Karlsruhe Institute of Technology

KASTEL

Universally Composable Verifiable Random Oracles

Master's Thesis of

Tim Scheurer

at the Department of Informatics
KASTEL – Institute of Information Security and Dependability

Reviewer:	Prof. Dr. Jörn Müller-Quade
Second reviewer:	Prof. Dr. Thorsten Strufe
Advisor:	M.Sc. Michael Klooß

18. April 2022 – 18. October 2022

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 18.10.2022

. . . .

(Tim Scheurer)

Abstract

Random Oracles are frequently used in cryptography to construct very efficient instantiations of powerful cryptographic primitives. Nevertheless, this practice is not sound in general as the uninstantiability result by Halevi *et al.* (JACM '04) for random oracles by any function family shows.

The Random Oracle Model can be made sound by instantiating the random oracle not with a locally computable hash-function, but by using an *interactive* protocol. In reality, such an interactive protocol may involve a trusted server that is reachable at some address over the internet. This server could use one of the usual techniques such as *lazy sampling* or the evaluation of a pseudo-random function to provide the random oracle.

One obvious drawback of this approach is a large amount of interaction each time a random oracle output is involved in some computation. We want to reduce this interaction to a minimum. First, it is clear that to circumvent the impossibility result from above, a party holding some input and wishing to know the corresponding random oracle output has to interact with another party in some way. This is, however, not the only way in which random oracles are commonly used within cryptographic protocols. Another use case first has a party A query the oracle on some input, thereby receiving a hash. A subsequently sends both input and hash (all in the context of some protocol) to a second party B and wants to convince B of the fact that the oracle has been queried correctly. A simple way for B to check correctness is to simply recompute the hash, but in our setting, this entails interaction.

The wish to allow this second use case to be non-interactive leads to the notion of a *Verifiable Random Oracle* (VRO) as an augmentation of a random oracle. Abstractly, a VRO consists of two oracles. The first part behaves like a random oracle whose output is augmented with a *proof of correct evaluation*. Using this proof, the second oracle can be used to publicly verify the correct evaluation of the random function. While this formulation does not inherently force verification to be non-interactive, the introduction of explicit proofs does at least allow for it.

In this thesis, we first formalize the notion of a VRO in the Universal-Composability framework of Canetti (FOCS '01). We then apply VROs to two cryptographic applications formulated in the Random Oracle Model and show that they remain sound. To show that our definition is sensible, we give several protocols realizing the ideal VRO functionality ranging from protocols for a single trusted party to distributed protocols allowing for some malicious corruption. We also compare VROs to existing primitives.

Zusammenfassung

Random Oracles werden häufig in der Kryptographie eingesetzt um sehr effiziente Instanziierungen mächtiger kryptographischer Primitive zu konstruieren. Jedoch ist diese Praxis im Allgemeinen nicht zulässig wie verschiedene Nicht-Instanziierungs-Ergebnisse für Random Oracles mittels lokal berechenbarer Familien von Funktionen durch Halevi *et al.* (JACM '04) zeigt.

Die Random Oracle Modell kann sicher eingesetzt werden, indem Random Oracles nicht mit einer lokal berechenbaren Hashfunktion, sondern stattdessen mit einem *interaktiven* Protokoll instanziiert werden. In der realen Welt könnte solch ein interaktives Protokoll beispielsweise aus einem vertrauenswürdigen Server, welcher über das Internet erreichbar ist, bestehen. Dieser Server würde sodann eine der bekannten Techniken wie *lazy sampling* oder das Auswerten einer Pseudo-Zufälligen Funktion verwenden, um die Funktionalität eines Random Oracle bereitzustellen.

Ein klarer Nachteil dieses Ansatzes ist die große Menge an Interaktion, die bei jeder Berechnung, die eine Auswertung des Random Oracle beinhaltet, nötig ist. Wir wollen diese Interaktion auf ein Minimum reduzieren. Um obiges Unmöglichkeitsresultat zu umgehen, muss die Auswertung des Random Oracle auf einer frischen Eingabe Interaktion der auswertenden Partei mit einer anderen Partei beinhalten. Dies ist jedoch nicht der einzige Verwendungszweck von Random Oracles, der häufig in kryptographischen Protokollen auftritt. Bei einem weiteren solchen Zweck wertet zunächst eine Partei A das Orakel auf einer Eingabe aus und erhält einen Hashwert. Im Anschluss sendet A Eingabe und Ausgabe (im Kontext eines Protokolls) an eine zweite Partei B und möchte B davon überzeugen, dass das Random Oracle korrekt ausgewertet wurde. Eine einfache Möglichkeit dies zu prüfen besteht darin, dass B selbst eine Auswertung des Random Oracle auf der erhaltenen Eingabe tätigt und die beiden Ausgaben vergleicht. In unserem Kontext benötigt dies jedoch erneut Interaktion.

Der Wunsch diesen zweiten Verwendungszweck nicht-interaktiv zu machen führt uns zum Begriff eines *Verifiable Random Oracle* (VRO) als Erweiterung eines Random Oracle. Abstrakt besteht ein VRO aus zwei Orakeln. Das erste Orakel verhält sich wie ein Random Oracle dessen Ausgabe um einen *Korrektheitsbeweis* erweitert wurde. Mit Hilfe dieses Beweises kann das zweite Orakel dazu verwendet werden öffentlich die korrekte Auswertung des Random Oracle zu verifizieren. Obwohl diese Orakel-basierte Formulierung nicht notwendigerweise nicht-interaktive Verifikation besitzt, so erlaubt jedoch die Einführung epliziter Korrektheitsbeweise dies.

In dieser Masterarbeit formalisieren wir zunächst den Begriff eines VRO im Universal-Composability Framework von Canetti (FOCS '01). Danach wenden wir VROs auf zwei kryptographische Anwendungen an, die in ihrer ussprünglichen Formulierung das Random

Oracle Modell verwenden, und zeigen, das deren Sicherheitseigenschaften erhalten bleiben. Um zu zeigen, dass unsere Definition realisierbar ist, konstruieren wir mehrere Protokolle, die die ideale VRO Funktionalität realisieren. Diese reichen von Protokollen für eine einzelne vertrausenswürdige Partei bis hin zu verteilten Protokollen, die eine gewisse Menge an böswilliger Korruption erlauben. Wir vergleichen weiterhin VROs mit ähnlichen existierenden Primitiven.

Contents

Abstract i

Zusammenfassung iii

1. Introduction 1
- 1.1. Motivation 1
- 1.2. Goal 3
- 1.3. Technical Overview 4
  - 1.3.1. The Ideal Functionality 4
  - 1.3.2. Applications 5
  - 1.3.3. Challenges 6
  - 1.3.4. Instantiations 7
- 1.4. Related Work 10
- 1.5. Contribution 11
- 1.6. Organization 11

2. Preliminaries 13
- 2.1. Definitions and Notation 13
- 2.2. The Random Oracle Model 14
- 2.3. Multi-Party Computation 14
- 2.4. The Universal Composability Framework 15
  - 2.4.1. The Universal-Composition Theorem and Hybrid Models 18
  - 2.4.2. Some Ideal Functionalities and Notation 19
- 2.5. Cryptographic Primitives 21
  - 2.5.1. Verifiable Random Functions 21
  - 2.5.2. Simulatable Verifiable Random Functions 22
  - 2.5.3. Fully-Homomorphic Encryption 23
  - 2.5.4. Proof Systems and Related Notions 27
  - 2.5.5. Extractable Non-Interactive Witness-Indistinguishable Arguments 32
  - 2.5.6. Sigma Protocols 34

3. The Verifiable Random Oracle Model 37
- 3.1. The Ideal VRO Functionality 37
  - 3.1.1. The Actual Functionality 37
  - 3.1.2. Comparison with Existing Functionalities 41
  - 3.1.3. Design Decisions 42
- 3.2. The VROM 45
- 3.3. Comparing Random Oracles to Verifiable Random Oracles 46

3.4. An Alternative Version 48

4. Applications 49

4.1. Full-Domain Hash 49
4.1.1. Definition of Security 50
4.1.2. FDH in the ROM 52
4.1.3. FDH in the VROM 52
4.1.4. Proof of Security 55
4.1.5. Final Thoughts 58
4.2. The Fischlin Transformation 58
4.2.1. Definition of Security 59
4.2.2. The Fischlin Transformation in the ROM 62
4.2.3. The Fischlin Transformation in the VROM 69
4.2.4. Proof of Security 70
4.2.5. Universally Composable Transferable Zero-Knowledge 74
4.2.6. In the VROM 83
4.2.7. Removing the Need for Unique Proofs 86
4.2.8. Final Thoughts 88

5. VRO Instantiations 91

5.1. ROM Instantiation 91
5.1.1. Small Codomains Using Truncation 92
5.2. Trusted Party Instantiation 94
5.3. Allowing Corruption 99
5.3.1. Distributing Protocols for Trusted Parties 99
5.3.2. The PRF Construction 100
5.4. Adding Privacy Using FHE-Encryption 102
5.4.1. Goals 103
5.4.2. Rationale for FHE 103
5.4.3. Building Blocks 103
5.4.4. The Protocol 105
5.4.5. Proof of Security 108
5.4.6. Reducing to Semantic Security 125
5.4.7. Using Singly-Homomorphic Encryption 126
5.4.8. Relying on Preprocessing 126
5.4.9. Eliminating Secure Channels 126
5.4.10. Analyzing Efficiency 127
5.4.11. Analyzing Scalability 128
5.5. Relaxing the VRO 129
5.5.1. Revisiting the PRF Construction 129
5.6. Strengthening the VRO 133
5.6.1. Stronger Proofs 133
5.6.2. Hiding Proofs 135
5.7. Hybrid Instantiations 135
5.8. Multiple Sessions 139

5.9. Semi-Honest Adversaries 140
5.10. General MPC 141
5.10.1. Client-Server Protocols 142
5.10.2. General Multi-Party Protocols 144

6. Related Primitives **145**
6.1. Comparing VRO Definitions 145
6.1.1. Syntax 145
6.1.2. Security 146
6.1.3. Comparison 147
6.1.4. Other Differences 148
6.2. Comparing VRO and OPRF Variations 149
6.2.1. OPRF Variations 149
6.2.2. Comparison 155
6.2.3. Evaluating the FHE Construction 156
6.3. Generic Constructions from OPRF 156
6.3.1. Relation between OPRF and Hybrid Instantiations 159
6.4. Using a Concrete OPRF 159
6.5. VOPRF from VRO 161
6.5.1. Naive Approach 161
6.5.2. Arguments Against an Unconditional Construction 162
6.5.3. Relying on Computational Assumptions 162

7. Future Work **163**
7.1. Adaptive Adversaries 163
7.2. Standalone Security 163
7.3. More Tasks 164
7.4. Weaker Randomness Guarantees 164
7.5. Global VRO 164
7.6. More Efficient Instantiations 164

8. Conclusion **165**

Bibliography **167**

A. Appendix **173**
A.1. Standard Definitions 173
A.1.1. Pseudo-Random Functions 173
A.1.2. Trapdoor One-Way Permutations 173
A.1.3. Digital Signature Schemes 174
A.2. An Ideal VRO Functionality with Algorithmic Verification 176
A.3. Additional Remarks 179
A.3.1. Remarks about the FHE Protocol 179
A.3.2. Remarks About OPRF Variants 182

A.4. Relaxing the VRO (Continued) 183
A.4.1. Leaking Only the Hash 183
A.4.2. Leaking Both Input and Hash 183
A.4.3. Leaking Only the Input 186
A.5. An Attack on the Randomized Fischlin Transform 187
A.6. Reducing to Semantic Security 192
A.7. Simplifying the FHE Construction for Semi-Honest Adversaries 196
A.8. Using VROs in OPRF Protocols 197

List of Figures

1.1. Typical interaction between two parties $\mathcal{A}$ and $\mathcal{B}$ using a random oracle RO. 3

2.1. The UC experiment with real adversary $\mathcal{A}$ and protocol π (left) and with simulator $\mathcal{S}$ and ideal functionality $\mathcal{F}$ (right). The multiple lines indicate that $\mathcal{Z}$ may interact with multiple protocol parties. 16
2.2. The security game for a simulatable VRF. 23
2.3. The zero-knowledge security game for NIZKs in the ROM. 29
2.4. The two UC Zero-Knowledge functionalities we use. The parts not highlighted in gray make up the NIZK functionality. Including them yields the TZK functionality. Where a highlighted passage succeeds an underlined one, the highlighted portion is meant to replace the underlined one. . . . 32
2.5. The WI game for NIWI protocols. 34
2.6. Schematic run of a Σ-protocol. 34

3.1. The UC Verifiable Random Oracle functionality. 39
3.2. The UC Signature functionality. 40

4.1. The FDH-ROM procedures. 52
4.2. The FDH-VROM procedures. 53
4.3. The original (left) and the randomized version (right) of the Fischlin prover. 65
4.4. The Fischlin verifier in the ROM. 66
4.5. The Fischlin zero-knowledge simulator in the ROM. 67
4.6. The online extractor in the ROM. 69
4.7. The Fischlin prover in the VROM. 71
4.8. The Fischlin verifier in the VROM. 71
4.9. The Fischlin online extractor in the VROM. 71
4.10. The Fischlin zero-knowledge simulator in the VROM. 73

5.1. The algorithms run by a party $\mathcal{P}$. 95
5.2. The algorithms run by the trusted party $\mathcal{T}$. 95
5.3. The client algorithms of π_{PRF} 101
5.4. The server algorithms of π_{PRF} from the perspective of server $\mathcal{S}_n$ 102
5.5. The UC Zero-Knowledge functionality. 105
5.6. The UC Bulletin-Board functionality. 106
5.7. The client algorithms for π_{FHE} from the perspective of a party $\mathcal{P}$. 109
5.8. The server algorithms for π_{FHE} from the perspective of server $\mathcal{P}_n$. 110
5.9. The algorithms used by the simulator. 112
5.10. The Prove algorithm sent by the simulator. 131
5.11. The client algorithms for π_{Hyb}. 137

5.12. Visualization of the typical flow of messages for a hash query by a caller C on input q with servers $\mathcal{S}_1, \ldots, \mathcal{S}_n$ and ideal functionality $\mathcal{F}$. Time advances from left to right. 143

6.1. The Pseudorandomness game for a gVRO. 147
6.2. The Weak Unforgeability game for a gVRO. 147
6.3. An ideal OPRF functionality for evaluating a random function. 151
6.4. An ideal OPRF functionality parametrized by a PRF F which it evaluates. 151
6.5. An ideal VOPRF functionality (taken from [2], figure 1). 153
6.6. A second ideal VOPRF functionality (taken from [66], figure 2). 154

A.1. The PRF security game. 174
A.2. The TDOWP security game. 174
A.3. The sEUF-CMA security game. 175
A.4. An alternative variant of the $\mathcal{F}_{VRO}$ functionality where proofs are verified algorithmically. 178
A.5. The UC Key-Registration functionality. 181

List of Protocols

1.1. Simplified pseudo-code for the PRF Construction. 8
1.2. Simplified pseudo-code for the augmented PRF Construction. 9

4.1. The sEUF-CMA security game in the VROM. 51
4.2. The reduction from EUF-CMA in the VROM to EUF-CMA in the ROM. For space reasons we use the following abbreviations: `HashProof`(`HP`), `Hash`(`H`), `Hashing`(`HA`), `SimInfo`(`SI`), `Verify`(`VFY`), `Verified`(`V`), `Key`(`K`). 56
4.3. The reduction from online extractability in the ROM to online extractability in the VROM. 75

1. Introduction

In this chapter, we first motivate our approach and our goals for this thesis. Then we give a technical overview of our results. We briefly discuss related work before stating our contribution. We conclude this chapter by laying out the organization of the remaining parts of this thesis.

1.1. Motivation

While giving all parties of a protocol oracle access to a random function had been previously done, *random oracles* (RO) and the *random oracle model* (ROM) have been explicitly defined in the seminal paper by Bellare and Rogaway in 1993 [7]. Since then, random oracles have been used to construct many efficient constructions for cryptographic primitives. These include BLS short signatures [12] where signatures consist of a single group element and which in addition allow for *aggregation* as well as *batch verification* of signatures, and OEAP encryption [6]. For some primitives, either the first known construction was in the ROM and only later constructions in the standard model[1] were found, e.g. the first HIBE by Gentry and Silverberg [56] and later the scheme by Boneh and Boyen [11], or only ROM-based constructions are efficient, e.g. compare the efficient RSA-based signature scheme PSS [8] and the (inefficient) scheme by Hohenberger and Waters [65].

Ideal Properties of Random Oracles The strength of random oracles lies in their ideal properties which are much stronger than standard model properties such as *collision resistance* or *universal one-wayness* [76]. The main properties which can be identified are *observability/extractability* and *programmability*. Informally, the former means that a party claiming to know the RO response for some input x either has queried the RO on x, in which case a reduction algorithm providing the RO can *extract* x, or otherwise, the party (even allowing it to be unbounded) has a chance no better than guessing of knowing the correct hash. Programmability, on the other hand, even allows for active interference by the entity providing the oracle. It allows setting outputs of the RO to some chosen value under the restriction that they remain independent and uniform. This gives the reduction great power, e.g. by allowing it to learn the pre-image of each response under some one-way permutation or to embed a challenge to a hard problem.

Intuitively, both of these properties crucially rely on the fact that the RO is provided as an oracle, which in a reduction is provided by the reduction algorithm. And indeed, Halevi *et al.* [22] showed that the *random oracle methodology* of instantiating random oracles with hash function families $\mathcal{H} = \{H_{\mathsf{pk}}\}_{\mathsf{pk} \in \mathcal{PK}}$ is in general unsound. This has sparked a controversy over the continued use of random oracles with opponents arguing

[1]That is, the model where there are no random oracles.

that provable security is necessary while proponents refer to the fact that no practical attack on a cryptographic primitive relied on the discrepancy between the hash function used and a RO. Some even argue that (more complicated) schemes designed to not rely on random oracles have introduced additional weaknesses which did not exist in the original RO-based schemes [70].

Introducing Interaction As has been shown to be the case in various cryptographic contexts, the above discussion can be wholly circumvented by using *interaction* to evaluate the RO. The attack in [22] is based on the fact that the adversary is given the code of the hash function that is used within the attacked protocol. By restricting its access to this code, e.g. by letting it be executed by a trusted third party. When using a *pseudo-random function* as the hash function, this approach can be shown to securely instantiate a RO.

Such interaction, while allowing for provably secure schemes, is oftentimes either unwanted or impossible in the case of isolated systems. Situations where interaction is unwanted include non-interactive zero-knowledge proofs, e.g. those achieved by using the Fiat-Shamir transform [44]. Their main purpose lies in generating proofs π, attesting to the correctness of some statement x, such that π can at some later point in time be verified without the verifier having to contact any other party. While a fully interactive RO may still have the advantage over interactive proofs of reducing the required availability of the prover[2], this situation seems unsatisfactory. For the prover, on the other hand, some amount of interaction may be tolerable. Proofs can be generated once and later be non-interactively verified by many other parties.

An Alternative Use Case of Random Oracles The main insight on the path to achieving partially non-interactive use of random oracles is the following. In many protocols, parties which collectively could be called *verifiers* (e.g. of signatures, zero-knowledge proofs, etc.) only contain instructions involving a random oracle RO of the form $h \stackrel{?}{=} \mathsf{RO}(q)$ and where both q and h were in some way computed from the parties' input or the protocol messages received so far. A typical such interaction is shown in Figure 1.1. The party $\mathcal{A}$ has input q and evaluates RO at this input, thereby obtaining the hash h. Both data are sent within some protocol message m to a second party $\mathcal{B}$ which aborts if the evaluation was done incorrectly. Based on the extractability of random oracles described above, such comparisons can only succeed with large probability if q has previously been submitted to the RO.[3] This submission was presumably done by another protocol party wishing to convince the verifier that it used the honest RO output in its computation.

Adding an Interface for Verification Our approach is based on this last observation. Informally, we augment the notion of a random oracle RO by another oracle ROVfy. As before, RO is used to query the RO on inputs q. In addition to the hash h, we allow RO to output a second value π. The oracle ROVfy receives as input a triple (q, h, π) and returns a bit $b \in \{0, 1\}$ indicating whether $h = \mathsf{RO}(q)$ holds (with respect to π). While we have seen that queries to RO necessarily involve interaction, our goal is to instantiate ROVfy without having to resort to interaction. To be called a *proof*, π should satisfy some

[2]With this we mean the ability for a verifier to interact with the prover, i.e. over some communication network.

[3]Briefly assuming the common case of a super-polynomial codomain.

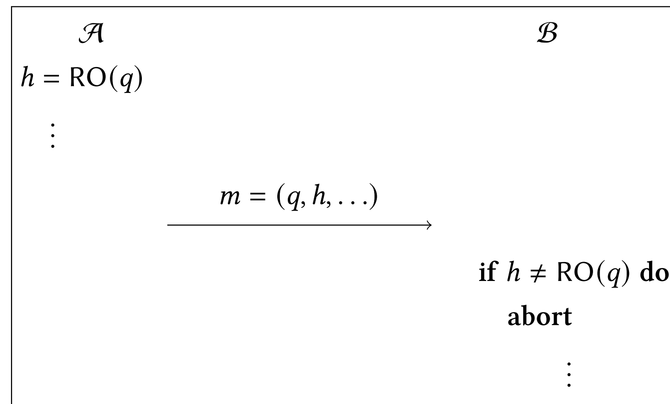


Figure 1.1.: Typical interaction between two parties $\mathcal{A}$ and $\mathcal{B}$ using a random oracle RO.

form of unforgeability condition. As a minimal requirement, it should be computationally infeasible (or even impossible) to come up with a proof π' attesting to the fact that h' is the hash associated to q, if in reality $h = \mathrm{RO}(q)$ with $h \neq h'$. In addition to this basic property, we have identified further security properties and captured them within an ideal functionality in the UC framework. This, among others, includes modeling verification of proofs as a local computation which may not involve any interaction with other parties.

1.2. Goal

We describe the goals we are striving to achieve in this thesis with our model and instantiations thereof.

Modular Analysis Random oracles are almost exclusively used within other protocols. As VROs can be viewed as a replacement for random oracles in certain contexts, the same holds for them. The soundness of this practice relies on strong composability guarantees by the (V)RO. Definitions that are based on games played between a challenger and an adversary do not provide any composability guarantees *per se*. Simulation-based definitions are intended to provide the required composability but differ in their strength. While *stand-alone* simulation-based security is secure under *non-concurrent* composition (see [57]), i.e. only one protocol may be executing at a time, but multiple protocols may be "active" at the same time, UC security allows for arbitrary composition. As the typical use of random oracles is concurrent with the surrounding protocol, only striving for the strongest form of composability, i.e. UC, seems to be sufficient.

Provable Security In contrast to the heuristic security guarantees provided by instantiating random oracles using locally computable families of hash functions, we aim to realize our VRO definition using a provably secure protocol. Such a protocol can thus be employed in contexts in which one does not wish to rely on heuristics.

Corruption Resistance To further increase the resilience, we, apart from realizations involving a single trusted server, seek to reduce trust in individual parties by constructing distributed protocols for a VRO. These protocols should be resistant to some form of

passive and/or active corruption by the parties providing the VRO. For parties using the VRO, no amount of deviation from the honest protocol should allow them to compromise security.

1.3. Technical Overview

In this section, we give a high-level technical overview of our definitions and constructions.

1.3.1. The Ideal Functionality

We give an overview of the ideal VRO functionality. The formal definition will be in the UC framework by Canetti [18]. The description given here conveys the main intended behavior, but exact security guarantees regarding the use of the functionality within other protocols can only be derived from a formal specification and have to be viewed in the context of the execution model defined by the UC framework.

Introduction to the UC Framework We assume familiarity with the UC framework but quickly recount the main structure. Security of a protocol π is defined by comparing an execution of π to an execution where the participants interact with an ideal functionality $\mathcal{F}$ capturing the intended behavior. The former is referred to as the *real interaction* and the latter as the *ideal interaction*. In the real interaction, there is an additional entity $\mathcal{A}$ called the *adversary* which tries to break the security of π by corrupting parties according to some corruption model. In our case we allow the adversary to do *static malicious* corruptions which means that the adversary can control the complete behavior of corrupted parties, but can only corrupt parties which have not yet participated otherwise in the protocol. As the ideal interaction is secure by definition, to argue that π is as secure while being attacked by $\mathcal{A}$ we compare the two interactions. If the interactions are indistinguishable then π has to be as secure as running the ideal interaction. Indistinguishability is defined with respect to an additional entity $\mathcal{Z}$ called the *environment* which executes either a session involving π or $\mathcal{F}$ and gives input and receives output from uncorrupted parties. It also interacts with $\mathcal{A}$ and can communicate with it throughout the execution. As $\mathcal{A}$ only exists in the real world, an *ideal adversary* $\mathcal{S}$ also called the *simulator* is introduced. $\mathcal{S}$, while only interacting with $\mathcal{F}$, has to give $\mathcal{Z}$ the same interface as if it was interacting with $\mathcal{A}$ and π. $\mathcal{Z}$ acts as an "interactive distinguisher" and π is secure if the output distributions of $\mathcal{Z}$ are indistinguishable when comparing runs of the real and the ideal interaction.

Description of the Functionality Ideal functionalities in the UC framework are structured into what we chose to call *tasks*. We now describe the tasks of our ideal VRO functionality which is called $\mathcal{F}_{VRO}$. In addition to allowing for the retrieval of some initialization information, e.g. a key required during the otherwise non-interactive verification process, the functionality essentially allows parties to submit queries for two different tasks. The first task consists in querying the random oracle portion of the VRO on some input q and receiving both the output h and proof π.[4] $\mathcal{F}_{VRO}$ guarantees consistency; queries for the same input q made by different parties in the same session of $\mathcal{F}_{VRO}$ receive the same

[4] $\pi = \bot$ is allowed.

h. Outputs for different inputs behave as if they were sampled independently from the codomain $\mathcal{H}$ and different instances of $\mathcal{F}_{VRO}$ are totally independent. As the UC adversary is in control of the network, to model the necessary interactiveness of this task we have to allow the adversary to prohibit the delivery of individual responses.

So far we have not described how proofs are generated by $\mathcal{F}_{VRO}$. As in this idealized context, proofs are merely strings without any inherent meaning, we let their concrete form be chosen by the adversary. There are some further technicalities to this which we will describe in the main part of the thesis. The real definition involves letting the adversary provide an algorithm Prove for generating proof strings which can be run within $\mathcal{F}_{VRO}$ without involving the adversary except during an initialization step. Briefly, this is necessary as we want to allow proof strings for input q and output h to depend on both of these data, but do not wish to provide them to the adversary.

For technical reasons we also have to provide *some* leakage to the adversary. Namely, for each hash query with input q by some party $\mathcal{P}$ we hand the length $||q||$ of q to the adversary. The reason for the latter is that during simulation the simulator has to be able to extract q from any corrupted party making a hash query. At the same time, an honest party doing a hash query has to be able to hide its input q from the adversary in the real execution. If we want to allow verifiable random oracles with *a priori* unbounded input length, these two requirements are incompatible unless we add the aforementioned leakage. Informing the simulator about the identity of $\mathcal{P}$ is necessary to allow the simulator to provide an accurate simulation towards the (also simulated) adversary.

Verification requests contain an input q, the supposed hash h, a proof π, and potentially a verification key vk obtained during initialization. Answers to such requests contain a single bit b. Receiving $b = 1$ signals to the party making the request that h is the correct hash associated with q in this session of $\mathcal{F}_{VRO}$. On the other hand, $b = 0$ can mean one of two things. Either h is not the correct hash of q, or π is an incorrect proof. These two properties ensure that proofs are perfectly unforgeable. Note that we explicitly allow verification requests for correct (q, h), but containing some π which was not previously output by $\mathcal{F}_{VRO}$ to verify under certain conditions. This relaxation is similar to the distinction between EUF-CMA-secure and sEUF-CMA-secure signature schemes.

To force protocols attempting to realize the functionality to have a non-interactive verification step, the adversary is not given the ability to delay responses to verification requests. These are guaranteed to be answered immediately. As part of this modeling, we use the framework developed by Camenisch *et al.* in [16] which allows forcing the adversary to answer certain types of messages within the same activation. Any protocol for $\mathcal{F}_{VRO}$ must thus realize verification using only local computations and possibly calls to ideal functionalities which themselves are guaranteed to respond and can not be delayed by the adversary.

1.3.2. Applications

Next, we apply $\mathcal{F}_{VRO}$ to two cryptographic applications.

Full-Domain Hash The first is the *full-domain hash* (FDH) signature scheme. Briefly, a signature of a message m is obtained by hashing $h = \mathsf{RO}(m)$ using the random oracle

RO and then computing the pre-image of h under a trap-door one-way permutation (TDOWP) f, i.e. $\sigma = f^{-1}(h)$. In the security proof of FDH, essential use is made of both the extractability as well as the programmability of the RO. The reduction is given a value x in the domain of the TDOWP and is successful if it can compute $f^{-1}(x)$. To achieve this while having to answer queries to the signing oracle it has to provide to the underlying FDH adversary, x is embedded at some randomly chosen RO query m. All other queries are answered in such a way that the reduction knows a pre-image under f of the returned hash. Using this pre-image, signing queries for RO input corresponding to it can be answered with a valid signature. If the FDH adversary is successful, it will produce a pair (m^*, σ^*) such that σ^* is a valid signature for m^* and m^* has not been submitted to the signing oracle. The reduction now hopes for $m = m^*$ as then σ^* is the pre-image of x under f. Note that extractability has indeed been used to ensure that m^* is among the queries, except with probability one over the cardinality of the domain of the TDOWP.

In the main part of this thesis, we show how to modify this variant of FDH in the ROM to rely on $\mathcal{F}_{VRO}$ instead. The modification is based on the observation that verifying a signature σ for message m consists in checking $\mathrm{RO}(m) = f(\sigma)$. Hence, we let the signer include a proof π attesting to $(m, \mathrm{RO}(m))$ as well as $\mathrm{RO}(m)$ itself in the signature. The new verification procedure involves performing the equality check not by re-querying m, but by verifying π. We additionally show that $\mathrm{RO}(m)$ can be excluded from the signature while retaining security.

Fischlin Transformation The second application is to the Fischlin transform, which is a generic transformation for a certain class of Σ-protocols to make them non-interactive in the ROM. The differences between it and the Fiat-Shamir transform lie in the *online extractability* of the Fischlin transform. Informally, this requires being able to extract witnesses for valid proofs from the RO queries of the prover and in particular without having to rewind the prover. We show that also in this case the random oracle can soundly be replaced with $\mathcal{F}_{VRO}$. This application was the main motivation for restricting the leakage of $\mathcal{F}_{VRO}$ upon each query to the length of the input. Leaking the full input would allow the adversary to extract witnesses from honest provers, based on the online extractability of the Fischlin-transformed protocol.

We even go a step further and show that both protocols transformed with the Fischlin transform in the ROM as well as those using $\mathcal{F}_{VRO}$ realize an ideal functionality $\mathcal{F}_{TZK}$ modeling transferable zero-knowledge proofs.[5] Note that for this application, a slight modification to the $\mathcal{F}_{VRO}$-based version of the Fischlin transform has to be made to reestablish the non-malleability of proofs required by $\mathcal{F}_{TZK}$. This property is destroyed by naïvely including proofs produced by $\mathcal{F}_{VRO}$ in the zero-knowledge proof. Our solution to this problem relies on strong one-time signatures.

1.3.3. Challenges

Before we describe the protocols with which $\mathcal{F}_{VRO}$ can be realized, we go over the main technical difficulties posed by our definition.

[5] $\mathcal{F}_{TZK}$ is a variant of a non-interactive zero-knowledge proof functionality $\mathcal{F}_{NIZK}$ which we have defined for this thesis.

- **Unpredictability:** Hash values have to be *offline unpredictable* to the adversary/environment. This means that the adversary can not be allowed to possess knowledge of any keying material which would allow it to predict the hash h corresponding to an input q without actually querying the VRO. Such keying material can not exist in the ideal interaction as there hashes are chosen lazily by $\mathcal{F}_{VRO}$, hence the simulator can not provide such information to the environment.

- **Non-Interactive Verification:** Answers to verification requests have to be either locally computable or at least they can not be interfered with by the adversary.

- **Unforgeable Proofs:** It has to be infeasible for the adversary, based on the knowledge it obtains either by partially controlling the entities computing hash values or by interacting with honest protocol parties in an arbitrary manner, to come up with a valid proof π for some (q, h) such that h is not the *true* hash value for q.

- **Input/Output Hiding:** As stated above, the only information the adversary learns upon a hash query is the length of the input, the identity of the party making the query, and the fact that a hash query has occurred. This means both the full q and h have to be hidden from the adversary in any protocol trying to realize $\mathcal{F}_{VRO}$. This prevents any protocol where q is sent in the clear to some set of servers, even if no single server can compute h and the proof π on its own. As we will see, this turns out to be the main obstacle in realizing $\mathcal{F}_{VRO}$.

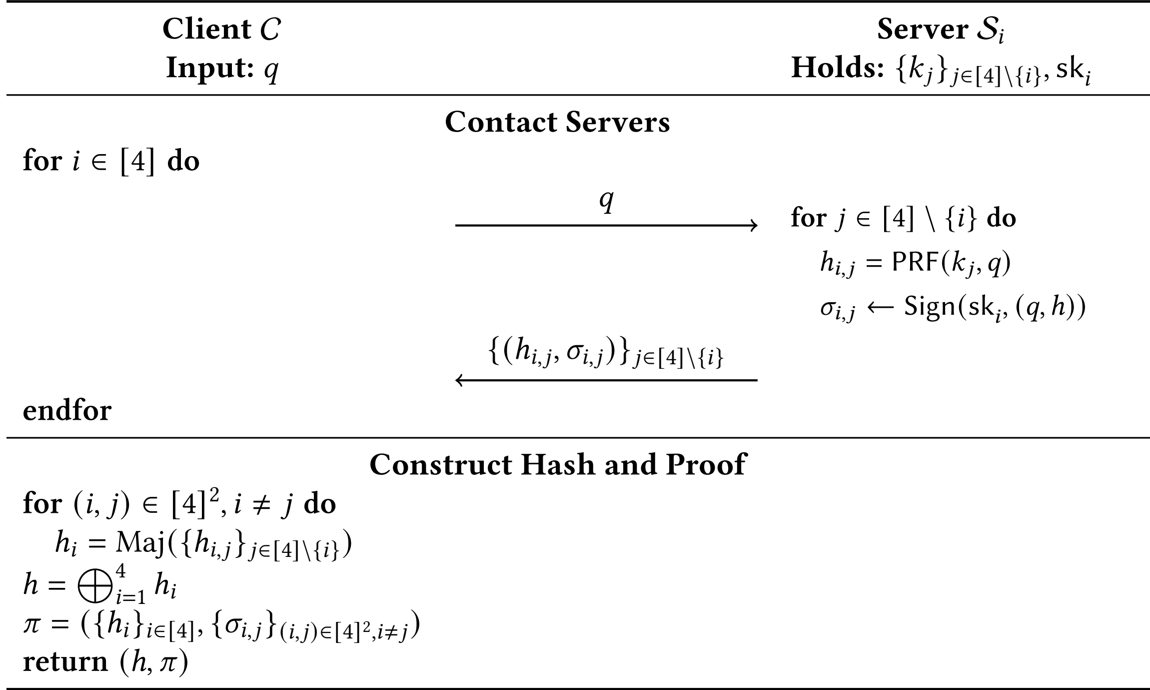
1.3.4. Instantiations

We now first describe a protocol that achieves the first three properties. We then modify it to also achieve the third property using primitives such as *fully-homomorphic encryption* (FHE) and *extractable witness-indistinguishable arguments* (NIWI). The first one of these constructions is essentially the *PRF construction* from [35], the second protocol is original to this thesis.

PRF Construction The PRF construction is in the client-server model, meaning the main functionality of the VRO is provided by a fixed number of servers with well-known identities. While it in principle works with any number n of servers and allows for $t < n/3$ malicious parties, the runtime of each party scales exponentially with t and so the construction is only practical for small t. In the following we describe the protocol for the case of $n = 4$ and $t = 1$.

During a trusted initialization stage, each of the four servers $\mathcal{S}_i$, $i \in [4]$, is given three PRF keys k_j with $j \neq i$, i.e. all but the one corresponding to its identity. In addition, every server generates a key-pair $(\mathsf{sk}_i, \mathsf{vk}_i)$ for a signature scheme SIG. The vk_i are made public. To create a hash and proof for input q, the party making the request sends q in the clear to all the $\mathcal{S}_i$. Server $\mathcal{S}_i$ evaluates the PRF on q for all of the keys in its possession, obtaining values $h_{i,j} = \mathsf{PRF}(k_j, q)$. It then signs the messages (q, h_j) using sk_i and obtains signatures[6] $\sigma_{i,j}$. All h_j and σ_j are returned.

[6]There is a technicality here which we will resolve in the main part of the thesis.



Protocol 1.1.: Simplified pseudo-code for the PRF Construction.

The responses are collected. From every three PRF outputs which according to the protocol have been made with respect to the same key k_i, the correct $h_i = \mathsf{PRF}(k_i, q)$ is computed via a majority vote. The result is guaranteed to be correct as at most one output is wrong. The final hash h is then computed as $h = \bigoplus_{i=1}^{4} h_i$. Valid proofs for (q, h) have to consist of four values h_i summing to h as well as (at least) two signatures valid under verification keys associated to two different servers for messages (q, h_i). Pseudo-code for the generation of proofs is shown in Protocol 1.1. The function Maj is used to compute the most common value occurring in a set S.

This construction almost realizes $\mathcal{F}_{VRO}$ if we disregard hiding the input. As the corrupted server only holds three out of the total four PRF keys, it can not predict hash values on its own. This same fact can also be exploited by the simulator to program. The simulator can choose the outputs of the PRF instance corresponding to the unknown key freely as long as they are distributed correctly. In this way, it can bias the hash for input q to the hash h which it obtains from $\mathcal{F}_{VRO}$. This change remains hidden from the simulated adversary. Proofs are unforgeable by the security of the used signature scheme. Modifying an existing proof π for (q, h) to be valid for (q, h') with $h \neq h'$ requires forging a signature for one of the honest servers. Forging a proof for a yet unqueried q even requires forging two such signatures.

