathcal{S}$ sends $(\texttt{TestPwd}, sid, \mathcal{P}_0, \mathsf{pw}_0, \ell_{\mathcal{Z}})$ to $\mathcal{F}$[10].

Regarding the label, observe that $\mathcal{S}$'s $\texttt{NewKey}$ query will contain $\ell_{\mathcal{Z}}$ which was contained in the output of the honest $\mathcal{P}_0$ (cf. Figure 6·27). Since $\texttt{TestPwd}$ queries overwrite any existing labels, there will be an entry $(\mathcal{P}_1, \ell_{\mathcal{Z}})$ in $\Lambda_{\mathcal{L}}$ and thus, regarding the label, the output towards $\mathcal{Z}$ does not change compared to the previous game. Regarding the session key, we have to analyze different cases depending on whether $Y^*_{\mathcal{Z}}$ was generated using $\mathcal{F}_{\mathsf{IC}}$ or not. However, observe that the only changes of session keys between this and the previous game occur whenever a $\texttt{TestPwd}$ query of $\mathcal{S}$ causes a record to be marked as $\texttt{interrupted}$.

- There is an entry $(\mathsf{pw}_{\mathcal{Z}}||\hat{\ell}, *, *, \mathcal{E}, Y^*_{\mathcal{Z}})$ in Λ_{IC}: if $\mathsf{pw}_{\mathcal{Z}} = \mathsf{pw}_0$, the record is marked $\texttt{compromised}$ and the session key is not changed by $\mathcal{F}$. If $\mathsf{pw}_{\mathcal{Z}} \neq \mathsf{pw}_0$, on the other hand, the record is marked $\texttt{interrupted}$ and $\mathcal{F}$ hands out a random session key, as opposed to game $\mathbf{G}_4$. However, since the session key is distributed as before, $\mathcal{Z}$ can only detect this by reproducing

[8] This entry is unique due to simulation of $\mathcal{F}_{\mathsf{IC}}$ as described in game $\mathbf{G}_2$.

[9] Note that, since $\mathcal{F}$ does not leak any information at this point, $\mathcal{S}$ cannot depend on the outcome of a $\texttt{TestPwd}$ query.

[10] Letting $\mathcal{S}$ guess a pass-string that he actually knows seems a little artificial. Indeed, the simulation in this case will be changed in game $\mathbf{G}_{12}$ when $\mathcal{S}$ becomes oblivious of $\mathcal{P}_0$'s pass-string.

$\mathcal{P}_0$'s input $(sid, X^*, Y^*_{\mathcal{Z}}, \mathsf{CDH}(\mathcal{D}_{\mathsf{pw}_0||\ell}(X^*), \mathcal{D}_{\mathsf{pw}_0||\ell_{\mathcal{Z}}}(Y^*_{\mathcal{Z}}))$ to $\mathcal{F}_{\mathsf{RO}}$. Lemma 6 (see below) shows indistinguishability of game $\mathbf{G}_4$ and game $\mathbf{G}_5$.

- There is no entry $(*, *, *, \mathcal{E}, Y^*_{\mathcal{Z}})$ in Λ_{IC}: since the `TestPwd` query will result in a compromised record, the modified simulation has no impact on the output towards $\mathcal{Z}$ in this case.

The following lemma bounds the probability that an unsuccessful dictionary attack leads to a non-random looking session key. Since in this case the labels do not play any role (the encryption keys of the form $pass - string||label$ will not match regardless of the labels), we ignore them for the sake of simplicity.

Lemma 6. *If CDH holds in $\mathbb{G}$, then $\forall \mathsf{pw}_0, Y^*_{\mathcal{Z}} \leftarrow \mathcal{Z}$, where $Y^*_{\mathcal{Z}}$ is a ciphertext generated through $\mathcal{F}_{\mathsf{IC}}$ with some key $\mathsf{pw}_{\mathcal{Z}} \neq \mathsf{pw}_0$, $\mathsf{pw}_0 \in \mathbb{F}_p$, it holds that*

$$\Pr_{\mathbf{G}_5}[CDH(\mathcal{D}_{\mathsf{pw}_0}(X^*), \mathcal{D}_{\mathsf{pw}_0}(Y^*_{\mathcal{Z}})) \leftarrow \mathcal{Z}(X^*)] = negl(\lambda).$$

Proof. We create an attacker $\mathcal{B}_{\mathsf{CDH}}$ given a CDH instance $(g, A = g^a, B = g^b)$. $\mathcal{B}_{\mathsf{CDH}}$ runs $\mathcal{Z}$ simulating game $\mathbf{G}_5$ as follows: $\mathcal{B}_{\mathsf{CDH}}$ internally runs all of the participating machines, i.e. $\mathcal{S}$, $\mathcal{F}$ and the dummy parties as in game $\mathbf{G}_5$, but with some modifications. First, $\mathcal{B}_{\mathsf{CDH}}$ computes $X^* \leftarrow \mathcal{E}_{\mathsf{pw}_0}(A)$ and updates Λ_{IC} accordingly, aborting if there already was an entry $(\mathsf{pw}_0, A, *, \mathcal{E}, *)$. Upon receiving a query $\mathcal{D}_{\mathsf{pw}_0}(Y^*_{\mathcal{Z}})$, $\mathcal{B}_{\mathsf{CDH}}$ again aborts if there already is an entry $(\mathsf{pw}_0, *, *, \mathcal{E}, Y^*_{\mathcal{Z}})$. Otherwise, it draws $\beta \xleftarrow{\$} \mathbb{F}_q$ and sets the answer to this query to be Bg^{β}. This can happen multiple times (for different $Y^*_{\mathcal{Z}}$), and $\mathcal{B}_{\mathsf{CDH}}$ keeps track of the pairs $(\beta, Y^*_{\mathcal{Z}})$ in a list Λ_{CDH}. The last modification concerns the part of the simulator's code of game $\mathbf{G}_5$ where a value $Z \leftarrow \mathcal{D}_{\mathsf{pw}_0}(Y^*)^a$ needs to be computed, but note that $\mathcal{B}_{\mathsf{CDH}}$ does not know a. Instead, $\mathcal{B}_{\mathsf{CDH}}$ just sets $Z \leftarrow \perp$.

Finally, $\mathcal{B}_{\mathsf{CDH}}$ picks a random entry from Λ_{RO} asked by the environment, parses it as $((\mathsf{pw}_0, X^*, Y^*_{\mathcal{Z}}, Z), h)$, looks for an entry $(\beta, Y^*_{\mathcal{Z}})$ in Λ_{CDH} and outputs $Z/(g^a)^{\beta}$ as a CDH solution.

First, note that if $\mathcal{B}_{\mathsf{CDH}}$ does not abort, it perfectly emulates $\mathcal{Z}$'s view in game $\mathbf{G}_5$, since A, Bg^{β} are random in $\mathbb{G}$ and the record for $\mathcal{P}_0$ will be interrupted, which means that $\mathcal{F}$ will output a random session key for $\mathcal{P}_0$, overwriting

$\perp$. Second, $\mathcal{B}_{\mathsf{CDH}}$ only has to abort if there is a collision upon choosing random values from $\mathbb{G}$.

Assume that $\mathcal{Z}$ outputs $\mathsf{CDH}(\mathcal{D}_{\mathsf{pw}_0}(X^*), \mathcal{D}_{\mathsf{pw}_0}(Y^*_{\mathcal{Z}}))$ with non-negligible probability. This is only possible if $\mathcal{Z}$ asked both corresponding decryption queries. Existence of $(\mathsf{pw}_{\mathcal{Z}}, *, *, \mathcal{E}, Y^*_{\mathcal{Z}})$ with $\mathsf{pw}_{\mathcal{Z}} \neq \mathsf{pw}_0$ in Λ_{IC} ensures that the answer to $\mathcal{D}_{\mathsf{pw}_0}(Y^*_{\mathcal{Z}})$ can be chosen by $\mathcal{B}_{\mathsf{CDH}}$ as described above. Thus, $\mathcal{B}_{\mathsf{CDH}}$ finds a correct CDH solution with non-negligible probability $1/q_{\mathcal{Z}}$, where $q_{\mathcal{Z}}$ is the number of hash queries issued by $\mathcal{Z}$. □

Game $\mathbf{G}_6$: $\mathcal{S}$ handles dictionary attacks against the server $\mathcal{P}_1$ using the `TestPwd` interface. Again, in this game we only change the simulation. Analogously to game $\mathbf{G}_5$, we let $\mathcal{S}$ use the `TestPwd` interface upon receiving adversarially generated $X^*_{\mathcal{Z}}, \ell_{\mathcal{Z}}$ upon simulating $\mathcal{P}_1$. Observe that the only difference is due to the order of flows: if $\mathcal{S}$ extracts an incorrect pass-string, he produces Y^* using this wrong pass-string. However, Y^* will be distributed as before and again, $\mathcal{Z}$ can only detect the change by reproducing $\mathcal{P}_1$'s input to $\mathcal{F}_{\mathsf{RO}}$, namely $(sid, X^*_{\mathcal{Z}}, Y^*, \mathsf{CDH}(\mathcal{D}_{\mathsf{pw}_1||\ell_{\mathcal{Z}}}(X^*_{\mathcal{Z}}), \mathcal{D}_{\mathsf{pw}_1||\ell'}(Y^*))$.

Using an analogous argument to Lemma 6, indistinguishability from game $\mathbf{G}_5$ follows from the hardness of CDH in $\mathbb{G}$.

Game $\mathbf{G}_7$: $\mathcal{F}$ aligns session keys. Upon receiving a query $(\texttt{NewKey}, sid, \mathcal{P}_i, \ell', \mathsf{k}_i)$ from $\mathcal{S}$ for a session, if this query is *due* then output $(sid, \ell^*, \mathsf{k}^*)$ to $\mathcal{P}_i$ where k^* is the session key that was formerly sent to the other party and the label ℓ^* is derived as usual: if there is a record $(\mathcal{P}_{1-i}, \ell^*)$ in $\Lambda_{\mathcal{L}}$, extract ℓ^* from it; otherwise, set $\ell^* \leftarrow \perp$.

We now analyze distinguishability of this game from game $\mathbf{G}_6$. If $\mathcal{Z}$ tampered with the transcript, any player that received a modified message will not have a fresh record anymore (cf. simulation described in games $\mathbf{G}_5$ and $\mathbf{G}_6$) and the output of this player towards $\mathcal{Z}$ is not changed in this game. On the other hand,

if $\mathcal{Z}$ does not advise $\mathcal{A}$ to tamper with any message, $\mathcal{F}$ did not overwrite any labels and thus $\ell^* = \ell'$. Additionally, perfect correctness of the EKE2 protocol ensures that, in case of a due record, $\mathsf{k}_i = \mathsf{k}^*$.

Note that $\mathcal{F}$ still differs from the functionality $\mathcal{F}_{\ell\text{-iPAKE}}$ described in Figure 6·8 in some aspects. First, it does not output randomly generated session keys towards $\mathcal{Z}$ for honest sessions. Furthermore, it reports all pass-strings to $\mathcal{S}$. We will take care of these remaining differences in the next games.

Game $\mathbf{G}_8$: In some cases, $\mathcal{F}$ generates a random session key when the other party is corrupted. Upon receiving a `NewKey` query (`NewKey`, $sid, \mathcal{P}_i, \ell'$, k_i) from $\mathcal{S}$, if there is a fresh record of the form $(\mathcal{P}_i, \mathsf{pw}_i)$ in $\Lambda_{\mathcal{P}}$, and this is the first `NewKey` query for $\mathcal{P}_i$, $\mathcal{P}_i$ is honest and $\mathcal{P}_{1-i}$ corrupted and there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ in $\Lambda_{\mathcal{P}}$ with $\mathsf{pw}_i \neq \mathsf{pw}_{1-i}$, we let $\mathcal{F}$ pick a new random key k^* of length λ and send $(sid, \ell^*, \mathsf{k}^*)$ to $\mathcal{P}_i$, where ℓ^*, as usual, is taken from the list $\Lambda_{\mathcal{L}}$ or set to be $\perp$.

The simulation ensures that the record $(\mathcal{P}_i, \mathsf{pw}_i)$ is either compromised or interrupted (cf. description of the simulator in games $\mathbf{G}_5$ and $\mathbf{G}_6$). Thus, the modification has no effect since it only concerns fresh records.

Game $\mathbf{G}_9$: $\mathcal{F}$ generates a random session key for an honest session. Upon receiving a `NewKey` query (`NewKey`, $sid, \mathcal{P}_i, \ell', \mathsf{k}_i$) from $\mathcal{S}$, if there is a fresh record of the form $(\mathcal{P}_i, \mathsf{pw}_i)$ in $\Lambda_{\mathcal{P}}$, and this is the first `NewKey` query for $\mathcal{P}_i$, both parties are honest and the `NewKey` query is not *due*, we let $\mathcal{F}$ pick a new random key k^* of length k and send $(sid, \ell^*, \mathsf{k}^*)$ to $\mathcal{P}_i$, where ℓ^*, as usual, is taken from the list $\Lambda_{\mathcal{L}}$ or set to be $\perp$.

In other words, $\mathcal{F}$ now generates a random session key upon a first `NewKey` query for an honest party $\mathcal{P}_i$ with fresh record $(\mathcal{P}_i, \mathsf{pw}_i)$ where the other party

is also honest, if (at least) one of the following events happens:

1. There is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ in $\Lambda_{\mathcal{P}}$ with $\mathsf{pw}_i \neq \mathsf{pw}_{1-i}$;
2. No output was sent to the other party yet;
3. If there was output to the other party, the record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ in $\Lambda_{\mathcal{P}}$ was not fresh and thus interrupted or compromised at that time

In all of these cases, $\mathcal{S}$ chose a fresh k_i following a uniform distribution and ℓ' was contained in the `NewSession` query of $\mathcal{P}_i$'s partner, thus $\ell' = \ell^*$. Regarding the session key, $\mathcal{Z}$ can only notice a difference if it reproduces k_i by computing $\mathcal{P}_i$'s input $(sid, X^*, Y^*, \mathsf{CDH}(\mathcal{D}_{\mathsf{pw}_i||\ell}(X^*), \mathcal{D}_{\mathsf{pw}_i||\ell'}(Y^*))$ to $\mathcal{F}_{\mathsf{RO}}$ and sending it to $\mathcal{F}_{\mathsf{RO}}$ via the adversary $\mathcal{A}$.

The following lemma bounds the probability that a session key of an unattacked session does not look random.

Lemma 7. *If CDH holds in $\mathbb{G}$, then $\forall \mathsf{pw}, \ell, \ell' \leftarrow \mathcal{Z}$ with $\mathsf{pw} \in \mathbb{F}_p$ and $\ell, \ell' \in \mathcal{L}$ it holds that*

$$\Pr_{\boldsymbol{G}_9}[\mathit{CDH}(\mathcal{D}_{\mathsf{pw}||\ell}(X^*), \mathcal{D}_{\mathsf{pw}||\ell'}(Y^*)) \leftarrow \mathcal{Z}(X^*, Y^*)] = negl(\lambda).$$

Proof. We only sketch the proof since it is similar to the proof of Lemma 6. Namely, the strategy of embedding (randomized versions of) a CDH challenge into the simulation of game $\mathbf{G}_9$ is done by just encrypting both CDH challenge elements to obtain X^* and Y^*. For the final argument, note that $\perp$ is not seen by $\mathcal{Z}$ since it is either replaced using a random session key or a previously computed key. □

It follows that game $\mathbf{G}_8$ and game $\mathbf{G}_9$ are indistinguishable.

Game $\mathbf{G}_{10}$: $\mathcal{F}$ always takes all labels from the list $\Lambda_{\mathcal{L}}$. We modify $\mathcal{F}$ as follows: if $\mathcal{F}$ outputs $(sid, \ell', \mathsf{k}_i)$ towards $\mathcal{P}_i$ where ℓ', k_i are taken from a query $(\mathtt{NewKey}, sid, \mathcal{P}_i, \ell', \mathsf{k}_i)$ from $\mathcal{S}$, $\mathcal{F}$ extracts ℓ^* from a record $(\mathcal{P}_{1-i}, \ell^*)$ in $\Lambda_{\mathcal{L}}$ or

sets $\ell^* \leftarrow \perp$ if such a record does not exist. $\mathcal{F}$ then outputs $(sid, \ell^*, \mathsf{k}_i)$ towards $\mathcal{P}_i$. We additionally modify $\mathcal{S}$ to remove the labels from the NewKey queries altogether.

First observe that we can remove the labels from the NewKey queries because, in this and the past games, we ensured that $\mathcal{F}$ now does not access this label anymore. However, we still have to argue indistinguishability of the current and the previous game. The cases where k_i of $\mathcal{S}$ is relayed by $\mathcal{F}$ towards $\mathcal{P}_i$ are the following:

- $\mathcal{P}_i$ has a compromised record
- $\mathcal{P}_i$ is corrupted
- $\mathcal{P}_i$ has a fresh record, its partner is corrupted and has a record with a matching pass-string

In the first case, we have that $\ell' = \ell^*$ since the label ℓ' outputted by $\mathcal{P}_i$ was also contained in a TestPwd query by $\mathcal{S}$ and overwrote any existing label send by $\mathcal{P}_i$'s partner. For the second case, observe that since we restrict to static corruption, corrupted players will not have records in $\Lambda_{\mathcal{P}}$ and thus this case will never happen. In the third case, corruption of the partner ensures that $\mathcal{S}$ issued a TestPwd query which overwrote any existing label with ℓ', so $\ell' = \ell^*$ as well.

Observe that now $\mathcal{F}$ acts like $\mathcal{F}_{\ell\text{-iPAKE}}$ regarding the output of session keys and labels. The only remaining difference is that the NewSession queries still contain the pass-strings of the parties. In the next games, we will make the simulation independent of these pass-strings.

Game $\mathbf{G}_{11}$: Simulate without pw_1 if server $\mathcal{P}_1$ is honest. In case of receiving a $(\texttt{NewSession}, sid, \mathsf{pw}_1, \ell')$ from an honest $\mathcal{P}_1$ playing the role of a server,

we modify $\mathcal{F}$ by forwarding only $(\texttt{NewSession}, sid, \mathcal{P}_1, \ell')$ to $\mathcal{S}$. We now have to modify $\mathcal{S}$ to proceed simulation without knowing pw_1. Upon receiving $(\texttt{NewSession}, sid, \mathcal{P}_1, \ell')$ from $\mathcal{F}$ for an honest $\mathcal{P}_1$, we let $\mathcal{S}$ draw uniformly at random a "dummy" pass-string $\mathsf{pw}_\mathcal{S}$ and proceed the simulation of $\mathcal{P}_1$ using $\mathsf{pw}_\mathcal{S}$ as a pass-string.

We first note that, if at any time $\mathcal{S}$ sends a $\texttt{NewKey}$ query to $\mathcal{F}$ containing a session key k_1 for $\mathcal{P}_1$, this session key is only seen by $\mathcal{Z}$ if $\mathcal{P}_1$'s record is compromised. Otherwise, we thus only have to argue indistinguishability of the transcripts of game $\mathbf{G}_{10}$ and game $\mathbf{G}_{11}$.

- $\mathcal{Z}$ sends $\ell_\mathcal{Z}, X^*_\mathcal{Z}$, there is a record $(\mathsf{pw}_\mathcal{Z}||\hat{\ell}, X, *, \mathcal{E}, X^*)$ in Λ_{IC} for some $\hat{\ell} \in \mathcal{L}$ and $\mathsf{pw}_\mathcal{Z} = \mathsf{pw}_1$: since $\mathcal{S}$ will issue a $\texttt{TestPwd}$ query that will result in a compromised record (cf. simulation described in game $\mathbf{G}_6$), nothing is changed since $\mathsf{pw}_\mathcal{S}$ was never used, and Y^* is generated using the correct pass-string $\mathsf{pw}_\mathcal{Z}$.
- $\mathcal{Z}$ sends $\ell_\mathcal{Z}, X^*_\mathcal{Z}$, there is a record $(\mathsf{pw}_\mathcal{Z}||\hat{\ell}, X, *, \mathcal{E}, X^*)$ in Λ_{IC} for some $\hat{\ell} \in \mathcal{L}$ and $\mathsf{pw}_\mathcal{Z} \neq \mathsf{pw}_1$: since $\mathcal{P}_1$ will receive a random session key from $\mathcal{F}$ in this case (the record will be marked $\texttt{interrupted}$), we only have to argue indistinguishability of Y^* generated with $\mathsf{pw}_\mathcal{Z}||\ell'$ instead of $\mathsf{pw}_1||\ell'$. Simulation of $\mathcal{F}_{\mathsf{IC}}$ ensures that Y^* is distributed uniformly random as before. Observe that here it is crucial that even for a corrupted session, an interrupted record lets the functionality hand out a random session key, since $\mathcal{S}$ has no means to decide whether it has to output a session key for $\mathcal{P}_1$ that matches the session key that $\mathcal{Z}$ can compute on behalf of $\mathcal{P}_0$ from the message (ℓ', Y^*).
- $\mathcal{Z}$ sends $\ell_\mathcal{Z}, X^*_\mathcal{Z}$ and no $\mathcal{E}$ record: the simulation described in game $\mathbf{G}_6$ tells $\mathcal{S}$ to issue a $\texttt{TestPwd}$ query, but now using $\mathsf{pw}_\mathcal{S}$ instead of pw_1. If, coinci-

dentally, $\mathsf{pw}_1 = \mathsf{pw}_{\mathcal{S}}$, nothing changes. On the other hand, if $\mathsf{pw}_1 \neq \mathsf{pw}_{\mathcal{S}}$, $\mathcal{P}_1$ obtains a random session key from $\mathcal{F}$ as opposed to the game before and Y^* is created using $\mathsf{pw}_{\mathcal{S}}||\ell'$ instead of $\mathsf{pw}_1||\ell'$. This can only be detected if $\mathcal{Z}$ reproduces $\mathcal{P}_1$'s input to $\mathcal{F}_{\mathsf{RO}}$ from game $\mathbf{G}_{10}$, which happens only with negligible probability according to Lemma 8 (see below).

- both parties honest and no injections: $\mathcal{P}_1$ will obtain a uniformly random session key from $\mathcal{F}$ in this case, and thus the only difference is that Y^* was created using $\mathsf{pw}_{\mathcal{S}}||\ell'$ instead of $\mathsf{pw}_1||\ell'$. Again, this is indistinguishable since Y^* is distributed exactly as before.

The following lemma bounds the probability that an injected X^* that was not obtained using encryption leads to a non-random looking session key.

Lemma 8. *If CDH holds in $\mathbb{G}$, then $\forall \mathsf{pw}_1, \ell', \ell_{\mathcal{Z}}, X^*_{\mathcal{Z}} \leftarrow \mathcal{Z}$, where $X^*_{\mathcal{Z}}$ was not generated using $\mathcal{F}_{\mathsf{IC}}$, $\ell', \ell_{\mathcal{Z}} \in \mathcal{L}$ and $\mathsf{pw}_1 \in \mathbb{F}_p$, it holds that*

$$\Pr_{\boldsymbol{G}_{11}}[\mathit{CDH}(\mathcal{D}_{\mathsf{pw}_1||\ell_{\mathcal{Z}}}(X^*_{\mathcal{Z}}), \mathcal{D}_{\mathsf{pw}_1||\ell'}(Y^*)) \leftarrow \mathcal{Z}(Y^*)] = negl(\lambda).$$

Proof. Note that the only difference to Lemma 6 is that this time, no record $(*, *, *, \mathcal{E}, X^*_{\mathcal{Z}})$ exists so the fact that $\mathcal{B}_{\mathsf{CDH}}$ is able to embed an element of its CDH challenge into $\mathcal{D}_{\mathsf{pw}_1}(X^*_{\mathcal{Z}})$ is even more obvious. The rest of the proof is analogously to Lemma 6. □

Game $\mathbf{G}_{12}$: Simulate without pw_0 if client $\mathcal{P}_0$ is honest. In a similar fashion, we now let $\mathcal{F}$ cut the pass-string from `NewSession` queries to an honest $\mathcal{P}_0$. We again have to modify $\mathcal{S}$ to proceed simulation without knowing pw_0. Upon receiving $(\texttt{NewSession}, sid, \mathcal{P}_0, \ell)$ from $\mathcal{F}$ for an honest $\mathcal{P}_0$, we let $\mathcal{S}$ draw uniformly at random a "dummy" pass-string $\mathsf{pw}_{\mathcal{S}}$. $\mathcal{S}$ proceeds the simulation of $\mathcal{P}_0$ using $\mathsf{pw}_{\mathcal{S}}$ as a pass-string.