Augmenting the PRF Construction We now augment the above construction to also hide the input. Our protocol employs strong cryptographic primitives and it is interesting future work to either provide a simpler way to hide the input in the above protocol or to come up with another protocol altogether.

Client C		**Server** $\mathcal{S}_i$
Input: q		
Holds: $\mathsf{vk} = \{\mathsf{vk}_l\}_{l \in [4]}$		**Holds:** $\{k_j\}_{j \in [4] \setminus \{i\}}, \mathsf{sk}_i$
	Contact Servers	
$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{FHE.Gen}(1^\lambda)$		
$c \leftarrow \mathsf{Enc}(\mathsf{pk}, q)$		
for $i \in [4]$ **do**		
	Prove knowledge of sk $\longrightarrow$	
	(pk, c) $\longrightarrow$	**for** $j \in [4] \setminus \{i\}$ **do**
		$C_1 = \mathsf{PRF}(k_j, \cdot)$
		$C_2 = \mathsf{Sign}(\mathsf{sk}_i, (\cdot, \cdot))$
		$c_{h,i,j} = \mathsf{Eval}(\mathsf{pk}, C_1, c)$
		$c_{\sigma,i,j} \leftarrow \mathsf{Eval}(\mathsf{pk}, C_2, c, c_{h,i,j})$
	$\longleftarrow$ $\{(c_{h,i,j}, c_{\sigma,i,j})\}_{j \in [4] \setminus \{i\}}$	
endfor		
	Construct Hash and Proof	
for $(i, j) \in [4]^2, i \neq j$ **do**		
$\quad h_{i,j} \leftarrow \mathsf{Dec}(\mathsf{sk}, c_{h,i,j})$		
$\quad \sigma_{i,j} \leftarrow \mathsf{Dec}(\mathsf{sk}, c_{\sigma,i,j})$		
$\quad h_i = \mathsf{Maj}(\{h_{i,j}\}_{j \in [4] \setminus \{i\}})$		
for $i \in [4]$ **do**		
$\quad w_i = \{\sigma_{i,j} \mid i \in [4] \setminus \{i\}\}$		
$\quad x_i = (\mathsf{vk}, (q, h_i))$		
$\quad \pi_i = \mathsf{NIWI}(x_i, w_i)$		
$h = \bigoplus_{i=1}^{4} h_i$		
$\pi = (h_i, \pi_i)_{i \in [4]}$		
return (h, π)		

Protocol 1.2.: Simplified pseudo-code for the augmented PRF Construction.

Roughly, to hide the input q we let the party making the hash query encrypt it under the public key pk of an FHE scheme that it has generated. Let c be the resulting ciphertext. The server then executes the same protocol as above but does so on c using the homomorphic properties of the FHE scheme. There are, however, two problems with this basic approach. First, as q is now no longer given in the clear to the servers, the simulator has to be given a way to extract it. There are two main ways to solve this, both involve a zero-knowledge proof of knowledge by the party making the request. We can either require a proof of knowledge of the plaintext itself, or we can require a proof of knowledge of the FHE decryption key. As the latter is computationally more efficient, especially if we want to allow long inputs, we have chosen this solution. Some care has to be taken here regarding to invalid ciphertexts being provided by corrupted parties.

Another difficulty concerns the structure of proofs. If we want to use the semantic security of the FHE scheme to argue that the corrupted server does not gain information about q, we can not let proofs produced by honest parties depend on the information provided by the corrupted server. The exact argument for this is given in the main body, but it informally goes as follows: In the game-hop where we wish to use semantic security, the reduction does not have access to the FHE decryption key sk but has to produce proofs which are distributed correctly. The structure of proofs, however, depends on whether a corrupted server did its homomorphic computation honestly or not. If it did, signatures by this server have to be included in proofs. If it did not, none can be included. Without being able to decrypt, the reduction is unable to differentiate between a server following or not following the protocol.

We establish this independence in the following way. We no longer include the signatures themselves as this leaks the identity of the signers. Instead, we only include a NIWI proof which proves that the creator of the proof *was in possession* of enough signatures to create a valid signature-based proof. In the security proof, we can then, in one of the game-hops, switch to a hybrid where proofs are computed only from signatures by honest servers. The environment is unable to tell the difference between these two games based on the witness-indistinguishability of the NIWI. Simplified pseudo-code for the augmented protocol is shown in Protocol 1.2.

In the main body of the thesis we prove that these changes are sufficient to realize $\mathcal{F}_{VRO}$.

1.4. Related Work

There are many primitives which involve the generation of (pseudo-)random bits destined to be used in some higher-level protocol. Many of these primitives such as *pseudo-random generators* [10], *pseudo-random functions* [58] and its verifiable variant, or protocols for *oblivious evaluation* of pseudo-random functions (OPRF), involve a single entity which is in possession of a secret key. Only to parties not knowing this key does the generated output look random. VROs, on the other hand, are unkeyed primitives and are provided as a public service to the parties in a protocol. Nonetheless, all of the just-mentioned primitives may themselves serve as building blocks in protocols that realize a VRO and indeed we will do so in later parts of this thesis. OPRF protocols in particular are conceptionally close, but some major differences remain. An extensive comparison will be drawn in Chapter 6.

There have also been other attempts to securely instantiate random oracles in certain contexts. These include the use of *correlation-intractable* hash functions in the context of Σ-protocols [20], *point-function obfuscation* [17], or *universal computational extractors* (UCE) [5] which can be used to replace random oracles in various applications. Our approach, however, is fundamentally different. While the aim of obtaining provable security instead of the heuristic security offered by random oracles is similar, we use interaction to be able to securely instantiate (a form of) random oracle. The other approaches mentioned, on the other hand, are trying to find standard model properties of hash functions which can be used to rid constructions of the need for random oracles altogether.

1.5. Contribution

Our contributions are both definitional as well as constructive. We give an ideal functionality $\mathcal{F}_{VRO}$ in the UC framework and argue that it captures all relevant security properties we intuitively expect it to have based on the motivation given in Section 1.1. We validate our definition by applying it to two cryptographic applications, FDH signatures and the Fischlin transformation. To be able to do this, we define security models for signatures and a certain class of zero-knowledge proofs in the presence of VROs instead of ROs. We then modify FDH signatures and the Fischlin transform to make use of VROs and proof that they retain their security. A third contribution consists in the concrete constructions realizing the definition we put forth. These constructions are applicable in different corruption scenarios up to allowing for a certain set of parties to be statically and maliciously corrupted. We further analyze variations of $\mathcal{F}_{VRO}$ in which we either weaken or strengthen its properties and investigate instantiations for them.

1.6. Organization

This thesis is organized as follows. In Chapter 2 we start by giving the necessary notational and definitional background information required to follow the rest of this thesis. Then, in Chapter 3 we formally define the verifiable random oracle model by giving an ideal verifiable random oracle functionality in the UC framework and compare the result to the random oracle model. In Chapter 4 we apply our definition to two applications that originally used the random oracle model and which we adapt to use verifiable random oracles instead. We turn to concrete instantiations of verifiable random oracles in Chapter 5. First, we give a simple single-party instantiation based on either signatures and pseudorandom functions or simulatable and verifiable random functions. Then we give a more involved construction in a client-server model and which allows for some amount of static malicious corruption of the servers and all of the clients. We also analyze relaxed variations of our definition of verifiable random oracles which possess weaker security guarantees but may be easier to instantiate securely. We show that this is indeed true by proving that an existing protocol designed for a weaker, game-based definition of verifiable random oracles securely realizes this relaxed definition. In Chapter 6 we investigate the relation

of verifiable random oracles to other primitives defined for similar purposes. We draw a conclusion in Chapter 8.

2. Preliminaries

In this chapter, we give the necessary background information used in later chapters.

2.1. Definitions and Notation

In this section, we introduce basic notation. For a probability distribution $\mathcal{D}$ we write $s \leftarrow\!\!\$\ \mathcal{D}$ to generate a sample s distributed according to $\mathcal{D}$. If S is a set, $|S|$ is the cardinality of S; we also apply this notation to vectors v to denote the number of components $|v$. We denote by $s \leftarrow\!\!\$\ S$ sampling an element $s \in S$ according to the uniform distribution. We will sometimes extend this to allow sampling a subset $\{e_1, e_2, \cdots, e_i\} \in \mathcal{P}(S)$ of cardinality i by writing $\{e_1, e_2, \ldots, e_i\} \leftarrow\!\!\$\ S$, assuming $|S| \geq i$. For $n \in \mathbb{N}$, we define $[n] := \{1, 2, \ldots, n\}$. For a string $s \in \{0, 1\}^*$ let $\|s\|$ be the length of s.

Let $\mathcal{A}$ be a probabilistic Turing machine. If $\mathcal{A}$ runs in polynomial time in the length of its input we call it a *probabilistic polynomial-time* (PPT) Turing machine. We write $y = \mathcal{A}(x; r)$ to denote (deterministically) running $\mathcal{A}$ on input x and random coins r and producing output y. With $x \leftarrow \mathcal{A}(x)$ we denote the process of first sampling coins r of the appropriate length and running $\mathcal{A}(x; r)$. While algorithms representing cryptographic primitives will usually have this *uniform* shape, adversarial algorithms, i.e. distinguishers, will be *non-uniform.* A non-uniform machine $\mathcal{A}$ receives an additional input adv, the so-called *advice*, of length $p(\|x\|)$ and where p is a polynomial associated with $\mathcal{A}$. The advice given to $\mathcal{A}$ is the same for all inputs of the same length. Alternatively, a non-uniform algorithm can be thought of as a family $\{C_n \mid n \in \mathbb{N}, |C| \leq p(n)\}$ of polynomial-sized circuits.

We let λ be the *security parameter* and 1^λ its unary representation. We implicitly (and sometimes explicitly) give 1^λ as the first input to any PPT algorithm $\mathcal{A}$ to allow it to run in time that is at least polynomial in λ.

If $f : \mathbb{N} \to [0, 1]$ is a function mapping from the natural numbers into the unit interval, we call f *negligible* if for all $d \in \mathbb{N}$ there exists a $n_0 \in \mathbb{N}$ such that for all $n > n_0$ it holds that $f(n) < n^{-d}$. We also call g *overwhelming* iff $1 - g$ is negligible.

A random variable $X : \Omega \to \mathbb{R}$ is a function from the sample space Ω of some probability space $(\Omega, \Sigma, \Pr)$ to the real numbers. Given two random variables X and Y, we can define the *total variation distance between* X *and* X, $\Delta(X, Y)$ as

$$\begin{aligned}\Delta(X, Y) &= \sum_{\mu \in \mathbb{R}} |\,\mathbf{Pr}[X = \mu] - \mathbf{Pr}[Y = \mu]| \\ &= \max_{A \subseteq \mathbb{R}} |\,\mathbf{Pr}[Y \in A] - \mathbf{Pr}[Y \in A]|\end{aligned}$$

A collection of random variables X_λ indexed by $\lambda \in \mathbb{N}$ is called an *ensemble of random variables.* We also require ensembles $X_{\lambda,z}$ which are indexed by larger sets than $\mathbb{N}$ such as pairs $(\lambda, z) \in \mathbb{N} \times \{0,1\}^*$. Given two ensembles $\mathcal{X} = \{X_{\lambda,z}\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*}$ and $\mathcal{Y} = \{Y_{\lambda,z}\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*}$ we call $\mathcal{X}$ and $\mathcal{Y}$ *statistically close*, iff

$$\forall\, z \in \{0,1\}^* : \Delta(X_{\lambda,z}, Y_{\lambda,z}) \leq \mathsf{negl}(\lambda)$$

for some negligible function $\mathsf{negl}(\lambda)$.

For a weaker notion of indistinguishability we also define *computational indistinguishability* of ensembles $\mathcal{X} = \{X_{\lambda,z}\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*}$ and $\mathcal{Y} = \{Y_{\lambda,z}\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*}$, in symbols $\mathcal{X} \overset{c}{\approx} \mathcal{Y}$. For $\mathcal{X}$ and $\mathcal{Y}$ to be computationally indistinguishable, we require that for any non-uniform algorithm $\mathcal{A}$ outputting a bit, for any $d \in \mathbb{N}$, for all $\lambda \in \mathbb{N}$ and all $z \in \cup_{\kappa\leq\lambda^d}\{0,1\}^\kappa$ it holds that:

$$\mathsf{Adv}^{\mathrm{dist}}_{\mathcal{A},\mathcal{X},\mathcal{Y}}(\lambda) := \left|\mathbf{Pr}\left[\mathcal{A}(1^\lambda, X_{\lambda,z}) = 1\right] - \mathbf{Pr}\left[\mathcal{A}(1^\lambda, Y_{\lambda,z}) = 1\right]\right| < n^{-d}$$

where the probability is taken over the random coins of $\mathcal{A}$ and the distributions of $X_{\lambda,z}$ and $Y_{\lambda,z}$.

Note that these past two definitions for general ensembles indexed by the set $\mathbb{N} \times \{0,1\}^*$ naturally descend to the ordinary case of ensembles indexed by $\mathbb{N}$ alone.

Remark 2.1.1. From this definition, we can see why adversaries in this document will be non-uniform machines. Imagine for a moment that computational indistinguishability was to only quantify over (uniform) PPT adversaries $\mathcal{A}$. For $\mathcal{A}$ to break the computational indistinguishability of two ensembles $\mathcal{X}$ and $\mathcal{Y}$ would then require that there exists a $d \in \mathbb{N}$ and infinitely many $\lambda_i \in \mathbb{N}$, $i \in \mathbb{N}$, such that for each of these λ_i there exists a corresponding $z_i \in \cup_{\kappa\leq\lambda_i^d}\{0,1\}^\kappa$ for which

$$\left|\mathbf{Pr}\left[\mathcal{A}(1^\lambda, X_{\lambda,z}) = 1\right] - \mathbf{Pr}\left[\mathcal{A}(1^\lambda, Y_{\lambda,z}) = 1\right]\right| \geq n^{-d}.$$

To use the distinguishing ability of $\mathcal{A}$ within some reduction $\mathcal{B}$ to a hard problem, $\mathcal{B}$ would have to know the association from λ_i to z_i to let $\mathcal{A}$ execute on the correct samples. As z_i will generally be difficult to compute from λ_i it seems like we have to provide it as an additional advice to $\mathcal{A}$ (and also $\mathcal{B}$ during the reduction such that it may indeed run $\mathcal{A}$).

2.2. The Random Oracle Model

A random oracle H is an oracle providing a truly random function $H : \mathcal{D} \rightarrow \mathcal{H}$ from some domain $\mathcal{D}$ to a codomain $\mathcal{H}$. Giving a party oracle access to H essentially means that there exists a box which, given some $x \in \mathcal{D}$ returns $H(x)$ without revealing any other information. The *Random Oracle Model* (ROM) [7] describes a model for the definition of cryptographic protocols where all parties are given access to a single random oracle.

2.3. Multi-Party Computation

In the simplest case, *Multi-Party Computation* (MPC) is concerned with the task of n parties $\mathcal{P}_1$ through $\mathcal{P}_n$ each holding (private) input x_i for some (randomized) function

$f : \mathbb{N} \times (\{0,1\}^*)^n \times \{0,1\}^* \rightarrow \{0,1\}^n$ which receives as input a security parameter, the different parties inputs, and random bits, and produces output $f(\lambda, x_1, \ldots, x_n, r)_i$ for party $\mathcal{P}_i$.

To compute f, the parties engage in some kind of protocol π involving sending messages over some kind of communication channels, e.g. an authenticated point-to-point network, and local computations based on the parties' input, random coins, and the messages received so far. At the end of the computation, each party $\mathcal{P}_i$ outputs some result y_i.

A number of parties may be corrupted by an adversary $\mathcal{A}$ which in the mildest case follows the protocol honestly and tries to extract information from the view of the parties it controls after the protocol has concluded (semi-honest adversary). $\mathcal{A}$ may also arbitrarily deviate from the protocol, sending messages of its choice (malicious adversary).

Some of the properties we would like π to have in the presence of an adversary $\mathcal{A}$ corrupting a number of parties are the following:

- **Input Privacy:** Any function of some honest parties input that the adversary can compute, it can already compute from the corrupted parties outputs.
- **Input Independence:** The adversary can choose its inputs as a function of the honest parties' inputs.
- **Fairness:** The honest parties receive output if and only if the adversary receives the output of the corrupted parties.
- **Guaranteed Output Delivery:** The honest parties always receive their outputs, independent of the behavior of the adversary.

The definition of security of a protocol for an MPC task depends on the specific set of properties one wishes to achieve. What they all have in common is that they compare an execution of π with adversary $\mathcal{A}$ to an interaction where, essentially, the honest parties hand their inputs to a trusted party computing f. The inputs for the corrupted parties are provided by another machine called the *simulator* $\mathcal{S}$ and π is called secure if the joint distribution over the outputs of the honest parties and $\mathcal{A}/\mathcal{S}$ is indistinguishable between the two kinds of interactions. We will not make this definition more explicit as we will later be working with the stronger definition for MPC which we define in the next section.

Note that MPC can be generalized from secure function evaluation of a function f to the computation of *reactive functionalities* $\mathcal{F}$ which keep an internal state and can be queried iteratively. It is common for only some of the parties to send input and receive output to and from $\mathcal{F}$ in any given iteration.

2.4. The Universal Composability Framework

In this section we define the *Universal Composability* (UC) [18] framework for specifying functionalities $\mathcal{F}$ as well as the UC notion for what it means for a protocol π to securely realize a functionality $\mathcal{F}$. This notion follows the *simulation paradigm*.

Traditionally, protocols and their security were analyzed in isolation, both from instances of other protocols, but also from additional sessions of the same protocol, i.e. parallel

Figure 2.1.: The UC experiment with real adversary $\mathcal{A}$ and protocol π (left) and with simulator $\mathcal{S}$ and ideal functionality $\mathcal{F}$ (right). The multiple lines indicate that $\mathcal{Z}$ may interact with multiple protocol parties.

composition. It is well-known that proving the security of a protocol in this stand-alone sense has (in general) little meaning for the security of the protocol in larger contexts. A well-known example involves the parallel composition of just two instances of a zero-knowledge protocol and where this completely breaks the security guarantees by leaking the full witness.

Ideal and Real Interaction UC, on the other hand, defines the security of protocols in such a way that they retain their security properties even when run concurrently with an arbitrary, *a priori* unbounded, number of sessions of the same or other protocols. This is achieved by first defining an ideal world where (dummy) parties receive their input and privately give it to a trusted party implementing the functionality $\mathcal{F}$. Based on the received input, $\mathcal{F}$ may generate output for some of the parties, which is again privately returned. To model some form of allowed corruption even in this ideal world, $\mathcal{F}$ interacts with the ideal-world adversary $\mathcal{S}$ (which is also called the simulator). For example, $\mathcal{S}$ may receive the length of a message sent via an ideal encrypted message-transfer functionality where it is well-known that in general the length of the message can not be hidden by an encryption scheme allowing for arbitrarily long messages to be transmitted. If the length of the message was not leaked, there would not exist a protocol realizing the functionality. $\mathcal{S}$ may also be able to control when the delivery of outputs occurs.

Apart from this ideal world, there is the real world with real parties running a protocol π and communicating over some network. Instead of a simulator, there is an adversary $\mathcal{A}$ which, depending on the corruption model, is given varying amounts of influence over the protocol execution, e.g. by being allowed to delay and deliver messages or to corrupt parties and from this point on act on their behalf.

Defining Security What we would like to have is that running a session of π with $\mathcal{A}$ is "essentially as secure" as having the parties interact with $\mathcal{F}$ and $\mathcal{S}$. As the ideal world interaction is secure by definition this would mean that also π securely realizes $\mathcal{F}$. To formalize this, another machine $\mathcal{Z}$ called the *environment* is introduced. The environment's job is to "play" with a single session, called the *test session*, either of the protocol π with adversary $\mathcal{A}$ or the ideal protocol involving $\mathcal{F}$ and simulator $\mathcal{S}$. $\mathcal{Z}$ spawns protocol parties, generates inputs for and collects outputs from the honest parties, and interacts with an adversary continuously throughout the protocol execution. At the end of the session $\mathcal{Z}$ generates some output corresponding to its decision as to which of the two

cases occurred. π is then said to UC-realize $\mathcal{F}$ if the output distributions for the two cases are computationally indistinguishable.

In more detail, $\mathcal{Z}$ also receives auxiliary input $z \in \{0,1\}^*$. We then define $\mathrm{REAL}_{\pi,\mathcal{Z},\mathcal{A}}(z)$ to be the random variable consisting of the output of $\mathcal{Z}$ when interacting with a session of π with adversary $\mathcal{A}$ where the randomness is over the random coin tosses of all involved PPT machines. For the ideal interaction we define $\mathrm{IDEAL}_{\mathcal{F},\mathcal{Z},\mathcal{S}}(z)$ to be the random variable consisting of the output of $\mathcal{Z}$ when interacting with an instance of $\mathcal{F}$ with adversary/simulator $\mathcal{S}$. $\mathrm{REAL}_{\pi,\mathcal{Z},\mathcal{A}}$ and $\mathrm{IDEAL}_{\mathcal{F},\mathcal{Z},\mathcal{S}}$ denote the ensembles of random variables $\{\mathrm{REAL}_{\pi,\mathcal{Z},\mathcal{A}}(z)\}_{z\in\{0,1\}^*}$ and $\{\mathrm{IDEAL}_{\mathcal{F},\mathcal{Z},\mathcal{S}}\}_{z\in\{0,1\}^*}$ respectively. Having introduced this notation we are ready to define what it means for a protocol π to *UC-realize* a UC-functionality $\mathcal{F}$.

Definition 2.4.1 (UC-Realization I)**.** A protocol π *UC-realizes* a functionality $\mathcal{F}$ iff for all PPT $\mathcal{A}$ there exists a PPT $\mathcal{S}$ such that for every auxiliary-input PPT machine $\mathcal{Z}$, $\mathrm{REAL}_{\pi,\mathcal{Z},\mathcal{A}} \overset{c}{\approx} \mathrm{IDEAL}_{\mathcal{F},\mathcal{Z},\mathcal{S}}$.

This definition requires the construction of a simulator $\mathcal{S}$ for any adversary $\mathcal{A}$. One important simplification to the proof strategy is possible by realizing that it is already enough to give a single simulator $\mathcal{S}_{\mathcal{D}}$ for the so-called *dummy adversary* $\mathcal{D}$. All the adversary $\mathcal{D}$ does is relay messages between the environment $\mathcal{Z}$ and the protocol parties. This effectively gives $\mathcal{Z}$ control over the actions of all corrupted parties based on all information which the adversary is able to observe during the protocol execution. We make the following definition to capture this fact.

Definition 2.4.2 (UC-Realization II)**.** A protocol π *UC-realizes* a functionality $\mathcal{F}$ iff for the dummy adversary $\mathcal{D}$ there exists a PPT $\mathcal{S}_{\mathcal{D}}$ such that for every auxiliary-input PPT machine $\mathcal{Z}$

$$\mathrm{REAL}_{\pi,\mathcal{Z},\mathcal{A}} \overset{c}{\approx} \mathrm{IDEAL}_{\mathcal{F},\mathcal{Z},\mathcal{S}}$$

Corruption The UC framework is very expressive with respect to the corruption of both ideal functionalities and real protocol parties. As is usually done, we will later appeal to the more intuitive notions of *static* and *malicious* corruption but will define them here in some more detail.

Formally, each party $\mathcal{P}$ is an interactive Turing machine endowed with different tapes. One of these tapes is called the *backdoor tape*. This tape is used for communication with the adversary. A specification of how messages received on the backdoor tape are processed is thus formally part of a specific protocol and not of the UC framework itself. The notion of static corruption is then defined as having parties only react to backdoor messages if they receive such a message during their first activation, i.e. when they have not yet participated in the actual protocol. Adaptive corruption, on the other hand, allows corruptions to take place at any point during the protocol execution. A party may react in one of several ways to such a corruption taking place. In the case of semi-honest corruption, the party will still follow the protocol, but leaks any information it receives to the adversary. For malicious corruption, the party first sends its current state to the adversary (in the adaptive case) and from then onwards the adversary is allowed to freely choose messages sent by the party.

For ideal functionalities, we use the standard notion of corruption. The simulator can choose which parties it corrupts by sending a message containing the identity of the party to the functionality. From that point on, all communication that previously was between the now corrupted party and the functionality is now between the simulator and the functionality, i.e. the simulator has to provide input and receives the output. Note that this means the functionality is aware of the set of currently corrupted parties[1] and its behavior can depend on it. For example, there could exist a threshold t such that, if more than t parties are corrupted, the functionality loses all its security guarantees. This could be modeled by leaking some kind of otherwise secret information to the simulator.

In practice, this latter point is rarely explicitly utilized. Rather than saying a protocol π, e.g. for general function evaluation, realizes a functionality $\mathcal{F}$ which leaks all inputs once more than t parties are corrupted, one says that π realizes another functionality $\mathcal{F}'$ (which never leaks inputs), but only as long as less than t parties are corrupted. The threshold for the number of corrupted parties is thus framed as an external assumption. As the two ways of stating such a property of π are ultimately equivalent, we will later also make use of the more simple way. In particular, this allows for more succinct specifications of functionalities.

2.4.1. The Universal-Composition Theorem and Hybrid Models

The strength of working within the UC framework lies in its powerful composition theorem. We have already stated that the goal of the UC framework consists in giving a way to design protocols that can be arbitrarily composed with both further instances of itself as well as other protocols. This notion of arbitrary (or universal) composability of protocols is formalized in the *Universal-Composition Theorem*. So far we have only defined two kinds of worlds, the real world involving the PPT machines $\mathcal{Z}$ and $\mathcal{A}$ as well as any protocol parties of some session of a protocol π, and the ideal world involving the PPT machines $\mathcal{Z}$ and $\mathcal{S}$ and the ideal functionality $\mathcal{F}$. To be able to formulate the composition theorem in its most general form we first have to define the intermediate notion of a *$\mathcal{F}$-hybrid model* for some functionality $\mathcal{F}$. In such a hybrid model there exist real protocol parties, an adversary $\mathcal{A}$, as well as an unlimited number of instances of $\mathcal{F}$. Protocol parties are able to give input and receive output from these instances and $\mathcal{A}$ is given the corresponding corruption information. The notion of hybrid models generalizes to multiple hybrid-functionalities in an obvious way. An important fact to note is that in the bare UC framework, the environment is not allowed to directly interact with ideal functionalities.

The setting of the composition theorem is the following. There is a protocol ξ in some $\mathcal{F}$-hybrid model and UC-realizing a functionality $\mathcal{H}$, i.e. the description of ξ involves protocol parties interacting with instances of $\mathcal{F}$. Let π be a protocol UC-realizing $\mathcal{F}$ in some $\mathcal{G}$-hybrid model and define a third protocol $\xi^{\mathcal{F}\to\pi}$ which is constructed from ξ by replacing sessions of $\mathcal{F}$ by sessions of π. The following theorem is taken from [18].[2].

[1] There is also a reporting mechanism through which the environment can retrieve this set. This is necessary to force the simulator to not corrupt too many parties compared to the real execution.

[2] Note that we have chosen to ignore the notion of *subroutine-respecting* protocols for simplicity It essentially requires protocols to not pass information outside its current session, except as output generated by the *main machines* of the session.

Theorem 2.4.3 (Universal-Composition Theorem)**.** *If ξ is a protocol which UC-realizes $\mathcal{H}$ in the $\mathcal{F}$-hybrid model and π is a protocol that UC-realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model then $\xi^{\mathcal{F}\to\pi}$ UC-realizes $\mathcal{H}$ in the $\mathcal{G}$-hybrid model.*

As a special case of the above theorem, if π UC-realizes $\mathcal{F}$ without relying on another functionality $\mathcal{G}$, $\xi^{\mathcal{F}\to\pi}$ UC-realizes $\mathcal{H}$.

The composition theorem allows the analysis of complex protocols in a modular fashion. First, the complete protocol π for a functionality $\mathcal{F}$ is formulated and proven secure in some hybrid-model using functionalities $\mathcal{F}_1, \mathcal{F}_2, \ldots$. Then the $\mathcal{F}_i$ are themselves shown to be realized by protocols π_i, maybe themselves making use of further ideal functionalities. This process is repeated in the desired granularity. At the end, the composition theorem is used to show the security of $\pi' = \pi^{\mathcal{F}_1\to\pi_1, \mathcal{F}_2\to\pi_2, \ldots}$.

2.4.2. Some Ideal Functionalities and Notation

We now introduce some of the functionalities used in this thesis. The rest will be introduced as we require them. First, we define some shorthand for defining functionalities.

Notation The real-world adversary $\mathcal{A}$ is usually given control of the network. For protocols π which involve communication between multiple parties, this means that $\mathcal{A}$ can always execute a denial-of-service attack. As any kind of attack by $\mathcal{A}$ has to be executable by the simulator $\mathcal{S}$, $\mathcal{S}$ often has to be given similar abilities when formulating some functionality $\mathcal{F}$. To keep descriptions of functionalities brief, we introduce some shorthand notation.

To let the simulator

- delay the delivery of some message $(\texttt{Task}, sid, m)$ by $\mathcal{F}$ to a party $\mathcal{P}$ we write *send private delayed output to $\mathcal{P}$*. This is translated to: Let $\mathcal{F}$ send a message $(\texttt{PrivDelay}, \texttt{Task}, sid, \mathcal{P})$ to $\mathcal{S}$. Upon receiving a response `ok`, send a message $(\texttt{Task}, sid, m)$ to $\mathcal{P}$.
- delay the delivery of some message $(\texttt{Task}, sid, m)$ by $\mathcal{F}$ to a party $\mathcal{P}$ and also leak m to $\mathcal{S}$ we write *send public delayed output to $\mathcal{P}$*. This is translated as in the first case except for using `PubDelay` and including m in the message by $\mathcal{F}$ to $\mathcal{S}$.
- provide an immediate response to a message by $\mathcal{F}$, we write *wait for a message m from the adversary.*

The concept of immediate responses by the simulator/environment is formally defined in [16] by introducing so-called *responsive environments* and restricting to them as the set of environments considered Definition 2.4.2. It is required to force the simulator to immediately answer *urgent requests* by a functionality. In this way, modeling artifacts resulting from simulators being able to do other work after receiving such a request, but before answering, are avoided. Requiring immediate responses should be restricted to *meta-information* exchanged between functionalities and the simulator as such information does not correspond to actual network communication in the real protocol. One important fact to notice is that messages requiring immediate responses do not make a specific task

interactive as it does not inherently give the simulator the ability to delay responses by the functionality to honest parties.

Remark 2.4.4. Technically speaking, $\mathcal{S}$ has to be able to respond to delaying messages in any order. By simply responding with `ok`, $\mathcal{F}$ has no way of knowing which message it should deliver. This can be solved by using some kind of identifier that is included in both messages, e.g. a simple counter. It is understood that such a mechanism is used anytime the description of a functionality contains a phrase such as *wait for a response*.

Standard Functionalities The first functionality is the standard random oracle functionality $\mathcal{F}_{RO}$ which is parametrized by a domain $\mathcal{D}$ and codomain $\mathcal{H}$. Notably, the adversary is not involved in any way. For protocols π trying to UC-realize $\mathcal{F}_{RO}$ this has the following implications. First, the real-world adversary is not allowed to gather any information on oracle inputs by and oracle outputs to honest parties using a session of π. The UC execution model in the ideal world also *guarantees* that parties receive their output. This immediately disqualifies any protocol where oracle queries are not merely local evaluations of a public function as usually the real-world adversary is given the ability to indefinitely delay messages sent over the communication network. Together with the uninstantiability result for the ROM this essentially means that $\mathcal{F}_{RO}$ can not be UC-realized. Protocols in the $\mathcal{F}_{RO}$-hybrid model should thus be seen as providing the same heuristic security guarantees as for the traditional RO model.

The $\mathcal{F}_{RO}$ functionality

Init Initialise an empty list $\mathcal{L} \leftarrow \emptyset$

Query On input $(\texttt{Query}, sid, q)$ for $q \in \mathcal{D}$ from some party $\mathcal{P}$, if there does not exist a $(q, h) \in \mathcal{L}$, let $h \leftarrow\!\!\$\ \mathcal{H}$ and store (q, h) in $\mathcal{L}$. Return $(\texttt{Answer}, sid, q, h)$.

One possible relaxation of $\mathcal{F}_{RO}$ is $\mathcal{F}_{RO}^{d}$ where the d means that we allow the adversary to observe when oracle queries are happening and *delay* responses, i.e. to decide when to deliver the output. $\mathcal{F}_{RO}^{d}$ is shown below. This formulation allows the protocol where a trusted party does lazy sampling or computes oracle outputs using a PRF to UC-realize it.

The $\mathcal{F}_{RO}^{d}$ functionality

Init Initialise an empty list $\mathcal{L} \leftarrow \emptyset$

Query On input $(\texttt{Query}, sid, q)$ for $q \in \mathcal{D}$ from some party $\mathcal{P}$, if there does not exist a $(q, h) \in \mathcal{L}$, let $h \leftarrow\!\!\$\ \mathcal{H}$ and store (q, h) in $\mathcal{L}$. Send private delayed output $(\texttt{Answer}, sid, q, h)$ to $\mathcal{P}$.

In UC, different modes of communication are captured by way of ideal functionalities. $\mathcal{F}_{AUTH}$ allows some party $\mathcal{A}$ to send a single message m to another party $\mathcal{B}$ in an ideally authenticated way. The adversary is able to observe m and is in control of when to deliver it, but it can not change m to some other message $m' \neq m$ or deliver a message to $\mathcal{B}$ without $\mathcal{A}$ having indeed sent it. Assuming authenticated channels can thus be translated to working in the $\mathcal{F}_{AUTH}$-hybrid model.

Communication that is both authentic as well as confidential is modeled by the functionality $\mathcal{F}_{SMT}$ for *secure message transfer*. The adversary is still allowed to delay the delivery of the message, but instead of learning the full message m, it only learns $l(m)$ where l is a function parametrizing the leakage of $\mathcal{F}_{SMT}$, e.g. the length of the transmitted message. If the formulation of a protocol assumes *secure channels* can be translated to working in the $\mathcal{F}_{SMT}$-hybrid model.

This concludes our introduction to the UC framework. We note that we did not intend to give a fully rigorous definition of the UC framework and refer to [18] for details. In particular, we did not answer questions regarding how activations of parties are scheduled or how the number of computation steps a party may make is determined. We strongly believe that all protocols contained in this thesis are efficient with respect to all definitions of efficient polynomial-time computation which have been used in the context of UC.

2.5. Cryptographic Primitives

In this section, we will define the various cryptographic (game-based) primitives and their game-based security notions which we will use and refer to throughout this thesis.

The standard definitions for *pseudo-random functions*, sEUF-CMA-secure *digital signature schemes* and *trapdoor one-way permutations* can be found in Appendix A.1.

2.5.1. Verifiable Random Functions

An augmented version of pseudo-random function are *verifiable* (pseudo-)random functions (VRF) [74]. With normal PRFs, the pseudo-randomness of outputs crucially depends on the fact that the key is unknown. But then a party receiving outputs from the holder of a PRF key has to trust that the evaluation was done correctly. If some security guarantee of the receiver relies on the correctness of the received outputs, using a PRF is insufficient. VRFs augment PRFs by adding three algorithms $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{Verify})$.

The first is a key generation algorithm Gen which produces evaluation keys ek and verification keys vk. Eval combines the evaluation of the underlying function

$$\mathsf{PRF} : \mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$$

with the generation of proofs π, i.e. for $(y, \pi) \leftarrow \mathsf{Eval}(\mathsf{ek}, x)$ it holds that $y = \mathsf{PRF}(\mathsf{ek}, x)$ (here we assume that the whole ek is used as key to PRF). Proofs can then be verified using Verify, which outputs a value in $\{0, 1\}$, and vk. Apart from the already stated correctness condition involving PRF, the following also has to hold

$$\forall \lambda \in \mathbb{N}\, \forall\, (\mathsf{ek}, \mathsf{vk}) \leftarrow \mathsf{Gen}(1^\lambda)\, \forall\, x \in \mathcal{X} : \mathsf{Verify}\, (\mathsf{vk}, \mathsf{Eval}(\mathsf{ek}, x)) = 1 \tag{2.1}$$

Security is defined two-fold. On the one hand, pseudo-randomness of Eval when restricted to the first output component is defined as in the PRF case. On the other hand, it is required that Verify for any verification key vk and each $x \in \mathcal{X}$ accepts proofs for at most a single $y \in \mathcal{Y}$, i.e. we require a kind of perfect unforgeability, even an unbounded adversary is unable to find valid proofs for wrong outputs.

Definition 2.5.1 (Verifiable Random Function)**.** A VRF VRF is *secure* iff the underlying PRF is secure and it holds that

$$\forall \lambda \in \mathbb{N}\, \forall\, (\mathsf{ek}, \mathsf{vk}) \leftarrow \mathsf{Gen}(1^\lambda)\, \forall\, (x, y_1, \pi_1, y_2, \pi_2) : \\ \mathsf{Verify}(\mathsf{vk}, x, y_1, \pi_1) = 1 \wedge \mathsf{Verify}(\mathsf{vk}, x, y_2, \pi_2) \Rightarrow y_1 = y_2$$

2.5.2. Simulatable Verifiable Random Functions

In some cases, the guarantees provided by the above definition of VRFs are not enough. For example, publishing a verification key vk *commits* the sender to the whole function as can be seen from the perfect unforgeability we require of VRFs. While it may seem like this is a good and useful property, proving the security of a surrounding protocol using a VRF sometimes requires being able to "break" a primitive (think of simulating zero-knowledge proofs) which in the case of VRFs corresponds to being able to forge proofs for arbitrary pairs (x, y).

Simulatability For this to be possible, however, the reduction or the *simulator* has to be able to gain an advantage over an honest party setting up an instance of a VRF. One natural way to achieve this is by working in the *common reference string* (CRS) model where each party has access to some string σ which is chosen according to some known distribution $\mathcal{D}$ and in a real protocol execution is generated by an honest party. During the security game, however, the simulator is the one providing σ and in turn only has to ensure that σ is chosen from some distribution $\mathcal{D}'$ with $\mathcal{D}' \overset{c}{\approx} \mathcal{D}$. This also allows the simulator to gain backdoor information τ associated with σ. Using the backdoor τ as well as the changed distribution $\mathcal{D}'$ of reference strings, the simulator is able to simulate proofs.