Additionally, we further change $\mathcal{S}$ in case of a dictionary attack against client $\mathcal{P}_0$, i.e., upon receiving $\ell_{\mathcal{Z}}, Y^*_{\mathcal{Z}}$ from $\mathcal{Z}$. After submitting a `TestPwd` query with

an extracted $\mathsf{pw}_{\mathcal{Z}}$, we let $\mathcal{S}$ now choose $x' \stackrel{\$}{\leftarrow} \mathbb{F}_P$ and add $(\mathsf{pw}_{\mathcal{Z}}||\ell, g^{x'}, x', \perp, X^*)$ to Λ_{IC} and proceed the simulation of $\mathcal{P}_0$ using x' instead of x.

- $\mathcal{Z}$ sends $\ell_{\mathcal{Z}}, Y^*_{\mathcal{Z}}$, there is a record $(\mathsf{pw}_{\mathcal{Z}}||\hat{\ell}, Y, *, \mathcal{E}, Y^*)$ in Λ_{IC} for some $\hat{\ell} \in \mathcal{L}$ and $\mathsf{pw}_{\mathcal{Z}} = \mathsf{pw}_0$: $\mathcal{S}$ will issue a `TestPwd` query that will result in a compromised record (cf. simulation described in game $\mathbf{G}_5$), resulting in a session key that is computed using $\mathsf{pw}_{\mathcal{Z}}$ instead of $\mathsf{pw}_{\mathcal{S}}$. Additionally, X^* is generated using the incorrect pass-string $\mathsf{pw}_{\mathcal{S}}$. However, adjusting Λ_{IC} as described above still allows $\mathcal{S}$ to know the exponent of $\mathcal{D}_{\mathsf{pw}_{\mathcal{Z}}||\ell}(X^*)$ and continue the simulation, making it look like $\mathsf{pw}_{\mathcal{Z}}$ was used from the beginning. $\mathcal{Z}$'s view is distributed exactly as before since x', x are both uniformly random in $\mathbb{F}_P$.
- $\mathcal{Z}$ sends $\ell_{\mathcal{Z}}, Y^*_{\mathcal{Z}}$, there is a record $(\mathsf{pw}_{\mathcal{Z}}||\hat{\ell}, Y, *, \mathcal{E}, Y^*)$ in Λ_{IC} for some $\hat{\ell} \in \mathcal{L}$ and $\mathsf{pw}_{\mathcal{Z}} \neq \mathsf{pw}_0$: since $\mathcal{P}_0$ will receive a random session key from $\mathcal{F}$ in this case (the record will be interrupted), we only have to argue indistinguishability of X^* generated with $\mathsf{pw}_{\mathcal{S}}|\ell$ instead of $\mathsf{pw}_0||\ell$. Simulation of $\mathcal{F}_{\mathsf{IC}}$ ensures that Y^* is distributed uniformly random as before. Again, it is crucial here that $\mathcal{F}$ helps $\mathcal{S}$ by randomizing the session key if needed.
- $\mathcal{Z}$ sends $\ell_{\mathcal{Z}}, Y^*_{\mathcal{Z}}$ and no $\mathcal{E}$ record: the simulation described in game $\mathbf{G}_5$ tells $\mathcal{S}$ to issue a `TestPwd` query, but now using $\mathsf{pw}_{\mathcal{S}}||\ell$ instead of $\mathsf{pw}_0||\ell$. If, coincidentally, $\mathsf{pw}_0 = \mathsf{pw}_{\mathcal{S}}$, nothing changes. On the other hand, if $\mathsf{pw}_0 \neq \mathsf{pw}_{\mathcal{S}}$, $\mathcal{P}_0$ obtains a random session key from $\mathcal{F}$ as opposed to the game before and X^* was created using $\mathsf{pw}_{\mathcal{S}}||\ell$ instead of $\mathsf{pw}_0||\ell$. This can only be detected if $\mathcal{Z}$ reproduces $\mathcal{P}_0$'s input to $\mathcal{F}_{\mathsf{RO}}$ from game $\mathbf{G}_{11}$, which happens only with negligible probability using an argument very similar to Lemma 8.
- both parties honest and no injections: $\mathcal{P}_0$ will obtain a uniformly random

session key from $\mathcal{F}$ in this case, and thus the only difference is that X^* was created using $\mathsf{pw}_{\mathcal{S}}||\ell$ instead of $\mathsf{pw}_0||\ell$. Again, this is indistinguishable since X^* is distributed exactly as before.

Observe that in game $\mathbf{G}_{12}$, $\mathcal{F} = \mathcal{F}_{\ell\text{-iPAKE}}$[11], and thus the theorem follows. The complete description of the simulator of game $\mathbf{G}_{12}$ interacting with $\mathcal{F}_{\ell\text{-iPAKE}}$ and $\mathcal{Z}$ is given in Figure 6·29.

The simulator $\mathcal{S}$, initialized with a security parameter λ, first runs a group generation algorithm using $\lambda$to obtain a cyclic group $\mathbb{G}$ with generator g of order q with $\log_2(q) \geq \lambda$. Then, $\mathcal{S}$ initializes the dummy adversary $\mathcal{A}$. $\mathcal{S}$ then interacts with an ideal functionality $\mathcal{F}_{\ell\text{-iPAKE}}$ and an environment $\mathcal{Z}$ via the following queries:

- **Upon receiving a query $(\texttt{NewSession}, sid, \mathcal{P}_i, \ell)$ from $\mathcal{F}_{\ell\text{-iPAKE}}$:**
  - initialize a party $\mathcal{P}_i$, connect it to $\mathcal{A}$ and proceed the UC protocol execution described in Figure 6·27 using $\mathsf{pw}_{\mathcal{S}} \xleftarrow{\$} \mathbb{F}_p$ as pass-string and $\mathcal{S}$'s random coins. (Cf. games $\mathbf{G}_0$,$\mathbf{G}_{11}$ and $\mathbf{G}_{12}$.)
- **Upon receiving a query (sid, m) from any entity:** (Cf. games $\mathbf{G}_1$ and $\mathbf{G}_2$.)
  If $m \notin \langle g \rangle^3$, then abort. Else:
  - if there is an entry (m, h) then reply with (sid, h).
  - else, choose $h \xleftarrow{\$} \mathbb{F}_q$ and abort if there already is an entry $(*, *, h)$ in Λ_{RO}. Else, reply with (sid, h).
- **If an internally simulated party $\mathcal{P}_i$ produces an output $(sid, \ell', \mathsf{k}_i)$:**
  Send $(\texttt{NewKey}, sid, \mathcal{P}_i, \mathsf{k}_i)$ to $\mathcal{F}_{\ell\text{-iPAKE}}$. (Cf. game $\mathbf{G}_1$.)
- **If $\mathcal{Z}$ sends $(sid, \ell_{\mathcal{Z}}, Z^*_{\mathcal{Z}})$ to an honest party $\mathcal{P}_i$:**
  - if $(\mathsf{pw}_{\mathcal{Z}}||\hat{\ell}, *, *, \mathcal{E}, Z^*_{\mathcal{Z}}) \in \Lambda_{IC}$ for any $\hat{\ell} \in \mathcal{L}$, then send $(\texttt{TestPwd}, sid, \mathcal{P}_i, \mathsf{pw}_{\mathcal{Z}}, \ell_{\mathcal{Z}})$ to $\mathcal{F}_{\ell\text{-iPAKE}}$ and proceed the simulation of $\mathcal{P}_i$ with $\mathsf{pw}_{\mathcal{Z}}$. (Cf. games $\mathbf{G}_5$ and $\mathbf{G}_6$.)
  - if $(*, *, *, \mathcal{E}, Z^*_{\mathcal{Z}}) \notin \Lambda_{IC}$, then send $(\texttt{TestPwd}, sid, \mathcal{P}_i, \mathsf{pw}_{\mathcal{S}}, \ell_{\mathcal{Z}})$ to $\mathcal{F}_{\ell\text{-iPAKE}}$. (Cf. games $\mathbf{G}_5$ and $\mathbf{G}_6$.)
  - if $\mathcal{P}_i$ was started with input $(\texttt{NewSession}, sid, \mathcal{P}_i, \ell)$, choose $x' \xleftarrow{\$} \mathbb{F}_P$, add $(\mathsf{pw}_{\mathcal{Z}}||\ell, g^{x'}, x', \perp, Z^*)$ to Λ_{IC} and proceed as if x' was the value drawn uniformly random from $\mathbb{F}_q$ at the beginning of the simulation of $\mathcal{P}_i$. (Cf. game $\mathbf{G}_{12}$.)
- **Upon receiving a query $(sid, \texttt{Encrypt}, k, m)$ from any entity:** (Cf. games $\mathbf{G}_1$ and $\mathbf{G}_2$.)
  If $k \notin \mathbb{F}_p$ or $m \notin \mathbb{G}$ then abort. Else:
  - if there is an entry $(k, m, *, *, c)$ in Λ_{IC}, reply with (sid, c)
  - else, choose $c \xleftarrow{\$} \mathbb{G} \setminus \{1\}$. If there already is a record $(*, *, *, *, c)$ then abort. Else, add $(k, m, \perp, \mathcal{E}, c)$ to Λ_{IC} and reply with (sid, c).
- **Upon receiving a query $(sid, \texttt{Decrypt}, k, c)$ from any entity:** (Cf. games $\mathbf{G}_1$ and $\mathbf{G}_2$.)
  If $k \notin \mathbb{F}_p$ or $c \notin \mathbb{G}$ then abort. Else:
  - if there is an entry $(k, m, *, *, c)$ in Λ_{IC}, reply with (sid, m).
  - else, choose $\alpha \xleftarrow{\$} \mathbb{F}_p^*$. If there already is a record $(*, g^\alpha, *, *, *)$ then abort. Else, add $(k, g^\alpha, \alpha, \mathcal{D}, c)$ to Λ_{IC} and reply with (sid, g^α).
- **Upon receiving a query $(sid, \texttt{crs})$ from any entity: (Cf. game $\mathbf{G}_1$.)**
  - reply with $(sid, (g, q))$.

Additionally, $\mathcal{S}$ forwards all other instructions from $\mathcal{Z}$ to $\mathcal{A}$ and reports all output of $\mathcal{A}$ towards $\mathcal{Z}$. Instructions of corrupting a player are only obeyed if they are received before the protocol started, i.e., before $\mathcal{S}$ received any `NewSession` query from $\mathcal{F}_{\ell\text{-iPAKE}}$.

Figure 6·29: The Simulator $\mathcal{S}$ for the EKE2 Protocol

[11]We note that we can, w.l.o.g, assume that there are no `NewSession` queries from $\mathcal{Z}$ to corrupted parties. Thus, it is enough to remove the pass-strings from the `NewSession` queries given as input from $\mathcal{Z}$ to honest parties.

6.12 Appendix G: Proof of Theorem 14

For the proof, we describe an honest execution of the protocol $\mathsf{fPAKE}_{\mathsf{RSS}}$ in the UC framework in Figure 6·30. See (Freire et al., 2014) for a detailed description on how to execute protocols within the UC framework. This real protocol execution will be the starting point for our proof. We then proceed in a series of games, to end up with the ideal execution running with only dummy parties, a simulator and the ideal functionality $\mathcal{F}^{M}_{\mathsf{fPAKE}}$. For convenience, we refer to a received protocol message as *adversarially generated* if it was not produced by either $\mathcal{P}_0$ or $\mathcal{P}_1$. We also refer to a query $(\texttt{NewKey}, sid, \mathcal{P}_i, \mathsf{k}_i)$ from the adversary $\mathcal{S}$ with an honest party $\mathcal{P}_i$ as *due* if

- there is a `fresh` record of the form $(\mathcal{P}_i, \mathsf{pw}_i)$
- this is the first `NewKey` query for $\mathcal{P}_i$
- there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $d(\mathsf{pw}_i, \mathsf{pw}_{1-i}) \leq \delta$ and $\mathcal{P}_{1-i}$ is honest
- a key k_{1-i} was sent to the other party while $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ was `fresh` at the time.