In more detail, algorithms (Setup, SimSetup, SimGen, SimProve) are added where Setup is the honest setup algorithm for generating reference strings, SimSetup outputs simulated reference strings σ and backdoors τ, SimGen on input (σ, τ) generates simulated key-pairs (ek, vk). The algorithm SimProve, on input (σ, τ), an evaluation key ek output by SimGen, and any pair (x, y), outputs simulated proofs π. The other algorithms Gen, Eval, and Verfiy are extended to also take σ as input.

Security We follow the definitions in [27]. Correctness is extended to also hold for simulated reference strings, simulated keys, and simulated proofs. Security of the honest algorithms remains unchanged and security in the simulated case is defined via indistinguishability of two games $\mathrm{G}^{\mathrm{sVRF}}_{\mathcal{A},\mathsf{sVRF}}(\lambda, 0)$ and $\mathrm{G}^{\mathrm{sVRF}}_{\mathcal{A},\mathsf{sVRF}}(\lambda, 1)$ shown in Figure 2.2. In both games, the adversary $\mathcal{A}$ receives parameters σ and a verification key vk. It is further given access to an oracle that replies to inputs x with their evaluation y and a proof π. In $\mathrm{G}^{\mathrm{sVRF}}_{\mathcal{A},\mathsf{sVRF}}(\lambda, 0)$, the challenger uses the real algorithms while in $\mathrm{G}^{\mathrm{sVRF}}_{\mathcal{A},\mathsf{sVRF}}(\lambda, 1)$ it uses the algorithms which allow simulation. Furthermore, evaluation queries in $\mathrm{G}^{\mathrm{sVRF}}_{\mathcal{A},\mathsf{sVRF}}(\lambda, 1)$ are answered via lazy sampling of values in the range accompanied by simulated proofs.

Definition 2.5.2 (Simulatable VRF)**.** A simulatable VRF sVRF is *secure* iff no PPT adversary $\mathcal{A}$ can distinguish between $\mathrm{G}^{\mathrm{sVRF}}_{\mathcal{A},\mathsf{sVRF}}(\lambda, 0)$ and $\mathrm{G}^{\mathrm{sVRF}}_{\mathcal{A},\mathsf{sVRF}}(\lambda, 1)$ shown in Figure 2.2, except with negligible probability, and for all non-simulated parameters σ and therefrom derived verification keys vk, the same perfect unforgeability as in Definition 2.5.1 holds.

$\mathsf{G}^{\mathsf{sVRF}}_{\mathcal{A},\mathsf{sVRF}}(\lambda, 0)$	$\mathsf{G}^{\mathsf{sVRF}}_{\mathcal{A},\mathsf{sVRF}}(\lambda, 1)$
1 : $\sigma \leftarrow \mathsf{Setup}(1^\lambda)$	1 : $L[\cdot] \leftarrow \bot$
2 : $(\mathsf{ek}, \mathsf{vk}) \leftarrow \mathsf{Gen}(1^\lambda, \sigma)$	2 : $(\sigma, \tau) \leftarrow \mathsf{SimSetup}(1^\lambda)$
3 : $b \leftarrow \mathcal{A}^{\mathsf{Eval}_1(\cdot)}(1^\lambda, \sigma, \mathsf{vk})$	3 : $(\mathsf{ek}, \mathsf{vk}) \leftarrow \mathsf{SimGen}(1^\lambda, \sigma, \tau)$
4 : **return** b	4 : $b \leftarrow \mathcal{A}^{\mathsf{Eval}_2(\cdot)}(1^\lambda, \sigma, \mathsf{vk})$
	5 : **return** b

$\mathsf{Eval}_1(x)$	$\mathsf{Eval}_2(x)$
1 : $(y, \pi) \leftarrow \mathsf{Eval}(\sigma, \mathsf{ek}, x)$	1 : $y = L[x]$
2 : **return** (y, π)	2 : **if** $y = \bot$ **do**
	3 : $y \leftarrow\$ \mathcal{Y}$
	4 : $L[x] = y$
	5 : **fi**
	6 : $\pi \leftarrow \mathsf{SimProve}(\sigma, \tau, \mathsf{ek}, x, y)$
	7 : **return** (y, π)

Figure 2.2.: The security game for a simulatable VRF.

2.5.3. Fully-Homomorphic Encryption

Encryption schemes are usually employed to guarantee both the *confidentiality* as well as the *integrity* or *non-malleability* of messages from a sender to a receiver. In some cases, it can be advantageous to only require confidentiality and explicitly allow the malleability of ciphertexts. The simplest case thereof are so-called *(singly) homomorphic encryption schemes* $\mathsf{HE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ where both the message space $\mathcal{M}$ as well as the ciphertext space C are equipped with group structures $\oplus$ and $\otimes$ respectively and the following holds for all messages $m_1, m_2 \in \mathcal{M}$

$$\forall \lambda \in \mathbb{N} \, \forall (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda) : \mathsf{Enc}(\mathsf{pk}, m_1) \otimes \mathsf{Enc}(\mathsf{pk}, m_2) = \mathsf{Enc}(\mathsf{pk}, m_1 \oplus m_2) \quad (2.2)$$

This kind of structure can for example be found in the encryption schemes by El'Gamal [42] or Paillier [78] and allows for the construction of primitives such as *homomorphic commitment schemes*. A stronger variant of this homomorphic property is achieved by so-called *fully-homomorphic encryption schemes* (FHE). In that case there are (essentially) two *compatible* group operations $\boxplus_1$ and $\boxplus_2$ for $\boxplus \in \{\oplus, \otimes\}$ on the message and ciphertext space, i.e. there exist ring structures on both spaces and Enc mediates a *ring homomorphism* from $\mathcal{M}$ to C. For most FHE schemes this analogy is not quite perfect. While for most schemes, the operations on the message space indeed form a ring (or even a field), homomorphic evaluation on ciphertexts is done by introducing special algorithms Eval_1 and Eval_2 and such that instead of (2.2) the following holds

$$\forall \lambda \in \mathbb{N}, i \in \{1,2\}, (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda) : \\ \mathsf{Dec}\left(\mathsf{sk}, \mathsf{Eval}_i\left(\mathsf{pk}, \mathsf{Enc}(\mathsf{pk}, m_1), \mathsf{Enc}(\mathsf{pk}, m_2)\right)\right) = m_1 \oplus_i m_2 \tag{2.3}$$

Many FHE schemes have the simplest possible message space consisting of single bits $b \in \{0,1\}$. The two operations then correspond to addition and multiplication modulo two, i.e. $\mathcal{M}$ is considered to be $\mathbb{F}_2$, the finite field with two elements. Ciphertext spaces, on the other hand, vary widely.

Encryption of longer messages is possible by encrypting them bit-wise. Based on this observation, we define what in later sections will be understood to be meant when we speak of an FHE scheme.

Definition 2.5.3 (Fully-Homomorphic Encryption Scheme)**.** An FHE scheme FHE is a tuple of algorithms $(\mathsf{Gen}, \mathsf{Enc}_1, \mathsf{Dec}_1, \mathsf{Enc}, \mathsf{Dec},$
$\mathsf{Eval}_1, \mathsf{Eval}_2, \mathsf{Eval})$ with the following behaviors

- $\mathsf{Gen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$: on input the security parameter 1^λ, the key generation algorithm outputs a public key pk and a secret key sk.
- $\mathsf{Enc}_1(\mathsf{pk}, b) \to c$: on input a public key pk and a bit $b \in \{0,1\}$, the bit-encryption algorithm outputs a ciphertext c.
- $\mathsf{Dec}_1(\mathsf{sk}, c) \to b$: on input a secret key sk and a ciphertext $c \in \mathrm{im}(\mathsf{Enc}_1(\mathsf{pk}, \cdot))$ in the image of the encryption algorithm for the corresponding public key pk, the bit-decryption algorithm outputs a bit b. If c is not in this set, $\perp$ is output.
- $\mathsf{Enc}(\mathsf{pk}, m) \to c$: on input a public key pk and a message $m \in \{0,1\}^{\|m\|}$, the encryption algorithm outputs the vector $(c_i)_{1 \le i \le \|m\|}$ where $c_i \leftarrow \mathsf{Enc}_1(\mathsf{pk}, m_i)$ for $1 \le i \le \|m\|$.
- $\mathsf{Dec}(\mathsf{sk}, c) \to m$: on input a secret key sk and a ciphertext $c = (c_1, c_2, \ldots, c_n)$, the decryption algorithm outputs either the vector $m = (m_i)_{i \in [n]}$ where $m_i = \mathsf{Dec}_1(\mathsf{sk}, c_i)$ for $i \in [n]$ or $m = \perp$ if for some $j \in [n]$, $\perp \leftarrow \mathsf{Dec}_1(\mathsf{sk}, c_j)$.
- $\mathsf{Eval}_i(\mathsf{pk}, c_1, c_2) \to c$: on input a public key pk and two ciphertext vectors c_1, c_2 of length one, the bit-evaluation algorithm outputs a ciphertext vector c of length one. If both input ciphertexts are valid encryptions of b_1 and b_2 respectively with respect to pk, then c is a valid ciphertext of $b_1 \oplus_i b_2$ with respect to pk.
- $\mathsf{Eval}(\mathsf{pk}, C, c) \to c'$: on input a public key pk, a (boolean) circuit $C : \{0,1\}^n \to \{0,1\}^k$ with in-degree n and out-degree k, and a ciphertext vector c of length n, the evalutation algorithm outputs a ciphertext vector c' of length n. If c is a valid ciphertext of a plaintext $m \in \{0,1\}^n$ with respect to pk then c' is a valid ciphertext of the plaintext $C(m) \in \{0,1\}^k$ with respect to pk.

Remark 2.5.4. We will usually suppress Enc_1, Dec_2 and $\mathsf{Eval}_{1/2}$ and let FHE only consist of $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$.

The main use case for FHE, and also what we will later use it for, lies in outsourcing computation. One party is in possession of some data m and some other party holds a circuit C and possibly more data m' such that $|m| + |m'|$ is the in-degree of C. The first party encrypts m and sends the result to the party holding C. This party encrypts m' using the first parties' public key and evaluates C on the two ciphertexts. The result of the evaluation is sent back and decrypted, thereby recovering $C(m, m')$.

Security A common security notion for FHE schemes is the usual IND-CPA definition. This can even be simplified to require that no distinguisher be able to distinguish encryptions of zero from encryptions of one, given the public key, i.e. that the following hold:

$$\left\{\mathsf{Enc}_1(\mathsf{pk}, 0) \mid (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)\right\} \overset{c}{\approx} \left\{\mathsf{Enc}_1(\mathsf{pk}, 1) \mid (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda)\right\}$$

Security for longer messages then follows by a standard hybrid-argument.

This seems to be a minimal assumption for FHE to be useful and we will always assume it. While it is clear that, due to their inherent malleability, FHE schemes can not be IND-CCA2-secure, there have been proposed and achieved various intermediate notions including IND-CCA1. For the type of FHE scheme we have defined above and which allow for the evaluation of circuits of unbounded depth, public keys are often augmented by an encryption of the secret key to enable execution of a *bootstrapping* process [53]. Most FHE schemes are *noise-based* and homomorphic evaluations increase this noise. This limits the amount of homomorphic computation which can be done to a ciphertext. Bootstrapping circumvents this limitation by evaluating the decryption circuit homomorphically using the encrypted secret key, thereby lowering the noise and allowing for further evaluations. The soundness of giving the adversary such an encrypted secret key is usually not proven but assumed. This assumption is called *circular security* and is a form of security under *key-dependent messages* (KDM).

Circuit Privacy In some applications, including ours, we do not want an evaluated ciphertext to leak information about the evaluator's private input m' and/or the circuit C (which may be viewed as a "secret" circuit $C' = C(\cdot, m')$ known only to the evaluator). Such a property is called *circuit privacy*. There, a simulator Sim can generate simulated ciphertexts given only the result $C(x)$ of the computations and which are indistinguishable (for some notion) from evaluated ciphertexts $\mathsf{Eval}(\mathsf{pk}, C, x)$. As the simulator does not receive C this means that evaluated ciphertexts are "zero-knowledge" with respect to C.

Semi-Honest Security There are several notions of this. We begin by defining the weakest definition we will be using called *semi-honest statistical circuit privacy.*

Definition 2.5.5 (Semi-Honest Statistical Circuit Privacy)**.** An FHE scheme FHE has *semi-honest statistical circuit privacy*, if there exists a PPT simulator Sim and a negligible function $\mathsf{negl}(\lambda)$, such that for all $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}$, all messages $m_1, \dots, m_i$, all circuits $C : \{0,1\}^i \to \{0,1\}^o$, the following holds

$$\Delta(\mathsf{Sim}(1^\lambda, \mathsf{pk}, C(m_1, m_2, \dots, m_i)), \mathsf{Eval}(\mathsf{pk}, C, (\mathsf{Enc}(\mathsf{pk}, m_1), \dots, \mathsf{Enc}(\mathsf{pk}, m_i)))) \leq \mathsf{negl}(\lambda)$$

Remark 2.5.6. Observe that this definition clearly holds for schemes where evaluated ciphertexts and fresh encryptions of the contained result are statistically close under the assumption that the un-evaluated ciphertext and the public key are well-formed. In that case, Sim simply returns $\mathsf{Enc}(\mathsf{pk}, C(m_1, m_2, \dots, m_i)$. This is achieved for example by the original FHE scheme by Gentry [53] using *noise flooding*, but also by the schemes in [84, 37].

Malicious Security Definition 2.5.5 only considers well-formed public keys and well-formed ciphertexts. In some contexts, i.e. when the holder of a key-pair or a party providing a ciphertext is corrupted, this assumption may no longer hold. In such situations, a stronger notion called *malicious statistical circuit privacy* may be required. There, also potentially malformed public keys pk^* and ciphertexts c^* may be given to Sim.

We do not give a full definition for the following reason. The usual definition for maliciously circuit private FHE schemes has been made with one-round (i.e. two-message) two-party computation protocols in the plain model in mind. Due to an impossibility result, such a protocol can not offer full malicious security with polynomial-time simulation for both the client holding the input and the server holding the circuit. For this reason, the simulator is allowed to be unbounded. The security guaranteed by the existence of such a simulator is then that for any public key pk^* and ciphertext c^*, *some* effective input x^* is determined. For a discussion of this subject, see [77].

We note that a generic method to upgrade semi-honest circuit privacy to malicious circuit privacy is by letting the owner of a public key pk prove knowledge of the corresponding secret key sk as well as the fact that the provided ciphertext is well-formed. In that case, the impossibility result can be circumvented by either: (1) using NIZK proofs, thereby retaining the two-message structure but not the plain model (2) using interactive zero-knowledge proofs, thereby retaining the plain model but not the two-message structure. By employing one of these two strategies, polynomial-time simulation can be recovered.

Another advantage of this approach is the fact that by using a UC-secure zero-knowledge protocol, simulation in the UC sense can be achieved. Informally, this means that the simulator will have to be able to explicitly efficiently recover the effective input x^* from some (pk^*, c^*) supplied by a corrupted party. As we will be using FHE to realize a UC-functionality we will later explicitly upgrade a semi-honestly circuit private FHE scheme using UC-secure zero-knowledge protocols.

Remark 2.5.7. There also exist relaxed notions of privacy where the circuit C is public and only the additional input y of the evaluator is supposed to be protected. The above definition for malicious circuit privacy would then be changed to allow the simulator to receive also the circuit C, but still keep the input hidden. Such a notion would be sufficient for our purposes. To be specific, C will either be the evaluation circuit of a PRF where the server holds the key and a client supplies encrypted input or C will be the signing algorithm of a signature scheme where a server holds the signing key and a client supplies an encrypted message. For simplicity, we will, however, use the above definition.

Multi-Hop Schemes FHE schemes, as we have defined them above, allow circuits of unbounded depth to be evaluated. They also allow chaining of evaluations, i.e let c be a ciphertext with respect to a public key pk. Then these schemes allow to first evaluate

$c' \leftarrow \mathsf{Eval}(\mathsf{pk}, C, c)$ and later to also compute $c'' \leftarrow \mathsf{Eval}(\mathsf{pk}, C', c')$ for two circuits C and C' which can be composed. Adding malicious circuit privacy can, however, lead to this no longer being possible in the sense that circuit privacy may be broken if multiple evaluations are chained together [77]. Schemes where chaining of evaluations is possible are called *multi-hop*. Schemes where this is not possible are called *single-hop*. While we will later indeed be required to execute two evaluations in sequence, they will be executed by the same party and can thus be considered single-hop. There can not be a situation in which a second, honest evaluator is denied circuit privacy by receiving a non-well-formed ciphertext from a first, corrupted evaluator. Either both evaluations are done honestly, in which case the single-hop malicious circuit privacy protects both evaluations, or both are done by corrupted parties, in which case there is no honest party to protect. We can thus make use of more efficient single-hop constructions.

Ciphertext Spaces There exist various ciphertext spaces in different algebraic settings. As with ordinary encryption schemes, some of them possess ciphertext spaces that are hard to recognize and/or hard to sample from. In some applications where the provider of a ciphertext to which some evaluation is applied may be corrupted, invalid ciphertexts can be problematic. One solution involves letting providers of ciphertexts prove their well-formedness in zero-knowledge to the evaluator. For simplicity, we will later assume that all well-formed ciphertexts (which are required to be efficiently recognizable) are decryptable. We describe an FHE scheme having this property as having a full ciphertext space. Such schemes exist, e.g. see the scheme in [55].

Remark 2.5.8. As a slight relaxation, we could require the space C_{pk} of valid ciphertexts under a given public key pk to be efficiently recognizable under the assumption that pk is itself well-formed. If an evaluator was then first proven the well-formedness of the public key, it could efficiently reject non-well-formed ciphertexts. Under both assumptions, efficient recognizability of ciphertexts given a well-formed key or a full ciphertext space, proving knowledge of the secret key is sufficient to upgrade semi-honest to malicious circuit privacy.

2.5.4. Proof Systems and Related Notions

An (interactive) proof system (IPS) $(\mathcal{V}, \mathcal{P})$ for a language L allows a *prover* $\mathcal{P}$ to convince a *verifier* $\mathcal{V}$ (interactively) of the fact that a certain *statement* x belongs to L, i.e. that $x \in L$. While in general only $\mathcal{V}$ is required to be PPT and $\mathcal{P}$ is allowed to be unbounded, we restrict our attention to proof systems where both parties are PPT and L is an NP-language with associated witness-relation $\mathcal{R}_L \subset \{0,1\}^* \times \{0,1\}^*$. The common input for both parties is then a purported statement x and the prover additionally receives a witness w (assuming $x \in L$).
The two principal properties of a plain IPS (informally) are

- **Completeness:** For all $(x, w) \in \mathcal{R}_L$, the verifier $\mathcal{V}$ on input x, when interacting with the honest prover $\mathcal{P}$ on input (x, w), accepts, except with negligible probability.
- **Soundness:** For all $x \notin L$, no PPT $\mathcal{P}^*$ in the role of the prover can make the honest prover $\mathcal{V}$ accept, except with negligible probability.

The negligible probability is in this case not with respect to a security parameter, but with respect to $\|x\|$.

Remark 2.5.9. Requiring that soundness hold only with respect to PPT machines $\mathcal{P}^*$ as opposed to arbitrary machines leads to *argument systems* instead of proof systems. As the distinction is not important to the content of this thesis we will usually speak of *proofs* even though *arguments* are sufficient.

Zero-Knowledge A plain IPS for an NP-language L is relatively uninteresting as the honest prover may just send its witness to the verifier and the verifier accepts if and only if the witness is correct. More interesting proof systems are those endowed with a *zero-knowledge* property [60]. Informally, for an IPS to be zero-knowledge requires the verifier to learn nothing from an accepting interaction except that the statement x belongs to L.

As we will only require exact definitions for specialized notions of zero-knowledge in this thesis[3], we keep the following definition rather informal.

Generally, zero-knowledge requires for every PPT adversarial verifier $\mathcal{V}^*$ the existence of a machine called a *simulator* $\mathsf{Sim}_{\mathcal{V}^*}$, which is often only required to run in *expected polynomial time* and which on input a $x \in L$ generates transcripts of conversations between $\mathcal{V}^*$ and $\mathcal{P}$. These transcripts are required to be indistinguishable from true transcripts between these two machines, for some notion of indistinguishability.

Non-Interactiveness So far, all IPS were allowed to consist of multiple rounds of interaction between $\mathcal{V}$ and $\mathcal{P}$. In its most extreme form, an IPS only has the prover send a single message to the verifier, i.e. it is *non-interactive*. This message is hence called the proof π and can be interpreted as having been computed with some PPT algorithm Prove by the prover. The verifier, after receiving π, checks its validity with respect to the common input x by running another PPT algorithm Verify, which outputs a decision-bit b.

In this case, we can altogether dispense with the two parties and only consider properties of Prove and Verify. Completeness and soundness are defined analogously to the interactive case. But when we try to define *non-interactive zero-knowledge proofs* (NIZK) we run into an impossibility result in the plain model [59], i.e. the model where there is no shared trusted setup such as a common reference string (CRS) and also no random oracles.

NIZK in the CRS Model An exact definition then necessarily depends on the particular model that is chosen and for each model, there exists a whole zoo of definitions. Consider the conceptually simplest CRS model. In the honest case, a trusted party generates the CRS and hands it to two parties. The zero-knowledge simulator, on the other hand, is allowed to generate its own reference string as long as it is distributed computationally indistinguishable from honest reference strings. This allows the simulator to retain a trapdoor which is subsequently used to create fake proofs for statements $x \in L$ without requiring a witness w. In some definitions, the simulated CRS is allowed to depend on x while in others it has to be selected without seeing the statement. A NIZK in the CRS model may only allow a single statement to be proven with respect to a single CRS or polynomially many.

[3]For ROM/UC NIZKs and Σ-protocols to be exact.

$\mathsf{G}^{\text{zk-rom}}_{\mathcal{D},\mathcal{P}}(\lambda, 0)$	$\mathsf{G}^{\text{zk-rom}}_{\mathcal{D},\mathsf{Sim}}(\lambda, 1)$	
1 : $b \leftarrow \mathcal{D}^{H,\mathcal{P}}(1^\lambda)$	1 : $b \leftarrow \mathcal{D}^{H_{\mathsf{Sim}},\mathcal{P}_{\mathsf{Sim}}}(1^\lambda)$	
2 : **return** b	2 : **return** b	
$\mathcal{P}(x, w)$	$H_{\mathsf{Sim}}(x)$	$\mathcal{P}_{\mathsf{Sim}}(x, w)$
1 : **if** $(x, w) \notin \mathcal{R}_L$ **do**	1 : $h = \mathsf{Sim}(\mathsf{RO}, x)$	1 : **if** $(x, w) \notin \mathcal{R}_L$ **do**
2 : **return** $\perp$	2 : **return** h	2 : **return** $\perp$
3 : **fi**		3 : **fi**
4 : $\pi \leftarrow \mathsf{Prove}(x, w)$		4 : $\pi \leftarrow \mathsf{Sim}(x)$
5 : **return** π		5 : **return** π

Figure 2.3.: The zero-knowledge security game for NIZKs in the ROM.

NIZK in the ROM The augmentation of the plain model for which we require an exact definition of NIZK is the assumed existence of a random oracle H.[4] This is a common way to achieve NIZK from interactive zero-knowledge in practice due to the efficiency of the Fiat-Shamir transform [44]. A similar variety of definitions exists in this setting and we have selected a common one. Both Prove and Verify are allowed to contain instructions of the form $h = H(x)$ to indicate random oracle queries. Completeness and soundness remain as in the plain model.

For the notion of zero-knowledge we consider, there has to exist a PPT simulator Sim such that no PPT distinguisher $\mathcal{D}$ can distinguish between the following two environments:

1. In the first, $\mathcal{D}$ interacts with an honest random oracle H and an oracle $\mathcal{P}$ which on input $(x, w) \in \mathcal{R}_L$ outputs $\pi \leftarrow \mathsf{Prove}^H(x, w)$ and $\perp$ otherwise.

2. In the second, $\mathcal{D}$ interacts with an oracle H_{Sim} and an oracle $\mathcal{P}_{\mathsf{Sim}}$ which on input $(x, w) \in \mathcal{R}_L$ outputs $\mathsf{Sim}(x)$ and $\perp$ otherwise.

The two games are shown in Figure 2.3.

Remark 2.5.10. The wrapper around Sim is necessary because, presumably, L is a hard language to decide and the prover and hence also the simulator is only required to work on inputs in the language. Also note that Sim is stateful and queries to both H_{Sim} and $\mathcal{P}_{\mathsf{Sim}}$ use and update the same state, e.g. the random oracle may be programmed adaptively depending on which statements proofs are requested for. Consistency of H_{Sim} is ensured as without loss of generality the distinguisher either only queries each input once and caches the result for subsequent use or may use discrepancies between responses to the same input to distinguish.

Definition 2.5.11 (ROM NIZK). A pair of PPT algorithms $\mathsf{NIZK} = (\mathsf{Prove}, \mathsf{Verify})$ is called a NIZK in the random oracle model if it is complete, sound, and there exists a PPT machine

[4]With some domain and codomain specified by the concrete protocol.

Sim such that for all PPT distinguishers $\mathcal{D}$

$$\left|\mathbf{Pr}\left[\mathrm{G}^{\mathrm{zk-rom}}_{\mathcal{D},\mathcal{P}}(\lambda.0)\right] - \mathbf{Pr}\left[\mathrm{G}^{\mathrm{zk-rom}}_{\mathcal{D},\mathsf{Sim},1}(\lambda, 1)\right]\right|$$

is negligible in λ.

Proofs of Knowledge Sometimes a stronger property than *soundness*, which guards the verifier against malicious provers trying to prove false statements $x \notin L$, called *knowledge soundness* is required. A proof system having this property not only ensures that no false statements can be proven, but also that the prover is in possession of a witness w for x. The prover *knowing* a witness is formalized by *some* procedure using which a witness can be *extracted* from a prover with large enough probability related to the probability with which the malicious prover can make the honest verifier accept. Again there exist a multitude of game-based definitions ranging from *extraction with rewinding* in the plain model, over *online extraction* without rewinding in the ROM, to stronger kinds of extractability where extraction of a witness has to be possible adaptively multiple times and after letting the adversary see a polynomial number of simulated proofs. We require concrete definitions for one of these notions which we give next.

Online Extractability Being able to extract a witness without rewinding the prover, at first sight, seems to defy the zero-knowledge property. And this would indeed be true if the extractor was not given any further abilities. In the case of online extractability, this additional advantage of the extractor lies in the fact that it has, in addition to a (supposedly valid) proof π for some statement x produced by an adversary $\mathcal{A}$, also the random oracle queries made by $\mathcal{A}$ while it came up with π. This information is in any other situation unavailable to the verifier or any other party.

There then has to exist an online extractor Ext such that for any (possibly unbounded) adversary $\mathcal{A}$ and a random oracle H the following holds. Let $(x, \pi) \leftarrow \mathcal{A}^H(1^\lambda)$ and let $Q_H(\mathcal{A})$ be the set of random oracle queries including their responses made by $\mathcal{A}$. Let then $w \leftarrow \mathsf{Ext}(x, \pi, Q_H(\mathcal{A}))$ and define the probability of success P_{Succ} of Ext as

$$\mathsf{P}_{Succ} = \mathbf{Pr}[(x, w) \in \mathcal{R}_L \vee \mathsf{Verify}(x, \pi) = 0]$$

Definition 2.5.12 (Online-Extractable NIZK)**.** A NIZK $\mathsf{NIZK} = (\mathsf{Prove}, \mathsf{Verify})$ is called *online extractable* if there exists a PPT extractor Ext such that its probability of success P_{Succ} in the above experiment is overwhelming.

Universally Composable NIZK We also require a definition of NIZK in the UC framework. It comes in the form of an ideal functionality $\mathcal{F}_{NIZK}$ given in Figure 2.4 (based on a functionality from [29]). $\mathcal{F}_{NIZK}$ in a single session allows multiple provers $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$ to prove multiple statements towards a set of verifiers $\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_n$. In particular, proofs are transferable, i.e. a proof created once can be verified by multiple verifiers.

To create a proof for a statement $x \in L$, the prover has to submit x together with a witness w such that $(x, w) \in \mathcal{R}_L$. Proofs consist of proof strings π. To model that these strings have no inherent meaning associated with them, they are provided by the simulator $\mathcal{S}$ after seeing the (true) statement x. To verify a proof, the verifier $\mathcal{V}$ submits the proof

string π as well as the statement x to $\mathcal{F}_{NIZK}$. If π was previously output by $\mathcal{F}_{NIZK}$ itself, it is immediately accepted. To answer verification requests for proofs for which this is not the case, the simulator is asked to provide a witness w for x. It has to extract w from the pair (x, π) and any additional knowledge it has obtained, e.g. extracted from an ideal random oracle functionality that it provides to the environment.

We quickly describe how the different game-based notions for completeness, soundness, and zero-knowledge proof of knowledge are captured by $\mathcal{F}_{NIZK}$.

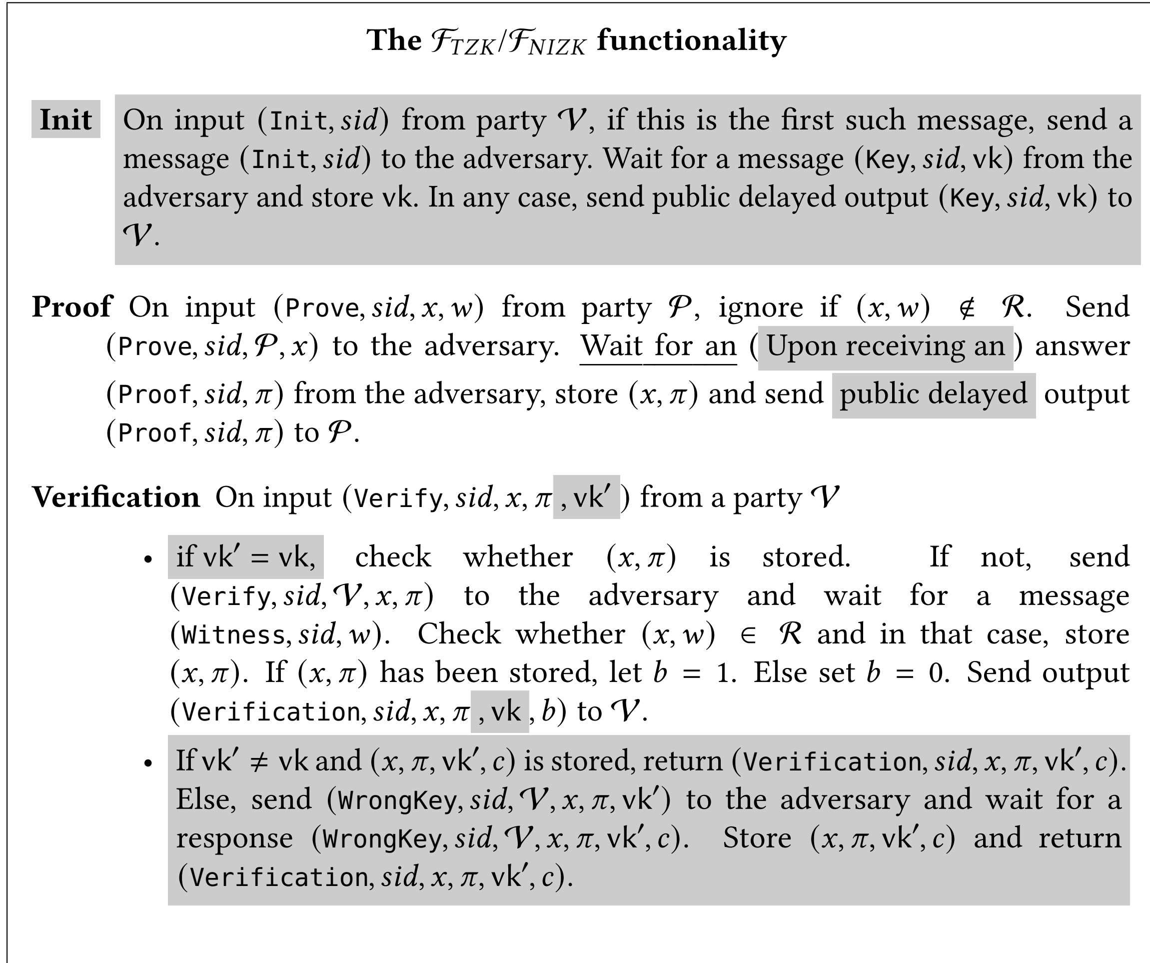
- **Completeness:** Any proof π for a statement x returned by $\mathcal{F}_{NIZK}$ is stored and when the verifier later provides the same (x, π), it is accepted.
- **Soundness:** Bare soundness is captured by the fact that the verification bit is always determined as $\mathcal{R}(x, w)$. Hence, if $x \notin L$, no proof for x will be accepted.
- **Zero-Knowledge:** The witness w is only given to $\mathcal{F}_{NIZK}$, but the adversary has to provide the proof π without knowing w. Hence π can not leak any information regarding w to the verifier.
- **Proof of Knowledge:** The prover has to present a valid witness w for the proof statement x to $\mathcal{F}_{NIZK}$. It thus has to know all of w and w can easily be extracted by having the extractor impersonate $\mathcal{F}_{NIZK}$, i.e. when the extractor is the simulator for some protocol in the $\mathcal{F}_{NIZK}$-hybrid model.

A Relaxed Functionality In Chapter 4 we will try and construct protocols which UC-realize some form of zero-knowledge functionality and which make use of a verifiable random oracle. In its most commonly used form, $\mathcal{F}_{NIZK}$ however does require both the creation of proofs as well as their verification to always succeed. On the other hand, using the oracle portion of a VRO will have to entail interaction. This means that the UC adversary is able to deny access to it as it will be in full control of the network. The success of proof generation could thus not be guaranteed if we suppose for a moment that this requires access to the random oracle. Similarly, before a party is able to check VRO proofs, it may have to retrieve a verification key. Again, this may involve interaction.

One way to weaken $\mathcal{F}_{NIZK}$ to allow protocols making use of a VRO to realize it would be to allow the simulator to decide when to deliver answers to both proof generation as well as verification queries. We feel like this would lose most of the spirit of non-interactive zero-knowledge. For proof generation, we do not see any other way. It will have to involve queries to the random oracle (if we suppose that no other setup is used) and as such has to be delayable. Verification of proofs, however, could be made non-interactive if access to the VRO verification key was ensured.

Based on these thoughts we have come up with variant of the $\mathcal{F}_{NIZK}$ functionality which we call *transferable zero-knowledge* $\mathcal{F}_{TZK}$. The changes we have made to $\mathcal{F}_{NIZK}$ are highlighted in Figure 2.4 and are as follows:

- There exists an additional task `Init` which is used to retrieve a verification key `vk`. The form of the key itself is chosen by the adversary. The adversary is able to decide when to deliver individual response messages.



The $\mathcal{F}_{TZK}/\mathcal{F}_{NIZK}$ functionality

Init On input (Init, sid) from party $\mathcal{V}$, if this is the first such message, send a message (Init, sid) to the adversary. Wait for a message (Key, sid, vk) from the adversary and store vk. In any case, send public delayed output (Key, sid, vk) to $\mathcal{V}$.

Proof On input (Prove, sid, x, w) from party $\mathcal{P}$, ignore if $(x, w) \notin \mathcal{R}$. Send (Prove, $sid, \mathcal{P}, x$) to the adversary. Wait for an (Upon receiving an) answer (Proof, sid, π) from the adversary, store (x, π) and send public delayed output (Proof, sid, π) to $\mathcal{P}$.

Verification On input (Verify, sid, x, π, vk′) from a party $\mathcal{V}$

- if vk′ = vk, check whether (x, π) is stored. If not, send (Verify, $sid, \mathcal{V}, x, \pi$) to the adversary and wait for a message (Witness, sid, w). Check whether $(x, w) \in \mathcal{R}$ and in that case, store (x, π). If (x, π) has been stored, let $b = 1$. Else set $b = 0$. Send output (Verification, sid, x, π, vk, b) to $\mathcal{V}$.
- If vk′ ≠ vk and $(x, \pi, \mathsf{vk}', c)$ is stored, return (Verification, $sid, x, \pi, \mathsf{vk}', c$). Else, send (WrongKey, $sid, \mathcal{V}, x, \pi, \mathsf{vk}'$) to the adversary and wait for a response (WrongKey, $sid, \mathcal{V}, x, \pi, \mathsf{vk}', c$). Store $(x, \pi, \mathsf{vk}', c)$ and return (Verification, $sid, x, \pi, \mathsf{vk}', c$).

Figure 2.4.: The two UC Zero-Knowledge functionalities we use. The parts not highlighted in gray make up the NIZK functionality. Including them yields the TZK functionality. Where a highlighted passage succeeds an underlined one, the highlighted portion is meant to replace the underlined one.

- Responses to proof generations are delivered by the adversary.
- Verification queries contain an additional input vk′. If the provided key is equal to the stored verification key, then $\mathcal{F}_{TZK}$ behaves like $\mathcal{F}_{NIZK}$. For different vk′, verifications are still consistent, two queries for the same input yield the same output, but the result is fully determined by the adversary.

2.5.5. Extractable Non-Interactive Witness-Indistinguishable Arguments

In some situations the full strength of zero-knowledge is unnecessary and the weaker property of a proof/argument system being *witness-indistinguishable* (WI) may be sufficient. The general structure is unchanged with respect to NIZK, i.e. there are algorithms Prove and Verify. Prove, on input $(x, w) \in \mathcal{R}_L$ for some NP-relation $\mathcal{R}_L$, produces a proof

$\pi \leftarrow \mathsf{Prove}(x, w)$. Verify, on input (x, π), outputs a bit b indicating whether the proof was accepted or not.

While in contrast to NIZK, NIWI can exist in the plain model, we need additional properties which require us to work in the CRS model. The CRS σ is provided by an algorithm Setup and given to both Prove and Verify as an additional input.

The four properties a NIWI has to possess in our setting are the following. Note that extractability already implies soundness, but we state it for completeness. The definitions are based on those in [64].

Definition 2.5.13 (Extractable NIWI)**.** A non-interactive argument system $\mathsf{NIWI} = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ is called an *extractable non-interactive witness-indistinguishable argument system in the CRS model* for a relation $\mathcal{R}_L$ with associated language L, if it has the following properties.