We also define a masking function that reveals the positions of the identical bits:

$$msk\mathsf{pw}\mathsf{pw}' := \{i | \mathsf{pw}_i = \mathsf{pw}'_i, i \in [n]\}$$

Game $\mathbf{G}_0$: The real protocol execution. This is the real execution of $\mathsf{fPAKE}_{\mathsf{RSS}}$ where the environment $\mathcal{Z}$ runs the protocol (cf. Figure 6·10) with parties $\mathcal{P}_0$ and $\mathcal{P}_1$, both having access to an ideal ℓ-iPAKE functionality $\mathcal{F}_{\ell\text{-}\mathsf{iPAKE}}$, and an adversary $\mathcal{A}$ that, w.l.o.g., can be assumed to be the dummy adversary as shown in (Canetti, 2001, section 4.4.1).

Game $\mathbf{G}_1$: Modeling the ideal layout. We first make some purely conceptual changes that do not modify the input/output interfaces of $\mathcal{Z}$. We add one relay (also referred to as *dummy party*) on each of the wires between $\mathcal{Z}$ and a party. We also add one relay covering all the wires between the dummy parties and

The parties $\mathcal{P}_0, \mathcal{P}_1$ are running with $\mathcal{F}_{\ell\text{-iPAKE}}$.

Protocol Steps:

1. When a party $\mathcal{P}_i$, $i \in \{0,1\}$, receives an input $(\texttt{NewSession}, sid, \mathsf{pw}_i, \texttt{sender})$ from $\mathcal{Z}$, it does the following:
   - compute $(\mathsf{vk}, \mathsf{sk}) \overset{\$}{\leftarrow} \mathsf{SigGen}(1^\lambda)$
   - query n times $\mathcal{F}_{\ell\text{-iPAKE}}$ with $(\texttt{NewSession}, sid, (\mathsf{pw}_i)_t, \mathsf{vk})$, $t = 1, ..., n$, receiving back (sid, ℓ_t, K_t) as answer
   - choose $U \overset{\$}{\leftarrow} \mathbb{F}_q$
   - compute $C \leftarrow \mathsf{Share}(U)$
   - compute $E \leftarrow C + (K_t)_{t=1}^n$
   - compute $\sigma_E \leftarrow \mathsf{Sign}(\mathsf{vk}, E)$
   - send $(sid, E, \sigma_E, \mathsf{vk})$ to $\mathcal{P}_{1-i}$
   - set $\mathsf{k} \leftarrow U$
   - send (sid, k) towards $\mathcal{Z}$ and terminate the session.
2. When a party $\mathcal{P}_i$, $i \in \{0,1\}$, receives an input $(\texttt{NewSession}, sid, \mathsf{pw}_i, \texttt{receiver})$ from $\mathcal{Z}$, it does the following:
   - query n times $\mathcal{F}_{\ell\text{-iPAKE}}$ with $(\texttt{NewSession}, sid, (\mathsf{pw}_i)_t, \varepsilon)$, $t = 1, ..., n$, receiving back (sid, ℓ_t, K'_t) as answer
   - if not all ℓ_t are equal or $\ell_1 \neq \mathcal{VK}$, then abort
3. When $\mathcal{P}_i$, who already obtained an input $(\texttt{NewSession}, sid, \mathsf{pw}_i, \texttt{receiver})$ and thus holds a vector (sid, ℓ_t, K'_t) obtained from $\mathcal{F}_{\ell\text{-iPAKE}}$, receives a message $(sid, E, \sigma_E, \mathsf{vk})$ from $\mathcal{P}_{1-i}$, it does the following:
   - set $K' := (K'_t)_{t\in[n]}$
   - abort if $\mathsf{vk} \neq \ell_1$
   - abort if $\mathsf{Vfy}(\mathsf{vk}, \sigma_E, E) = 0$
   - compute $U' \leftarrow \mathsf{Reconstruct}(E - K')$
   - set $\mathsf{k} \leftarrow U'$
   - send (sid, k) towards $\mathcal{Z}$ and terminate the session.

Figure 6·30: A UC Execution of $\mathsf{fPAKE}_{\mathsf{RSS}}$

real parties and call it $\mathcal{F}$ (and let $\mathcal{F}$ relay messages according to the original wires). We group all the formerly existing instances except for $\mathcal{Z}$ into one machine and call it $\mathcal{S}$. Note that this implies that $\mathcal{S}$ executes the code of the ℓ-iPAKE functionality $\mathcal{F}_{\ell\text{-iPAKE}}$. The differences are depicted in Figure 6·20 with $\mathcal{F}_{\mathsf{OT}}$ replaced by $\mathcal{F}_{\ell\text{-iPAKE}}$.

Game $\mathbf{G}_2$: Building $\mathcal{F}^M_{\mathsf{fPAKE}}$. In this game, we start modeling $\mathcal{F}^M_{\mathsf{fPAKE}}$. First, we let $\mathcal{F}$ maintain a list of tuples of the form $(\mathcal{P}_i, \mathsf{pw}_i)$. Upon receiving a query of the form $(\texttt{NewSession}, sid, \mathsf{pw}_i, \texttt{role})$ from party $\mathcal{P}_i$, if this is the first `NewSession`-query, or if this is the second `NewSession`-query and there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$, then $\mathcal{F}$ records $(\mathcal{P}_i, \mathsf{pw}_i)$ and marks this record as `fresh`. In any case the query $(\texttt{NewSession}, sid, \mathcal{P}_i, \mathsf{pw}_i, \texttt{role})$ is relayed to $\mathcal{S}$. Now that $\mathcal{F}$ knows about pass-strings, we can add a `TestPwd` interface to $\mathcal{F}$ as described in Figure 6·2, using leakage functions L^M_c, L^M_m and L^M_f. We let $\mathcal{S}$ parse outputs towards $\mathcal{F}$ as $(\texttt{NewKey}, sid, \mathcal{P}_i, \mathsf{k}_i)$ by adding the `NewKey` tag and the name of the party who produced the output. Additionally, we let $\mathcal{F}$ translate this back to (sid, k_i), send it to $\mathcal{Z}$ via $\mathcal{P}_i$ and mark the corresponding record as `completed`.

None of these modifications changes the output towards $\mathcal{Z}$ compared to the previous game $\mathbf{G}_1$.

Game $\mathbf{G}_3$: $\mathcal{F}$ generates a random session key for an interrupted session. Upon receiving a query $(\texttt{NewKey}, sid, \mathcal{P}_i, \mathsf{k}_i)$ from $\mathcal{S}$, if there is a record of the form $(\mathcal{P}_i, \mathsf{pw}_i)$ that is marked as `interrupted`, and this is the first `NewKey` query for $\mathcal{P}_i$, we let $\mathcal{F}$ output a random session key of length λ to $\mathcal{P}_i$. Otherwise, it continues to relay k_i.

Since the simulators described in game $\mathbf{G}_2$ and game $\mathbf{G}_3$ do not make use of the `TestPwd` interface, none of the records of $\mathcal{F}$ are marked as `interrupted` and

thus the output towards $\mathcal{Z}$ is equally distributed in both games.

Game $\mathbf{G}_4$: $\mathcal{S}$ handles dictionary attacks using the `TestPwd` interface. In this game, we only change the simulation. Consider the following setting: $\mathcal{P}_i$ obtained input $(\texttt{NewSession}, sid, \mathsf{pw}_i, \texttt{role})$ and $\mathcal{P}_{1-i}$ is corrupted and already provided its inputs to $\mathcal{F}_{\ell\text{-iPAKE}}$. In this situation, $\mathcal{S}$ will proceed simulation of $\mathcal{P}_i$ as follows:

$\mathcal{S}$ assembles $\mathsf{pw}_{\mathcal{Z}} \in \mathbb{F}_p^n$ from the queries to $\mathcal{F}_{\ell\text{-iPAKE}}$ that $\mathcal{P}_{1-i}$ issued. $\mathcal{S}$ sends $(\texttt{TestPwd}, sid, \mathcal{P}_i, \mathsf{pw}_{\mathcal{Z}})$ to $\mathcal{F}$, obtaining either "wrong guess", "correct guess" and perhaps also a mask $M \subseteq [n]$ from $\mathcal{F}$. If $\mathcal{S}$ does not receive a mask, $\mathcal{S}$ is not modified further. Else, let $I := [n] \setminus M$ the set of mismatching indices, and $d := |I| \leq \gamma$ their number. $\mathcal{S}$ sets up keys $K, K' \in \mathbb{F}_q^n$ with $K_t = K'_t \stackrel{\$}{\leftarrow} \mathbb{F}_q$ for the matching indices $t \in M$ and $K_t, K'_t \stackrel{\$}{\leftarrow} \mathbb{F}_p^2$ for the mismatching indices $t \in I$, where K' denotes the $\mathcal{F}_{\ell\text{-iPAKE}}$ output of $\mathcal{P}_{1-i}$. $\mathcal{S}$ now continues the simulation of $\mathcal{P}_i$ using K as output of $\mathcal{F}_{\ell\text{-iPAKE}}$.

We have to analyze different cases depending on the different outcomes of `TestPwd`. However, note that the modifications only have an impact on the output k_i of $\mathcal{P}_i$ if the record gets `interrupted`, and only affect the transcript if the answer to the `TestPwd` query contains a mask. Considering the case where `TestPwd`

- outputs $\vec{m}$ and sets the record `compromised`, i.e. $d \leq \gamma$ since the distribution of K, K' only depends on the mask of the pass-strings, the view of $\mathcal{Z}$ is identically distributed in game $\mathbf{G}_4$ and game $\mathbf{G}_3$;
- outputs "wrong guess" and sets the record `interrupted`, i.e. $d > \gamma$: $\mathcal{P}_i$ will now obtain a randomly chosen session key from $\mathcal{F}$, substituting the key k_i computed by $\mathcal{S}$. If $\mathcal{P}_i$ obtained $\texttt{role} = \texttt{sender}$, the output in game $\mathbf{G}_4$ and game $\mathbf{G}_3$ is equally distributed since the honest sender outputs a

random $\mathbb{F}_q$ value according to the protocol description. If $\mathcal{P}_i$ obtained $\texttt{role} = \texttt{receiver}$, both outputs are indistinguishable with overwhelming probability due to the smoothness of the RSS, since in game $\mathbf{G}_3$ at least $\gamma + 1$ shares are random.

Game $\mathbf{G}_5$: Excluding man-in-the-middle attacks. Again, in this game, we only change the simulation. We now consider the case where $\mathcal{Z}$ injects a message into a session where both parties are honest. We modify $\mathcal{S}$ as follows: upon receiving an adversarially generated $(sid, M_{\mathcal{Z}}, \sigma_{\mathcal{Z}}, \mathsf{vk}_{\mathcal{Z}})$ from $\mathcal{Z}$ intended for party $\mathcal{P}_i$, $\mathcal{S}$ aborts.

Observe that the simulation is only changed compared to the previous game if it is not aborted due to protocol instructions. This means that both games are equal unless all checks pass, especially $\mathsf{Vfy}(\mathsf{vk}_{\mathcal{Z}}, \sigma_{\mathcal{Z}}, M_{\mathcal{Z}}) = 1$. Any distinguisher between game $\mathbf{G}_5$ and game $\mathbf{G}_4$ can thus be turned into a forger of a valid message w.r.t the verification key of an honest party. Indinstinguishability thus follows from the security of the one-time signature scheme.

Game $\mathbf{G}_6$: $\mathcal{F}$ aligns session keys. Upon receiving a query $(\texttt{NewKey}, sid, \mathcal{P}_i, \mathsf{k}_i)$ from $\mathcal{S}$, if this query is *due* then output (sid, k_{1-i}) to $\mathcal{P}_i$ where k_{1-i} is the session key that was formerly sent to the other party.