- **Correctness:** For all $\sigma \leftarrow \mathsf{Setup}(1^\lambda)$, all $(x, w) \in \mathcal{R}_L$, and all honestly generated proofs $\pi \leftarrow \mathsf{Prove}(\sigma, x, w)$, it holds that $\mathsf{Verify}(\sigma, x, \pi) = 1$.

- **Computational Soundness:** For all PPT adversaries $\mathcal{A}$ the probability

$$\mathbf{Pr}\Big[\sigma \leftarrow \mathsf{Setup}(1^\lambda); (x, \pi) \leftarrow \mathcal{A}(\sigma) : \mathsf{Verify}(\sigma, x, \pi) = 1 \wedge x \notin L\Big]$$

  is negligible in λ.

- **Witness-Indistinguishability:** No PPT adversary $\mathcal{A}$ can distinguish the games $\mathrm{G}^{\mathrm{wit-ind}}_{\mathcal{A},\mathsf{NIWI}}(\lambda, 0)$ and $\mathrm{G}^{\mathrm{wit-ind}}_{\mathcal{A},\mathsf{NIWI}}(\lambda, 1)$ shown in Figure 2.5, except with negligible probability. We also define

$$\mathsf{Adv}^{\mathrm{wit-ind}}_{\mathcal{A},\mathsf{NIWI}}(\lambda) := \Big|\mathbf{Pr}\big[\mathrm{G}^{\mathrm{wit-ind}}_{\mathcal{A},\mathsf{NIWI}}(\lambda, 0) = 1\big] - \mathbf{Pr}\big[\mathrm{G}^{\mathrm{wit-ind}}_{\mathcal{A},\mathsf{NIWI}}(\lambda, 1) = 1\big]\Big|$$

- **Extractability:** There exists an alternative PPT setup algorithm ExtSetup (called E_1 below) which produces outputs (σ, τ) such that

$$\{\sigma \mid (\sigma, \tau) \leftarrow \mathsf{ExtSetup}(1^\lambda)\}_{\lambda \in \mathbb{N}} \overset{c}{\approx} \{\sigma \mid\leftarrow \mathsf{Setup}(1^\lambda)\}_{\lambda \in \mathbb{N}}$$

  There also exists a PPT algorithm $\mathsf{Extract}$ (E_2 below) such that for all PPT $\mathcal{A}$:

$$\mathbf{Pr}\Big[(\sigma, \tau) \leftarrow \mathsf{E}_1(1^\lambda); (x, \pi) \leftarrow \mathcal{A}(\sigma); w \leftarrow \mathsf{E}_2(\sigma, \tau, x, \pi) : \mathsf{Verify}(\sigma, x, \pi) = 1 \Rightarrow (x, w) \in \mathcal{R}_L\Big]$$

  is overwhelming in λ.

$\mathsf{G}^{\mathrm{wit-ind}}_{\mathcal{A},\mathsf{NIWI}}(\lambda, 0)$	$\mathsf{G}^{\mathrm{wit-ind}}_{\mathcal{A},\mathsf{NIWI}}(\lambda, 1)$
1 : $\sigma \leftarrow \mathsf{Setup}(1^\lambda)$	1 : $\sigma \leftarrow \mathsf{Setup}(1^\lambda)$
2 : $(x, w_0, w_1, \mathsf{state}) \leftarrow \mathcal{A}(1^\lambda, \sigma)$	2 : $(x, w_0, w_1, \mathsf{state}) \leftarrow \mathcal{A}(1^\lambda, \sigma)$
3 : $\pi \leftarrow \mathsf{Prove}(\sigma, x, w_0)$	3 : $\pi \leftarrow \mathsf{Prove}(\sigma, x, w_1)$
4 : **return** $\mathcal{A}(\pi, \mathsf{state})$	4 : **return** $\mathcal{A}(\pi, \mathsf{state})$
5 : $\wedge\ (x, w_0) \in \mathcal{R}_L \wedge (x, w_1) \in \mathcal{R}_L$	5 : $\wedge\ (x, w_0) \in \mathcal{R}_L \wedge (x, w_1) \in \mathcal{R}_L$

Figure 2.5.: The WI game for NIWI protocols.

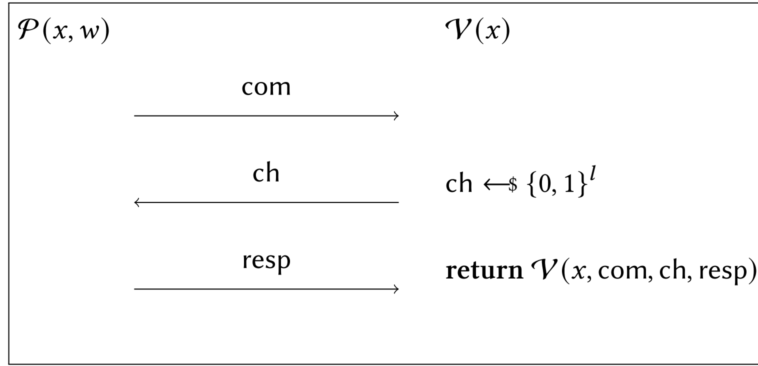


Figure 2.6.: Schematic run of a Σ-protocol.

2.5.6. Sigma Protocols

Sigma protocols (Σ-protocols) [32] are a special kind of interactive protocols for an NP-language L with witness-relation $\mathcal{R}_L$ between a prover $\mathcal{P}$ and verifier $\mathcal{V}$ having a certain set of properties. Their name derives from the communication pattern where the first and third messages are sent from the prover to the verifier and the second message is sent in the other direction, thereby tracing a Σ through the space-time diagram of the interaction (if we imagine the verifier at the end responding with whether the proof was accepted or not).

The messages are usually given the names com for *commitment*, ch for *challenge*, and resp for *response*. Σ-protocols are *public coin*, meaning that the challenge message by the verifier is randomly chosen from $\{0,1\}^l$ for some challenge length $l \in \mathbb{N}$ and independent of the received commitment.[5] We call a triple $(\mathrm{com}, \mathsf{ch}, \mathrm{resp})$ a *transcript*. Based on the fact that ch is chosen independent of com, the behavior of $\mathcal{V}$ is essentially determined by a single deterministic algorithm which receives as input the statement x and such a transcript and outputs a bit. By abuse of notation, we will call this algorithm $\mathcal{V}$ as well. A (well-formed) transcript $\tau = (\mathrm{com}, \mathsf{ch}, \mathrm{resp})$ can be either *accepting* and *non-accepting*, depending on the output of $\mathcal{V}(x, \tau)$.

[5]The challenge space may also be a more general set C. For example the Schnorr protocol [82] for discrete logarithms has $C = \mathbb{Z}_q$

There are slightly deviating collections of properties defining Σ-protocols in the literature. We make the following definition.

Definition 2.5.14 (Non-Asymptotic Σ-Protocol)**.** An interactive proof system $(\mathcal{V}, \mathcal{P})$ having the structure outlined above is called a non-asymptotic Σ-protocol for the relation $\mathcal{R}_L$ with associated language L, if it has the following properties.

- **Completeness:** For every $x \in L$, in an interaction between $\mathcal{V}$ on input x and $\mathcal{P}$ on input (x, w) and such that $(x, w) \in \mathcal{R}_L$, $\mathcal{V}$ accepts.
- **Special Soundness:** There exists a PPT *special soundness extractor* Ext which, given two accepting transcripts $\tau_1 = (\mathsf{com}, \mathsf{ch}, \mathsf{resp})$, $\tau_2 = (\mathsf{com}, \mathsf{ch}', \mathsf{resp}')$ with $\mathsf{ch} \neq \mathsf{ch}'$ for some x, outputs $w \leftarrow \mathsf{Ext}(x, \tau_1, \tau_2)$ with $(x, w) \in \mathcal{R}_L$.
- **Special Honest-Verifier Zero-Knowledge:** There exists a PPT *special zero-knowledge simulator* Sim. On input $x \in L$ and any challenge $\mathsf{ch} \in \{0,1\}^l$, $(\mathsf{com}, \mathsf{ch}, \mathsf{resp}) \leftarrow \mathsf{Sim}(x, \mathsf{ch})$ is an accepting transcript and transcripts by Sim are distributed identically to honest transcripts.[6]

Note that the special soundness property implies that Σ-protocols are proofs of knowledge. Similarly, the special honest-verifier zero-knowledge is a strengthening of ordinary honest-verifier zero-knowledge as there the simulator may choose the challenge by itself, as long as the produced distribution is correct.

Asymptotic Definition While these three properties make sense in a non-asymptotic setting, we also require two properties that do not. For this reason, we now let λ be the security parameter and add 1^λ as common input for both $\mathcal{P}$ and $\mathcal{V}$. Making the definition asymptotic involves letting $l = l(\lambda)$ be a function of the security parameter.[7] We also split the relation $\mathcal{R}_L$ into efficiently computable relations $\mathcal{R}_\lambda$, i.e. such that $\mathcal{R}_L = \cup_{\lambda=0}^{\infty} \mathcal{R}_\lambda$. The relations $\mathcal{R}_\lambda$ can be thought of as containing those pairs (x, w) for which $\|x\| = \lambda$. For simplicity, we assume that $\mathcal{R}_\lambda$ can be deduced from any $(x, w) \in \mathcal{R}_\lambda$.

We now amend the previous definition.

Definition 2.5.15 (Asymptotic Σ-Protocol)**.** An interactive proof system $(\mathcal{V}, \mathcal{P})$ satisfying Definition 2.5.14 for a relation $\mathcal{R}_L = \cup_{\lambda=0}^{\infty}$ is called an (asymptotic) Σ-protocol, if it in addition has the following properties.

- **Super-Logarithmic Commitment Min-Entropy:** The min-entropy of a probability distribution $\mathcal{D}$ is a measure for the *best-guess predictability* of $\mathcal{D}$. It is defined as $\max_k -\log(p_k)$ where the p_k are the probabilities associated with the outcomes of $\mathcal{D}$. Super-logarithmic min-entropy of commitments then means that the first message by the prover $\mathsf{com} \leftarrow \mathcal{P}(1^\lambda, x, w)$ for $(x, w) \in \mathcal{R}_\lambda$ is guessable using polynomially many guesses only with negligible probability.

[6] Of course for this to hold we have to restrict honest transcripts to those with the same challenge.

[7] This is not technically necessary to obtain an asymptotic definition, but not doing so leads to a larger (in particular, constant) *knowledge error*.

- **Quasi-Unique Responses:** No PPT adversary $\mathcal{A}$, on input 1^λ, can produce a $(x, \mathrm{com}, \mathrm{ch}, \mathrm{resp}, \mathrm{resp}')$ such that $\mathrm{resp}' \neq \mathrm{resp}$ and both $(\mathrm{com}, \mathrm{ch}, \mathrm{resp}')$ and $(\mathrm{com}, \mathrm{ch}', \mathrm{resp}')$ are accepting transcripts for x, except with negligible probability.

Remark 2.5.16. We briefly exemplify the difference between an asymptotic and non-asymptotic definition for the Schnorr discrete-logarithm protocol. A non-asymptotic relation $\mathcal{R}$ may have the form

$$\mathcal{R} = \{(g^x, x) \mid \langle g \rangle = \mathbb{G}, |\mathbb{G}| = q, x \in \mathbb{Z}_q\}$$

for some prime number q. For an asymptotic relation. Let $\mathcal{R}_{asymp} = \cup_{\lambda=0}^{\infty} \mathcal{R}_\lambda$ where $\mathcal{R}_\lambda$ has the same form as $\mathcal{R}$ and for q a λ-bit prime. In each case a prover, on input (g^x, x), would set $\mathrm{com} = g^r$ for r a random exponent from the same group as x, but only in the asymptotic case can commitments have high min-entropy as a function of λ (indeed they have fixed min-entropy in the non-asymptotic case).

3. The Verifiable Random Oracle Model

In this chapter, we introduce the *Verifiable Random Oracle Model* (VROM). We start in Section 3.1 by defining an ideal functionality $\mathcal{F}_{VRO}$ formulated in the UC framework. The decisions we made during the design of $\mathcal{F}_{VRO}$ and their consequences for protocols using $\mathcal{F}_{VRO}$ are given in Section 3.1.3. In Section 3.2 we describe how $\mathcal{F}_{VRO}$ fits into protocols defined either within the UC framework or within a game-based context. Section 3.3 contains a comparison between a random oracle and an instance of $\mathcal{F}_{VRO}$ as well as an investigation of transformations from protocols using the former to protocols using the latter. We conclude the chapter by giving a second ideal functionality providing stronger privacy guarantees during proof verification, but which is less general in that it restricts the class protocols which are able to realize it.

3.1. The Ideal VRO Functionality

We now define a VRO by means of an ideal functionality $\mathcal{F}_{VRO}$ in the UC framework. As we have described in Section 2.4, each instance of a functionality $\mathcal{F}$ is associated with a session identifier *sid*. There are parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ interacting with $\mathcal{F}$ by sending messages to and receiving responses from $\mathcal{F}$. $\mathcal{F}$ also interacts with an ideal adversary/simulator which both interacts with $\mathcal{F}$ on behalf of corrupted parties as well as receives and provides backdoor information from/to $\mathcal{F}$.

3.1.1. The Actual Functionality

There are three types of queries the protocol parties can make: initialization queries, hash queries, and verification queries. We describe each of them in turn in text. The formal specification of $\mathcal{F}_{VRO}$ is shown in Figure 3.1.

- **Parameters:** $\mathcal{F}_{VRO}$ is parametrized by a *domain*[1] $\mathcal{X}$ and a *codomain* $\mathcal{H}$ which serve as input and hash spaces for the random oracle portion of $\mathcal{F}_{VRO}$. Both $\mathcal{X}$ and $\mathcal{H}$ will usually consist of bit-strings of some fixed or bounded length. We allow $\mathcal{X}$ to have infinite cardinality, i.e. to be the set $\{0,1\}^*$ of arbitrary bit-strings. To allow uniform sampling, $\mathcal{H}$ is required to be finite (for constant security parameter).

- **Initialization:** Initiated by a party $\mathcal{P}$ sending a message $(\texttt{Init}, sid)$ to $\mathcal{F}_{VRO}$. Upon the first such message, $\mathcal{F}_{VRO}$ asks the adversary to provide a string vk representing the verification key as well as the description of a stateless Turing machine Prove.

[1] We use $\mathcal{X}$ to let us later use $\mathcal{D}$ to denote the UC dummy adversary.

This response has to be sent immediately. $\mathcal{P}$ receives as response receives a message $(\texttt{Key}, sid, \mathsf{vk})$, the time of delivery of which is decided upon by the adversary[2].

- **Hash Queries:** Initiated by a party $\mathcal{P}$ sending a message $(\texttt{Query}, sid, q)$ to $\mathcal{F}_{VRO}$, where q is from $\mathcal{X}$. Upon each first such message containing some q, an independent and uniform element from $\mathcal{H}$ is drawn and associated with q. A message containing only the length $\|q\|$ of q as well as the identity of $\mathcal{P}$ is given to the adversary which (not necessarily immediately) responds with a message containing a string s. A proof string π is computed as $\pi \leftarrow \mathsf{Prove}(q, h, s)$, a record containing (q, h, π) is stored, and returned within a message $(\texttt{Answer}, sid, q, h, \pi)$ to $\mathcal{P}$. The adversary again decides when this message is delivered, but can only observe the identity $\mathcal{P}$ of the recipient as well as the `Answer` portion of the message.

- **Verification Queries:** Initiated by a party $\mathcal{P}$ sending a message $(\texttt{Verify}, sid, q, h, \pi, \mathsf{vk}')$ to $\mathcal{F}_{VRO}$, where q is from $\mathcal{X}$, h is from $\mathcal{H}$, and π and vk' are arbitrary bit-strings. A bit b signaling the validity of π as a proof for the fact that h is the correct hash for q is determined as follows. First, if there has been a previous verification query containing the same data, b is set to the same value. Notice that this is independent of whether vk' is equal to the correct key vk. Else, if vk' *is* equal to vk, and either q was never contained as input in a hash query or h is not the correct hash associated with q, then b is set to 0. If b is still not set at this point, the adversary is given the ability to fully determine b after seeing all of q, h, π, and vk'. Again, this response by the adversary is required to be immediate. In any case $\mathcal{F}_{VRO}$ stores a record containing $(q, h, \pi, \mathsf{vk}', b)$ for use in later queries. $\mathcal{F}_{VRO}$ sends a message $(\texttt{Verified}, sid, q, h, \pi, \mathsf{vk}', b)$ to $\mathcal{P}$ and without giving the adversary the ability to deliver it.

In short, parties can request the verification key vk for a session sid, can query values q from the domain $\mathcal{X}$ and receive the corresponding hash h from $\mathcal{H}$ and a proof π of correct evaluation, and can present an input value q, a purported hash h and proof π attesting to this fact, as well as the verification key vk and check whether the proof was correct.

Remark 3.1.1. We note that there are different ways of specifying an ideal functionality $\mathcal{F}$ in the literature. The first (and "correct" one with respect to the UC specification [18]) is the following. For each session of $\mathcal{F}$ with session identifier sid there exists a different interactive Turing machine instance (ITI) $\mathcal{M}_{sid}^{\mathcal{F}}$ which is aware both of the functionality it is providing as well as its sid. Parties wishing to communicate "with $\mathcal{F}$" in session sid by sending a message m simply send m to $\mathcal{M}_{sid}^{\mathcal{F}}$. In particular, the delivered message does not contain sid. A textual description using this style is then always implicitly parametrized by some session identifier sid, but protocol messages do not contain sid. Another way considers all sessions of $\mathcal{F}$ to be provided by the same physical machine. Thus, to differentiate between different sessions, each message has to contain sid. Textual descriptions in this style often require $\mathcal{F}$ to store "tuples of the form $(sid, \ldots)$" to keep track of the different states of all the multiplexed sub-functionalities. A third intermediate way, which we will be using, does include sid in messages, but only uses them to signify the

[2]The adversary is allowed to see the full message in this case, but of course, it has itself determined vk.

The $\mathcal{F}_{VRO}$ functionality

Initialization Upon receiving a value $(\texttt{Init}, sid)$ from party $\mathcal{P}$, if this is the first time that $(\texttt{Init}, sid)$ was received, send $(\texttt{Init}, sid)$ to the adversary. Wait for an answer $(\texttt{Init}, sid, \mathsf{Prove}, \mathsf{vk})$ where Prove is the description of a stateless PPT TM and vk is a string, store these data. Send public delayed output with either the just received or stored vk as $(\texttt{Key}, sid, \mathsf{vk})$ to $\mathcal{P}$.

Hashing Upon receiving a value $(\texttt{Hash}, sid, q)$ from party $\mathcal{P}$, if there is no stored $(\texttt{ver}, q, h, \pi, \mathsf{vk}, 1)$ for some h and π, let $h \leftarrow \mathcal{H}$. In either case proceed to send the adversary a message $(\texttt{Hashing}, sid, \mathcal{P}, \|q\|)$. Upon receiving an answer $(\texttt{SimInfo}, sid, \mathcal{P}, s)$, let $\pi \leftarrow \mathsf{Prove}(q, h, s)$. If there is no tuple $(\texttt{ver}, q, h, \pi, \mathsf{vk}, 0)$ stored, store [a] $(\texttt{ver}, q, h, \pi, \mathsf{vk}, 1)$ and send private delayed output $(\texttt{HashProof}, sid, q, h, \pi)$ to $\mathcal{P}$. If there is such a tuple, halt.

Verification Upon receiving a value $(\texttt{Verify}, sid, q, h, \pi, \mathsf{vk}')$ from some party $\mathcal{V}$, do the following:[b]

- If there exists a stored record $(\texttt{ver}, q, h, \pi, \mathsf{vk}', b)$, set $f \leftarrow b$ *(Consistency)*.
- Else, if $\mathsf{vk}' = \mathsf{vk}$ and no record $(\texttt{ver}, q, h, *, \mathsf{vk}, 1)$ exists, set $f \leftarrow 0$ *(Unforgeability)*.
- Else, send $(\texttt{Verify}, sid, q, h, \pi, \mathsf{vk}')$ to the adversary and wait for an answer $(\texttt{Verified}, sid, b)$, set $f \leftarrow b$.
- Then, and if it is not yet stored, store $(\texttt{ver}, q, h, \pi, \mathsf{vk}', f)$ and return $(\texttt{Verified}, sid, q, h, \pi, f)$ to $\mathcal{V}$.

[a] We implicitly assume that there was a previous `Init` message when receiving the first `Hash` message. If there has not been such a message, $\mathcal{F}_{VRO}$ does the initialization now.

[b] Again, assume a previous `Init` message.

Figure 3.1.: The UC Verifiable Random Oracle functionality.

The $\mathcal{F}_{Sig}$ functionality

Key Generation Upon receiving a value ($\mathtt{KeyGen}, sid$) from $\mathcal{P}$, verify that this is the first request and $sid = (\mathcal{P}, sid')$ for some sid'. If not, then ignore the request. Else, hand ($\mathtt{KeyGen}, sid$) to the adversary. Upon receiving ($\mathtt{VerificationKey}, sid, \mathsf{vk}$) from the adversary, store vk and output ($\mathtt{VerificationKey}, sid, \mathsf{vk}$) to $\mathcal{P}$.

Signature Generation Upon receiving a value ($\mathtt{Sign}, sid, m$) from $\mathcal{P}$, verify that $sid = (\mathcal{P}, sid')$ for some sid'. If not, then ignore the request. Else, send ($\mathtt{Sign}, sid, m$) to the adversary. Upon receiving ($\mathtt{Signature}, sid, m, \sigma$) from the adversary, verify that no entry $(m, \sigma, \mathsf{vk}, 0)$ recorded. If it is, then output an error message to $\mathcal{P}$ and halt. Else, output ($\mathtt{Signature}, sid, m, \sigma$) to $\mathcal{P}$, and record the entry $(m, \sigma, \mathsf{vk}, 1)$.

Signature Verification Upon receiving a value ($\mathtt{Verify}, sid, \sigma, \mathsf{vk}'$) from party $\mathcal{P}$, verify that vk is recorded. If not, output ($\mathtt{Verified}, sid, m, \sigma, \mathsf{vk}', 0$) to $\mathcal{P}$. Else, hand ($\mathtt{Verify}, sid, m, \sigma, \mathsf{vk}'$) to the adversary. Upon receiving ($\mathtt{Verified}, sid, m, \phi$) from the adversary do:

- If $\mathsf{vk}' = \mathsf{vk}$ and the entry $(m, \sigma, \mathsf{vk}, 1)$ is recorded, then set $f = 1$. (*Completeness*)
- Else, if $\mathsf{vk}' = \mathsf{vk}$, the signer is not corrupted, and no entry $(m, \sigma', \mathsf{vk}, 1)$ for any σ' is recorded, then set $f = 0$ and record the entry $(m, \sigma, \mathsf{vk}, 0)$. (*Unforgeability*)
- Else, if there is an entry $(m, \sigma, \mathsf{vk}', f')$, then set $f = f'$. (*Consistency*)
- Else, let $f = \phi$ and record the entry $(m, \sigma, \mathsf{vk}', \phi)$.

Output ($\mathtt{Verified}, sid, m, f$) to $\mathcal{P}$.

Figure 3.2.: The UC Signature functionality.

session of the party to which the message is sent, otherwise it follows the first paradigm. The reason for this being that it is easier to send a message ($\texttt{Task}, sid, m$) to a functionality $\mathcal{G}$ rather than sending ($\texttt{Task}, m$) to the instance of $\mathcal{G}$ in session sid.

3.1.2. Comparison with Existing Functionalities

The ideal functionality $\mathcal{F}_{VRO}$ is reminiscent of other existing functionalities which, speaking loosely, create and output *proof strings* asserting some statement and which are supposed to be publicly verifiable by parties other than the original creator of the proof. In the following, we will compare $\mathcal{F}_{VRO}$ to such functionalities. The first is a functionality $\mathcal{F}_{Sig}$, capturing the security provided by signature schemes, and $\mathcal{F}_{NIZK}$, a functionality for multi-proof, transferable non-interactive zero-knowledge proofs. For reference, $\mathcal{F}_{Sig}$ is shown in Figure 3.2. $\mathcal{F}_{NIZK}$ has already been defined in Section 2.5.4.

$\mathcal{F}_{Sig}$ is essentially taken from [23], but we apply the framework for immediate responses to model that these are supposed to be local computations (this did not seem to be a concern for the original authors).

The differences between $\mathcal{F}_{VRO}$ and $\mathcal{F}_{Sig}$ are the following:

- **Distinguished Signer:** With $\mathcal{F}_{Sig}$, there is a distinguished party $\mathcal{S}$, the signer. Only $\mathcal{S}$ can execute the $\texttt{KeyGen}$ to successfully retrieve the verification key vk as well as generate signatures. It is part of the surrounding protocol to authentically transfer this key to the prospective verifier. Also, the verification of signatures is affected by whether $\mathcal{S}$ is corrupted or not. With $\mathcal{F}_{VRO}$, on the other hand, there is no such distinguished party "owning" any session. Any party can request to retrieve the verification key and request proofs. In addition, $\mathcal{F}_{VRO}$ is independent of the corruption status of any party.

- **Hashes:** There is nothing within $\mathcal{F}_{Sig}$ which corresponds to the association of hashes h to inputs q as done by $\mathcal{F}_{VRO}$.

- **Signature/Proof Generation:** Signature and proof requests are similar in some ways but different in others. Both signature strings and proof strings are ultimately determined by the simulator. $\mathcal{F}_{Sig}$, however, leaks the whole message m to the simulator and lets the simulator (adaptively) respond with the signature string σ for m. Proof strings output by $\mathcal{F}_{VRO}$ also depend on the "message" q (as well as the associated hash h), but are computed by the initially provided Prove algorithm, i.e. the simulator has to commit to a strategy for the generation of proofs strings once and for all at the start of the session. On the one hand, $\mathcal{F}_{VRO}$ leaks less about the input—only the length of q—but has to also provide the identity of the party requesting the proof to the simulator.

- **Unforgeability Guarantees:** Due to the existence of hash values, verification of proofs differs slightly from the verification of signatures. Consistency is guaranteed by both functionalities, verifications for the same inputs yield the same result, independent of the identity of the party making the request. The unforgeability

guarantees differ slightly. $\mathcal{F}_{Sig}$ guarantees that for no message m for which no signature was previously generated, a valid signature can be found (assuming that the sender is not corrupted). Unforgeability for $\mathcal{F}_{VRO}$ consists of two sub-cases. The rejection of any proof for a yet unqueried q corresponds to the single case of $\mathcal{F}_{Sig}$ while requiring that there can only be valid proofs for the h assigned to q has no direct correspondence.

Altogether, $\mathcal{F}_{VRO}$ can succinctly be described as, on input q, providing signatures over (q, h) where h is computed from q using a random function RF and where the choice of RF can not be influenced from outside $\mathcal{F}_{VRO}$. In addition, $\mathcal{F}_{VRO}$ is augmented by privacy regarding the input values.

$\mathcal{F}_{NIZK}$ is conceptually much further away from $\mathcal{F}_{VRO}$. One difference that could have inspired our design of $\mathcal{F}_{VRO}$ is the absence of a verification key. Proofs by $\mathcal{F}_{NIZK}$ can be transferred to and verified by any party without the verifier having to obtain the correct verification key. In most concrete protocols, however, this is bought by relying on either a common reference string or a random oracle. Access to either of these can not be denied by the adversary. To remain flexible we have chosen not to go down this path and retain an explicit initialization step.

3.1.3. Design Decisions

In this section, we will explain $\mathcal{F}_{VRO}$ in more detail with respect to what the description means in the UC framework.

Delaying Initialization We allow the simulator to decide whether and when to answer initialization requests on a per-request basis. This models the fact that in a real protocol the delivery of, e.g. the verification key, might occur by network communication. As the real adversary is usually (and also in all our later instantiations) given the ability to decide when to deliver network messages and do so per message, allowing this is necessary.

Code Upload We let the simulator provide the Prove algorithm during the initialization stage of $\mathcal{F}_{VRO}$. This technique is called *code upload* in the UC literature and models two security properties: (1) it forces the simulator to commit to a strategy for generating proofs at the onset of the session, (2) it allows hiding from the simulator either when certain ideal tasks are invoked or at least allows to hide certain kinds of information, e.g. in our case proofs π have to depend on q and h, but we do not wish to let the simulator gain full knowledge of these data.

Verification Keys One might ask whether the existence of explicit verification keys is required. Indeed, we can imagine a functionality $\mathcal{F}_{VRO}^{no-vk}$ which does not reference any vk or vk'. Such a functionality would then either (1) only consist of two tasks for proof generation and verification or (2) still contain the same three tasks of $\mathcal{F}_{VRO}$, but a response to an `Init` message merely acts as a receipt that the requesting party has been initialized and may thus generate proofs and verify proof. Both of these cases have some deficiencies either in requiring stronger assumptions to realize them or in their usability. In some more detail:

(1) Presume for a moment that a protocol ξ which realizes $\mathcal{F}_{VRO}^{no-vk}$ itself requires a party $\mathcal{V}$ trying to verify a proof π for some (q, h) to know a verification key vk.[3] As verification queries have to always succeed without delay while starting from (q, h, π) alone, $\mathcal{V}$ has to be able to obtain vk in the same manner. In practice, this will require assuming an ideal distribution mechanism for vk. We do not wish to restrict protocols to making such pre-processing assumptions.

(2) First, in this case, $\mathcal{F}_{VRO}^{no-vk}$ has to ensure that parties making a verification query have previously completed initialization. Again this is due to the fact that this may involve retrieval of a physical verification key without which the verification can not succeed. If this is ensured, a major drawback of such a functionality is that higher-level protocols can not transfer the ability to verify proofs by transmitting verification keys among parties. As we think that this is a common occurrence (and indeed we will make use of this in Chapter 4), we have opted not to employ such a $\mathcal{F}_{VRO}^{no-vk}$.

Statefulness We require Prove to be stateless. For Prove to be stateless means that proofs π can only depend on q, h, and s, but not on any information associated with previous queries. Note that s can not be used to outsource the state to the simulator as the simulator does not learn q and h for hash queries made by honest parties. It may keep some state based on information such as the lengths of past queries, however.

Incorrect Verification Key For verification queries where the provided verification key vk′ differs from the correct key vk, no guarantees regarding the unforgeability of proofs are made. This captures the fact that each verifier has to be sure that the verification key it uses to verify proofs is authentic. This is similar to the case of signature schemes where keys are not inherently associated with parties and verifiers have to obtain authentic keys to have any guarantee that a message was indeed signed by some other party. Concretely, this allows the adversary to claim any triple (q, h, π) to constitute a valid input-hash-proof triple under such a key. It also allows the adversary to circumvent the unforgeability property regarding multiple valid proofs for a single q but different h_1, h_2.

Alternative Proofs and Malleability Our formulation allows for verification queries on input (q, h, π) to return accept, even though π was not generated within $\mathcal{F}_{VRO}$, but only when q was queried before and h is the correct hash for q. We allow this as it does not contradict any of our unforgeability requirements and allows for simpler instantiations. The distinction is identical to the one between strong and non-strong existentially unforgeable signature schemes.

Verification Queries As we have mentioned before while motivating the VROM, we do wish to have verification of proofs (q, h, π) to be non-interactive, i.e. have it be a local computation in any protocol realizing $\mathcal{F}_{VRO}$. This is achieved by not giving the simulator the ability to decide when responses to such queries are delivered. Note that we *do* notify the simulator for some of the verification requests and have $\mathcal{F}_{VRO}$ wait for a response

[3]As we will see, this is generally the case.

before delivering the result. By using the mechanism of requiring immediate responses by the simulator, this can not be used to induce a (visible) delay in the delivery of responses.

Whenever the simulator *is* notified, it gains knowledge of the full input to the verification request, i.e. (q, h, π). The reason for this weakened privacy when compared to hash queries is two-fold. For the primary reason, see the discussion in Section 3.4. A simpler reason is that proofs by $\mathcal{F}_{VRO}$ are publicly verifiable. As such, we feel that it makes sense to assume that the adversary could in practice gain access to this information.

Hash Queries For hash queries, on the other hand, we do inform the simulator using `Hashing` messages upon each query. This models the fact that such queries are allowed to be interactive. We also leak the length $||q||$ of the input q as it is infeasible to hide the length of inputs from the adversary, e.g. by encrypting them, if the domain $\mathcal{X}$ contains strings of arbitrary length. Again due to the control over the network exhibited by the adversary, we have to allow the simulator to decide when responses are delivered.

Halting There is one condition from which $\mathcal{F}_{VRO}$ can not recover and has to halt, namely when Prove generates a proof π for input (q, h) and for which there exists a stored record $(\texttt{ver}, q, h, \pi, \mathsf{vk}, 0)$. Proceeding by outputting π to the party making the query and recording $(\texttt{ver}, q, h, \pi, \mathsf{vk}, 1)$ would either break consistency or completeness should (q, h, π, vk) be later used in a verification query. As such, we choose to let $\mathcal{F}_{VRO}$ halt.

Simulation Information We allow the simulator to supply a string s which is later given to Prove using the `SimInfo` messages to model the fact that the adversary may have some influence on the specific form of proofs produced in any protocol realizing $\mathcal{F}_{VRO}$. Note that due to the completeness condition, s may not influence whether the produced proof is valid or not. We believe this to be reasonable due to the fact that the `Query` task is already interactive and the fact that similar functionalities which produce a kind of *proof/signature string* give total control over the concrete form of such strings to the simulator.

We remark that we were not required to use non-trivial s within our main instantiation. We chose to retain it for two reasons:

- To keep $\mathcal{F}_{VRO}$ as general as possible and allow greater flexibility for future protocols.
- While we did not make use of s in our main instantiation of $\mathcal{F}_{VRO}$, we used it to show that the protocol on which this instantiation is based realizes a relaxed variant of $\mathcal{F}_{VRO}$ which we will define later.

Let us quickly sketch how s is used in the latter. Proofs in this relaxed protocol consist of a number of signatures. These are produced by a set of servers, some of which are corrupted. Corrupted servers can choose whether to submit valid signatures or not. Independent of this choice, valid proofs are produced. Nonetheless, the distribution of proofs hinges on how exactly the corrupted servers behave during any given interaction. In the ideal interaction, Prove has to produce proofs that are consistent with the strategy of the real-world adversary $\mathcal{A}$. The simulator $\mathcal{S}$ can, by simulating an instance of $\mathcal{A}$, gain knowledge of how $\mathcal{A}$ behaves upon each hash query and can also obtain the signatures in case $\mathcal{A}$ follows the protocol. $\mathcal{S}$ can then use s (containing the signatures by $\mathcal{A}$) to give Prove the ability to produce correctly distributed proofs.

Completeness Proofs π which are generated by $\mathcal{F}_{VRO}$ itself are guaranteed to be accepted when later submitted in a verification query containing also the correct verification key vk. This is a natural completeness condition.

Unforgeability This is modeled by the fact that $\mathcal{F}_{VRO}$ never accepts a proof as valid for a value q for which the hash h has not even been determined yet, nor for a hash h' which differs from the correct hash h assigned to q. The need for the second condition is clear, without it no verifier receiving a valid proof for some pair (q, h) could be sure that h is the correct output for q. The first condition could seem quite strong. It means that even for polynomially small codomains $\mathcal{H}$, no valid proofs for some input q should be efficiently computable, although the (future) hash h for q can be guessed with non-negligible probability.

In particular, this means that there can be no instantiation of $\mathcal{F}_{VRO}$ where the size of $\mathcal{H}$ is polynomial and proofs are empty as in that case predicting h's and forging proofs for unqueried values q are equivalent. We note that this could be solved by allowing forged proofs to be found with a certain probability related to the size of the codomain. To simplify the description of $\mathcal{F}_{VRO}$ we have decided not to include such a mechanism.

3.2. The VROM

Just as the ROM in the UC setting can be defined as working in the $\mathcal{F}_{RO}$-hybrid model, we can define the VROM in the UC setting to correspond to the $\mathcal{F}_{VRO}$-hybrid model.[4] In this thesis, however, we wish to also apply VROs in a game-based context. The definition of the ROM in a game-based setting is trivial and unambiguous. All parties gain oracle access to the same random function. In particular, there is no adversary which is able to influence the behavior of the oracle, prohibit certain parties from obtaining their requested output, or learn about inputs given to and outputs obtained from it. $\mathcal{F}_{VRO}$, on the other hand, explicitly interacts with an adversary, leaking to it information and giving it power over the execution.

The Adversary Interface We hence have to provide this *adversary interface* to some party participating in the game. Thankfully, most games explicitly involve a single adversarial entity which is the natural choice for receiving this interface. As we are working with an ideal functionality, all powers given to the adversary are by definition permitted and so this should not lead to unnatural attacks, e.g. by the adversary not delivering any responses to `Init` requests made by honest parties. We will see that such behavior is usually directly unfavorable to the goal the adversary is trying to achieve. Imagine for example the EUF-CMA game and imagine a signature scheme where verification involves the verification of a proof obtained from $\mathcal{F}_{VRO}$. If the adversary was not to deliver the response to the `Init` query by the challenger (note here that even the challenger must make such a query and is not simply "given" the verification key of $\mathcal{F}_{VRO}$), then there is no way for the adversary to, at the end of the game, provide a forged signature σ^* which the challenger will deem valid and which thus would let the adversary win.

[4]We will speak of the VROM or the $\mathcal{F}_{VRO}$-hybrid model interchangeably from now on.

Immediate Responses We have modeled some of the interactions between $\mathcal{F}_{VRO}$ and the adversary as requiring immediate responses. In the UC framework, this is achieved by restricting the class of adversaries considered in the notion of UC-realization. Similar techniques are known in the game-based setting, e.g. restricting adversaries to some notion of admissible adversary behaving in a certain well-behaved way. Hence, we will usually restrict to adversaries which immediately reply. Note that as this condition is efficiently checkable, we could instead incorporate such a check into the challenger and end up with an equivalent notion.

Multiple Sessions Another difference between the UC and game-based context, which also exists in the ROM, is the following. By working in the $\mathcal{F}_{RO}$-hybrid model, protocols are free to make calls to $\mathcal{F}_{RO}$ in arbitrary sessions $sid_1, \ldots, sid_n$. In the game-based ROM, on the other hand, parties are given oracle access to what can be described as a single session of $\mathcal{F}_{RO}$. We choose to make a similar restriction for the game-based VROM. Formally, we let the challenger of the respective game run a single instance of $\mathcal{F}_{VRO}$ (we can let the adversary choose the session identifier in some cases) and give the adversary oracle access to the aforementioned adversary interface. The challenger in addition makes the ordinary interface of $\mathcal{F}_{VRO}$ accessible to any of the game procedures it executes during the game, e.g. within a signing oracle provided to the adversary.