We now analyze distinguishability of this game from game $\mathbf{G}_5$. If $\mathcal{Z}$ tampered with the transcript, the simulation in game $\mathbf{G}_5$ ensures that the simulation aborts and there is thus no $\texttt{NewKey}$ query for $\mathcal{P}_i$. On the other hand, if $\mathcal{Z}$ does not advise $\mathcal{A}$ to tamper with any message, perfect correctness of $\mathsf{fPAKE}_{\mathsf{RSS}}$ protocol ensures that, in case of a due record where the parties hold close passstrings $\mathsf{pw}_i, \mathsf{pw}_{1-i}$ with $d(\mathsf{pw}_i, \mathsf{pw}_{1-i}) \leq n - r$, the output of $\mathcal{F}$ towards $\mathcal{Z}$ is the same as in the previous game $\mathbf{G}_5$. Observe that perfect correctness directly

follows from the perfect correctness of $\mathcal{F}_{\ell\text{-iPAKE}}$ and the r-robustness of the secret sharing, which is *always* able to correct up to $n-r$ errors.

Note that $\mathcal{F}$ still differs from the functionality $\mathcal{F}_{\mathsf{fPAKE}}^{M}$ in some aspects. First, it does not output randomly generated session keys towards $\mathcal{Z}$ for honest sessions. Furthermore, it reports all pass-strings to $\mathcal{S}$. We will take care of these remaining differences in the next games.

Game $\mathbf{G}_7$: In some cases, $\mathcal{F}$ generates a random session key when the other party is corrupted. Upon receiving a `NewKey` query $(\texttt{NewKey}, sid, \mathcal{P}_i, \mathsf{k}_i)$ from $\mathcal{S}$, if there is a fresh record of the form $(\mathcal{P}_i, \mathsf{pw}_i)$, and this is the first `NewKey` query for $\mathcal{P}_i$, $\mathcal{P}_i$ is honest and $\mathcal{P}_{1-i}$ corrupted and there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $d(\mathsf{pw}_i, \mathsf{pw}_{1-i}) > \delta$, we let $\mathcal{F}$ pick a new random key k from $\mathbb{F}_q$ and send (sid, k) to $\mathcal{P}_i$.

The simulation ensures that the record $(\mathcal{P}_i, \mathsf{pw}_i)$ is either compromised or interrupted (cf. description of the simulator in game $\mathbf{G}_4$). Thus, the modification has no effect since it only concerns fresh records.

Game $\mathbf{G}_8$: $\mathcal{F}$ generates a random session key for an honest session. Upon receiving a `NewKey` query $(\texttt{NewKey}, sid, \mathcal{P}_i, \mathsf{k}_i)$ from $\mathcal{S}$, if there is a fresh record of the form $(\mathcal{P}_i, \mathsf{pw}_i)$, and this is the first `NewKey` query for $\mathcal{P}_i$, both parties are honest and the `NewKey` query is not *due*, we let $\mathcal{F}$ pick a new random key k from $\mathbb{F}_q$ and send (sid, k) to $\mathcal{P}_i$.

In other words, $\mathcal{F}$ now generates a random session key upon a first `NewKey` query for an honest party $\mathcal{P}_i$ with fresh record $(\mathcal{P}_i, \mathsf{pw}_i)$ where $\mathcal{P}_{1-i}$ is also honest, if (at least) one of the following events happen:

1. There is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $d(\mathsf{pw}_i, \mathsf{pw}_{1-i}) > \delta$; then, the probability that k_i was already random in game $\mathbf{G}_7$ is overwhelming due to the $r-1$-

smoothness of the RSS on random secrets. Note that to apply this property it is crucial that both parties are honest and thus the value U is randomly chosen.

2. No session key was sent to $\mathcal{P}_{1-i}$ yet; we just have to consider the case where there is a record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ with $d(\mathsf{pw}_i, \mathsf{pw}_{1-i}) \leq \delta$ since we already dealt with the other case in the first event. Due to the r-robustness of the RSS, the session key in the previous game was U, which is distributed uniformly random in $\mathbb{F}_q$.
3. If there was a session key sent to $\mathcal{P}_{1-i}$, the record $(\mathcal{P}_{1-i}, \mathsf{pw}_{1-i})$ was not fresh and thus interrupted or compromised at that time; since our simulation never issues `TestPwd` queries for honest sessions (in fact, game $\mathbf{G}_5$ states that $\mathcal{S}$ aborts upon man-in-the-middle attacks with overwhelming probability), this event can not happen in our simulation.

Game $\mathbf{G}_9$: Simulating without pass-string if both parties are honest. In case of receiving a $(\mathtt{NewSession}, sid, \mathsf{pw}_i, \mathtt{role})$ from an honest $\mathcal{P}_i$, we modify $\mathcal{F}$ by forwarding only $(\mathtt{NewSession}, sid, \mathcal{P}_i, \mathtt{role})$ to $\mathcal{S}$. We now have to modify $\mathcal{S}$ to proceed simulation without knowing pw. Upon receiving $(\mathtt{NewSession}, sid, \mathcal{P}_i, \mathtt{role})$ from $\mathcal{F}$ for an honest $\mathcal{P}_i$, we let $\mathcal{S}$ draw uniformly at random a "dummy" pass-string $\mathsf{pw}_{\mathcal{S}}$ and proceed the simulation of $\mathcal{P}_i$ using $\mathsf{pw}_{\mathcal{S}}$ as a pass-string.

We first observe that $\mathcal{Z}$ is oblivious of k_i contained in the $(\mathtt{NewKey}, sid, \mathcal{P}_i, \mathsf{k}_i)$ query that $\mathcal{S}$ will eventually send to $\mathcal{F}$ during the simulation (since $\mathcal{F}$ never lets the simulator determine k_i for an honest session). We thus only have to consider the case where $\mathcal{P}_i$ has $\mathtt{role} = \mathtt{sender}$. We show that $\mathcal{Z}$, knowing $\mathsf{pw}_i, \mathsf{pw}_{1-i}$ and seeing two transcripts, cannot tell which one was generated using $\mathsf{pw}_i, \mathsf{pw}_{1-i}$ and which one was generated using $\mathsf{pw}_{\mathcal{S}}, \mathsf{pw}_{1-i}$ with a random $\mathsf{pw}_{\mathcal{S}}$ unknown to $\mathcal{Z}$. But this is trivial since the distribution of the values U, K does not depend

on the pass-strings: U is randomly chosen from $\mathbb{F}_q$. $\mathcal{F}_{\ell\text{-iPAKE}}$ ensures that K is randomly chosen from $\mathbb{F}_q^n$.

Game $\mathbf{G}_{10}$: Simulating without pass-string if someone is corrupted. Upon receiving ($\texttt{NewSession}, sid, \mathsf{pw}_i, \texttt{role}$) from $\mathcal{P}_i$ where $\mathcal{P}_{1-i}$ is corrupted, we modify $\mathcal{F}$ to only relay ($\texttt{NewSession}, sid, \mathcal{P}_i, \texttt{role}$) to $\mathcal{S}$. Additionally, we let $\mathcal{S}$ draw uniformly at random a "dummy" pass-string $\mathsf{pw}_\mathcal{S}$ and proceed the simulation of $\mathcal{P}_i$ using $\mathsf{pw}_\mathcal{S}$ as a pass-string. Note that due to the simulation described in game $\mathbf{G}_4$, $\mathcal{S}$ will ask a $\texttt{TestPwd}$ query, and after this query the simulation described in that game is already independent of pw_i except when $\mathcal{F}$'s reply does not contain a mask, i.e., $d(\mathsf{pw}_i, \mathsf{pw}_{1-i}) > \gamma$. In this case, we now let $\mathcal{S}$ set the output of $\mathcal{F}_{\ell\text{-iPAKE}}$ towards $\mathcal{P}_i$ to be a random $K \xleftarrow{\$} \mathbb{F}_q^n$.

Regarding indistinguishability, first note that in any case the input of $\mathcal{P}_i$ to $\mathcal{F}_{\ell\text{-iPAKE}}$ does not impact any values and thus we only have to argue further in case $\mathcal{S}$ is modified. Thus, $\mathcal{P}_i$'s record will get interrupted and $\mathcal{P}_i$ will obtain a uniformly random session key from $\mathcal{F}$, meaning that we only have to consider the case where $\mathcal{P}_i$ has $\texttt{role} = \texttt{sender}$ and argue indistinguishability of E, σ_E, vk generated with either K depending on pw_i (as in the previous game) or $K \xleftarrow{\$} \mathbb{F}_q^n$ (as in the current game). Opposed to the situation in game $\mathbf{G}_9$, note that now $\mathcal{Z}$ knows K'.

Since $d(\mathsf{pw}_i, \mathsf{pw}_{1-i}) > \gamma = n - t - 1$, at most t components of K' are the same as K in $\mathbf{G}_9$ with large probability $1 - \frac{n-t}{q}$, and thus w.h.p. $\mathcal{Z}$ learns at most t components of C. Hence the transcript of the current and previous games are indistinguishable due to the strong t-privacy of the RSS.

Observe that now $\mathcal{F}$ is equal to $\mathcal{F}_{\mathsf{fPAKE}}^M$ and $\mathcal{S}$ is equal to the simulator described in Figure 6·31. The theorem thus follows.

The simulator $\mathcal{S}$, initialized with a security parameter λ, initializes the dummy adversary $\mathcal{A}$. $\mathcal{S}$ emulates an ideal labeled iPAKE functionality $\mathcal{F}_{\ell\text{-iPAKE}}$ as depicted in Figure 6·26 for all calling entities in the system[a]. Additionally, $\mathcal{S}$ interacts with an ideal functionality $\mathcal{F}^{M}_{\mathsf{fPAKE}}$ and a distinguisher, the environment $\mathcal{Z}$, via the following queries:

- **Upon receiving a query** $(\texttt{NewSession}, \boldsymbol{sid}, \boldsymbol{\mathcal{P}_i}, \texttt{role})$ **from** $\boldsymbol{\mathcal{F}^{M}_{\mathsf{fPAKE}}}$**:** initialize a party $\mathcal{P}_i$ and connect it to $\mathcal{A}$.
  - If $\mathcal{P}_{1-i}$ is honest, $\mathcal{S}$ proceeds the UC protocol execution as described in Figure 6·30 using $\mathsf{pw}_{\mathcal{S}} \stackrel{\$}{\leftarrow} \mathbb{F}_p$ as pass-string for $\mathcal{P}_i$. (Cf. game $\mathbf{G}_9$.)
  - If $\mathcal{P}_{1-i}$ is corrupted, then $\mathcal{S}$ waits until $\mathcal{P}_{1-i}$ submitted n queries to $\mathcal{F}_{\ell\text{-iPAKE}}$ and then assembles $\mathsf{pw}_{\mathcal{Z}} \in \mathbb{F}_p^n$ from them. $\mathcal{S}$ sends $(\texttt{TestPwd}, sid, \mathcal{P}_i, \mathsf{pw}_{\mathcal{Z}})$ to $\mathcal{F}^{M}_{\mathsf{fPAKE}}$. If $\mathcal{S}$ receives back a mask M, let $I := [n] \setminus M$, and $\mathcal{S}$ sets up n $\mathbb{F}_q$-keys K with $K_t = K'_t \ \forall t \in I$ and $K_t \stackrel{\$}{\leftarrow} \mathbb{F}_p \neq K'_t \ \forall t \in M$, where K' denotes the output of $\mathcal{F}_{\ell\text{-iPAKE}}$ towards $\mathcal{P}_{1-i}$. $\mathcal{S}$ now continues the simulation of $\mathcal{P}_i$ using K as outputs of $\mathcal{F}_{\ell\text{-iPAKE}}$. (Cf. game $\mathbf{G}_4$.) If $\mathcal{S}$ does not receive a mask, $\mathcal{S}$ sets the output of $\mathcal{F}_{\ell\text{-iPAKE}}$ towards $\mathcal{P}_i$ to be $K \stackrel{\$}{\leftarrow} \mathbb{F}_q^n$. (Cf. game $\mathbf{G}_{10}$.)
- **If an internally simulated party** $\mathcal{P}_i$ **produces an output** $(\boldsymbol{sid}, \mathsf{k}_i)$**:** Send $(\texttt{NewKey}, sid, \mathcal{P}_i, \mathsf{k}_i)$ to $\mathcal{F}^{M}_{\mathsf{fPAKE}}$.
- **If** $\mathcal{Z}$ **sends** $(sid, M_{\mathcal{Z}}, \sigma_{\mathcal{Z}}, \mathsf{vk}_{\mathcal{Z}})$ **to an honest party** $\mathcal{P}_i$**:** if $\mathcal{P}_{1-i}$ is honest, $\mathcal{S}$ aborts after the Vfy step in the protocol, regardless of its outcome. (Cf. game $\mathbf{G}_5$.)