Remark 3.2.1. Note that this last change is introduced mostly in order to simplify reductions as they only have to provide a single instance of the VRO. It has generally no security implications as the attacked protocol does not itself rely on information provided by any other sessions and different sessions are perfectly isolated. Also, note that this does not mean that "real" protocols are stated obliviously with respect to the session of $\mathcal{F}_{VRO}$ they are using.

3.3. Comparing Random Oracles to Verifiable Random Oracles

In this section, we compare random oracles and verifiable random oracles. We investigate in which situations verifiable random oracles can be used to replace random oracles and also the other way around.

Differences There are a number of differences between the RO functionality $\mathcal{F}_{RO}$ and the VRO functionality $\mathcal{F}_{VRO}$. Each of these differences may affect the security of a protocol trying to use $\mathcal{F}_{VRO}$ instead of $\mathcal{F}_{RO}$. We have identified the following list:

- Queries to $\mathcal{F}_{RO}$ always succeed in producing an answer while the same is true only for the `Verify` task of $\mathcal{F}_{VRO}$.
- The simulator is not informed of queries to $\mathcal{F}_{RO}$ while it is informed upon each type of query (initialization, hashing, verification) to $\mathcal{F}_{VRO}$.
- The only output generated by $\mathcal{F}_{RO}$ are hash values h associated to some input q while $\mathcal{F}_{VRO}$ returns other additional kinds of data including verification keys and proof strings π.

An Unsound Transformation There is no general, sound transformation from protocols defined in the $\mathcal{F}_{RO}$-hybrid model (or the oracle-based ROM) to protocols that are instead defined in the $\mathcal{F}_{VRO}$-hybrid model. Rather, for each protocol there exists a multitude of ways of replacing random oracle queries with a mixture of hash and verification queries made to $\mathcal{F}_{VRO}$. The simplest of these consists in replacing every `Query` message on some input q to $\mathcal{F}_{RO}$ by the corresponding `Hash` message to $\mathcal{F}_{VRO}$ and ignoring the proof π contained in the response. While such a *canonical* transformation is always possible, it may not yield the best results. First, this transformation by no means transforms a protocol π_{RO} secure in the $\mathcal{F}_{RO}$-hybrid model UC-realizing some functionality $\mathcal{F}$ into a protocol π_{VRO} in the $\mathcal{F}_{VRO}$-hybrid model which also UC-realizes $\mathcal{F}$.

The smaller of two reasons for this lies in the delayability of responses to `Hash` messages by the $\mathcal{F}_{VRO}$ adversary. This alone may not break the transformation, however, as long as all tasks of $\mathcal{F}$ which require protocol parties in π_{RO} to make queries to $\mathcal{F}_{RO}$ are themselves delayable by the simulator for $\mathcal{F}$. A larger reason for why the transformation may lead to π_{VRO} not realizing $\mathcal{F}$ lies in the leaked information which the $\mathcal{F}_{VRO}$-adversary is provided with upon each `Hash` message. While it may not seem like leaking the identity of each party making a query and the length of the input, one can easily imagine protocols that are secure using $\mathcal{F}_{RO}$, but which become completely insecure when providing the adversary with this information. For example, imagine a (contrived) signature protocol where signing involves making a number of random oracle queries and where from the lengths of the queries the signing key can be reconstructed. There exist secure signature protocols in the ROM where this is possible (indeed, any secure signature scheme in the ROM can be changed as to allow this), but the canonically transformed protocol would be totally insecure as it leaks the signing key to the adversary whenever any party tries to sign a message.

It thus seems like any transformed protocol (using any kind of transformation from ROM to VROM, not necessarily the canonical transformation) has to be separately analyzed to ensure that the additional abilities gained by the adversary do not interfere with the provided security guarantees.

When restricting to the canonical transformation, a (non-formal) set of sufficient requirements for π_{RO} seems to be the following:

- All tasks of $\mathcal{F}$ which are realized by a protocol party in π_{RO} querying the random oracle can be delayed by the simulator interacting with $\mathcal{F}$.
- The adversary can already gain knowledge of the lengths of random oracle queries and the identity of the party making them in the original protocol.

Adding Verification Queries When the transformation also includes replacing some instructions of the form $h = H(q)$ with `Verify` queries to $\mathcal{F}_{VRO}$, additional requirements have to be added. To enable a party to execute verification queries, it first has to either execute the `Init` task or has to be provided the verification key authentically via some other method. In addition, having physical proof strings may open further attack vectors to the adversary which were previously not available. For verification queries to be useful, most of the time the proof is provided to the verifier by including it in some protocol

message by another party. If, in the original protocol, this message was supposed to be non-malleable, then adding a proof string π may break this non-malleability as we explicitly allow the simulator for $\mathcal{F}_{VRO}$ to claim proofs as valid if they do not conflict with the unforgeability properties. Again, it seems to be the case that each replacement of a hash query by a verification query and the necessary changes to the protocol required by having to distribute the corresponding proof string has to be analyzed separately to ensure that no attack of this type is possible.

As we will see in the next chapter, going in the other direction, an instance of $\mathcal{F}_{RO}$ can be seen as an idealized version of an instance of $\mathcal{F}_{VRO}$ by employing an instantiation of $\mathcal{F}_{VRO}$ where proofs are empty and verifications consist in re-querying on the same input and comparing the result.[5] This means an instance of $\mathcal{F}_{VRO}$ can be replaced by an instance of $\mathcal{F}_{RO}$ without any restrictions. We will formally describe this instantiation in Section 5.1.

3.4. An Alternative Version

In the version of $\mathcal{F}_{VRO}$ we have defined above, there is an asymmetry between how answers to hash and verification queries are generated. Proofs are generated by running the initially provided algorithm Prove while proofs are verified by, in some cases, letting the adversary determine whether they are accepted adaptively. It then seems like we may require the simulator to provide a second algorithm Verify during initialization. This would remove the just mentioned adaptivity while in addition having the adversary not even be aware of when a verification query has occurred, thereby further strengthening the fact that verification should correspond to a local operation.

We have defined such an alternative version of $\mathcal{F}_{VRO}$ in Appendix A.2. We show that some expressive power is lost by this definition and argue why we ultimately chose the more complex and in some aspects less private formulation of $\mathcal{F}_{VRO}$ above.

[5]There is a minor technical issue here, see the discussion at the end of Section 5.1 and our solution in Section 5.1.1.

4. Applications

In this chapter, we validate the VROM. We present two applications where random oracles are used and modify them to instead use verifiable random oracles. The first application is the well-known *full-domain hash* or *hash-then-invert* construction for signature schemes, the security proof of which makes essential use of the programmability of random oracles. As a second application, we chose the Fischlin transform which, similar to the Fiat-Shamir transform, is used to make a certain class of interactive zero-knowledge protocols non-interactive in the ROM. Each model, be it the ROM or the plain model, comes with its own specific notions of security. For both applications, we thus begin by defining a notion of security in the VROM. In both cases, we base this new definition on the existing ROM definition and extend it to encompass the additional capabilities of the VROM. Having fixed our notion of security, we first recapitulate the formulation of each of the applications in the ROM. Based on this description we translate them into the VROM. As there are several choices involved in this translation, we justify these choices. We then prove that the adapted schemes satisfy our new security definitions in the VROM.

In the case of the Fischlin transform we prove an additional result, namely that the Fischlin transform even results in UC transferable zero-knowledge proofs (TZK) protocols. For a definition of the $\mathcal{F}_{TZK}$ functionality see Section 2.5.4. We recall that it is a slight variation of the UC NIZK functionality $\mathcal{F}_{NIZK}$ such that it can be realized by protocols in the $\mathcal{F}_{VRO}$-hybrid model where the generation of proofs can be delayed by the adversary. We then prove that the Fischlin transform, when instantiated in the VROM (after a small, inexpensive modification) still yields UC TZKs. Note that in this latter case we do not have to define a separate notion of security for the VROM setting. Both random oracles, as well as verifiable random oracles, can be represented uniformly as ideal functionalities. These functionalities can then be employed in protocols realizing the *same* ideal TZK functionality.

4.1. Full-Domain Hash

Full-Domain Hash (FDH) [7] is a generic construction for building signature schemes from any family of trapdoor one-way permutations. The construction involves a hash function which, in general, has to be modeled as a random oracle to show security[1] [39]. This makes FDH a worthwhile application to apply our techniques.

This section is organized as follows. We begin by stating our notion of security for signature schemes in the ROM which is the usual EUF-CMA notion. Then we adapt this notion for signature schemes in the VROM. After that we recall the definition of FDH in the ROM, which we will call FDH-ROM in the following, and the adapted version

[1]Note that this result only holds if access to the permutation is restricted to be black-box.

FDH-VROM. The section concludes with a security proof for FDH-VROM with respect to the just-defined VROM notion of security for signature schemes.

4.1.1. Definition of Security

In this section, we state the adapted security notion we wish signature schemes in the VROM to fulfill. We have already defined EUF-CMA-security for signature schemes in the plain model, but here we are working in the ROM and thus recall the exact definition before adapting it to the VROM.

The ROM Definition In the game-based formulation, an attacker $\mathcal{A}$ is first given the verification key vk of the signature scheme. It has also access to a signing oracle, which, on input a message m, returns an honestly generated signature σ for m. At some point, the attacker outputs a pair (m^*, σ^*) and wins if it never queried its signing oracle on m^*. In the ROM, the attacker additionally has access to the same random oracle H with respect to which the signatures produced by its signing oracle can be verified.

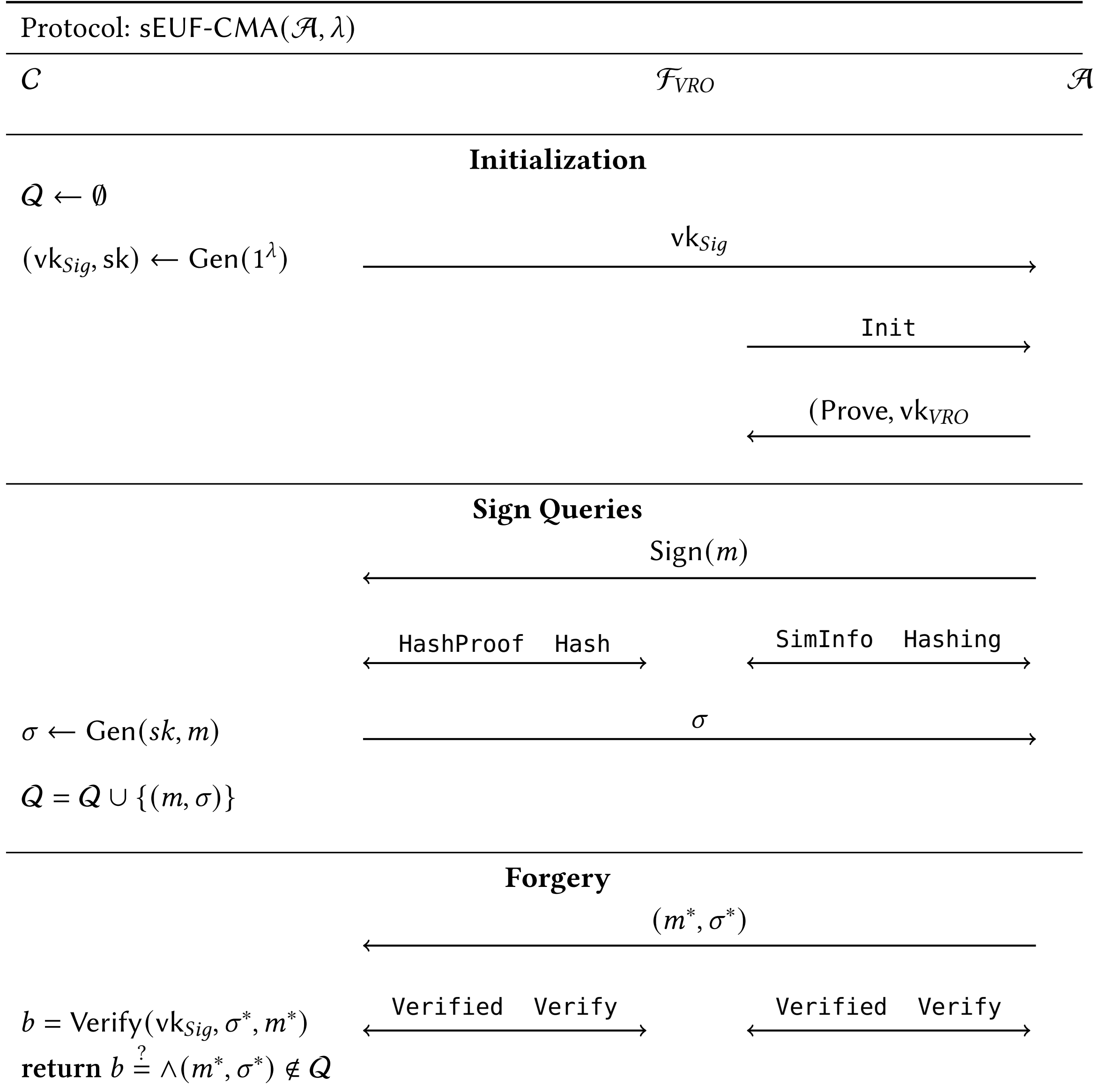
The VROM Definition Adapting the above to the VROM, we give the attacker $\mathcal{A}$ access to the VRO used by the signing oracle at all times. As per the discussion in Section 3.2, this access also contains the interface afforded to the adversary by $\mathcal{F}_{VRO}$. That means $\mathcal{A}$ is asked to initialize $\mathcal{F}_{VRO}$ by providing am algorithm Prove and verification key vk. $\mathcal{A}$ receives `Hashing` messages and is supposed to provide `SimInfo` responses to them. $\mathcal{A}$ is formally also tasked with delivering messages whenever this is required by $\mathcal{F}_{VRO}$, although not delivering messages is not in $\mathcal{A}$'s interest in the present setting and so we could just assume that all messages are being delivered. Also as per Section 3.2, we change the way session identifiers are handled. In the present game-based context we formally fix the session by removing *sid* from all messages exchanged between $\mathcal{F}_{VRO}$ and any protocol party $\mathcal{P}$. This is in line with the usual game-based ROM formulation where a single random oracle is provided to all parties. The winning condition still involves the adversary presenting a pair (m^*, σ^*), but the challenger checking the validity of σ^* may also involve verification queries made to $\mathcal{F}_{VRO}$.

The complete interaction between the challenger C and $\mathcal{A}$ is shown in Protocol 4.1. Double-ended arrows indicate bi-directional communication between $\mathcal{F}_{VRO}$, $\mathcal{A}$, and C.

Definition 4.1.1 (VROM Digital Signature Scheme)**.** Let $\mathsf{SIG} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ be a signature scheme in the VROM. SIG is sEUF-CMA-secure iff for all PPT adversaries $\mathcal{A}$ the probability

$$\mathbf{Pr}[\mathsf{sEUF\text{-}CMA}(\mathcal{A}, \lambda) = 1]$$

is negligible in λ. SIG is EUF-CMA-secure iff the probability is negligible in the altered game where the check $(m^*, \sigma^*) \notin Q$ is replaced by the check that m^* is not among the first components of elements in Q.



Protocol 4.1.: The sEUF-CMA security game in the VROM.

```
Gen(1^λ)                                Verify^H(vk, m, σ)
1 : (f, f^{-1}) ← TDP.Gen(1^λ)          1 : f = parse(vk)
2 : vk = f                              2 : h = f(σ)
3 : sk = f^{-1}                         3 : return h =? H(m)
4 : return (vk, sk)

Sign^H(sk, m)
1 : h = H(m)
2 : f^{-1} = parse(sk)
3 : return f^{-1}(h)
```

Figure 4.1.: The FDH-ROM procedures.

4.1.2. FDH in the ROM

We recall the definition of FDH in the ROM. Let FDH-ROM = (Gen, Sign, Verify) denote the algorithms defining the signature scheme. Let $\mathcal{M}$ be the message space and $\mathcal{X}$ the signature space. The required building blocks are the following:

- A family TDP of trapdoor one-way permutations with key-generation algorithm TDP.Gen and domain $\mathcal{X}$.
- A hash function H mapping the message space $\mathcal{M}$ to the signature space $\mathcal{X}$, in the following modeled as a random oracle.

The Protocol Key generation consists in running TDP.Gen(1^λ) to obtain a permutation key f and the corresponding trapdoor f^{-1} allowing efficient evaluation of the inverse permutation. The verification key vk is set to the permutation key f and the signing key sk is the trapdoor f^{-1}. Signing a message m using the signing key sk $= f^{-1}$ first hashes m using H, obtaining a digest $x = H(m)$. The signature then is the pre-image $\sigma = f^{-1}(x)$ of x under f. A signature σ given message m and verification key vk $= f$ is valid iff $f(H(m)) = \sigma$. Exact descriptions of the algorithms are shown in Figure 4.1.

Note that both signer and verifier make one hash query each for signing a given message/checking the signature of a given message. Both queries are on the message m.

4.1.3. FDH in the VROM

In this section, we propose an adapted formulation of FDH in the VROM. As can be seen above, a party verifying a signature σ on message m merely has to recompute $H(m)$ to check an equality involving the hash. This allows us to replace it with a verification query to $\mathcal{F}_{VRO}$. For this to work, however, we have to augment the signature to include both the hash $H(m)$ as well as the proof π. The query during signing has to be replaced by a hash query as its input is the (fresh) message m. Key generation also has to be changed

$\mathsf{Gen}(1^\lambda, sid)$	$\mathsf{Verify}(\mathsf{vk}, m, \sigma)$
1 : $(f, f^{-1}) \leftarrow \mathsf{TDP.Gen}(1^\lambda)$	1 : $(f, v, sid) = \mathtt{parse}(\mathsf{vk})$
2 : $(\mathtt{Init}, sid) \longrightarrow \mathcal{F}_{VRO}$	2 : $(\sigma', h, \pi) = \mathtt{parse}(\sigma)$
3 : $(\mathtt{Key}, sid, v) \longleftarrow \mathcal{F}_{VRO}$	3 : $h' = f(\sigma')$
4 : $\mathsf{vk} = (f, v, sid)$	4 : **if** $h' \neq h$ **do**
5 : $\mathsf{sk} = (f^{-1}, sid)$	5 : **return** 0
6 : **return** $(\mathsf{vk}, \mathsf{sk})$	6 : **fi**
	7 : $(\mathtt{Verify}, sid, m, h', \pi, v) \longrightarrow \mathcal{F}_{VRO}$
$\mathsf{Sign}(\mathsf{sk}, m)$	8 : $(\mathtt{Verified}, sid, q, h', \pi, \tau) \longleftarrow \mathcal{F}_{VRO}$
1 : $(f^{-1}, sid) = \mathtt{parse}(\mathsf{sk})$	9 : **return** $\tau \stackrel{?}{=} 1$
2 : $(\mathtt{Hash}, sid, m) \longrightarrow \mathcal{F}_{VRO}$	
3 : $(\mathtt{HashProof}, sid, m, h, \pi) \longleftarrow \mathcal{F}_{VRO}$	
4 : $\sigma = f^{-1}(h)$	
5 : **return** (σ, h, π)	

Figure 4.2.: The FDH-VROM procedures.

to choose a session identifier sid for $\mathcal{F}_{VRO}$ and this identifier has to be available during signing and verification.

Let FDH-VROM = (Gen, Sign, Verify) denote the algorithms defining the signature scheme. Let $\mathcal{M}$ be the message space and $\mathcal{X}$ the signature space. The required building blocks are as follows:

- As before, a trapdoor one-way permutation TDP on the domain $\mathcal{X}$ and with key-generation algorithm TDP.Gen.
- The ideal functionality $\mathcal{F}_{VRO}$, parametrized with domain $\mathcal{M}$ and codomain $\mathcal{X}$.

The Protocol Compared to the ROM variant, signing and verification keys are augmented by including in them a session identifier sid, which was provided as an additional input to Gen. To sign a message m, the signer sends the message $(\mathtt{Hash}, sid, m)$ to $\mathcal{F}_{VRO}$ and receives a response $(\mathtt{HashProof}, sid, m, h, \pi)$. She computes $\sigma' = f^{-1}(h)$ and returns the signature $\sigma = (\sigma', h, \pi)$. To verify a pair (m, σ) with $\sigma = (\sigma', h, \pi)$, if this is the first verification, the verifier sends a message $(\mathtt{Init}, sid)$ to $\mathcal{F}_{VRO}$ and receives an answer $(\mathtt{Key}, sid, \mathsf{vk})$. He then sends a message $(\mathtt{Verify}, sid, m, h, \pi, \mathsf{vk})$ to $\mathcal{F}_{VRO}$ and receives a response $(\mathtt{Verified}, sid, m, h, \pi, , b)$. If $b = 0$, return 0. Else return whether $f(\sigma')$ is equal to h. Formal descriptions of the algorithms are given in Figure 4.2.

Remark 4.1.2. To keep the protocol generally applicable, we have described it without fixing a single session of $\mathcal{F}_{VRO}$ and providing oracle access for hashing and verification. For proving security below we remove sid from keys and give the adversary access to a fixed session of $\mathcal{F}_{VRO}$ instead.

Removing the Hash We can actually remove the hash h from the signature. The verifier then only has to check that π is a correct proof under verification key v for input m and hash $f(\sigma')$. This transformation is analogous to leaving out the challenge in Σ-protocols that have been transformed via the Fiat-Shamir transform because it is already determined by public information, in our case the message [3].

We show the following lemma:

Lemma 4.1.3. FDH-VROM *without hashes included in signatures is* EUF-CMA*-secure in the VROM, iff* FDH-VROM *with hashes included in signatures is.*

Proof. To see why this is valid, imagine that there exists an algorithm $\mathcal{A}$ producing forgeries of the form $\sigma = (x, \pi)$ for message m. This means that π is a verifying proof for hash $f(x)$ and input m. We claim that we can construct from $\mathcal{A}$ another adversary $\mathcal{B}$ for the variant where hashes are included in signatures. $\mathcal{B}$ behaves as follows:

- $\mathcal{B}$ runs a simulated copy of $\mathcal{A}$.
- Upon receiving a verification key vk from its challenger, $\mathcal{B}$ gives vk to $\mathcal{A}$.
- $\mathcal{B}$ gives $\mathcal{A}$ access to $\mathcal{F}_{VRO}$ by connecting $\mathcal{A}$ to the interface of its own instance of $\mathcal{F}_{VRO}$ which $\mathcal{B}$ is given by its challenger.
- Upon a signing request by $\mathcal{A}$ on a message m, $\mathcal{B}$ first gives m to its own signing oracle. After receiving a signature $\sigma = (\sigma', h, \pi)$, $\mathcal{B}$ constructs $\sigma^* = (\sigma', \pi)$ and gives σ^* to $\mathcal{A}$.
- Upon receiving a purported forgery (m^*, σ^*) with $\sigma^* = (\sigma', \pi^*)$, $\mathcal{B}$ executes a `Hash` query on m^*, obtaining a hash h^*. $\mathcal{B}$ outputs $(m^*, (\sigma', h^*, \pi^*))$ to its challenger.

We have to show that $\mathcal{B}$ outputs a valid forgery with non-negligible probability if $\mathcal{A}$ does. So let us suppose that $\mathcal{A}$ produces a forgery $(m, \sigma) = (m, (x, \pi))$, i.e. the verification request with input $(m, f(x), \pi, vk)$ returns 1. Let $(m, \sigma') = (m, (x, h, \pi))$ be the output of $\mathcal{B}$ where h is the hash part of the response by the VRO to a query on input q. Whether this signature is correct is determined by whether $h = f(x)$ and the verification query on input (m, h, π, vk) returns 1. As we know that this verification query with h replaced by $f(x)$ does in fact return 1 we have to check that $h = f(x)$ holds with great enough probability, but this is implied by the fact that the ideal functionality $\mathcal{F}_{VRO}$ never returns 1 on verification requests for pairs (q, h) where either the hash of q has not been fixed yet or has been fixed to a value $h' \neq h$. Both cases might occur, because $\mathcal{A}$ might not have queried the VRO on m. □

Issues of Interaction Note that with the changes we have made, key generation and signing are no longer purely non-interactive (where this definition of non-interactive includes random oracle calls as they are guaranteed to be answered). Requiring interaction during signing is necessary, the signer has to learn the hash of its likely fresh message. Retrieving the verification data from $\mathcal{F}_{VRO}$ could be shifted to the verification step, thereby making key generation non-interactive again, but this seems to be a strictly worse option as it nullifies what we gain by the inclusion of proofs in signatures.

4.1.4. Proof of Security

In this section, we prove that FDH-VROM (with hashes included in signatures) does fulfill the EUF-CMA definition in the VROM given in Section 4.1.1. We mainly have to prove that the augmented signatures do not make it substantially easier to forge signatures by using the properties of the ideal functionality $\mathcal{F}_{VRO}$. We reduce the security of the scheme to the security of the original scheme in the ROM.

Remark 4.1.4. While FDH-ROM is sEUF-CMA secure and even has *unique signatures*, our definition of $\mathcal{F}_{VRO}$ has (perfectly) malleable proofs.[2] By including these malleable proofs in signatures we can not hope to proof any stronger notion than EUF-CMA security for FDH-VROM.

We prove the following theorem.

Theorem 4.1.5. *The signature scheme* FDH-VROM *with hashes included in signatures is* EUF-CMA*-secure in the VROM.*

Proof. We consider an adversary $\mathcal{A}$ in the VROM and construct another adversary $\mathcal{B}$ in the ROM such that if $\mathcal{A}$ has a non-negligible advantage, then so has $\mathcal{B}$. We construct $\mathcal{B}$ by having it internally run a copy of $\mathcal{A}$. $\mathcal{B}$ has access to a signing oracle Sign and a random oracle H while $\mathcal{A}$ expects access to an instance of $\mathcal{F}_{VRO}$, including the adversary interface, as well as a signing oracle Sign′.

As we have described in Section 3.2, the adversaries $\mathcal{A}$ which we have to consider answer immediately when this is required by $\mathcal{F}_{VRO}$. This still allows $\mathcal{A}$ to delay responses to `Init` or `Hash` messages sent to $\mathcal{F}_{VRO}$. We now argue that in the present situation, not delivering all messages immediately only lowers the probability of winning for $\mathcal{A}$. Formally, for any adversary $\mathcal{A}$ which is allowed to delay messages we construct another adversary $\hat{\mathcal{A}}$ which immediately delivers all messages and such that the probability of $\hat{\mathcal{A}}$ to win the EUF-CMA game in the VROM is at least as high as the corresponding probability of $\mathcal{A}$.

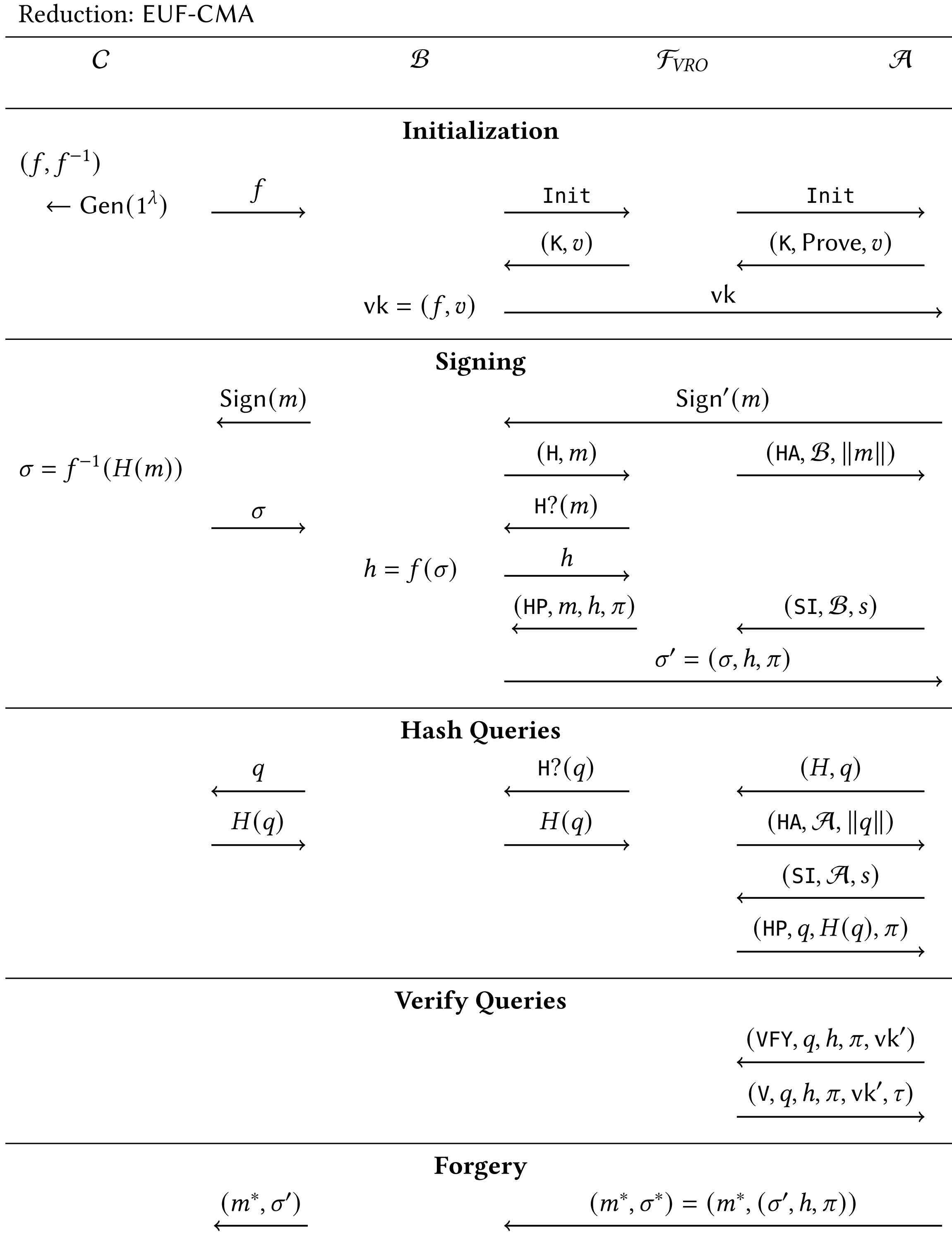
$\hat{\mathcal{A}}$ behaves as follows: It acts as a channel between $\mathcal{A}$ and the challenger. Whenever $\hat{\mathcal{A}}$ is asked by $\mathcal{F}_{VRO}$ to deliver some message m, it does so immediately. When $\mathcal{A}$ finally asks for m to be delivered, $\hat{\mathcal{A}}$ does nothing. When $\hat{\mathcal{A}}$ receives a signature σ for some message m, it checks whether $\mathcal{A}$ has allowed the contained proof π to be delivered and only sends σ to $\mathcal{A}$ when this has happened. The same mechanism is used for proofs requested by $\mathcal{A}$ directly from $\mathcal{F}_{VRO}$ as well as any time the verification key vk is retrieved by $\mathcal{A}$.

It is easy to see that the view provided to $\mathcal{A}$ by $\hat{\mathcal{A}}$ is as if $\mathcal{A}$ had interacted with the challenger itself. As such, by outputting whatever $\mathcal{A}$ outputs, $\hat{\mathcal{A}}$ has a larger or equal probability of winning.

Now we describe $\mathcal{B}$ in detail. $\mathcal{B}$ behaves as follows:

- $\mathcal{B}$ runs a simulated copy of $\mathcal{A}$.
- $\mathcal{B}$ provides access to an honest instance of $\mathcal{F}_{VRO}$, except that instead of sampling the hash h for some input q as $h \leftarrow\$ \mathcal{X}$, $\mathcal{B}$ sets $h = H(q)$.

[2]With this we mean that, once a proof π for (q, h) has been output by $\mathcal{F}_{VRO}$, the adversary is free to let $\mathcal{F}_{VRO}$ accept *any* π' as a valid proof for (q, h).



Protocol 4.2.: The reduction from EUF-CMA in the VROM to EUF-CMA in the ROM. For space reasons we use the following abbreviations: `HashProof(HP)`, `Hash(H)`, `Hashing(HA)`, `SimInfo(SI)`, `Verify(VFY)`, `Verified(V)`, `Key(K)`.

- Upon receiving a TDOWP key f:
  1. $\mathcal{B}$ simulates a `Init` query to $\mathcal{F}_{VRO}$.
  2. $\mathcal{A}$ responds by providing a message containing (Prove, v) and allows the response (Key, v) for $\mathcal{B}$ to be delivered, $\mathcal{B}$ constructs the verification key $\mathsf{vk} = (f, v)$.
  3. $\mathcal{B}$ sends vk to $\mathcal{A}$.
- Upon receiving a $\mathsf{Sign}'(m)$ request from $\mathcal{A}$:
  1. $\mathcal{B}$ executes a $\mathsf{Sign}(m)$ query itself, obtaining a signature σ.
  2. It then makes a simulated hash query for m to $\mathcal{F}_{VRO}$ and receives (h, π) (where $h = H(m)$ as per the way $\mathcal{F}_{VRO}$ is simulated).
  3. $\mathcal{A}$ receives the signature $\sigma' = (\sigma, h, \pi)$.
- Upon receiving a purported forgery (m^*, σ^*) from $\mathcal{A}$:
  1. $\mathcal{B}$ parses σ^* as (σ, h, π).
  2. And returns (m^*, σ) to its own challenger.

The full reduction is shown in Protocol 4.2. To simplify the code for $\mathcal{B}$, especially with respect to the instance of $\mathcal{F}_{VRO}$ it has to simulate towards $\mathcal{A}$, we show $\mathcal{F}_{VRO}$ in a separate column and let messages $\mathsf{H?}(q)$ sent from $\mathcal{F}_{VRO}$ to $\mathcal{B}$ denote $\mathcal{F}_{VRO}$ asking for the hash h it should set for q. In every other aspect $\mathcal{F}_{VRO}$ behaves honestly.

Proving Success We now prove that if $\mathcal{A}$ has a non-negligible probability of producing a forgery then so does $\mathcal{B}$. We first show that the $\mathcal{B}$ simulates a valid environment for $\mathcal{A}$ and then that any valid forgery output by $\mathcal{A}$ leads to $\mathcal{B}$ outputting a valid forgery as well.

First, as H is a random oracle, the instance of $\mathcal{F}_{VRO}$ which $\mathcal{B}$ provides for $\mathcal{A}$ behaves exactly like an honest instance. The answers to Sign' queries are also correct. Let m be a queried message and $\sigma' = (\sigma, h, \pi)$ be the response by $\mathcal{B}$. Assuming that $\mathcal{A}$ allows the hash query of $\mathcal{B}$ for m to succeed, $f(\sigma)$ is equal to h and π is a valid proof for the fact that the hash of m is equal to h (under verification key v). As these are exactly the conditions that are checked by the verification algorithm of FDH-VROM, the returned signature σ' is valid with respect to the verification key vk and the simulated $\mathcal{F}_{VRO}$. Notice that π is distributed correctly by running the Prove provided by $\mathcal{A}$ on m, h and s (where $\mathcal{A}$ was also allowed to provide s).

We now turn to analyzing the probability of success of $\mathcal{B}$ in breaking EUF-CMA in the ROM. Suppose we are given a message and signature pair (m^*, σ^*) by $\mathcal{A}$ and where $\mathcal{A}$ has never made a Sign' query for m^*. Let again $\mathsf{vk} = (f, v)$ be the verification key. We can expand σ^* to (σ', h, π). If $\mathcal{A}$ would have won the EUF-CMA game, then it must hold that $f(\sigma') = h$ and π is a proof such that if $\mathcal{B}$ was to send a message $(\mathsf{Verify}, m^*, h, v, \pi)$ to $\mathcal{F}_{VRO}$, $\mathcal{F}_{VRO}$ would accept. By the verification portion of the code of $\mathcal{F}_{VRO}$, the latter implies that

- a hash query has been previously made for m.

- h is the hash that has been associated with q upon the first such hash query.

As m^* has not been included in any previous Sign' query, this query was done by $\mathcal{A}$. By the manner in which $\mathcal{B}$ supplies hashes to be used by $\mathcal{F}_{VRO}$, it then holds that $h = H(m^*)$. Combining this with $f(\sigma') = h$ we obtain $f(\sigma') = H(m^*)$ which is exactly the equation to verify signatures within FDH-ROM. This shows that $\mathcal{B}$ has the same probability of producing a forgery as $\mathcal{A}$. □

By combining Theorem 4.1.5 and Lemma 4.1.3 we obtain the following corollary.

Corollary 4.1.6. *The signature scheme* FDH-VROM *without hashes included in signatures is* EUF-CMA-*secure in the VROM.*

4.1.5. Final Thoughts

As we saw in the last section, there is essentially nothing to prove to show that FDH can be transferred into the VROM. This is because of the perfect unforgeability properties of $\mathcal{F}_{VRO}$ which yield that every verifying signature (σ', h, π) has to contain the true hash of the image of the first component of the signature under f. What one should keep in mind is that this only holds with respect to the ideal $\mathcal{F}_{VRO}$. Once one replaces $\mathcal{F}_{VRO}$ by some protocol π UC-realizing it, these properties may degrade to only hold in a computational sense. The $(\texttt{Hashing}, \dots)$ messages we provided to the attacker were not helpful to him, because the signatures already contain the hash contained in these messages (and the attacker knows the message and so could just query the VRO directly). Similarly, the ability to delay the delivery of responses to $\mathcal{F}_{VRO}$ queries could not help the adversary in the present context.

4.2. The Fischlin Transformation

In this section we apply the VROM to a more involved example, the Fischlin transformation [45]. The Fischlin transform is used to transform a certain class of Σ-protocols into non-interactive zero-knowledge proofs of knowledge (NIZKPoK) in the ROM. In contrast to other kinds of transforms which achieve the same goal, such as the Fiat-Shamir transform [44], whose knowledge extractor relies on rewinding techniques and the forking lemma [80], the Fischlin transform has the nice property that the produced NIZKPoKs are *online extractable.* As we have already defined in Chapter 2, online extractability requires the existence of an efficient machine Ext, the *online extractor*, which, when given the output (x, π) and list of random oracle queries Q made by a cheating prover $\mathcal{A}$, can extract a witness w for the proved statement x, assuming that π is a valid proof.