Additionally, $\mathcal{S}$ forwards all other instructions from $\mathcal{Z}$ to $\mathcal{A}$ and reports all output of $\mathcal{A}$ towards $\mathcal{Z}$. Instructions of corrupting a player are only obeyed if they are received before the protocol started, i.e., before $\mathcal{S}$ received any `NewSession` query from $\mathcal{F}^{M}_{\mathsf{fPAKE}}$.

[a] An entity is any internally simulated ITM such as parties or the real-world adversary as well as ITMs outside $\mathcal{S}$ such as the distinguisher $\mathcal{Z}$.

Figure 6·31: The Simulator $\mathcal{S}$ for $\mathsf{fPAKE}_{\mathsf{RSS}}$

6.13 Appendix H: A Natural (But Failed) Approach to fPAKE

A natural idea for building a fPAKE is the use of a fuzzy extractor (Dodis et al., 2004; Boyen, 2004), that allows to extract a common secret from two strings close enough, and to compose it with a regular PAKE. This approach was introduced in (Boyen et al., 2005) (Section 4). Their protocol uses the code-offset construction of a fuzzy

$\mathcal{P}_0(\mathsf{pw} \in \mathbb{F}_q^n)$		$\mathcal{P}_1(\mathsf{pw}' \in \mathbb{F}_q^n)$
$w \xleftarrow{\$} \mathbb{F}_q^k, c \leftarrow \mathsf{Share}(w) \in \mathbb{F}_q^n$		
$s \leftarrow c + \mathsf{pw}$	$\xrightarrow{\quad s \quad}$	$c' \leftarrow s + \mathsf{pw}'$
		$w' \leftarrow \mathsf{Reconstruct}(c')$
K	$\xleftrightarrow{\mathsf{PAKE}(w, w')}$	K'
$\mathsf{sk} \leftarrow K$		$\mathsf{sk}' \leftarrow K'$

Figure 6·32: A First Natural Construction (with code-offset fuzzy sketch and PAKE)

sketch (Dodis et al., 2004), a.k.a. fuzzy commitment (Juels and Wattenberg, 1999), to implement a fuzzy-extractor as a two-party primitive. It is presented in Figure 6·32.

Theorem 17. *The construction from Figure 6·32 cannot securely realize $\mathcal{F}_{\mathsf{fPAKE}}^{P}$.*

Proof. Consider the following attack by $\mathcal{Z}$. $\mathcal{Z}$ sends a randomly chosen pw as input to an honest $\mathcal{P}_0$ and obtains a sketch s from $\mathcal{A}$. It then computes $c \leftarrow s - \mathsf{pw}$ and outputs 1 if c is in the image of Share. In the real world, this happens with probability 1. Now assume there is a simulator $\mathcal{S}$ outputting a simulated sketch $\tilde{s}$ in the ideal world. Since $\mathcal{S}$ does not get to learn pw unless it succeeds at a `TestPwd` query, observe that this output may not depend on pw except with some small (but non-negligible) probability p, namely the probability of guessing a pass-string that makes $\mathcal{F}_{\mathsf{fPAKE}}^{P}$ output pw. Thus, with probability $1 - p \approx 1$, $\tilde{c} := \tilde{s} - \mathsf{pw}$ is randomly distributed in $\mathbb{F}_q^n$ and lies in the image of Share only with probability $1/q^{mn-l}$. More formally, the probability that $\mathcal{Z}$ outputs 1 in the ideal world is

$$\begin{aligned}
\Pr[\tilde{c} \in Im(\mathsf{Share})] &= \Pr[\tilde{c} \in Im(\mathsf{Share}) | \mathcal{S} \text{ depends on } \mathsf{pw}] \cdot p \\
&\quad + \Pr[\tilde{c} \in Im(\mathsf{Share}) | \mathcal{S} \text{ does not depend on } \mathsf{pw}] \cdot (1 - p) \\
&\leq p + 1/q^{mn-l}(1 - p) \approx p.
\end{aligned}$$

□

References

Abdalla, M., Catalano, D., Chevalier, C., and Pointcheval, D. (2008). Efficient two-party password-based key exchange protocols in the UC framework. In Malkin, T., editor, *CT-RSA 2008*, volume 4964 of *LNCS*, pages 335–351. Springer, Heidelberg.

Afshar, A., Hu, Z., Mohassel, P., and Rosulek, M. (2015). How to efficiently evaluate RAM programs with malicious security. In Oswald, E. and Fischlin, M., editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 702–729. Springer, Heidelberg.

Ananth, P. and Vaikuntanathan, V. (2019). Optimal bounded-collusion secure functional encryption. In *TCC 2019, Part I*, LNCS, pages 174–198. Springer, Heidelberg.

Ateniese, G., Camenisch, J., Joye, M., and Tsudik, G. (2000). A practical and provably secure coalition-resistant group signature scheme. In Bellare, M., editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 255–270. Springer, Heidelberg.

Au, M. H., Tsang, P. P., Susilo, W., and Mu, Y. (2009). Dynamic universal accumulators for DDH groups and their application to attribute-based anonymous credential systems. In Fischlin, M., editor, *CT-RSA 2009*, volume 5473 of *LNCS*, pages 295–308. Springer, Heidelberg.

Badrinarayanan, S., Goyal, V., Jain, A., and Sahai, A. (2016). Verifiable functional encryption. In Cheon, J. H. and Takagi, T., editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 557–587. Springer, Heidelberg.

Baldimtsi, F., Camenisch, J., Dubovitskaya, M., Lysyanskaya, A., Reyzin, L., Samelin, K., and Yakoubov, S. (2017). Accumulators with applications to anonymity-preserving revocation. *IACR Cryptology ePrint Archive*, 2017:43.

Baldimtsi, F., Canetti, R., and Yakoubov, S. Universally composable accumulators.

Ball, M., Malkin, T., and Rosulek, M. (2016). Garbling gadgets for Boolean and arithmetic circuits. In Weippl, E. R., Katzenbeisser, S., Kruegel, C., Myers, A. C., and Halevi, S., editors, *ACM CCS 2016*, pages 565–577. ACM Press.

Barak, B., Canetti, R., Lindell, Y., Pass, R., and Rabin, T. (2005). Secure computation without authentication. In Shoup, V., editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 361–377. Springer, Heidelberg.

Bari, N. and Pfitzmann, B. (1997). Collision-free accumulators and fail-stop signature schemes without trees. In Fumy, W., editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 480–494. Springer, Heidelberg.

Beaver, D., Micali, S., and Rogaway, P. (1990). The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*, pages 503–513. ACM Press.

Bellare, M., Hoang, V. T., and Rogaway, P. (2012). Foundations of garbled circuits. In Yu, T., Danezis, G., and Gligor, V. D., editors, *ACM CCS 2012*, pages 784–796. ACM Press.

Bellare, M., Pointcheval, D., and Rogaway, P. (2000). Authenticated key exchange secure against dictionary attacks. In Preneel, B., editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 139–155. Springer, Heidelberg.

Bellovin, S. M. and Merritt, M. (1992). Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *1992 IEEE Symposium on Security and Privacy*, pages 72–84. IEEE Computer Society Press.

Benaloh, J. C. and de Mare, M. (1994). One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In Helleseth, T., editor, *EUROCRYPT'93*, volume 765 of *LNCS*, pages 274–285. Springer, Heidelberg.

Bennett, C. H., Brassard, G., and Robert, J.-M. (1988). Privacy amplification by public discussion. *SIAM Journal on Computing*, 17(2):210–229.

Bitansky, N., Canetti, R., Kalai, Y. T., and Paneth, O. (2014). On virtual grey box obfuscation for general circuits. In Garay, J. A. and Gennaro, R., editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 108–125. Springer, Heidelberg.

Blanton, M. and Hudelson, W. M. (2009). Biometric-based non-transferable anonymous credentials. In *Information and Communications Security*, pages 165–180. Springer.

Borisov, N., Goldberg, I., and Brewer, E. (2004). Off-the-record communication, or, why not to use PGP. In *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 77–84. ACM.

Boudot, F. (2000). Efficient proofs that a committed number lies in an interval. In Preneel, B., editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 431–444. Springer, Heidelberg.

Boyen, X. (2004). Reusable cryptographic fuzzy extractors. In Atluri, V., Pfitzmann, B., and McDaniel, P., editors, *ACM CCS 2004*, pages 82–91. ACM Press.

Boyen, X., Dodis, Y., Katz, J., Ostrovsky, R., and Smith, A. (2005). Secure remote authentication using biometric data. In Cramer, R., editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 147–163. Springer, Heidelberg.

Boyko, V., MacKenzie, P. D., and Patel, S. (2000). Provably secure password-authenticated key exchange using Diffie-Hellman. In Preneel, B., editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 156–171. Springer, Heidelberg.

Brostoff, S. and Sasse, M. (2000). Are passfaces more usable than passwords?: A field trial investigation. *People and Computers*, pages 405–424.

Cachin, C., Micali, S., and Stadler, M. (1999). Computationally private information retrieval with polylogarithmic communication. In Stern, J., editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 402–414. Springer, Heidelberg.

Camacho, P. (2009). On the impossibility of batch update for cryptographic accumulators. Cryptology ePrint Archive, Report 2009/612. `http://eprint.iacr.org/2009/612`.

Camacho, P. and Hevia, A. (2010). On the impossibility of batch update for cryptographic accumulators. In Abdalla, M. and Barreto, P. S. L. M., editors, *LATINCRYPT 2010*, volume 6212 of *LNCS*, pages 178–188. Springer, Heidelberg.

Camacho, P., Hevia, A., Kiwi, M. A., and Opazo, R. (2008). Strong accumulators from collision-resistant hashing. In Wu, T.-C., Lei, C.-L., Rijmen, V., and Lee, D.-T., editors, *ISC 2008*, volume 5222 of *LNCS*, pages 471–486. Springer, Heidelberg.

Camenisch, J., Casati, N., Groß, T., and Shoup, V. (2010). Credential authenticated identification and key exchange. In Rabin, T., editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 255–276. Springer, Heidelberg.

Camenisch, J., Drijvers, M., and Tackmann, B. (2019). Multi-protocol UC and its use for building modular and efficient protocols. *IACR Cryptology ePrint Archive*, 2019:65.

Camenisch, J., Kohlweiss, M., and Soriente, C. (2009). An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In Jarecki, S. and Tsudik, G., editors, *PKC 2009*, volume 5443 of *LNCS*, pages 481–500. Springer, Heidelberg.

Camenisch, J., Krenn, S., and Shoup, V. (2011). A framework for practical universally composable zero-knowledge protocols. In Lee, D. H. and Wang, X., editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 449–467. Springer, Heidelberg.

Camenisch, J. and Lysyanskaya, A. (2002). Dynamic accumulators and application to efficient revocation of anonymous credentials. In Yung, M., editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 61–76. Springer, Heidelberg.

Camenisch, J. and Lysyanskaya, A. (2003). A signature scheme with efficient protocols. In Cimato, S., Galdi, C., and Persiano, G., editors, *SCN 02*, volume 2576 of *LNCS*, pages 268–289. Springer, Heidelberg.

Camenisch, J. and Michels, M. (1999a). Proving in zero-knowledge that a number is the product of two safe primes. In Stern, J., editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 107–122. Springer, Heidelberg.

Camenisch, J. and Michels, M. (1999b). Separability and efficiency for generic group signature schemes. In Wiener, M. J., editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 413–430. Springer, Heidelberg.

Camenisch, J. and Stadler, M. (1997). Efficient group signature schemes for large groups (extended abstract). In Kaliski Jr., B. S., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 410–424. Springer, Heidelberg.

Camenisch, J. and Van Herreweghen, E. (2002). Design and implementation of the idemix anonymous credential system. In Atluri, V., editor, *ACM CCS 2002*, pages 21–30. ACM Press.

Canetti, R. (2001). Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press.

Canetti, R. (2004). Universally composable signature, certification, and authentication. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, page 219. IEEE Computer Society.

Canetti, R. (2007). Obtaining universally compoable security: Towards the bare bones of trust (invited talk). In Kurosawa, K., editor, *ASIACRYPT 2007*, volume 4833 of *LNCS*, pages 88–112. Springer, Heidelberg.

Canetti, R., Dachman-Soled, D., Vaikuntanathan, V., and Wee, H. (2012). Efficient password authenticated key exchange via oblivious transfer. In Fischlin, M., Buchmann, J., and Manulis, M., editors, *PKC 2012*, volume 7293 of *LNCS*, pages 449–466. Springer, Heidelberg.

Canetti, R. and Fischlin, M. (2001). Universally composable commitments. In Kilian, J., editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 19–40. Springer, Heidelberg.

Canetti, R., Fuller, B., Paneth, O., Reyzin, L., and Smith, A. D. (2016). Reusable fuzzy extractors for low-entropy distributions. In Fischlin, M. and Coron, J.-S., editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 117–146. Springer, Heidelberg.

Canetti, R., Halevi, S., Katz, J., Lindell, Y., and MacKenzie, P. D. (2005). Universally composable password-based key exchange. In Cramer, R., editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 404–421. Springer, Heidelberg.

Canetti, R., Lindell, Y., Ostrovsky, R., and Sahai, A. (2002). Universally composable two-party and multi-party secure computation. In *34th ACM STOC*, pages 494–503. ACM Press.

Catalano, D. and Fiore, D. (2013). Vector commitments and their applications. In Kurosawa, K. and Hanaoka, G., editors, *PKC 2013*, volume 7778 of *LNCS*, pages 55–72. Springer, Heidelberg.

Chang, T. Y. (2011). An id-based multi-signer universal designated multi-verifier signature scheme. *Inf. Comput.*, 209(7):1007–1015.

Chaum, D. (1996). Private signature and proof systems. US Patent 5,493,614.

Choi, S. G., Katz, J., Kumaresan, R., and Zhou, H.-S. (2012). On the security of the "free-XOR" technique. In Cramer, R., editor, *TCC 2012*, volume 7194 of *LNCS*, pages 39–53. Springer, Heidelberg.

Chou, T. and Orlandi, C. (2015). The simplest protocol for oblivious transfer. In Lauter, K. E. and Rodríguez-Henríquez, F., editors, *LATINCRYPT 2015*, volume 9230 of *LNCS*, pages 40–58. Springer, Heidelberg.

Chow, S. S. M. (2006). Identity-based strong multi-designated verifiers signatures. In *Public Key Infrastructure, Third European PKI Workshop: Theory and Practice, EuroPKI 2006, Turin, Italy, June 19-20, 2006, Proceedings*, pages 257–259.

Chow, S. S. M. (2008). Multi-designated verifiers signatures revisited. *I. J. Network Security*, 7(3):348–357.

Cramer, R., Damgård, I. B., Döttling, N., Fehr, S., and Spini, G. (2015). Linear secret sharing schemes from error correcting codes and universal hash functions. In Oswald, E. and Fischlin, M., editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 313–336. Springer, Heidelberg.

Damgård, I., Haagh, H., Mercer, R., Nitulescu, A., Orlandi, C., and Yakoubov, S. (2019). Stronger notions and constructions for multi-designated verifier signatures. Cryptology ePrint Archive, Report 2019/1153. `https://eprint.iacr.org/2019/1153`.

Damgård, I. and Jurik, M. (2003). A length-flexible threshold cryptosystem with applications. In Safavi-Naini, R. and Seberry, J., editors, *ACISP 03*, volume 2727 of *LNCS*, pages 350–364. Springer, Heidelberg.

Damgård, I. and Triandopoulos, N. (2008). Supporting non-membership proofs with bilinear-map accumulators. Cryptology ePrint Archive, Report 2008/538. `http://eprint.iacr.org/2008/538`.

Daugman, J. (2004). How iris recognition works. *Circuits and Systems for Video Technology, IEEE Transactions on*, 14(1):21 – 30.

Derler, D., Hanser, C., and Slamanig, D. (2015). Revisiting cryptographic accumulators, additional properties and relations to other primitives. Cryptology ePrint Archive, Report 2015/087. `http://eprint.iacr.org/2015/087`.

Diffie, W. and Hellman, M. E. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654.

Dodis, Y., Kanukurthi, B., Katz, J., Reyzin, L., and Smith, A. (2012). Robust fuzzy extractors and authenticated key agreement from close secrets. *IEEE Transactions on Information Theory*, 58(9):6207–6222.

Dodis, Y., Ostrovsky, R., Reyzin, L., and Smith, A. (2008). Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. *SIAM Journal on Computing*, 38(1):97–139.

Dodis, Y., Reyzin, L., and Smith, A. (2004). Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In Cachin, C. and Camenisch, J., editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 523–540. Springer, Heidelberg.

Dupont, P.-A., Hesse, J., Pointcheval, D., Reyzin, L., and Yakoubov, S. (2018). Fuzzy password-authenticated key exchange. In Nielsen, J. B. and Rijmen, V., editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 393–424. Springer, Heidelberg.

Ellison, C., Hall, C., Milbert, R., and Schneier, B. (2000). Protecting secret keys with personal entropy. *Future Generation Computer Systems*, 16(4):311–318.

Fazio, N. and Nicolosi, A. (2003). Cryptographic accumulators: Definitions, constructions and applications.

Fiat, A. and Shamir, A. (1987). How to prove yourself: Practical solutions to identification and signature problems. In Odlyzko, A. M., editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg.

Freire, E. S. V., Hesse, J., and Hofheinz, D. (2014). Universally composable non-interactive key exchange. In Abdalla, M. and Prisco, R. D., editors, *SCN 14*, volume 8642 of *LNCS*, pages 1–20. Springer, Heidelberg.

Fujisaki, E. and Okamoto, T. (1997). Statistical zero knowledge protocols to prove modular polynomial relations. In Kaliski Jr., B. S., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 16–30. Springer, Heidelberg.

Garman, C., Green, M., and Miers, I. (2014). Decentralized anonymous credentials. In *NDSS 2014*. The Internet Society.

Gassend, B., Clarke, D. E., van Dijk, M., and Devadas, S. (2002). Silicon physical random functions. In Atluri, V., editor, *ACM CCS 2002*, pages 148–160. ACM Press.

Gasti, P., Sedenka, J., Yang, Q., Zhou, G., and Balagani, K. S. (2016). Secure, fast, and energy-efficient outsourced authentication for smartphones. *Trans. Info. For. Sec.*, 11(11):2556–2571.

Gennaro, R., Halevi, S., and Rabin, T. (1999). Secure hash-and-sign signatures without the random oracle. In Stern, J., editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 123–139. Springer, Heidelberg.

Ghosh, E., Ohrimenko, O., Papadopoulos, D., Tamassia, R., and Triandopoulos, N. (2016). Zero-knowledge accumulators and set algebra. In Cheon, J. H. and Takagi, T., editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 67–100. Springer, Heidelberg.

Gorbunov, S., Vaikuntanathan, V., and Wee, H. (2012). Functional encryption with bounded collusions via multi-party computation. In Safavi-Naini, R. and Canetti, R., editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 162–179. Springer, Heidelberg.

Goyal, V., Jain, A., Koppula, V., and Sahai, A. (2015). Functional encryption for randomized functionalities. In Dodis, Y. and Nielsen, J. B., editors, *TCC 2015, Part II*, volume 9015 of *LNCS*, pages 325–351. Springer, Heidelberg.

Han, J., Chung, A., Sinha, M. K., Harishankar, M., Pan, S., Noh, H. Y., Zhang, P., and Tague, P. (2018). Do you feel what i hear? Enabling autonomous IoT device pairing using different sensor types. In *IEEE Symposium on Security and Privacy*.

Han, J., Harishankar, M., Wang, X., Chung, A. J., and Tague, P. (2017). Convoy: Physical context verification for vehicle platoon admission. In *18th ACM International Workshop on Mobile Computing Systems and Applications (HotMobile)*.

Hofheinz, D. and Müller-Quade, J. (2004). Universally composable commitments using random oracles. In Naor, M., editor, *TCC 2004*, volume 2951 of *LNCS*, pages 58–76. Springer, Heidelberg.

Huang, Y., Katz, J., and Evans, D. (2012). Quid-Pro-Quo-tocols: Strengthening semi-honest protocols with dual execution. In *2012 IEEE Symposium on Security and Privacy*, pages 272–284. IEEE Computer Society Press.

Huang, Y., Katz, J., and Evans, D. (2013). Efficient secure two-party computation using symmetric cut-and-choose. In Canetti, R. and Garay, J. A., editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 18–35. Springer, Heidelberg.

Jakobsson, M., Sako, K., and Impagliazzo, R. (1996). Designated verifier proofs and their applications. In Maurer, U. M., editor, *EUROCRYPT'96*, volume 1070 of *LNCS*, pages 143–154. Springer, Heidelberg.

Juels, A. and Wattenberg, M. (1999). A fuzzy commitment scheme. In Motiwalla, J. and Tsudik, G., editors, *ACM CCS 99*, pages 28–36. ACM Press.

Katz, J. and Vaikuntanathan, V. (2011). Round-optimal password-based authenticated key exchange. In Ishai, Y., editor, *TCC 2011*, volume 6597 of *LNCS*, pages 293–310. Springer, Heidelberg.

Kolesnikov, V., Mohassel, P., and Rosulek, M. (2014). FleXOR: Flexible garbling for XOR gates that beats free-XOR. In Garay, J. A. and Gennaro, R., editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 440–457. Springer, Heidelberg.

Kolesnikov, V. and Rackoff, C. (2008). Password mistyping in two-factor-authenticated key exchange. In Aceto, L., Damgård, I., Goldberg, L. A., Halldórsson, M. M., Ingólfsdóttir, A., and Walukiewicz, I., editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 702–714. Springer, Heidelberg.

Kolesnikov, V. and Schneider, T. (2008). Improved garbled circuit: Free XOR gates and applications. In Aceto, L., Damgård, I., Goldberg, L. A., Halldórsson, M. M., Ingólfsdóttir, A., and Walukiewicz, I., editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, Heidelberg.

Laguillaumie, F. and Vergnaud, D. (2004). Multi-designated verifiers signatures. In López, J., Qing, S., and Okamoto, E., editors, *ICICS 04*, volume 3269 of *LNCS*, pages 495–507. Springer, Heidelberg.

Laguillaumie, F. and Vergnaud, D. (2007). Multi-designated verifiers signatures: anonymity without encryption. *Inf. Process. Lett.*, 102(2-3):127–132.

Li, J., Li, N., and Xue, R. (2007a). Universal accumulators with efficient nonmembership proofs. In Katz, J. and Yung, M., editors, *ACNS 07*, volume 4521 of *LNCS*, pages 253–269. Springer, Heidelberg.

Li, Y., Susilo, W., Mu, Y., and Pei, D. (2007b). Designated verifier signature: Definition, framework and new constructions. In *Ubiquitous Intelligence and Computing, 4th International Conference, UIC 2007, Hong Kong, China, July 11-13, 2007, Proceedings*, pages 1191–1200.

Lindell, Y. (2013). Fast cut-and-choose based protocols for malicious and covert adversaries. In Canetti, R. and Garay, J. A., editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 1–17. Springer, Heidelberg.

Lindell, Y. and Pinkas, B. (2011). Secure two-party computation via cut-and-choose oblivious transfer. In Ishai, Y., editor, *TCC 2011*, volume 6597 of *LNCS*, pages 329–346. Springer, Heidelberg.