We begin by defining online extractable NIZKPoKs in the VROM. The notions of security we consider are completeness, zero-knowledge, and online extractability. We base these definitions on the respective definitions in the ROM which are stated in Chapter 2. Then we recall definitions related to the Fischlin transform in the ROM from [45, 71] and give our adaptation of it to the VROM setting. We conclude by proving the Fischlin transform in the VROM secure with respect to our security definitions.

4.2.1. Definition of Security

In defining the security of NIZKPoKs, we consider completeness, zero-knowledge, and online extractability. We do not consider soundness separately, because it is implied by online extractability. For the complete definitions of these notions in the ROM, we refer back to Chapter 2.

There are multiple ways we might go about defining the security of a NIZKPoK in the VROM. The first is by extending the usual game-based definitions involving honest and cheating provers or verifiers, simulators, and extractors and describing how they interact in the presence of an instance of $\mathcal{F}_{VRO}$. This is the route chosen below. There are also ideal functionalities trying to capture either zero-knowledge in general or non-interactive zero-knowledge in particular. We will explore this setting in Section 4.2.5.

Completeness Completeness means that an honest prover will convince an honest verifier on an input $(x, w) \in \mathcal{R}_L$, either always in the perfect completeness case or with overwhelming probability in the statistical case. As a NIZKPoK which is the result of applying the Fischlin transform in general possesses a negligible completeness error[3], this is the notion we consider.

Recall that for completeness in the ROM we require that there exists a negligible function $\mathsf{negl}(\lambda)$ such that for a random oracle H and every $(x, w) \in \mathcal{R}_\lambda$ it holds that

$$\mathbf{Pr}\left[\pi \leftarrow \mathcal{P}^H(x, w) : \mathcal{V}^H(x, \pi) = 0\right] \leq \mathsf{negl}(\lambda)$$

where the probability is taken over the random coins of $\mathcal{P}^H$ as well as the randomness used to generate outputs of H.

In the completeness "game", both the prover and the verifier are honest. There thus does not immediately seem to exist any adversarial party to which we could make the adversary interface of $\mathcal{F}_{VRO}$ available. This did not pose any problems when working in the ROM where there exists a clear definition of a RO which is under no adversarial influence. The same is, however, not true in the VROM as $\mathcal{F}_{VRO}$ requires explicit initialization. One way to handle this is as follows: The verification key vk is set to $\perp$ and Prove, on input (q, h, s) always returns $\perp$ as well. There is, however, another possibility that seems to be more natural.

We consider an arbitrary adversary $\mathcal{A}$ which acts as a kind of external attacker, corrupting neither the prover nor the verifier, trying to break the completeness of the protocol by way of its corruption of $\mathcal{F}_{VRO}$. By considering such an $\mathcal{A}$ we formally give it the ability to delay the delivery of `HashProof` responses, supposedly to the prover, as well as `Key` responses, supposedly to the verifier. Delaying all such messages would immediately break the ordinary notion we have for completeness, i.e. requiring the verifier to end the interaction by accepting a proof it has received. We solve this issue by restricting the class of adversaries to those delivering all types of messages by $\mathcal{F}_{VRO}$ mentioned above. Note that this is in addition to the restriction to adversaries which provide immediate responses to certain requests by $\mathcal{F}_{VRO}$, see the discussion in Section 3.2. The abilities of $\mathcal{A}$ are thus essentially the following: (1) Initializing $\mathcal{F}_{VRO}$ by providing Prove and vk, (2) being notified upon each hash query made to $\mathcal{F}_{VRO}$ and where this notification includes

[3]We note that in [45], Fischlin describes a way to achieve perfect completeness.

learning the identity $\mathcal{P}$ of the party making the request as well as the length $\|q\|$ of the input, (3) being able to provide `SimInfo` messages s, (4) in some cases determining the bit in the response to proof verification queries.

Definition 4.2.1 (VROM Completeness). A NIZKPoK $\Pi = (\mathcal{P}, \mathcal{V})$ in the VROM is called complete, if for all PPT adversaries $\mathcal{A}$ which deliver all messages between $\mathcal{F}_{VRO}$ and the algorithms in Π there exists a negligible function $\mathsf{negl}(\lambda)$, such that for all $\lambda \in \mathbb{N}$ and all $(x, w) \in \mathcal{R}_\lambda$ the probability that in the following game $b = 1$ is output, is less than $\mathsf{negl}(\lambda)$.

- Give $\mathcal{A}$ the adversary interface for a single session of $\mathcal{F}_{VRO}$.
- Run $\mathcal{P}(1^\lambda, x, w)$ with oracle access to $\mathcal{F}_{VRO}$ until $\mathcal{P}$ outputs some π.
- Run $\mathcal{V}(1^\lambda, x, \pi)$ with oracle access to $\mathcal{F}_{VRO}$ until $\mathcal{V}$ outputs a bit b.
- Set b as the output of the game.

The probability is taken over the randomness used by $\mathcal{P}$ as well as $\mathcal{F}_{VRO}$.

Zero-Knowledge In the original paper [45], zero-knowledge is defined in a somewhat non-standard manner. It requires the existence of a zero-knowledge simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$ such that for all PPT distinguishers $\mathcal{D} = (\mathcal{D}_1, \mathcal{D}_2)$, the following two distributions are indistinguishable[4] :

- Let H be a random oracle, $(x, w, \delta) \leftarrow \mathcal{D}_1^H(1^\lambda)$, and $\pi \leftarrow \mathcal{P}^H(x, w)$.
  If $(x, w) \in \mathcal{R}_L$, output $\mathcal{D}_2^H(\pi, \delta)$.
- Let $(H_1, \sigma) \leftarrow \mathcal{S}_1(1^\lambda)$, $(x, w, \delta) \leftarrow \mathcal{D}_1^{H_1}(1^\lambda)$, and $(H_2, \pi) \leftarrow \mathcal{S}_2(\sigma, x)$.
  If $(x, w) \in \mathcal{R}_L$, output $\mathcal{D}_2^{H_2}(\pi, \delta)$.

We will, however, instead use the definition we have made in Section 2.5.4 as we find it easier to work with. Our definition is in some ways stronger than the definition shown above. Our simulator has to provide up to polynomially many proofs for statements that were adaptively selected. Nonetheless, the two definitions seem to be ultimately incomparable. In principle, a simulator for our definition can make the simulated proofs it produces depend on the random oracle inputs previously given to it by the distinguisher. This is allowed as both the proof-generation and random oracle share a common state. $\mathcal{S}_2$ above, however, can not do so as the random oracle queries made by $\mathcal{D}_1$ are perfectly hidden from it.

We are unaware of any interesting case where the first part of the simulator $\mathcal{S}_1$ chooses H_1 to be anything other than a simple PRF. The simulator we will give below could similarly be transformed into a simulator having the form above. As such, at least in this case working with our definition yields a stronger result.

[4]We have already slightly simplified the interaction here by explicitly checking that $(x, w) \in \mathcal{R}_L$, but it is easy to see that both definitions are equivalent.

The VROM Definition Using our definition, adapting to the VROM is now straightforward. We give the distinguisher $\mathcal{D}$ oracle access to a session of $\mathcal{F}_{VRO}$, including the adversary interface, as well as either an honest prover $\mathcal{P}$ or the simulator $\mathcal{S}$. In the case with the simulator, the simulator also implements the instance of $\mathcal{F}_{VRO}$ (called $\mathcal{F}_{VRO}^{\mathcal{S}}$ below), in the other case $\mathcal{F}_{VRO}$ is honest. The prover $\mathcal{P}$ also has oracle access to $\mathcal{F}_{VRO}$, but without the adversary interface.

Definition 4.2.2 (VROM Zero-Knowledge)**.** A NIZKPoK $\Pi = (\mathcal{P}, \mathcal{V})$ in the VROM is said to have the zero-knowledge property, if for every PPT $\mathcal{D}$ there exists a negligible function $\mathsf{negl}(\lambda)$ such that

$$\left|\mathbf{Pr}\left[\ \mathcal{D}^{\mathcal{F}_{VRO},\mathcal{P}^{\mathcal{F}}_{VRO}}(1^{\lambda}) = 1\ \right] - \mathbf{Pr}\left[\ \mathcal{D}^{\mathcal{F}^{\mathcal{S}}_{VRO},\mathcal{S}}(1^{\lambda}) = 1\ \right]\right| = \mathsf{negl}(\lambda)$$

Remark 4.2.3. We have in some sense recreated the usual UC setting here. The distinguisher corresponds to the environment and giving it the adversary interface of $\mathcal{F}_{VRO}$ can be seen as having this access provided through the dummy adversary (as it would have if we were working in the $\mathcal{F}_{VRO}$-hybrid model). $\mathcal{S}$ then corresponds to the simulator for the dummy adversary.

The Necessity of Hiding Inputs To prove zero-knowledge of the Fischlin transform in the VROM, it is essential that the distinguisher is not given the queries made by the prover or the simulator. Imagine that we were including the hash query input q in the $(\mathtt{Hashing}, \ldots)$ messages generated by $\mathcal{F}_{VRO}$. Now also imagine that the verifier is corrupted and is provided with these messages. By the online extractability of the Fischlin transform this would immediately allow the adversary to obtain a witness.

In the current setting, this manifests itself as the distinguisher being able to extract a witness from the observed hash queries when interacting with the honest prover. To keep both kinds of interactions indistinguishable, the simulator would thus have to be able to simulate these hash queries. The simulator, however, does not have the witness to do so. Thus, either breaking the underlying relation has to be easy or there can not exist an efficient simulator. Depending on which is the case would make the NIZKPoK either trivial or not zero-knowledge.

Online Extraction In the ROM, online extractability says that there exists a PPT algorithm Ext such that for any algorithm $\mathcal{A}$ the following holds. Let H be a random oracle, $(x, \pi) \leftarrow \mathcal{A}^H$ and Q_H be all the queries made by $\mathcal{A}$ to H and the corresponding answers. Then let $w \leftarrow \mathsf{Ext}(x, \pi, Q_H)$. The probability that $(x, w) \notin \mathcal{R}_L$ and $\mathcal{V}^H(x, \pi) = 1$ is negligible.

Online extractability is a soundness property and as such considers a cheating prover. We thus give the prover adversary access to the instance of $\mathcal{F}_{VRO}$ with which it interacts. Essentially, this means the adversary is the one providing the verification key and the code for generating proofs. The notification for each hash query does not provide any new information. Apart from this, the definition is as in the ROM, if we let Q_H be the list consisting of tuples (q, h, π), i.e. we augment them to also contain the obtained proof.

Definition 4.2.4 (VROM Online Extractability)**.** A NIZKPoK $\Pi = (\mathcal{P}, \mathcal{V})$ in the VROM is online extractable, if there exists a PPT algorithm Ext such that for all adversaries $\mathcal{A}$, the probability

$$\mathbf{Pr}\Big[(x,\pi) \leftarrow \mathcal{A}^{\mathcal{F}_{VRO}}(1^\lambda); w \leftarrow \mathsf{Ext}(1^\lambda, x, \pi, Q) \mid 0 = \mathcal{V}(1^\lambda, x, \pi) \wedge (x,w) \notin \mathcal{R}_\lambda\Big]$$

is negligible in λ where $\mathcal{A}$ is given the adversary access to $\mathcal{F}_{VRO}$ and Q is contains a tuple (q, h, π) for every hash query for input q and which returned (h, π) made by $\mathcal{A}$.

Remark 4.2.5. The definition given here requires that Ext work for any, not necessarily PPT, machine $\mathcal{A}$. This is fine when working with the ideal $\mathcal{F}_{VRO}$.

Remark 4.2.6. We may also give the extractor access to other types of queries made by $\mathcal{A}$, i.e. also verification queries. Our reasoning for not doing so is the following: First, only verification queries for the correct verification key vk could be of any use for $\mathcal{A}$ as it has to forge a proof verifiable with respect to vk. Second, only accepting verification queries are useful and these have to be preceded by a hash query for the same input.

4.2.2. The Fischlin Transformation in the ROM

We now define the Fischlin transform in the ROM. The basic idea to achieve online extractability without relying on rewinding techniques is by essentially forcing the prover to rewind itself. Informally, to compute a proof for some input (x, w), the prover tries to find inputs q of a specific format, which include a valid transcript of the underlying Σ-protocol, and for which $H(q) = 0$ where H is a random oracle. Depending on the output length of H, this will take more or fewer attempts. Furthermore, the q which together can be used to form a valid proof π are linked in such a way that they can only be efficiently computed and input into H if a witness for the statement to be proved is known. This then leads to the situation that any party not in possession of a witness w can not produce a valid proof π, and a party which does know a witness w is forced to give inputs to H which allow efficient computation of w. We now give a formal definition.

Let $\Sigma = (\mathcal{V}_\Sigma, \mathcal{P}_\Sigma)$ be a Σ-protocol for the language L with witness-relation $\mathcal{R}_L$. According to our definition from Chapter 2 this means:

- Σ is complete.
- Σ has quasi-unique responses.
- Σ has super-logarithmic commitment min-entropy.
- There exists a special zero-knowledge simulator $\mathcal{S}_\Sigma$, which on input any statement x and challenge ch outputs an accepting transcript (com, ch, resp).
- There exists a special soundness extractor Ext_Σ, which on input two accepting transcripts (com, ch, resp) and (com, ch′, resp′) for statement x with $\mathsf{ch} \neq \mathsf{ch}'$ outputs a witness w with $(x, w) \in \mathcal{R}_L$.

These restrictions to a sub-class of Σ-protocols to which the transform can be applied are taken from [45].

We further introduce parameters l, b, r and t which are functions of the security parameter λ. Let H be a random oracle with domain $\{0,1\}^*$. Then:

- l is the length of challenges of Σ, i.e. Σ has challenge space $\{0,1\}^l$.
- b is the length of outputs of H, i.e. H has codomain $\{0,1\}^b$.
- r and t are computational parameters of the Fischlin transform where r is the number of repetitions of the underlying Σ-protocol and t is a soundness parameter.

These parameters have to fulfill several relations:

- $l, b, r, s \in O(\log \lambda)$
- $br \in \omega(\log \lambda)$
- $2^{t-b} \in \omega(\log \lambda)$
- $b \leq t \leq l$

The above set of requirements is required for the applicability of the transformation as originally described by Fischlin [45] (call this the *original Fischlin transform*). Recently, Kondi *et al.* [71] have identified practical issues regarding the applicability of the original Fischlin transform. In particular, they show that it can be difficult to decide whether the required quasi-uniqueness property holds in certain contexts as it depends on what additional information the attacker $\mathcal{A}$ possesses. This leads to situations where quasi-uniqueness holds intuitively, but does not under more careful inspection. They also describe a practical attack on a class of protocols which have been transformed using the original Fischlin transform. It is based on the deterministic nature of the prover as defined by Fischlin and which allows an attacker to determine which witness was used by the prover to compute a proof in a transformed OR-protocol [30]. OR-protocols are used to prove logical disjunctions and require that the prover supply a witness for either term of the statement. It has to be stressed that this is *not* an attack on the original Fischlin transform itself. The protocol used in the attack violates the quasi-uniqueness property and so does not allow the original Fischlin transformation to be applied soundly. We will go into more detail after we have defined both versions of the transform. In [71] it is shown how to forego the context-dependent requirement of quasi-unique responses by introducing the notion of *strong special soundness* for Σ-protocols. It is defined as follows.

Definition 4.2.7 (Strong Special Soundness)**.** A Σ-protocol Σ for the NP-relation $\mathcal{R}$ has the *strong special soundness* property if there exists a PPT algorithm Ext such that for all x, two valid transcripts $t = (\mathsf{com}, \mathsf{ch}, \mathsf{resp})$ and $t' = (\mathsf{com}, \mathsf{ch}', \mathsf{resp}')$ with $t \neq t'$, and $w \leftarrow \mathsf{Ext}(x, t, t')$, it holds that $(x, w) \in \mathcal{R}$.

Remark 4.2.8. Note that having quasi-unique responses and ordinary special soundness implies strong special soundness. Given two transcripts $t = (\mathsf{com}, \mathsf{ch}, \mathsf{resp})$ and $t' = (\mathsf{com}, \mathsf{ch}', \mathsf{resp})$ with $t \neq t'$ and for some statement x, the strong special soundness extractor runs the special soundness extractor on (x, t, t'). This is valid as with overwhelming probability $\mathsf{resp} \neq \mathsf{resp}'$ due to the quasi-unique responses property.

The authors of [71] then show that there exists a transformation from Σ-protocols satisfying the same conditions as for the original Fischlin transform, except with strong special soundness replacing quasi-unique responses and ordinary special soundness, to online extractable NIZKPoKs where all security definitions are as defined in [45]. Informally, they randomize the original Fischlin prover and adjust the zero-knowledge simulator accordingly. For this reason, we call this the *randomized Fischlin transform*. The randomized Fischlin transform depends on the same parameters l, b, r, s, t defined above. In [71], identical conditions as above are imposed on these parameters. As we will show in Appendix A.5, the proof of the validity of the zero-knowledge simulator in [71] contains a small formal error that allows an attacker to distinguish honestly generated proofs from simulated proofs by detecting the programming of the random oracle by the simulator. We give one possible solution to resolve this issue. To do so, we invert the requirement $l \in O(\log \lambda)$ imposed by the original Fischlin transform and instead require $l \in \omega(\log \lambda)$. As we still require $t \in O(\log \lambda)$, we in addition replace the requirement $b \leq t \leq l$ by $b \leq t \leq m$ for $m \in O(\log \lambda)$.

Prover We first describe the prover in the original Fischlin transform. The prover $\mathcal{P}^H$ receives as input a pair $(x, w) \in \mathcal{R}_L$. It begins by running the prover $\mathcal{P}_\Sigma$ of the underlying Σ-protocol r times on (x, w) to generate r commitments com_i for $i \in [r]$. It combines all the commitments into a vector $\vec{\mathsf{com}} = (\mathsf{com}_1, \ldots, \mathsf{com}_r)$. For each repetition $i \in [r]$, $\mathcal{P}^H$ sequentially steps through all strings $\mathsf{ch}_i = 0, 1, \ldots, 2^t$ of length t and computes the response resp_i which makes $(\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i)$ a valid transcript for x. It continues to do this until it finds one for which $H(x, \vec{\mathsf{com}}, i, \mathsf{ch}_i, \mathsf{resp}_i) = 0^b$. If for one of the repetitions no such input is found, $\pi = \bot$ is output. Otherwise $\mathcal{P}^H$ outputs $\pi = (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i)_{i=1,2,\ldots,r}$.

The algorithm on the left in Figure 4.3 is the original definition from [45] which we have just described. The algorithm on the right is the randomized prover from [71]. As noted in the introduction to the current section, the right prover uses challenges of length $l \in \omega(\log \lambda)$ rather than $l \in O(\log \lambda)$ as the original Fischlin prover does. This only slightly increases the total size of proofs as the size of commitments is already required to be in $\omega(\log \lambda)$ due to the requirement on their min-entropy. As opposed to the original prover, to the randomized prover in each iteration of the loop, instead of incrementing the previous challenge by one, chooses a random challenge from the set of challenges that have not been previously tried.

We have only described the original prover for completeness, but will from now on restrict our attention the randomized prover. We note that the verifier described above is used in both variants of the transform. In particular, stepping through the challenges randomly does not affect the completeness error. In case no proof is computed, both provers will have tried the same number of challenges, albeit of different lengths. As each query to H produces independent random output, only the number of trials determines the probability of success.

$\mathcal{P}^H(x, w)$

1 : **for** $i \in \{1, 2, \ldots, r\}$ **do**
2 : $\quad \mathsf{com}_i \leftarrow \mathcal{P}_\Sigma(x, w)$
3 : **endfor**
4 : $\vec{\mathsf{com}} = (\mathsf{com}_1, \mathsf{com}_2, \ldots, \mathsf{com}_r)$
5 : **for** $i \in \{1, 2, \ldots, r\}$ **do**
6 : $\quad$ **for** $\mathsf{ch}_i \in \{0, 1\}^t$ **do**
7 : $\quad\quad \mathsf{resp}_i \leftarrow \mathcal{P}_\Sigma(x, w, \mathsf{com}_i, \mathsf{ch}_i)$
8 : $\quad\quad h = H(x, \vec{\mathsf{com}}, i, \mathsf{ch}_i, \mathsf{resp}_i)$
9 : $\quad\quad$ **if** $h = 0$
10 : $\quad\quad\quad$ **go to next** i
11 : $\quad\quad$ **fi**
12 : $\quad\quad$ **if** $\mathsf{ch}_i = 2^t - 1$
13 : $\quad\quad\quad$ **return** $\bot$
14 : $\quad\quad$ **fi**
15 : $\quad$ **endfor**
16 : **endfor**
17 : $\pi = (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i)_{i=1,2,\ldots,r}$
18 : **return** π

$\mathcal{P}^H(x, w)$

1 : **for** $i \in \{1, 2, \ldots, r\}$ **do**
2 : $\quad \mathsf{com}_i \leftarrow \mathcal{P}_\Sigma(x, w)$
3 : **endfor**
4 : $\vec{\mathsf{com}} = (\mathsf{com}_1, \mathsf{com}_2, \ldots, \mathsf{com}_r)$
5 : **for** $i \in \{1, 2, \ldots, r\}$ **do**
6 : $\quad \mathcal{E}_i = \emptyset$
7 : $\quad$ **while** $|\mathcal{E}_i| < 2^t$ **do**
8 : $\quad\quad \mathsf{ch}_i \leftarrow\!\!\$\ \{0, 1\}^l \setminus \mathcal{E}_i$
9 : $\quad\quad \mathsf{resp}_i \leftarrow \mathcal{P}_\Sigma(x, w, \mathsf{com}, \mathsf{ch}_i)$
10 : $\quad\quad h = H(x, \vec{\mathsf{com}}, i, \mathsf{ch}_i, \mathsf{resp}_i)$
11 : $\quad\quad$ **if** $h = 0$ **do**
12 : $\quad\quad\quad$ **go to next** i
13 : $\quad\quad$ **fi**
14 : $\quad\quad$ **if** $|\mathcal{E}_i| = 2^t - 1$ **do**
15 : $\quad\quad\quad$ **return** $\bot$
16 : $\quad\quad$ **fi**
17 : $\quad$ **endwhile**
18 : **endfor**
19 : $\pi = (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i)_{i=1,2,\ldots,r}$
20 : **return** π

Figure 4.3.: The original (left) and the randomized version (right) of the Fischlin prover.

```
V^H(x, π)
1 :  (com_i, ch_i, resp_i)_{i=1,2,...,r} = parse(π)
2 :  com⃗ = (com_1, com_2, ..., com_r)
3 :  for i ∈ {1, 2, ..., r} do
4 :     h = H(x, com⃗, i, ch_i, resp_i)
5 :     σ = V_Σ(x, com_i, ch_i, resp_i)
6 :     if h ≠ 0 ∨ σ = 0 do
7 :        return 0
8 :     fi
9 :  endfor
10 : return 1
```

Figure 4.4.: The Fischlin verifier in the ROM.

Remark 4.2.9. Notice how the original prover samples challenges from $\{0,1\}^t$ instead of $\{0,1\}^l$, although l is technically the challenge length of the underlying Σ-protocol. This is done for efficiency reasons. As long as t in conjunction with the other parameters fulfills the requirements laid out at the beginning of the current section, the prover can get away with only trying at most 2^t challenges and still achieve a negligible completeness error. This is not a problem as any Σ-protocol Σ_1 with challenges of length l_1 can be interpreted as Σ-protocol Σ_2 with challenges of length $l_2 \leq l_1$ by having the verifier of Σ_2 behave as the verifier in Σ_1, but fix the first $l_1 - l_2$ bits of each challenge to zeroes. These zeroes do not have to be communicated to the prover of Σ_2 but can rather be added to the truncated challenge by it when running the code of the underlying prover of Σ_1.

Verifier The verifier $\mathcal{V}^H$ receives as input a statement x and a purported proof $\pi = (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i)_{i=1,2,\ldots,r}$ and returns a bit $b \in \{0,1\}$. It outputs $b = 1$ if and only if for all of the r transcripts $(\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i)$, the verifier V_Σ of the underlying Σ-protocol accepts and in addition $H(x, \vec{\mathsf{com}}, i, \mathsf{ch}_i, \mathsf{resp}_i) = 0^b$ where $\vec{\mathsf{com}}$ is again the vector of all commitments contained in π. The code for the Fischlin verifier is shown in Figure 4.4.

Sum of Hashes In the original paper, the sum of the $H(x, \vec{\mathsf{com}}, i, \mathsf{ch}_i, \mathsf{resp}_i)$ only had to be less than some parameter S with $S \in \mathcal{O}(r)$. This was done to reduce the completeness error introduced by the transform, i.e. reduce the probability of not being able to produce a valid proof.

Requiring all of them to be 0^b still leaves the completeness error to be negligible and lets us save some space later. It can be seen by looking at the upper estimate for the completeness error from [45] which is given by

$$\exp\left(r \ln(\mathrm{e}(2S+1)) - (S+1)2^{t-b}\right).$$

Setting $S = 0$ leads to the expression

$$\exp\left(r \ln(\mathrm{e}) - 2^{t-b}\right) = \exp\left(r - 2^{t-b}\right).$$

```
S(x)
 1 : for i ∈ {1, 2, ..., r} do
 2 :   ch_i ←$ {0, 1}^t
 3 :   (com_i, resp_i) ← S_Σ(x, ch_i)
 4 : endfor
 5 : com⃗ = (com_1, com_2, ..., com_r)
 6 : for i ∈ {1, 2, ..., r} do
 7 :   h = (x, com⃗, i, ch_i, resp_i)
 8 :   if H_S is already defined on h do
 9 :     return ⊥
10 :   fi
11 :   define H_S(h) = 0
12 : endfor
13 : return (com_i, ch_i, resp_i)_{i=1,2,...,r}
```

Figure 4.5.: The Fischlin zero-knowledge simulator in the ROM.

By the requirements on r, t and b for the applicability of the Fischlin transform, $r \in \mathcal{O}(\log\lambda)$ and $2^{t-b} \in \omega(\log\lambda)$, the above term is negligible in the security parameter λ also when setting $S = 0$.

Simulator We directly give a simulator for our definition of zero-knowledge. We also make some efficiency changes to the original simulator afforded to us by simulating for the second prover from above. The code of the simulator is shown in Figure 4.5 and is essentially taken from [71], except that like the prover, the challenger samples challenges of super-logarithmic length l. Compared to the simulator for the original Fischlin transform, this simulator only has to program H on r inputs and uses fewer random bits.

As we have changed the zero-knowledge definition, the prover, and the simulation strategy with respect to [45] we have to prove that what we have defined is a valid simulator. We claim the following theorem to hold.

Theorem 4.2.10. *Let Σ be a Σ-protocol satisfying the requirements for the randomized Fischlin transform and let Σ' be the transformed protocol in the ROM. Then Σ' has the zero-knowledge property as defined in Section 4.2.1.*

Proof. We have shown above that the completeness error for the prover is negligible. The same is true for the simulator. For the simulator to not be able to produce a proof, it must have failed to program $H_{\mathcal{S}}$ at some input. Because the r commitments com_i contained in each hash query by the prover have entropy which is super-logarithmic in the security parameter, this only happens with negligible probability. We can in the following thus condition on no completeness error occurring. Note that this reasoning only shows that the distinguisher can not induce the simulator to fail using adaptive proof requests. It remains to be shown that they do not aid the distinguisher in other ways.

We turn to the distribution of proofs and focus our attention on a single com_i. As the honest prover steps through the challenges randomly, each challenge has the same chance of being included in the final proof. The simulator, on the other hand, chooses a single challenge uniformly. These two distributions are identical. Equality of proof distributions now follows from the (perfect) special zero-knowledge property of the Σ-protocol.[5]

This leaves the possibility of distinguishing both worlds by detecting the programming of the random oracle. Note that only distributional attacks have to be considered as we have chosen to let the simulator halt if it would have tried to change some already defined oracle output. If our simulator had chosen (pseudo-)random outputs to program, no argument would have to be made. Such programming is always allowed. But $\mathcal{S}$ programs using the fixed value 0^b which is decidedly not pseudo-random.

In the following, we will argue that even an unbounded distinguisher has negligible probability in distinguishing the simulated random oracle $H_{\mathcal{S}}$ from an honest random oracle H. First, consider a single proof for (x, w) being generated by $\mathcal{S}$ and $\mathcal{P}$ and where both parties use the same vector $\vec{\mathsf{com}} = (\mathsf{com}_1, \mathsf{com}_2, \dots, \mathsf{com}_r)$ of commitments. After having generated $\vec{\mathsf{com}}$, $\mathcal{P}$ will make a number of queries to H. The set of all possible such queries, at most 2^t of which are actually made by $\mathcal{P}$, is

$$Q = \left\{(x, \vec{\mathsf{com}}, i, \mathsf{ch}, \mathsf{resp}) \mid i \in [r], \mathsf{ch} \in \{0,1\}^l, \mathcal{V}^H(x, \mathsf{com}_i, \mathsf{ch}, \mathsf{resp}) = 1\right\}.$$

We also define

$$Q_i = \left\{(x, \vec{\mathsf{com}}, i, \mathsf{ch}, \mathsf{resp}) \mid \mathsf{ch} \in \{0,1\}^l, \mathcal{V}^H(x, \mathsf{com}_i, \mathsf{ch}, \mathsf{resp}) = 1\right\}.$$

Now, suppose that both $\mathcal{S}$ and $\mathcal{P}$ end up with the same proof $\pi = (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i)_{i=1,2,\dots,r}$.[6] As $\mathcal{P}$ has arrived at these challenges by sampling from Q until it has found enough inputs with hash 0^b, H will return a uniform random value on all inputs in Q. In short $H\restriction_{Q_i} : Q_i \to \{0,1\}^b$ is a random function and $q_i = (x, \vec{\mathsf{com}}, i, \mathsf{ch}_i, \mathsf{resp}_i)$ are random values such that $H(q_i) = 0^b$ holds[7]. $\mathcal{S}$, on the other hand, has sampled the ch_i randomly and then programmed $H_{\mathcal{S}}$ to output 0^b on q_i for $i \in [r]$. In short, $H_{\mathcal{S}} \restriction_{Q_i} : Q_i \to \{0,1\}^b$ has been sampled from the set of all functions from Q_i to $\{0,1\}^b$ and only then $H_{\mathcal{S}}(q_i)$ has been set to 0^b (alternatively, conditioned on these relations holding).

The distinguishers task (for a single proof) is then to distinguish r samples taken from either of these distributions (outside of all the Q_i, both H and $H_{\mathcal{S}}$ are identically distributed). But the statistical distance between the two distributions just described is less than 2^{-l} (probability mass of less than this amount has been shifted from hashes other than 0^b to 0^b) and therefore negligible. A polynomial-time distinguisher requesting $p = \mathsf{poly}(\lambda)$ proofs thus receives rp samples and so this type of attack adds at most $rp2^{-l}$ to the distinguishing advantage, which is still a negligible amount. □

[5] We note that computational special zero-knowledge would be sufficient, but would require a hybrid-argument to show that even the polynomially many samples observed by the distinguisher only help him negligibly.

[6] Recall that we have assumed that the prover always succeeds in producing a proof.

[7] This does not mean that we have conditioned on this to be the case.

```
Ext(x, π, Q_H)
1 : (com_i, ch_i, resp_i)_{i=1,2,...,r} = parse(π)
2 : com⃗ = (com_1, com_2, ..., com_r)
3 : for i ∈ {1, 2, ..., r} do
4 :   if (x, com⃗, i, ch*, resp*) ∈ Q_H, ch_i ≠ ch* do
5 :     return Ext_Σ(x, com_i, ch_i, ch*, resp_i, resp*)
6 :   fi
7 : endfor
8 : return ⊥
```

Figure 4.6.: The online extractor in the ROM.

Remark 4.2.11. Notice how this reasoning breaks down if we do not require $l \in \omega(\log \lambda)$. It would only mean that the statistical distance is bounded by $2^{-l} \in O(\frac{1}{\mathsf{poly}(\lambda)})$. The attack in Appendix A.5 can then quickly be described as follows: The distinguisher requests a proof for (x, w) and receives a proof $\pi = (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i)_{i=1,2,\dots,r}$. It picks some $j \in [i]$ and for all $\mathsf{ch} \in \{0,1\}^l$ with $\mathsf{ch} \neq \mathsf{ch}_j$ it computes the accepting transcript $(\mathsf{com}_j, \mathsf{ch}, \mathsf{resp})$. Then, it queries the random oracle on all the $(x, \vec{\mathsf{com}}, j, \mathsf{ch}, \mathsf{resp})$ and compares the distribution of hashes to the uniform distribution.

Online Extractor The code for the online extractor is shown in Figure 4.6. It works by looking through Q_H for an accepting transcript for a commitment com that is also included in the proof π, but which contains a different challenge. Using the special soundness extractor of the underlying Σ-protocol a witness for x is extracted.

We refer to [45] for a proof that Ext is a valid online extractor. We do remark, however, that the same proof applies in our setting, even though we have altered the definition of the prover, as online extractability considers arbitrary provers and we have left the verifier unchanged.

4.2.3. The Fischlin Transformation in the VROM

We are now ready to adapt the Fischlin transform from ROM to VROM. First, observe how the random oracle is used in the Fischlin transform. As in the FDH case, the prover is required to query the random oracle on fresh inputs. Hence, these queries have to be replaced by Hash queries to $\mathcal{F}_{VRO}$. The verifier, on the other hand, merely has to reconstruct inputs that have previously been queried and has to ensure that the corresponding outputs have a special form. In our case, that they all consist of zeroes. These queries can be replaced by verification queries to $\mathcal{F}_{VRO}$.

To allow the verifier to make these queries we augment proofs to also include r proofs π_i, $i \in [r]$. Proofs thus have the form $(\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i, \pi_i)_{i=1,2,\dots,r}$. Notice that again we can get away with not including any hashes as all of them are required to be a fixed value known to the verifier. The verifier first checks the validity of all the π_i and then verifies the validity of all the $(\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i)$ using $\mathcal{V}_\Sigma$ as before.

The complete code for the new prover, verifier, and online extractor is shown in Figures 4.7, 4.8 and 4.9. Note that the verifier has become semi-interactive. Upon verifying the first proof it receives, the verifier has to execute a `Init` request to retrieve the verification key vk. All subsequent proofs only require verifications of $\mathcal{F}_{VRO}$-proofs and are therefore non-interactive.

Prover The prover receives as input $(x, w) \in \mathcal{R}_L$ and has access to an instance of $\mathcal{F}_{VRO}$. It executes the code shown in Figure 4.7. The changes with respect to the ROM version consist in replacing H-queries with `Hash` queries to $\mathcal{F}_{VRO}$ and including the proofs π_i alongside the transcripts in the final proof π.

Verifier The verifier receives as input a pair (x, π) where π is a purported proof for $x \in L$. The verifier also has access to an instance of $\mathcal{F}_{VRO}$. The code for the verifier is shown in Figure 4.8. The changes with respect to the ROM version consist of the verifier having to retrieve the verification key vk and replacing all checks of the form $H(m) = 0^b$ by verification queries to $\mathcal{F}_{VRO}$.

Online Extractor Our changes to the ROM extractor are similarly straightforward. Let $\mathcal{A}$ be the adversary and let (x, π) be its output. Let also Q be the queries it made to $\mathcal{F}_{VRO}$ as tuples (q_j, h_j, π_j).

Ext operates as follows. It parses π as $(\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i, \pi_i)_{i=1,2,\ldots,r}$. Then it runs Ext_Σ on input $(x, (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i)_{i=1,2,\ldots,r}, Q')$ where Q' is constructed from Q by removing the last component. It receives output w and outputs it. The complete code of Ext is shown in Figure 4.9.

4.2.4. Proof of Security

In this section we prove the security of the transformed scheme using $\mathcal{F}_{VRO}$. Again we fix the session of $\mathcal{F}_{VRO}$ by removing *sid* from all messages. We proceed via three lemmas stating the completeness, zero-knowledge property, and online extractability.

Lemma 4.2.12. *Let Σ be a Σ-protocol satisfying the requirements for the randomized Fischlin transform and let Σ' be the transformed protocol in the VROM. Then Σ' is complete in the VROM according to Definition 4.2.1.*

Proof. We show that completeness is preserved. Let $\mathcal{A}$ be the restricted adversary (see Section 4.2.1). It is easy to see that the prover in that case will produce a proof with exactly the same probability as the ROM prover. As the verifier will be able to retrieve the verification key and by the perfect completeness guaranteed by $\mathcal{F}_{VRO}$, it will always output 1 if it receives a proof. Thus, the transformed scheme has the same negligible completeness error as before. □

Lemma 4.2.13. *Let Σ be a Σ-protocol satisfying the requirements for the randomized Fischlin transform and let Σ' be the transformed protocol in the VROM. Then Σ' has the zero-knowledge property in the VROM according to Definition 4.2.2.*

Proof. We have to show the existence of a simulator $\mathcal{S}$ for our definition of zero-knowledge in the presence of $\mathcal{F}_{VRO}$ with domain $\{0, 1\}^*$ and codomain $\{0, 1\}^b$. Let $\mathcal{D}$ be a distinguisher and $\mathcal{P}$ the honest prover. We have to describe the following things:

$\mathcal{P}^{\mathcal{F}_{VRO}}(x, w)$

1: **for** $i \in \{1, 2, \ldots, r\}$ **do**
2: $\quad \mathsf{com}_i \leftarrow \mathcal{P}_\Sigma(x, w)$
3: **endfor**
4: $\vec{\mathsf{com}} = (\mathsf{com}_1, \mathsf{com}_2, \ldots, \mathsf{com}_r)$
5: **for** $i \in \{1, 2, \ldots, r\}$ **do**
6: $\quad \mathcal{E}_i = \emptyset$
7: $\quad$ **while** $|\mathcal{E}_i| < 2^t$ **do**
8: $\quad\quad \mathsf{ch}_i \leftarrow\$ \{0,1\}^t \setminus \mathcal{E}_i$
9: $\quad\quad \mathsf{resp}_i \leftarrow \mathcal{P}_\Sigma(x, w, \mathsf{com}_i, \mathsf{ch}_i)$
10: $\quad\quad q = (x, \vec{\mathsf{com}}, i, \mathsf{ch}_i, \mathsf{resp}_i)$
11: $\quad\quad (\texttt{Hash}, sid, q) \rightarrow \mathcal{F}_{VRO}$
12: $\quad\quad (\texttt{HashProof}, sid, q, h, \pi_i) \leftarrow \mathcal{F}_{VRO}$
13: $\quad\quad$ **if** $h = 0$ **do**
14: $\quad\quad\quad$ **go to next** i
15: $\quad\quad$ **fi**
16: $\quad\quad$ **if** $|\mathcal{E}_i| = 2^t - 1$ **do**
17: $\quad\quad\quad$ **return** $\bot$
18: $\quad\quad$ **fi**
19: $\quad$ **endwhile**
20: **endfor**
21: $\pi = (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i, \pi_i)_{i=1,2,\ldots,r}$
22: **return** π

Figure 4.7.: The Fischlin prover in the VROM.

$\mathcal{V}^{\mathcal{F}_{VRO}}(x, \pi)$

1: **if** $\mathsf{vk} = \bot$ **do**
2: $\quad (\texttt{Init}, sid) \rightarrow \mathcal{F}_{VRO}$
3: $\quad (\texttt{VerificationKey}, sid, \mathsf{vk}) \leftarrow \mathcal{F}_{VRO}$
4: **fi**
5: $(\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i)_{i=1,2,\ldots,r} = \texttt{parse}(\pi)$
6: $\vec{\mathsf{com}} = (\mathsf{com}_1, \mathsf{com}_2, \ldots, \mathsf{com}_r)$
7: **for** $i \in \{1, 2, \ldots, r\}$ **do**
8: $\quad q = (x, \vec{\mathsf{com}}, i, \mathsf{ch}_i, \mathsf{resp}_i)$
9: $\quad (\texttt{Verify}, sid, q, 0^b, \pi_i, \mathsf{vk}) \rightarrow \mathcal{F}_{VRO}$
10: $\quad (\texttt{Verified}, sid, q, 0^b, \pi_i, \mathsf{vk}, \sigma) \leftarrow \mathcal{F}_{VRO}$
11: $\quad \zeta = \mathcal{V}_\Sigma(x, \mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i)$
12: $\quad$ **if** $\zeta = 0 \lor \sigma = 0$ **do**
13: $\quad\quad$ **return** 0
14: $\quad$ **fi**
15: **endfor**
16: **return** 1

Figure 4.8.: The Fischlin verifier in the VROM.

$\mathsf{Ext}(x, \pi, Q)$

1: $(\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i, \pi_i)_{i=1,2,\ldots,r} = \texttt{parse}(\pi)$
2: $\pi' = (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i)_{i=1,2,\ldots,r}$
3: $Q_H = \{(q, h) \mid (q, h, \pi) \in Q\}$
4: $w = \mathsf{Ext}_{ROM}(x, \pi', Q_H)$
5: **return** w

Figure 4.9.: The Fischlin online extractor in the VROM.

- How $\mathcal{F}_{VRO}^{\mathcal{S}}$, including the adversary interface, is provided to $\mathcal{D}$.
- How, given x, the proof π is simulated.

$\mathcal{S}$ behaves as follows:

- $\mathcal{F}_{VRO}^{\mathcal{S}}$ is an honest emulation of $\mathcal{F}_{VRO}$. Let L be the list of `ver`-tuples. All hash and verification queries of $\mathcal{D}$ are then answered honestly with respect to L.
- To program $\mathcal{F}_{VRO}^{\mathcal{S}}$ on input q to return h, $\mathcal{S}$ aborts if there already exists some $(\mathtt{ver}, q, h', \pi', \mathsf{vk}, 1)$ in L, i.e. if q has been queried before. If not, it simulates a hash query for q under identity $\mathcal{P}$ (i.e. the real prover) and receives `SimInfo` s from $\mathcal{D}$. It computes a proof $\pi \leftarrow \mathsf{Prove}(q, h, s)$ and stores $(\mathtt{ver}, q, h, \pi, \mathsf{vk}, 1)$ in L.
- Upon receiving input a statement x:
  - $\mathcal{S}$ chooses r challenges ch_i uniformly at random.
  - It runs the special zero-knowledge simulator $\mathcal{S}_\Sigma$ on (x, ch_i), obtaining transcripts $(\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i)$, $i \in [r]$.
  - $\mathcal{S}$ constructs $\vec{\mathsf{com}} = (\mathsf{com}_1, \mathsf{com}_2, \ldots, \mathsf{com}_r)$ and attempts to program $\mathcal{F}_{VRO}^{\mathcal{S}}$ on all inputs $(x, \vec{\mathsf{com}}, i, \mathsf{ch}_i, \mathsf{resp}_i)$, $i \in [r]$. If any fail, $\mathcal{S}$ aborts. Else, $\mathcal{S}$ for each $i \in [r]$ simulates a correctly distributed number of hash queries under the identity $\mathcal{P}$ and of the correct length. It samples random elements from $\{0,1\}^b$ until the all-zero string is found the first time to determine this number. Note that L is not affected during this process, $\mathcal{S}$ does not determine (and could not know) the concrete inputs for which it is simulating queries.

The code of $\mathcal{S}$ is shown in Figure 4.10. Note that we have suppressed the ability for $\mathcal{D}$ to delay the delivery of messages.

Proving Indistinguishability We have to show that the above is an indistinguishable simulation. First, $\mathcal{S}$ has the same strategy of which inputs to program as the simulator in the ROM. Hence it has the same probability of returning $\perp$ when asked to simulate a proof. This probability is again negligible and we can condition on the fact that both the honest prover as well as $\mathcal{S}$ always produce a proof. Note that this assumes that $\mathcal{D}$ allows $\mathcal{S}/\mathcal{P}$ to complete all of its hash queries, but not doing so affects both $\mathcal{P}$ and $\mathcal{S}$ in the same manner.

We turn to the distribution of proofs $\pi = (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i, \pi_i)_{i=1,2,\ldots,r}$. The portions of proofs apart from the π_i are correctly distributed by the proof of the validity of the zero-knowledge simulator in the ROM. But the π_i are generated by Prove on inputs $q = (x, \vec{\mathsf{com}}, i, \mathsf{ch}_i, \mathsf{resp}_i)$, $h = 0^b$, and s obtained from $\mathcal{D}$ after sending it a `Hashing` message containing $\mathcal{P}$ and $\|q\|$. This is the same process that is used by $\mathcal{P}$. Thus, proofs are distributed correctly.

What remains to be shown is that $\mathcal{F}_{VRO}^{\mathcal{S}}$ is a valid simulation of an honest $\mathcal{F}_{VRO}$. There are two differences:

$\mathcal{F}_{VRO}^{\mathcal{S}}(\texttt{Init})$

1 : $L \leftarrow \emptyset$
2 : $\texttt{Init} \rightarrow \mathcal{D}$
3 : $(\mathsf{Key}, \mathsf{Prove}, \mathsf{vk}) \leftarrow \mathcal{D}$
4 : **return** $(\mathsf{Key}, \mathsf{vk})$

$\mathcal{F}_{VRO}^{\mathcal{S}}(\texttt{Hash}, q)$

1 : **if** $\nexists(\texttt{ver}, q, h', \pi', \mathsf{vk}, 1) \in L$ **do**
2 : $\quad h \leftarrow \{0,1\}^b$
3 : **else**
4 : $\quad h = h'$
5 : $\quad (\texttt{Hashing}, \mathcal{D}, \|q\|) \rightarrow \mathcal{D}$
6 : $\quad (\texttt{SimInfo}, \mathcal{D}, s) \leftarrow \mathcal{D}$
7 : **fi**
8 : $\pi = \mathsf{Prove}(q, h, \pi)$
9 : **if** $(\texttt{ver}, q, h, \pi, \mathsf{vk}, 0) \in L$ **do**
10 : $\quad$ **halt**
11 : **fi**
12 : $L = L \cup \{(\texttt{ver}, q, h, \pi, \mathsf{vk}, 1)\}$
13 : **return** $(\texttt{HashProof}, q, h, \pi)$

$\mathcal{F}_{VRO}^{\mathcal{S}}(\texttt{Verify}, q, h, \pi, \mathsf{vk})$

1 : **if** $\exists(\texttt{ver}, q, h, \pi, \mathsf{vk}, b) \in L$ **do**
2 : $\quad$ **return** $(\texttt{Verified}, q, h, \pi, \mathsf{vk}, b)$
3 : **fi**
4 : **if** $\exists(\texttt{ver}, q, h, \pi', \mathsf{vk}, 1) \in L$ **do**
5 : $\quad (\texttt{Verify}, q, h, \pi, \mathsf{vk}) \rightarrow \mathcal{D}$
6 : $\quad (\texttt{Verified}, q, h, \pi, \mathsf{vk}, c) \leftarrow \mathcal{D}$
7 : $\quad$ **return** $(\texttt{Verifed}, q, h, \pi, \mathsf{vk}, c)$
8 : **fi**
9 : **return** $(\texttt{Verified}, q, h, \pi, \mathsf{vk}, 0)$
10 :

$\mathcal{S}(x)$

1 : **for** $r \in \{1, 2, \ldots, r\}$ **do**
2 : $\quad \mathsf{ch}_i \leftarrow\!\!\$ \{0,1\}^t$
3 : $\quad (\mathsf{com}_i, \mathsf{resp}_i) \leftarrow \mathcal{S}_\Sigma(x, \mathsf{ch}_i)$
4 : **endfor**
5 : $\vec{\mathsf{com}} = (\mathsf{com}_1, \mathsf{com}_2, \ldots, \mathsf{com}_r)$
6 : **for** $i \in \{1, 2, \ldots, r\}$
7 : $\quad q = (x, \vec{\mathsf{com}}, i, \mathsf{ch}_i, \mathsf{resp}_i)$
8 : $\quad$ **if** $\exists(\texttt{ver}, q, h', \pi', \mathsf{vk}, 1) \in L$ **do**
9 : $\quad\quad$ **return** $\perp$
10 : $\quad$ **fi**
11 : $\quad (\texttt{Hashing}, \mathcal{P}, \|q\|) \rightarrow \mathcal{D}$
12 : $\quad (\texttt{SimInfo}, \mathcal{P}, s) \leftarrow \mathcal{D}$
13 : $\quad \pi_i \leftarrow \mathsf{Prove}(q, 0^b, s)$
14 : $\quad L = L \cup \{(\texttt{ver}, q, 0^b, \pi_i, \mathsf{vk}, 1)\}$
15 : **endfor**
16 : **for** $i \in \{1, 2, \ldots, r\}$ **do**
17 : $\quad n_i = 0$
18 : $\quad$ **while** *true* **do**
19 : $\quad\quad m \leftarrow\!\!\$ \{0,1\}^b$
20 : $\quad\quad$ **if** $m = 0^b$ **do**
21 : $\quad\quad\quad$ **break**
22 : $\quad\quad$ **fi**
23 : $\quad\quad n_i = n_i + 1$
24 : $\quad$ **endwhile**
25 : $\quad \mathsf{len} = \|(x, \vec{\mathsf{com}}, i, \mathsf{ch}_i, \mathsf{resp}_i)\|$
26 : $\quad$ **for** $i \in \{1, 2, \ldots, n_i\}$ **do**
27 : $\quad\quad (\texttt{Hashing}, \mathcal{P}, \mathsf{len}) \rightarrow \mathcal{D}$
28 : $\quad\quad (\texttt{SimInfo}, \mathcal{P}, s) \leftarrow \mathcal{D}$
29 : $\quad$ **endfor**
30 : **endfor**
31 : $\pi = (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i, \pi_i)_{i=1,2,\ldots,r}$
32 : **return** π

Figure 4.10.: The Fischlin zero-knowledge simulator in the VROM.

- By programming 0^b as the output for r inputs for each requested proof for some statement x, $\mathcal{S}$ increases the fraction of such hashes over the expected number.

- For each $i \in [r]$, $\mathcal{P}$ queries inputs of the form $(x, \vec{\mathsf{com}}, i, \mathsf{ch}, \mathsf{resp})$ with $(\mathsf{com}_i, \mathsf{ch}, \mathsf{resp})$ an accepting transcript and while randomly stepping through the challenges ch until the output 0^b occurs. While doing so it adds tuples of the form $(\mathtt{ver}, \ldots, 1)$ to L. $\mathcal{S}$ for each $i \in [r]$ only adds one such tuple to L. It then randomly samples elements from $\{0,1\}^b$ until 0^b occurs the first time and each time sends a notification containing the correct information to $\mathcal{D}$, but without adding to L.

Both of these differences already existed in the ROM case. As $\mathcal{S}$ does, the ROM simulator for each generated proof programmed the random oracle for r randomly chosen challenges to return 0^b. It did not fix the hash values of any other inputs until finding one with the correct hash, as the honest prover would have done. The notifications to $\mathcal{D}$ do not contain any information which is not already known to $\mathcal{D}$, i.e. the identity of the prover $\mathcal{P}$ and a constant length len. Hence, the same argument from Section 4.2.2 also applies here.

Thus, the zero-knowledge properly follows from the zero-knowledge property of the ROM version and the properties of $\mathcal{F}_{VRO}$. □

Lemma 4.2.14. *Let Σ be a Σ-protocol satisfying the requirements for the randomized Fischlin transform and let Σ' be the transformed protocol in the VROM. Then Σ' is online extractable according to Definition 4.2.4.*

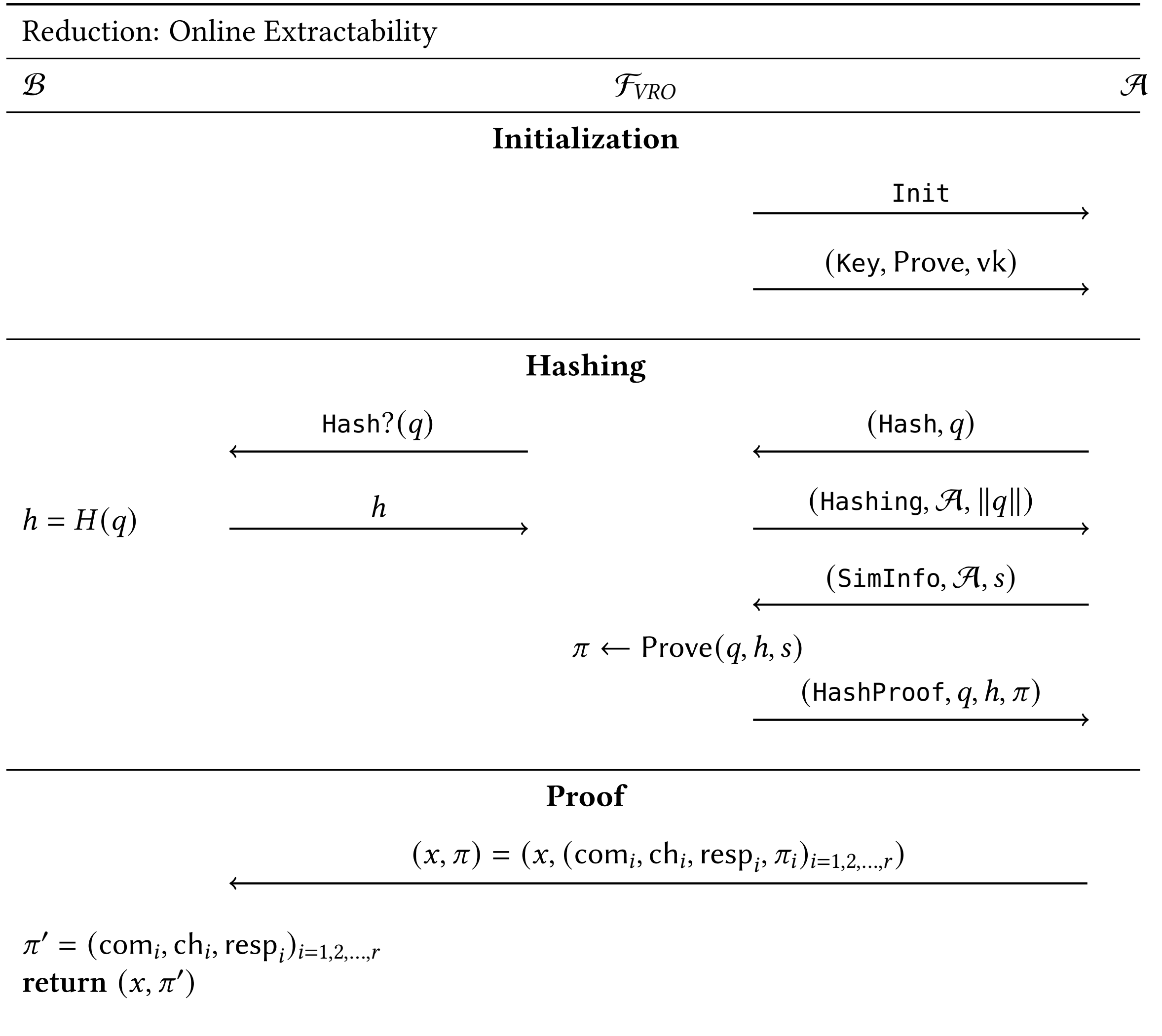
Proof. We claim that the extractor shown in Figure 4.9 is a valid extractor. Essentially we are saying that removing the proofs π_i with overwhelming probability produces a valid ROM proof with respect to the RO defined by the random function which $\mathcal{F}_{VRO}$ uses to assign hashes to inputs.

We proceed with a reduction. Let $\mathcal{A}$ be an adversary in the VROM. We construct an adversary $\mathcal{B}$ in the ROM. $\mathcal{B}$ provides $\mathcal{F}_{VRO}$ by using its random oracle H as the source of randomness. The interaction is shown in Protocol 4.3. Verification queries by $\mathcal{A}$ are not shown as they do not require any action by $\mathcal{B}$.

Let $(x, \pi) = (x, (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i)_{i=1,2,\ldots,r})$ be the output of $\mathcal{A}$ and assume that π is a valid proof for x with respect to $\mathcal{F}_{VRO}$ and verification key vk. This implies that $\mathcal{F}_{VRO}$ answers all messages $(\mathtt{Verify}, q_i, 0^b, \pi, \mathsf{vk})$ where $q_i = (x, \vec{\mathsf{com}}, i, \mathsf{ch}_i, \mathsf{resp}_i)$ with $(\mathtt{Verified}, q_i, 0^b, \pi, \mathsf{vk}, 1)$. Inspecting the code of $\mathcal{F}_{VRO}$ we see that this implies that 0^b is set as the hash for all the q_i. But by how the randomness of $\mathcal{F}_{VRO}$ is chosen this means that $H(q_i) = 0^b$ for all $i \in [r]$ and hence that the proof π' is a valid proof for x with respect to H. This shows that the probability of success of $\mathcal{A}$ is transferred to $\mathcal{B}$. □

4.2.5. Universally Composable Transferable Zero-Knowledge

In addition to the above, we want to investigate whether the Fischlin transform does realize an ideal *UC TZK* functionality $\mathcal{F}_{TZK}$ defined in Section 2.5.4 in either the $\mathcal{F}_{RO}$ or $\mathcal{F}_{VRO}$-hybrid model. Notice that this is a strictly stronger notion than what we have seen above. Online extractability is implied as the simulator $\mathcal{S}$ interacting with $\mathcal{F}_{TZK}$ is asked to extract witnesses from (x, π) alone. It is able to obtain the random oracle queries because



Protocol 4.3.: The reduction from online extractability in the ROM to online extractability in the VROM.

in the ideal world it is the entity providing (a simulated instance of) $\mathcal{F}_{RO}$ to the adversary. Our notion of zero-knowledge for NIZKs where the adversary is provided with an oracle producing proofs of true statements is also implied. $\mathcal{F}_{TZK}$, on input $(x, w) \in \mathcal{R}$, asks the simulator to provide a proof for x (without seeing w). Hence the simulator has to provide a simulated proof π for x which is returned to the environment. Again, due to being the entity providing $\mathcal{F}_{RO}$, the simulator is able to program. The environment can make multiple such queries and thus essentially has oracle access to proofs of *true statements*. In the real world, these will be honestly generated proofs while in the ideal world they are simulated. A successful distinguisher for the game-based definition of zero-knowledge would hence immediately yield a distinguishing environment that never verifies a proof.[8]

To show that the Fischlin transform produces UC TZKs, we have to prove that for any Σ-protocol $\Pi = (\mathcal{V}_\Sigma, \mathcal{P}_\Sigma)$ satisfying the prerequisites of the Fischlin transform, the transformed protocol $\Pi_{ROM} = (\mathcal{V}_{ROM}, \mathcal{P}_{ROM})$, when considered as a protocol π_{ROM} and in the multi-prover setting, π_{ROM} UC-realizes $\mathcal{F}_{TZK}$ in the $\mathcal{F}_{RO}$-hybrid model. Similarly, we want to show that $\Pi_{VROM} = (\mathcal{V}_{VROM}, \mathcal{P}_{VROM})$ with the VROM-adaptations and in the multi-prover setting defines a protocol π_{VROM} that UC-realizes $\mathcal{F}_{TZK}$ in the $\mathcal{F}_{VRO}$-hybrid model. As in the previous sections, all instances of $\mathcal{F}_{RO}$ and $\mathcal{F}_{VRO}$ are parametrized with domain $\{0, 1\}^*$ and codomain $\{0, 1\}^b$.

Remark 4.2.15. We note that we have defined the relaxation $\mathcal{F}_{TZK}$ of $\mathcal{F}_{NIZK}$ solely for the analysis of π_{VROM} as there the verifier will have to be able to retrieve a verification key from $\mathcal{F}_{VRO}$ and whether and when this succeeds is under the control of the adversary. Similarly, the delivery of proofs output by $\mathcal{F}_{VRO}$ are able to be delayed and as such we have to allow this for proofs produced by $\mathcal{F}_{TZK}$ as well. The protocol π_{ROM} does in fact UC-realize $\mathcal{F}_{NIZK}$ without any relaxations. We do not prove this separately, but it can be seen by inspecting proof for the $\mathcal{F}_{TZK}$ case.

The Protocols We first describe π_{ROM} and π_{VROM}. A party $\mathcal{P}$ running one of the protocols π_X, $X \in \{ROM, VROM\}$ and having a session identifier *sid* behaves as follows:

- On input $(\texttt{Init}, sid)$, if $X = ROM$ return $(\texttt{Key}, sid, \perp)$. Else, send $(\texttt{Init}, sid)$ to $\mathcal{F}_{VRO}$. Upon receiving a response $(\texttt{Key}, sid, \mathsf{vk})$, return it.
- On input $(\texttt{Prove}, sid, x, w)$, run $\pi \leftarrow \mathcal{P}_X(x, w)$ while providing access to the session with identifier *sid* of either $\mathcal{F}_{RO}$ or $\mathcal{F}_{VRO}$. Return $(\texttt{Proof}, sid, \pi)$.
- On input $(\texttt{Verify}, sid, x, \pi, \mathsf{vk})$, if $X = ROM$ and $\mathsf{vk} = \perp$ or $X = VROM$, run $\mathcal{V}_X(x, \pi)$ while providing access to the session with identifier *sid* of either $\mathcal{F}_{RO}$ or $\mathcal{F}_{VRO}$ as well as vk. Upon $\mathcal{V}_X$ outputting a bit b, return $(\texttt{Verification}, sid, x, \pi, \mathsf{vk}, b)$. If $X = ROM$ and $\mathsf{vk} \neq \perp$, return $(\texttt{Verification}, sid, x, \pi, \mathsf{vk}, 0)$.

In short, all parties have access to either $\mathcal{F}_{RO}$ or $\mathcal{F}_{VRO}$ and generate and verify proofs with respect to this shared functionality. Due to $\mathcal{F}_{RO}$ not having any verification keys, some simplifications were possible such as being able to reject any verification attempts for wrong keys.

[8]Note that this holds for every simulator and simulators for distinguishers and environments which only request proofs are in direct correspondence. Only then the claimed implication holds.

4.2.5.1. In the ROM

We begin in the ROM. First, we give a simulator $\mathcal{S}_{ROM}$ for the dummy adversary $\mathcal{D}$. Then we prove that $\mathcal{S}_{RO}$ is a valid simulator. $\mathcal{S}_{RO}$ makes use of the Fischlin simulator $\mathcal{S}_F$ described in Section 4.2.2, augmented by having it output a list $\mathcal{L}_{prog}$ of pairs (q, h) which describe how $\mathcal{S}_F$ wishes to program the random oracle for its current input $x \in L$. Let Ext be the online extractor in the ROM. For a list of pairs $\mathcal{T}$, let $\mathcal{T}_{fst}$ be the list of first components.

$\mathcal{S}_{ROM}$ behaves as follows:

- $\mathcal{S}_{ROM}$ simulates an instance of $\mathcal{D}$ and provides an instance of $\mathcal{F}_{RO}$ to it. $\mathcal{S}_{RO}$ keeps a list $\mathcal{L}$ of pairs (q, h) to answer random oracle queries.
- Upon receiving a message $(\texttt{Init}, sid)$ from $\mathcal{F}_{TZK}$, $\mathcal{S}_{ROM}$ responds with a message $(\texttt{Init}, sid, \bot)$.
- Upon receiving a message $(\texttt{Prove}, sid, \mathcal{P}, x)$ from $\mathcal{F}_{TZK}$, $\mathcal{S}_{ROM}$ simulates a proof π for x by first running $\mathcal{S}_F(x)$, obtaining $(\pi, \mathcal{Q})$. If $\mathcal{L}_{fst} \cap \mathcal{Q}_{fst}$ is non-empty, then $\mathcal{S}_{RO}$ outputs `abort` and halts. Otherwise it sets $\mathcal{L} = \mathcal{L} \cup \mathcal{Q}$ to add the freshly programmed entries. $\mathcal{S}_{RO}$ then sends a message $(\texttt{Proof}, sid, \mathcal{P}, \pi)$ to $\mathcal{F}_{TZK}$.
- Upon receiving a message $(\texttt{Verify}, sid, \mathcal{V}, x, \pi)$ from $\mathcal{F}_{NIZK}$, $\mathcal{S}_{ROM}$ checks whether π is a valid proof for x with respect to $\mathcal{F}_{RO}$ (and by adding pairs to $\mathcal{L}$ where this involves queries for inputs which have not been queried before) by running $\mathcal{V}_{RO}(x, \pi)$ with access to the RO represented by $\mathcal{L}$. If π is not a valid proof for x, $\mathcal{S}_{RO}$ sends $(\texttt{Witness}, sid, \bot)$ back to $\mathcal{F}_{NIZK}$. Else, $\mathcal{S}_{RO}$ lets $\mathcal{T} \subset \mathcal{L}$ be those queries in $\mathcal{L}$ which are prefixed by x. It then runs Ext on input $(x, \pi, \mathcal{T})$. Let w be the obtained output. If $w = \bot$, $\mathcal{S}_{RO}$ outputs `fail` and halts. Else, $\mathcal{S}_{RO}$ sends the message $(\texttt{Witness}, sid, w)$ to $\mathcal{F}_{NIZK}$.
- Upon receiving a message $(\texttt{WrongKey}, sid, \mathcal{V}, x, \pi, \mathsf{vk}')$ from $\mathcal{F}_{TZK}$, return $(\texttt{WrongKey}, sid, \mathcal{V}, x, \pi, \mathsf{vk}', 0)$.

There is no direct communication between parties and $\mathcal{D}$ has no adversarial influence on $\mathcal{F}_{RO}$ so there is nothing further for $\mathcal{S}_{ROM}$ to do than simulating $\mathcal{F}_{RO}$ and answer messages by $\mathcal{F}_{TZK}$.

Having defined $\mathcal{S}_{ROM}$, we now have to show that

- $\mathcal{S}_{ROM}$ attempts to extract witnesses for the correct set of `Verify` messages by $\mathcal{F}_{NIZK}$.
- `abort` happens with negligible probability. This event occurring implies that $\mathcal{S}_{ROM}$ was unable to program $\mathcal{F}_{RO}$ using the list returned by $\mathcal{S}_F$. We could lower its probability by re-running $\mathcal{S}_F$, but as a single run is already sufficient to achieve a negligible probability of aborting, we have chosen to keep the description simple.
- `fail` happens with negligible probability. In this event, $\mathcal{S}_{ROM}$ was unable to extract a witness from a valid statement/proof pair (x, π) where valid means that in the real interaction, the party trying to verify π would accept. $\mathcal{F}_{NIZK}$, on the other hand, will reject the proof without receiving a valid witness w.

- the proofs produced by $\mathcal{S}_{ROM}$ are computationally indistinguishable from those generated in the real interaction. Note that this is not immediately implied by the zero-knowledge property for the Fischlin transform which we have proven above as here the "adversary" has in addition access to an extraction oracle.

Remark 4.2.16. It may seem like $\mathcal{S}_{ROM}$ should not halt in the event of `fail` occurring as x may not be in the language and hence there does not exist a valid witness which could be extracted from π. But still, in the real interaction π would be accepted while $\mathcal{F}_{NIZK}$ rejects any proof for a $x \notin L$. As such, this event must occur only with negligible probability and $\mathcal{S}_{RO}$ is allowed to halt.

For the first point, we observe that the goal of $\mathcal{S}_{ROM}$ is to emulate a real protocol execution. As such, the initial check upon receiving some (x, π) that π is a valid proof for x with respect to the simulated random oracle is equivalent to letting the simulated instance of the party $\mathcal{V}$ making the verification query execute the real verification protocol on input $(\texttt{Verify}, sid, x, \pi)$. Only if $\mathcal{V}$ outputs $(\texttt{Verification}, sid, x, \pi, 1)$ does $\mathcal{S}_{RO}$ have to come up with a witness w with $(x, w) \in \mathcal{R}$ to make $\mathcal{F}_{NIZK}$ generate the same output. If instead $(\texttt{Verification}, sid, x, \pi, 0)$ is output by the simulated $\mathcal{V}$, $\mathcal{F}_{NIZK}$ will output the same message iff $\mathcal{S}_{RO}$ outputs an invalid witness. It achieves this by setting $w = \perp$ (which we assume to not be a valid witness for any x). Note that this discussion is independent of whether $x \in L$.

To show that no environment can tell a difference between the world where all honest parties execute the protocol π_{ROM} and generate and verify proofs according to it, and the world where proofs are simulated and verification involves extraction, we proceed via a series of hybrid interactions. We start from the real interaction involving $\mathcal{Z}$, $\mathcal{D}$ and a session of π_{ROM}. After each change, we argue that the distribution of outputs of $\mathcal{Z}$ in the new interaction is at least computationally indistinguishable from the previous interaction.

The first change we make is merely conceptual. We introduce a new entity C called the challenger. C will be the entity executing the different interactions and changes made to it. In the beginning, C will simply be executing the honest protocol on behalf of the honest parties and also provide an honest simulation of $\mathcal{F}_{RO}$ to $\mathcal{Z}$ and $\mathcal{D}$. Gradually, C will behave in ways in which no honest party could on its own behave. This does not matter as long as $\mathcal{Z}$ is unable to observe the changes based on the information known to it. After enough steps, C will behave in a way that is essentially identical to the ideal interaction. At that point, all that will be left to do is restructure C and split it into $\mathcal{F}_{TZK}$ and $\mathcal{S}_{RO}$, so that formally the ideal interaction has been reached.

Let $\text{REAL}^{\mathcal{F}_{RO}}_{\pi_{ROM},\mathcal{Z},\mathcal{D}}$ denote the ensemble of random variables in the real interaction where $\mathcal{Z}$ interacts with $\mathcal{D}$ and a session of π_{ROM} in the $\mathcal{F}_{RO}$-hybrid model. For some $i \geq 1$, let $\text{INT}^{\mathcal{F}_{RO}}_{i,C,\mathcal{Z},\mathcal{D}}$ denote the ensemble of random variables in interaction i between the challenger C, $\mathcal{Z}$ and $\mathcal{D}$. Third, let $\text{IDEAL}^{\mathcal{F}_{RO}}_{\mathcal{F}_{TZK},\mathcal{Z},\mathcal{S}_{ROM}}$ denote the ensemble of random variables in the ideal interaction involving $\mathcal{Z}$, $\mathcal{F}_{TZK}$ and $\mathcal{S}_{ROM}$.

Interaction 1: C executes an instance of $\mathcal{Z}$ (with the appropriate auxiliary input) and $\mathcal{D}$. It executes π_{ROM} honestly on behalf of all honest parties which are activated by

$\mathcal{Z}$ over the course of the interaction. The ideal random oracle $\mathcal{F}_{RO}$ is also simulated honestly.

Lemma 4.2.17. *The distribution of outputs of $\mathcal{Z}$ is identically distributed in the real interaction and **Interaction 1**, formally*

$$\mathrm{REAL}^{\mathcal{F}_{RO}}_{\pi_{ROM},\mathcal{Z},\mathcal{D}} = \mathrm{INT}^{\mathcal{F}_{RO}}_{1,C,\mathcal{Z},\mathcal{D}}$$

Proof. As we have already observed the introduction of C is merely syntactical and does not impact the view by $\mathcal{Z}$ of the interaction in any way. This proves the lemma. □

In the next step, we replace the way C lets honest parties generate proofs. Instead of running the honest prover, it uses the simulator. As the simulator expects to be able to program $\mathcal{F}_{RO}$, C also deviates from its honest emulation of $\mathcal{F}_{RO}$. Intuitively, this step corresponds to using our notion of zero-knowledge for NIZKPoKs where the distinguisher is able to adaptively request proofs for up to polynomially many inputs (x, w) while having access to the random oracle with respect to which the proofs are generated. Also, note that both C and the Fischlin simulator are able to answer random oracle and proof requests with respect to a shared state.

Interaction 2: C behaves as in **Interaction 1**, except in the following cases. Whenever $\mathcal{Z}$ asks an honest party to produce a proof for some input (x, w), $\mathcal{S}$ runs the Fischlin simulator $\mathcal{S}_F$, thereby obtaining a proof π. C also incorporates how $\mathcal{S}_F$ wishes to program the random oracle into its simulation of $\mathcal{F}_{RO}$. Whenever $\mathcal{S}_F$ tries to program $\mathcal{F}_{RO}$ on some input which is already set, C outputs `abort` and halts.

Lemma 4.2.18. *The distribution of outputs of $\mathcal{Z}$ in **Interaction 1** is computationally indistinguishable from the distribution in **Interaction 2**, formally*

$$\mathrm{INT}^{\mathcal{F}_{RO}}_{1,C,\mathcal{Z},\mathcal{D}} \stackrel{c}{\approx} \mathrm{INT}^{\mathcal{F}_{RO}}_{2,C,\mathcal{Z},\mathcal{D}}$$

Proof. We reduce to the zero-knowledge property in the ROM of Π_{ROM}. From any successful distinguisher $\mathcal{D}^*$ for the output distributions of $\mathcal{Z}$ in **Interaction 1** and **Interaction 2** we construct a distinguisher $\mathcal{D}'$ for the zero-knowledge property of Π_{ROM}. $\mathcal{D}'$ has access to either an oracle outputting honest proofs and an honest random oracle, or an oracle for simulated proofs and a simulated random oracle. It emulates C as in **Interaction 1** (say), except that it implements $\mathcal{F}_{RO}$ using its external random oracle and answers proof generation requests made to honest parties using proofs from its second oracle. In particular, proofs are verified using the honest verification algorithm from Π_{ROM} and with respect to whichever random oracle is provided. Once $\mathcal{Z}$ generates some output, $\mathcal{D}'$ gives it as input to $\mathcal{D}^*$ and outputs whatever $\mathcal{D}^*$ outputs.

When $\mathcal{D}'$ is interacting with honest oracles, **Interaction 1** is played. If, on the other hand, simulated oracles are supplied, **Interaction 2** is played. In each case the emulation

is perfect. By the assumed success of $\mathcal{D}^*$ in distinguishing outputs of $\mathcal{Z}$ and by the perfect emulation, this non-negligible probability of success perfectly transfers onto $\mathcal{D}'$. This contradicts the zero-knowledge property of Π_{ROM} which concludes the proof. □

We are now in a world where all proofs by honest parties have been replaced by simulated proofs, but verifications are still done as in the real interaction. This will now change. In the next interaction, honest verifiers will accept simulated proofs as valid but will behave differently when they receive a fresh proof. Assuming that such a proof verifies with respect to the honest verification algorithm of Π_{ROM}, C will run the Fischlin online extractor on all the queries made to $\mathcal{F}_{RO}$ which the environment has made through the dummy adversary. The answer to the verification request will then depend on whether a valid witness was able to be extracted.

Another thing to note is that in **Interaction 1**, the generation of proofs using the honest prover had a negligible completeness error, i.e. sometimes the proof returned even by an honest party would not verify using the honest verification algorithm. In **Interaction 2**, on the other hand, a proof which is output will always verify with respect to the current (programmed) state of $\mathcal{F}_{VRO}$and using the honest verification algorithm. This is, however, exchanged for a negligibly small change of C outputting `abort` and halting.

Interaction 3: C behaves as in **Interaction 2**, except in the following cases. Whenever an honest party, on input a pair (x, w) such that $\mathcal{R}(x, w) = 1$, outputs a (simulated) proof π, C stores the tuple (x, π). Now, whenever $\mathcal{Z}$ asks an honest party to verify some proof (x, π), C behaves as follows. If π does not verify with respect to x and π, the proof is rejected. Otherwise, C runs the online extractor Ext on (x, π) as well as the set Q of all queries made to $\mathcal{F}_{RO}$ so far. Let w be the result of this computation. If $(x, w) \notin \mathcal{R}$, π is rejected (even though the proof was valid!). Else, if $(x, w) \in \mathcal{R}$, (x, π) is added to the stored set of valid proofs and π is accepted.

Lemma 4.2.19. *The distribution of outputs of $\mathcal{Z}$ in **Interaction 2** is computationally indistinguishable from the distribution in **Interaction 3**, formally*

$$\mathrm{INT}^{\mathcal{F}_{RO}}_{2,C,\mathcal{Z},\mathcal{D}} \overset{c}{\approx} \mathrm{INT}^{\mathcal{F}_{RO}}_{3,C,\mathcal{Z},\mathcal{D}}$$

Proof. The fact that proofs generated by honest parties are immediately accepted is indistinguishable due to the fact that simulated proofs always verify (conditioned on no `abort` occurring). What remains to be shown is essentially that the online extractor works even against an adversary which is able to obtain simulated proofs for true statements, i.e. that it is *simulation-sound*. Plain simulation-soundness is formally not quite enough as the environment may try and verify multiple proofs for which extractions have to succeed. As we will see, this is not a problem due to the fact that extraction does not use rewinding.