Lindell, Y. and Pinkas, B. (2015). An efficient protocol for secure two-party computation in the presence of malicious adversaries. *Journal of Cryptology*, 28(2):312–350.

Lindell, Y. and Riva, B. (2014). Cut-and-choose Yao-based secure computation in the online/offline and batch settings. In Garay, J. A. and Gennaro, R., editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 476–494. Springer, Heidelberg.

Lipmaa, H. (2003). On diophantine complexity and statistical zero-knowledge arguments. In Laih, C.-S., editor, *ASIACRYPT 2003*, volume 2894 of *LNCS*, pages 398–415. Springer, Heidelberg.

Lipmaa, H. (2012). Secure accumulators from euclidean rings without trusted setup. In Bao, F., Samarati, P., and Zhou, J., editors, *ACNS 12*, volume 7341 of *LNCS*, pages 224–240. Springer, Heidelberg.

Marlinspike, M. (2013). Advanced cryptographic ratcheting.

Maurer, U. M. (1997). Information-theoretically secure secret-key agreement by NOT authenticated public discussion. In Fumy, W., editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 209–225. Springer, Heidelberg.

Mayrhofer, R. and Gellersen, H. (2009). Shake well before use: Intuitive and secure pairing of mobile devices. *IEEE Transactions on Mobile Computing*, 8(6):792–806.

McEliece, R. J. and Sarwate, D. V. (1981). On sharing secrets and Reed-Solomon codes. *Commun. ACM*, 24(9):583–584.

Micali, S., Rabin, M. O., and Vadhan, S. P. (1999). Verifiable random functions. In *40th FOCS*, pages 120–130. IEEE Computer Society Press.

Miers, I., Garman, C., Green, M., and Rubin, A. D. (2013). Zerocoin: Anonymous distributed E-cash from Bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411. IEEE Computer Society Press.

Ming, Y. and Wang, Y. (2008). Universal designated multi verifier signature scheme without random oracles. *Wuhan University Journal of Natural Sciences*, 13(6):685–691.

Mohassel, P. and Franklin, M. (2006). Efficiency tradeoffs for malicious two-party computation. In Yung, M., Dodis, Y., Kiayias, A., and Malkin, T., editors, *PKC 2006*, volume 3958 of *LNCS*, pages 458–473. Springer, Heidelberg.

Monrose, F., Reiter, M. K., and Wetzel, S. (2002). Password hardening based on keystroke dynamics. *International Journal of Information Security*, 1(2):69–83.

Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system.

Namecoin (https://www.namecoin.org/). Namecoin.

Ng, C. Y., Susilo, W., and Mu, Y. (2005). Universal designated multi verifier signature schemes. In *11th International Conference on Parallel and Distributed Systems, ICPADS 2005, Fuduoka, Japan, July 20-22, 2005*, pages 305–309.

Nguyen, L. (2005). Accumulators from bilinear pairings and applications. In Menezes, A., editor, *CT-RSA 2005*, volume 3376 of *LNCS*, pages 275–292. Springer, Heidelberg.

Nielsen, J. B. and Orlandi, C. (2009). LEGO for two-party secure computation. In Reingold, O., editor, *TCC 2009*, volume 5444 of *LNCS*, pages 368–386. Springer, Heidelberg.

Nisan, N. and Zuckerman, D. (1993). More deterministic simulation in logspace. In *25th ACM STOC*, pages 235–244. ACM Press.

Pappu, R., Recht, B., Taylor, J., and Gershenfeld, N. (2002). Physical one-way functions. *Science*, 297(5589):2026–2030.

Pinkas, B., Schneider, T., Smart, N. P., and Williams, S. C. (2009). Secure two-party computation is practical. In Matsui, M., editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, Heidelberg.

Pöhls, H. C. and Samelin, K. (2014). On updatable redactable signatures. In Boureanu, I., Owesarski, P., and Vaudenay, S., editors, *ACNS 14*, volume 8479 of *LNCS*, pages 457–475. Springer, Heidelberg.

Renner, R. and Wolf, S. (2004). The exact price for unconditionally secure asymmetric cryptography. In Cachin, C. and Camenisch, J., editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 109–125. Springer, Heidelberg.

Reyzin, L. and Yakoubov, S. (2016). Efficient asynchronous accumulators for distributed PKI. In Zikas, V. and De Prisco, R., editors, *SCN 16*, volume 9841 of *LNCS*, pages 292–309. Springer, Heidelberg.

Roth, R. (2006). *Introduction to Coding Theory*. Cambridge University Press, New York, NY, USA.

Sander, T. (1999). Efficient accumulators without trapdoor extended abstracts. In Varadharajan, V. and Mu, Y., editors, *ICICS 99*, volume 1726 of *LNCS*, pages 252–262. Springer, Heidelberg.

Seo, S., Hwang, J. Y., Choi, K. Y., and Lee, D. H. (2008). Identity-based universal designated multi-verifiers signature schemes. *Computer Standards & Interfaces*, 30(5):288–295.

Shailaja, G., Kumar, K. P., and Saxena, A. (2006). Universal designated multi verifier signature without random oracles. In *9th International Conference in Information Technology, ICIT 2006, Bhubaneswar, Orissa, India, 18-21 December 2006*, pages 168–171.

Shamir, A. and Tauman, Y. (2001). Improved online/offline signature schemes. In Kilian, J., editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 355–367. Springer, Heidelberg.

Shoup, V. (2001). A proposal for an ISO standard for public key encryption. Cryptology ePrint Archive, Report 2001/112. `http://eprint.iacr.org/2001/112`.

Slepak, G. (2013). Dnschain + okturtles. `http://okturtles.com/other/dnschain_okturtles_overview.pdf`.

Suh, G. E. and Devadas, S. (2007). Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th annual Design Automation Conference*, pages 9–14. ACM.

Tian, H. (2012). A new strong multiple designated verifiers signature. *IJGUC*, 3(1):1–11.

Tuyls, P., Schrijen, G. J., Skoric, B., van Geloven, J., Verhaegh, N., and Wolters, R. (2006). Read-proof hardware from protective coatings. In Goubin, L. and Matsui, M., editors, *CHES 2006*, volume 4249 of *LNCS*, pages 369–383. Springer, Heidelberg.

Vergnaud, D. (2006). New extensions of pairing-based signatures into universal designated verifier signatures. In Bugliesi, M., Preneel, B., Sassone, V., and Wegener, I., editors, *ICALP 2006, Part II*, volume 4052 of *LNCS*, pages 58–69. Springer, Heidelberg.

Wang, X., Ranellucci, S., and Katz, J. (2017). Authenticated garbling and efficient maliciously secure two-party computation. In Thuraisingham, B. M., Evans, D., Malkin, T., and Xu, D., editors, *ACM CCS 2017*, pages 21–37. ACM Press.

Woodage, J., Chatterjee, R., Dodis, Y., Juels, A., and Ristenpart, T. (2017). A new distribution-sensitive secure sketch and popularity-proportional hashing. In *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 682–710. Springer.

Wyner, A. D. (1975). The wire-tap channel. *The Bell System Technical Journali*, 54.

Yakoubov, S. (2017). A gentle introduction to Yao's garbled circuits. `http://web.mit.edu/sonka89/www/papers/2017ygc.pdf`.

Yakoubov, S., Fromknecht, C., and Velicanu, D. (2014). Certcoin: A namecoin based decentralized authentication system.

Yao, A. C.-C. (1986). How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press.

Yu, M.-D. M. and Devadas, S. (2010). Secure and robust error correction for physical unclonable functions. *IEEE Design & Test*, 27(1):48–65.

Zahur, S., Rosulek, M., and Evans, D. (2015). Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Oswald, E. and Fischlin, M., editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Heidelberg.

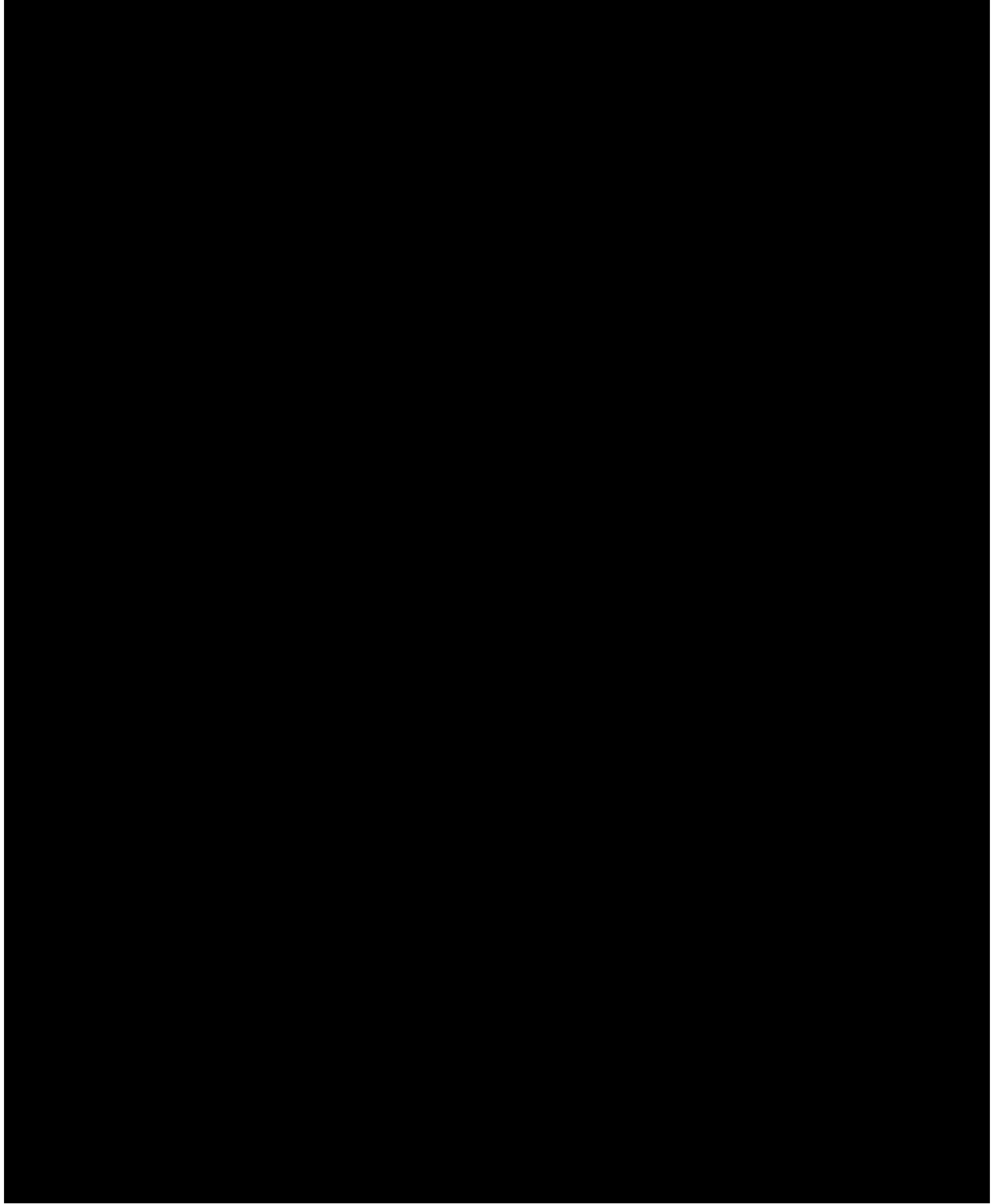
Zhang, Y., Au, M. H., Yang, G., and Susilo, W. (2012). (strong) multi-designated verifiers signatures secure against rogue key attack. In *Network and System Security - 6th International Conference, NSS 2012, Wuyishan, Fujian, China, November 21-23, 2012. Proceedings*, pages 334–347.

Zheng, Y. (1997). Digital signcryption or how to achieve cost(signature & encryption) $\ll$ cost(signature) + cost(encryption). In Kaliski Jr., B. S., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 165–179. Springer, Heidelberg.

Zviran, M. and Haga, W. J. (1993). A comparison of password techniques for multilevel authentication mechanisms. *The Computer Journal*, 36(3):227–237.

Chapter 7

Curriculum Vitae



253