We first argue that multiple extractions do not help the environment by using the fact that all inputs for the extractor are already known to the environment. Formally, let $\mathcal{A}$ be an adversary on the simulation-sound extractability of Π_{ROM} and let H be a random oracle. $\mathcal{A}$ is allowed to freely interleave queries of the form $(x, w) \in \mathcal{R}$, which are answered by

simulated proofs using $\mathcal{S}_F$, and queries of the form (x, π) where π was not among the proofs simulated for x. Queries of the latter form and where π is a valid proof for x lead to Ext being run on (x, π), and the set Q of all random oracle queries so far. If w with $(x, w) \in \mathcal{R}$ is obtained, the game continues. Otherwise, $\mathcal{A}$ wins. $\mathcal{A}$ loses once it halts and has not won already. We construct another adversary $\mathcal{B}$ on the simulation-sound extractability which is only allowed one query of the form (x, π) for a fresh π and which wins if π is valid and no witness for x can be extracted.
$\mathcal{B}$ behaves as follows:

- Queries to H made by $\mathcal{A}$ are answered by relaying them to $\mathcal{B}$'s own random oracle.
- Simulation queries (x, w) are relayed to the simulation oracle of $\mathcal{B}$ and the resulting proof π is returned to $\mathcal{A}$.
- Upon an extraction query (x, π) by $\mathcal{A}$ and if π is a valid proof for x, $\mathcal{B}$ computes $w \leftarrow \mathsf{Ext}(x, \pi, Q)$ where Q is the set of previous queries to H made by $\mathcal{A}$. If $w = \bot$, $\mathcal{B}$ outputs (x, π) to its challenger and halts. Otherwise, the simulation continues.

First, as long as $\mathcal{B}$ does not halt, the simulation of the environment expected by $\mathcal{A}$ is perfect. Second, whenever $\mathcal{B}$ generates output it wins. This shows that the probability of $\mathcal{B}$ to win is at least as large as that of $\mathcal{A}$. □

We have thus reduced simulation-sound extractability with multiple extraction queries to simulation-sound extractability with a single extraction query. The remaining two steps in the proof consist in proving that Π_{ROM} is simulation-sound with a single extraction query and reducing the computational indistinguishability of **Interaction 2** and **Interaction 3** to the simulation-soundness with a multiple extraction queries of Π_{ROM}.

First, we prove the former. For this we let $\mathcal{A}$ be an adversary on the simulation-sound extractability of Π_{ROM}. Let (x, π) be the output of $\mathcal{A}$ after interacting with an oracle which on input $(x, w) \in \mathcal{R}$ replies with $\pi \leftarrow \mathcal{S}_F$ and with a random oracle $H_{\mathcal{S}}$ which is also controlled by the simulator. Let further Q be the set of queries to $H_{\mathcal{S}}$ made by $\mathcal{A}$ during its runtime.

Assume that $\mathcal{A}$ wins, i.e. that π is a valid proof for x with respect to the final state of $H_{\mathcal{S}}$. We show that either π has previously been output by the simulation oracle on input (x, w) for some w or else that $w' \leftarrow \mathsf{Ext}(x, \pi, Q)$ such that $(x, w') \in \mathcal{R}$. We assume that not the former has occurred and show that the latter holds. There are three different cases to consider:

1. x is not in the language defined by $\mathcal{R}$.
2. x is a fresh statement for which no simulated proofs have been requested.
3. x was among the statements for which $\mathcal{A}$ previously requested simulated proofs.

Only the first of these cases immediately leads to $\mathcal{A}$ winning the game as in that case Ext must fail. For the remaining two cases, Ext has to be unable to extract a witness with non-negligible probability. The strategy from here on is then to show that the first case

occurs with negligible probability and the other two allow successful extraction, except with negligible probability.

We will start with the first one as it is the easiest. If $\mathcal{A}$ is able to produce a valid proof for an x not in the language, then this means $\mathcal{A}$ has broken (ordinary) simulation-soundness where no failed extraction is required for the adversary to win.[9] Even though in this game the simulator programs the random oracle, we show that this does not help $\mathcal{A}$. The idea is to simply replace simulated proofs and the simulated random oracle with their honest counterparts. This immediately reduces to the zero-knowledge property. In the new environment, $\mathcal{A}$ submits pairs of the form $(x, w) \in \mathcal{R}$ to an oracle that runs the honest prover and thus does not have to program the random oracle. This $\mathcal{A}$ could just have done by itself, i.e. $\mathcal{A}$ is really playing the ordinary soundness game where it has to produce a valid proof for a false statement without any additional help. As we know, ordinary soundness is implied by (ordinary) online extractability.

For the second case, we first argue that any previous requests for simulated proofs are of no use to $\mathcal{A}$. Primarily, this follows from the fact that all inputs checked by the verifier for hashing to 0^b are prefixed by the statement for which the proof was made. Therefore, all the programming done by the simulator can not have helped $\mathcal{A}$ in producing a valid proof.

This leaves us with the third case. Unlike in the previous case, we can not argue that past simulated proofs can not help $\mathcal{A}$. We can do so, however, for all the simulated proofs for statements $x' \neq x$. Let $\pi_1, \pi_2, \ldots, \pi_n$ be the previously simulated proofs for x with proof $\pi_i = (\mathsf{com}_{i,j}, \mathsf{ch}_{i,j}, \mathsf{resp}_{i,j})_{j=1,2,\ldots,r}$. Also let $\pi = (\mathsf{com}_j, \mathsf{ch}_j, \mathsf{resp}_j)_{j=1,2,\ldots,r}$ be the proof by $\mathcal{A}$. By the requirement that $\mathcal{A}$ has to produce a fresh proof, all the transcripts in the π_i differ from all the transcripts in π in at least one component[10]. First, if some π_j contains a transcript that differs from a transcript in π only in the challenge and/or the response, then this allows the extractor to extract by the strong special soundness property. We may thus assume that the vector of commitments $\vec{\mathsf{com}}$ differs from the commitment vector of all the π_i. As this vector is included in all the random oracle queries this shows that the previous proofs for x can not improve $\mathcal{A}$'s probability of success in this case and we are back to the second case which we have already analyzed.

We now are in the position where we have shown the simulation-sound online extractability under multiple extractions of the randomized Fischlin transform and may now use it to show that **Interaction 2** and **Interaction 3** are indistinguishable. From a distinguishing environment $\mathcal{Z}$, we construct a successful adversary $\mathcal{A}$. $\mathcal{A}$ simulates an instance of $\mathcal{Z}$ and uses its oracles to generate proofs and simulate $\mathcal{F}_{RO}$. Whenever $\mathcal{Z}$ makes a verification query for some (x, π) where π is a valid proof for x and π is different from all previously obtained proofs for x, $\mathcal{A}$ does the following. It collects all the random oracle queries Q which $\mathcal{Z}$ made so far and computes $w \leftarrow \mathsf{Ext}(x, \pi, Q)$. If $(x, w) \in \mathcal{R}$, $\mathcal{A}$ continues the simulation. In particular, it responds to the verification query for (x, π) by accepting. Otherwise, $\mathcal{A}$ outputs (x, π) to its challenger and halts.

[9] And where only proofs for true statements can be requested.

[10] For this to work it has to be ensured that transcripts contained in a proof can not be reordered while still yielding a valid proof, but this holds due to including the index of each transcript in the random oracle queries.

Now, as long as the simulation continues, the view of $\mathcal{Z}$ is exactly as in **Interaction 2**. It is also clear that $\mathcal{A}$ wins whenever it outputs some (x, π) as it runs the extractor on the same queries as later the challenger. Thus, the probability of the views of $\mathcal{Z}$ differing between **Interaction 2** and **Interaction 3** is lower than the advantage of $\mathcal{A}$ by the difference lemma. As we have assumed that $\mathcal{Z}$ is a distinguishing environment, but have also assumed that no $\mathcal{A}$ can be successful in attacking the simulation-sound online extractability, this is a contradiction. □

What remains to be done is to observe the remaining differences between **Interaction 3** and the ideal interaction. We make the following claim.

Lemma 4.2.20. *The distribution of outputs of $\mathcal{Z}$ is identically distributed in **Interaction 3** and the real interaction, formally*

$$\mathrm{INT}^{\mathcal{F}_{RO}}_{3,C,\mathcal{Z},\mathcal{D}} = \mathrm{REAL}^{\mathcal{F}_{RO}}_{\pi_{ROM},\mathcal{Z},\mathcal{D}}$$

In **Interaction 3**, proofs by honest parties are simulated using $\mathcal{S}_F$ and $\mathcal{F}_{RO}$ is programmed accordingly by C. This is exactly the process for generating proofs which is executed jointly by $\mathcal{F}_{TZK}$ and $\mathcal{S}_{ROM}$ in the ideal interaction. Similarly, verification of proofs in **Interaction 3** is as in the ideal interaction. First, all proofs which were previously generated by an honest party are immediately accepted. All other proofs are first checked for correctness using the honest verifier and the current state of $\mathcal{F}_{RO}$. If this check succeeds, a witness is tried to be extracted by running the same extraction algorithm on the same set of queries to $\mathcal{F}_{RO}$.

Note that during the whole proof we were able to ignore delayed delivery of proofs, the initialization task, as well as wrong verification keys being supplied to verify a proof as the reaction to these did not change throughout the different interactions. □

Combining the Lemmas 4.2.17 to 4.2.20, we have proven the following theorem.

Theorem 4.2.21. *The protocol π_{ROM} UC-realizes $\mathcal{F}_{TZK}$ in the $\mathcal{F}_{RO}$-hybrid model.*

4.2.6. In the VROM

Turning to the VROM setting, it seems like the same proof does not go through in the $\mathcal{F}_{VRO}$-hybrid model and with π_{VROM} as the protocol. First, not even ordinary simulation-soundness holds, because a cheating prover can simply alter a simulated proof $(\mathrm{com}_i, \mathrm{ch}_i, \mathrm{resp}_i, \pi_i)_{i=1,2,\ldots,r}$ by modifying one of the π_i to be valid for the same hash. Recall that this is explicitly allowed to the adversary by $\mathcal{F}_{VRO}$. One possible solution to this issue is requiring *unique proofs*. In this way, the non-malleability of proofs would be restored. In the following, we first show that adding this assumption does indeed allow π_{VROM} to UC-realize $\mathcal{F}_{TZK}$. Then in Section 4.2.7 we will give a more efficient alternative that works with the original $\mathcal{F}_{VRO}$, but involves a small change to the Fischlin transform itself.

Unique Proofs We thus proceed under the assumption that $\mathcal{F}_{VRO}$ proofs are unique, i.e during any session of $\mathcal{F}_{VRO}$ and for any pair (q, h) there is at most one π such that for the correct verification key vk the verification query on (q, h, π, vk) returns 1. This is formally enforced by changing the verification code of $\mathcal{F}_{VRO}$ such that the bit b which is currently

supplied by the adversary is always set to $b = 0$ whenever the verification key contained in the request is correct. For different keys $\mathsf{vk}' \neq \mathsf{vk}$ the adversary is still in full control, subject to consistency requirements.

As in the ROM, we give a simulator $\mathcal{S}_{VROM}$ for the dummy adversary $\mathcal{D}$. Then we prove that $\mathcal{S}_{VROM}$ is a valid simulator. $\mathcal{S}_{VROM}$ makes use of the Fischlin simulator $\mathcal{S}_F$, this time in the VROM, which we have described in Section 4.2.4. As $\mathcal{S}_F$ expects to be in control of $\mathcal{F}_{VRO}$, instead of letting $\mathcal{S}_F$ output a list to signal how it wishes to program $\mathcal{F}_{VRO}$, we give it the following abilities.

We allow it to

1. check whether some input q already has some hash h associated and if not to program $\mathcal{F}_{VRO}$ at input q.
2. create *faux* queries, i.e. queries which only consist in $\mathcal{F}_{VRO}$ sending `Hashing` messages to the adversary and waiting for the corresponding `SimInfo` responses, but which do not lead to any hash being set for any input.

All real or *faux* queries by $\mathcal{S}_F$ will be in the name of some party $\mathcal{P}$ which we give as an additional input. Should $\mathcal{S}_F$ not be able to simulate a proof we let it output `abort`.
$\mathcal{S}_{VROM}$ behaves as follows:

- $\mathcal{S}_{VROM}$ simulates an instance of $\mathcal{D}$ and provides an instance of $\mathcal{F}_{VRO}$ to it. As noted above, the simulator $\mathcal{S}_F$ is given partial control over this simulated instance of $\mathcal{F}_{VRO}$.
- Upon receiving a message $(\texttt{Init}, sid)$ from $\mathcal{F}_{TZK}$, $\mathcal{S}_{VROM}$ lets $\mathcal{F}_{VRO}$ send the same message to the adversary, asking it to provide a response $(\texttt{Init}, sid, \mathsf{Prove}, \mathsf{vk})$. Note that at this point $\mathcal{S}_{VROM}$ does not know the identity of the party which sent the initialization message to $\mathcal{F}_{TZK}$. $\mathcal{S}_{VROM}$ then sends $(\texttt{Init}, sid, \mathsf{vk})$ to $\mathcal{F}_{TZK}$. $\mathcal{S}_{VROM}$ is then asked by $\mathcal{F}_{TZK}$ to deliver the response to the initialization request while learning the identity $\mathcal{P}$ of the requesting party. It lets $\mathcal{F}_{VRO}$ simulate a delayed delivery for the message $(\texttt{Key}, sid, \mathsf{vk})$ and allows $\mathcal{F}_{TZK}$ to deliver it whenever the adversary does so.
- Upon receiving a message $(\texttt{Prove}, sid, \mathcal{P}, x)$ from $\mathcal{F}_{TZK}$, $\mathcal{S}_{VROM}$ simulates a proof by running $\mathcal{S}_F$ on input $(x, \mathcal{P})$. Once $\mathcal{S}_F$ creates output $\pi = (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i, \pi_i)$, this occurs once the adversary has delivered the answers to all the (real and *faux*) proof queries by $\mathcal{S}_F$ to $\mathcal{F}_{VRO}$, $\mathcal{S}_{VROM}$ sends $(\texttt{Proof}, sid, \mathcal{P}, x, \pi)$ to $\mathcal{F}_{TZK}$. If instead $\mathcal{S}_F$ has output `abort`, $\mathcal{S}_{VROM}$ outputs `abort` and halts. When asked by $\mathcal{F}_{TZK}$ to deliver π, it does so immediately.
- Upon receiving a message $(\texttt{Verify}, sid, \mathcal{V}, x, \pi)$ from $\mathcal{F}_{TZK}$, $\mathcal{S}_{VROM}$ checks whether π is a valid proof for x with respect to the current state of $\mathcal{F}_{VRO}$ and vk (as in the description of $\mathcal{F}_{TZK}$, we assume that vk has been set). If it is not, reply with $(\texttt{Verification}, sid, \mathcal{V}, x, \pi, 0)$. Else, run the VROM extractor Ext on x, π and the set Q of all hash queries made to $\mathcal{F}_{VRO}$. Let w be the output of Ext. $\mathcal{S}_{VROM}$ sends $(\texttt{Witness}, sid, w)$ to $\mathcal{F}_{TZK}$.

- Upon receiving a message (WrongKey, $sid, \mathcal{V}, x, \pi, \mathsf{vk}'$) from $\mathcal{F}_{TZK}$, return (WrongKey, $sid, \mathcal{V}, x, \pi, \mathsf{vk}', b$) where b is the result of running the honest verifier on (x, π), with verification key vk' and with access to $\mathcal{F}_{VRO}$.

To shorten the proof we only mention the differences to the proof for π_{ROM} here.

Interaction with the Adversary In π_{ROM}, initialization requests were answered immediately. With π_{VROM}, however, the initialization involves retrieval of the verification key vk from $\mathcal{F}_{VRO}$ and this retrieval can be delayed by the adversary. The simulator's simulation is perfect in this case, $\mathcal{S}_{VROM}$ allows $\mathcal{F}_{TZK}$ to deliver the key only once the simulated party has successfully executed what looks to the adversary like the honest initialization algorithm in π_{VROM}.

In π_{ROM}, both proof generation as well as verification were totally independent of the adversary. As such, $\mathcal{S}_{ROM}$ was able to immediately deliver proofs as well as provide answers to WrongKey messages without having to consult the adversary. In π_{VROM}, generation of proofs involves a (random) number of hash queries to $\mathcal{F}_{VRO}$ which involve the adversary and verification of proofs does as well (in our current setting only when the verification key contained in the request is incorrect). We have previously shown that $\mathcal{S}_F$ in the VROM simulates a correctly distributed number of hash queries. Again, $\mathcal{S}_{VROM}$ only allows $\mathcal{F}_{TZK}$ to deliver a proof π once the receiving party has successfully output a proof in the simulation.

Proof. The general structure of the proof is then identical to the proof for π_{ROM}. First, honestly generated proofs are replaced by simulated proofs. Again this can be reduced to the zero-knowledge property of Π_{VROM}. Here it is important that the zero-knowledge distinguisher is given the adversary interface of $\mathcal{F}_{VRO}$. Only then can the distinguisher which is built for the reduction adequately emulate the view of the distinguishing environment that it is simulating. We remark that this does not yet require $\mathcal{F}_{VRO}$ to have unique proofs. Second, verification of proofs is replaced by extraction. This is the step of the proof where the uniqueness of proofs is required. To reduce the simulation-sound online-extractability of Π_{VROM} to that of Π_{ROM} it has to be shown that the additional $\mathcal{F}_{VRO}$-proofs contained in the simulated proofs do not help the adversary $\mathcal{A}$. The condition that the proof output by $\mathcal{A}$ has to be fresh adds an additional case to be considered. Namely, the proof $\pi = (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i, \pi_i)_{i=1,2,\ldots,r}$ could differ from some previous proof $\pi' = (\mathsf{com}'_i, \mathsf{ch}'_i, \mathsf{resp}'_i, \pi'_i)_{i=1,2,\ldots,r}$ only in a single $\mathcal{F}_{VRO}$-proof $\pi_j \neq \pi'_j$. As long as π'_j is still accepted by $\mathcal{F}_{VRO}$ as a valid proof for input $(q, 0^b, \mathsf{vk})$ where $q = (x, \vec{\mathsf{com}}, j, \mathsf{ch}_j, \mathsf{resp}_j)$. But due to the uniqueness of proofs, such a case can not occur.

The remaining part of the reduction is then straightforward. $\mathcal{F}_{VRO}$ is simulated honestly, except that the hashes are set using queries to the external random oracle and proofs are simulated by augmenting the proofs received from the external oracle with honest proofs by $\mathcal{F}_{VRO}$. Finally, the proof returned by the adversary is stripped from its $\mathcal{F}_{VRO}$-proofs.

This concludes our outline of the proof. □

We have thereby shown the following theorem.

Theorem 4.2.22. *The protocol π_{VROM} UC-realizes $\mathcal{F}_{TZK}$ in der $\mathcal{F}_{VRO}$-hybrid model where $\mathcal{F}_{VRO}$ is required to have unique proofs.*

4.2.7. Removing the Need for Unique Proofs

The result from the last section is quite unsatisfactory. Unique proofs are a strong requirement and we would like to achieve the same result without them. One possibility is that instead of making a change to $\mathcal{F}_{VRO}$ in the form of unique proofs, we could make some changes to the Fischlin transform itself and thereby reinstate it resulting in UC TZKs when instantiated with $\mathcal{F}_{VRO}$.

Concretely, this would involve some way of adding non-malleability to proofs. One promising building block are *strong one-time signatures* (OTS), i.e. signature schemes for which an adversary is allowed to make one signing query m and wins if it can forge either a (fresh) signature for m or a new message m'. This primitive has been used in the past, e.g. in [81, 51, 63], to achieve similar goals to ours. The specific technique we use is, however, different.

As we have identified above, the problem with adding $\mathcal{F}_{VRO}$-proofs is that they are not included in hash queries which would protect their integrity. Let $\pi = (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i, \pi_i)_{i=1,2,\ldots,r}$ be proof as output by the current prover of Π_{VROM} and let $\mathsf{OTS} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ be an OTS. The idea is to augment π with a signature $\sigma \leftarrow \mathsf{OTS.Sign}(\mathsf{sk}, \pi)$ and the verification key vk. The verifier then in addition checks that σ is valid. Of course, this change alone would be totally insecure. The verification key vk and signature σ contained in the proof are not bound to π and so could be easily recomputed by an adversary. Below, we will fix this issue by making π itself depend on vk in a way that forging a valid proof π^* either implies breaking OTS or Π_{VROM}. Details follow.

The Augmented Protocol We make the above intuition formal and define an alternative randomized Fischlin transform $\Pi^*_{VROM} = (\mathcal{P}, \mathcal{V})$. First, we describe the prover $\mathcal{P}$. On input (x, w), $\mathcal{P}$:

- First generates an OTS key-pair $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{OTS.Gen}(1^\lambda)$.
- Then, $\mathcal{P}$ executes the prover of Π_{VROM}, except that vk is added as a prefix to all hash queries it makes.
- Let π be the resulting proof which we will call the *inner proof* from now on. It signs π and obtains a signature $\sigma \leftarrow \mathsf{OTS.Sign}(\mathsf{sk}, \pi)$.
- Finally, it outputs $\pi' = (\pi, \sigma, \mathsf{vk})$.

To verify a proof $(\pi, \sigma, \mathsf{vk})$, the verifier $\mathcal{V}$:

- First checks that $\mathsf{OTS.Verify}(\mathsf{vk}, \sigma, \pi) = 1$.
- After that, it executes the verifier from Π_{VROM}, except that vk is used as a prefix while verifying the $\mathcal{F}_{VRO}$-proofs contained in π.
- It accepts if and only if both verifications succeed.

Security First, it is clear that this transformation is still secure in the sense of our zero-knowledge definition. The simulator from Π_{VROM} additionally samples an OTS key-pair $(\mathsf{vk}, \mathsf{sk})$. When programming $\mathcal{F}_{VRO}$, vk is added as a prefix to all inputs. It then signs the simulated proof and includes the signature σ and vk in the final proof. Similarly, online extractability is essentially unchanged. After removing the signature, the OTS verification key, the $\mathcal{F}_{VRO}$-proofs, and the vk-prefix from all hash queries, the ordinary extractor can be used. This works because any valid augmented proof contains a valid non-augmented proof.

But what we are mainly interested in is showing simulation-sound online-extractability of Π^*_{VROM}. Let $\mathcal{A}$ be an adversary and (x, π) be its output with $\pi = (\pi', \sigma, \mathsf{vk})$. Let further $\mathsf{forge}_{\mathsf{OTS}}$ be the event that vk is contained in one of the simulated proofs which were previously received by $\mathcal{A}$. An adversary $\mathcal{B}$ on the security of OTS with an advantage

$$\mathsf{Adv}^{1-\mathrm{seuf}-\mathrm{cma}}_{\mathcal{B},\mathsf{OTS}}(\lambda) = \frac{1}{Q}\,\mathbf{Pr}\big[\mathsf{forge}_{\mathsf{OTS}}\big]$$

can be constructed in the obvious way and where Q is a polynomial upper bound on the number of simulated proofs requested by $\mathcal{A}$. It provides a simulation oracle to $\mathcal{A}$ and uses the verification key vk^* and single-use signing oracle it is provided with by its challenger to answer a randomly chosen query. Whenever $\mathcal{A}$ outputs a proof containing a forgery for vk^*, $\mathcal{B}$ outputs it to its challenger. By the assumed security of OTS, this implies that

$$\mathbf{Pr}\big[\mathsf{forge}_{\mathsf{OTS}}\big] \leq Q\,\mathsf{Adv}^{1-\mathrm{euf}-\mathrm{cma}}_{\mathcal{B},\mathsf{OTS}}(\lambda) \leq \mathsf{negl}(\lambda)$$

for some negligible function $\mathsf{negl}(\lambda)$. Conditioned on $\overline{\mathsf{forge}_{\mathsf{OTS}}}$, we thus have that vk is not contained in any previously received proofs. But vk is a prefix of every relevant hash query checked during the verification of π. We then claim that also the inner proof $\pi' = (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i, \pi_i)_{i=1,2,\ldots,r}$ contained in π is different from all simulated inner proofs

$$\pi_j = (\mathsf{com}_{j,i}, \mathsf{ch}_{i,j}, \mathsf{resp}_{i,j}, \pi_{i,j})_{i=1,2,\ldots,r}$$

where $j \in [Q]$, at least with overwhelming probability. For this not to be the case and indeed $\pi' = \pi_k$ for some $k \in [Q]$, the adversary would have to have chosen its vk such that replacing it for the verification key vk_k contained in the k'th simulated proof still yields a valid inner proof. Specifically, this requires simultaneously altering r inputs such that they still hash to 0^b. But the chance for this is 2^{-rb} for each try and so the probability of $\mathcal{A}$ succeeding in this task is upper-bounded by $Q\,2^{-rb}$ which is negligible due to the requirement $br \in \omega(\log \lambda)$ placed on the parameters. Conditioning also on this event, call it hash, not occurring, the inner proof π' is fresh. Additionally, as the π_i are not inputs to the hash queries, $\pi'' = (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i)_{i=1,2,\ldots,r}$ has to differ from the corresponding parts of the simulated proofs. From an attack of this kind by $\mathcal{A}$, an adversary C on the simulation-sound online extractability of Π_{ROM} can be constructed.

C behaves as follows:

- It simulates a copy of $\mathcal{A}$ and provides it with access to a session of $\mathcal{F}_{VRO}$ including the adversary interface as well as an oracle for simulation of proofs.

- The simulation of $\mathcal{F}_{VRO}$ is provided as follows. For some input $(\mathsf{vk}, x, \vec{\mathsf{com}}, i, \mathsf{ch}_i, \mathsf{resp}_i)$ for $i \in [r]$, C determines the hash by submitting $(x, \vec{\mathsf{com}}, i, \mathsf{ch}_i, \mathsf{resp}_i)$ to its external random oracle and using the obtained value h. All other aspects of the simulation are honest.

- Proofs for input (x, w) are simulated by first inputting (x, w) in the external simulation oracle. This yields a proof $\pi = (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i)_{i=1,2,\ldots,r}$.This proof is then used by the simulator for Π^*_{VROM} described above. In detail, a key-pair $(\mathsf{vk}, \mathsf{sk})$ for OTS is sampled, queries to $\mathcal{F}_{VRO}$ for $(\mathsf{vk}, \vec{\mathsf{com}}, i, \mathsf{ch}_i, \mathsf{resp}_i)$ are simulated, a correctly simulated number of *faux* hash queries is simulated, and finally a proof π' is computed from the resulting data in the obvious way.

- Once $\mathcal{A}$ outputs some pair (x^*, π^*), C parses $\pi^* = (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i, \pi_i)_{i=1,2,\ldots,r}$, constructs $\pi' = (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i)_{i=1,2,\ldots,r}$, and returns (x, π') to its challenger.

Due to the fact that inputs to the random oracle are truncated before their submission to the external oracle, the hash queries to $(\mathsf{vk}, \vec{\mathsf{com}}, i, \mathsf{ch}_i, \mathsf{resp}_i)$ are already programmed to return 0^b by the external simulator. For the same reason as well as the perfect unforgeability properties provided by $\mathcal{F}_{VRO}$, the proof which is finally output by C to its challenger is also valid. Conditioned on the event $\overline{\mathsf{forge}_{\mathsf{OTS}}} \wedge \overline{\mathsf{hash}}$, this proof will differ from all the previous simulated proofs received by C and so it will allow C to win with the same probability as $\mathcal{A}$. As we have shown that C has only a negligible chance of winning, the same is true for this remaining event in which $\mathcal{A}$ wins its game. By summing the advantages of $\mathcal{A}$ in the three exhaustive cases in which $\mathcal{A}$ wins and observing that their sum is negligible, we have shown the following theorem.

Theorem 4.2.23. *Let Π be a Σ-protocol in the ROM satisfying the requirements of the randomized Fischlin transform and let* OTS *be a strongly secure one-time signature scheme. Then the protocol Π^*_{VROM} obtained by the transformation described above is simulation-sound online extractable.*

As this was the only step in the proof in Section 4.2.6 where we used the uniqueness of proofs generated by $\mathcal{F}_{VRO}$, we obtain the following corollary if we let π^*_{VROM} be the UC-protocol in the multi-prover setting obtained from Π^*_{VROM} in the same manner as described at the beginning of Section 4.2.5.

Corollary 4.2.24. *The protocol π^*_{VROM} UC-realizes $\mathcal{F}_{TZK}$ in the $\mathcal{F}_{VRO}$-hybrid model.*

4.2.8. Final Thoughts

Completeness, online extractability, and zero-knowledge followed immediately from the correctness and unforgeability properties of $\mathcal{F}_{VRO}$ by slightly adapting the online extractor and zero-knowledge simulator used in the ROM. Proving that the randomized Fischlin transform produces UC TZKs was more involved due to the inherent malleability of the proofs output by $\mathcal{F}_{VRO}$. We showed two ways to alleviate this issue. First, by demanding unique proofs from $\mathcal{F}_{VRO}$. This allowed us to use the previously defined Fischlin transform in the VROM as-is but seems to greatly restrict the protocols realizing this version of

$\mathcal{F}_{VRO}$. We then showed how to lift this demand by making some small changes to the Fischlin transform. The main cost of the adaptation lay in the extra assumption of a strong one-time signature scheme.

5. VRO Instantiations

In this chapter, we will investigate different instantiations of $\mathcal{F}_{VRO}$. We start with an instantiation in the ROM and then show two instantiations that involve a single trusted party and do not require the ROM. Afterward, we build up to and finally give a construction involving multiple servers, some of which may be maliciously corrupted. We then analyze ways to relax $\mathcal{F}_{VRO}$ and what effect these changes have on possible instantiations. A special kind of instantiation, called a *hybrid instantiation*, which is secure under a stronger corruption model, is defined in Section 5.7 and a concrete hybrid instantiation is given. At the end of the chapter, we show how $\mathcal{F}_{VRO}$ can be realized using generic MPC.

5.1. ROM Instantiation

There is a natural instantiation of $\mathcal{F}_{VRO}$ using a random oracle $\mathcal{F}_{RO}$ with the same domain and codomain as the VRO. For technical reasons, we have to assume that the codomain has super-polynomial cardinality. Briefly, proofs are empty and hash queries as well as verification queries involve a hash query to $\mathcal{F}_{RO}$. In more detail, we define a protocol π_{RO} in the $\mathcal{F}_{RO}$-hybrid protocol. For a definition of $\mathcal{F}_{RO}$, see Section 2.4.2. A party $\mathcal{P}$ running π_{RO} behaves as follows:

- On input $(\texttt{Init}, sid)$, return $(\texttt{Key}, \bot)$.
- On input $(\texttt{Hash}, sid, q)$, send $(\texttt{Query}, sid, q)$ to $\mathcal{F}_{RO}$. Upon receiving an answer $(\texttt{Answer}, sid, q, h)$, output $(\texttt{HashProof}, sid, q, h, \bot)$.
- On input $(\texttt{Verify}, sid, q, h, \pi, \mathsf{vk})$, if either $\mathsf{vk} \neq \bot$ or $\pi \neq \bot$ hold, return $(\texttt{Verified}, sid, q, h, \pi, \mathsf{vk}, 0)$. Else, send $(\texttt{Query}, sid, q)$ to $\mathcal{F}_{RO}$. Upon receiving an answer $(\texttt{Answer}, sid, q, h')$, if $h \neq h'$ set $b = 0$. Else, set $b = 1$. In any case, return $(\texttt{Verified}, sid, q, h, \pi, \mathsf{vk}, b)$.

To prove the security of π_{RO}, we describe a simulator $\mathcal{S}$ for the dummy adversary $\mathcal{D}$. $\mathcal{S}$ behaves as follows:

- Upon receiving $(\texttt{Init}, sid)$ from $\mathcal{F}_{VRO}$, $\mathcal{S}$ sets $\mathsf{vk} = \bot$ and Prove as returning $\bot$ on every input (q, h, s). It sends $(\texttt{Init}, sid, \text{Prove}, \mathsf{vk})$ back to $\mathcal{F}_{VRO}$.
- Upon receiving $(\texttt{Hashing}, sid, \mathcal{P}, l)$ from $\mathcal{F}_{VRO}$, $\mathcal{S}$ immediately sets $s = \bot$ and responds with the message $(\texttt{SimInfo}, sid, \mathcal{P}, s)$.
- Upon receiving $(\texttt{Verify}, sid, q, h, \pi, \mathsf{vk}')$ from $\mathcal{F}_{VRO}$, $\mathcal{S}$ responds with the message $(\texttt{Verified}, sid, 0)$.

- The simulation of $\mathcal{F}_{RO}$ is provided as follows. Whenever a corrupted party sends a message $(\texttt{Query}, sid, q)$ and this is not the first query for q, $\mathcal{S}$ answers consistently. Else, $\mathcal{S}$ sends $(\texttt{Hash}, sid, q)$ to $\mathcal{F}_{VRO}$, answers the Hashing message as before, allows the response $(\texttt{HashProof}, sid, q, h, \pi)$ to be delivered and sets h as the hash for q.

- All outputs generated by $\mathcal{F}_{VRO}$ are delivered immediately.

It is easy to see that the only difference between simulation and the real interaction lies in the way verification queries are answered. In the real interaction, whenever h is the correct hash for q an empty proof $\pi = \bot$ is accepted for the empty verification key $\mathsf{vk} = \bot$, independent of whether q was previously hashed. In the ideal interaction, due to the unforgeability condition which is part of the verification procedure, *no* proof π for *no* hash h will be accepted for the correct verification key vk if q has not been previously hashed. As long as hashes are unpredictable, i.e. the codomain is super-polynomially large, this difference is unobservable with overwhelming probability, not even for an unbounded adversary (restricted to polynomially many verification queries).[1]

Due to this restriction, it is not quite valid to speak of $\mathcal{F}_{RO}$ as an idealized version of $\mathcal{F}_{VRO}$. We note, however, that this could be remedied, and $\mathcal{F}_{RO}$ be used to instantiate $\mathcal{F}_{VRO}$ for all potential codomains if we make the following change to how proofs are verified within $\mathcal{F}_{VRO}$. The idea is to allow the adversary to forge a proof π for some not previously queried q, but only for the correct hash h. The task of deciding whether to accept such a proof would again be given to the simulator as it does in the current version of $\mathcal{F}_{VRO}$ when an alternative proof or a proof for a wrong verification key is submitted.

We have chosen not to incorporate this change into $\mathcal{F}_{VRO}$ as it would further increase the complexity of its description. It also seems to be an artifact of π_{RO} having empty proofs as well as no verification key. In particular, all further instantiations presented in the following sections do not require restricting the codomain to be of super-polynomial size.

5.1.1. Small Codomains Using Truncation

There exists an alternative construction different from the intuitive way of instantiating $\mathcal{F}_{VRO}$ using an instance of $\mathcal{F}_{RO}$ we have just seen. In contrast to the construction above, no restrictions on the cardinality of the codomain have to be made. One drawback is, however, that proofs will no longer be empty strings and so the scheme will be less efficient. Combined with the fact that the above construction already works for super-polynomial codomains, the result from this section can be seen as a further mostly theoretical justification for the intuitively true fact that random oracles are idealized verifiable random oracles.

The Construction So let $\mathcal{H}$ be a codomain with $|\mathcal{H}| \in O(\mathsf{poly}(\lambda))$. For simplicity, we will in the following description restrict ourselves to strings of a constant length c, i.e. $\mathcal{H} = \{0, 1\}^c$. Our goal is to construct an instance of $\mathcal{F}_{VRO}$ with arbitrary domain $\mathcal{X}$ and codomain $\mathcal{H}$. To achieve this we will use an instance of $\mathcal{F}_{RO}$ with equal domain $\mathcal{D}$, but

[1] We note that, thus, this instantiation as a whole is secure against unbounded adversaries.

codomain $\mathcal{H}'$ with $\mathcal{H} \subset \mathcal{H}'$ and $|\mathcal{H}'| \in \omega(\mathsf{poly}(\lambda))$. Again for simplicity, we will let $\mathcal{H}' = \{0,1\}^\lambda$ consist of strings of length λ.

We define a protocol π'_{RO}. To generate a proof for some input q, a party running π'_{RO} still sends a query for q to $\mathcal{F}_{RO}$, receiving output $h' \in \{0,1\}^k$. Instead of letting all of h' be the hash associated with q, h' is first truncated to its first c bits, i.e. $h' = h||h^*$ for $h \in \{0,1\}^c$ and some $h^* \in \{0,1\}^{k-c}$. As noted before, proofs will no longer be empty, but will instead consist of the remaining portion h^*, i.e. $\pi = h^*$. To verify a proof (q, h, π), q is again queried to $\mathcal{F}_{RO}$, obtaining some h'. The decision of whether to accept or reject is then the result of the check $h||\pi = h'$.

The security of this scheme follows immediately from the original construction above. All we have done is we have shifted some of the hash into the proof. The simulator is adapted as follows:

- When asked to provide Prove and vk, $\mathcal{S}$ sets $\mathsf{vk} = \perp$ as before, but $\mathsf{Prove}(q, h, s)$ applies a $\mathsf{PRF} : \mathcal{K} \times \mathcal{X} \to \{0,1\}^{k-c}$ to q for some hard-coded key $k \in \mathcal{K}$. It outputs $\pi = \mathsf{PRF}(k, q)$ instead of outputting $\pi = \perp$ on all inputs. All responses to `Init` are delivered immediately.
- Upon receiving $(\texttt{Hashing}, sid, \mathcal{P}, l)$, $\mathcal{S}$ responds with $(\texttt{SimInfo}, sid, \mathcal{P}, \perp)$. When asked by $\mathcal{F}_{VRO}$ to deliver the proof, $\mathcal{S}$ does so.
- Queries to $\mathcal{F}_{RO}$ for some fresh input q are answered by first making a hash query for q to $\mathcal{F}_{VRO}$. Upon receiving the output h' and proof π, $\mathcal{S}$ sets $h = h'||\pi$ and returns h.
- When asked by $\mathcal{F}_{VRO}$ to determine the validity of some input (q, h, π, vk), $\mathcal{S}$ rejects the proof.

Note that our requirement on Prove to be stateless prohibits the use of true randomness to determine proofs. We are also unable to shift the responsibility of determining proofs onto $\mathcal{S}$ as the information within the `SimInfo` message can only depend on the length of q.

We forego rigorously proving the validity of $\mathcal{S}$, but note that after replacing PRF with a random function there remains no essential difference between the real and ideal interactions. We further note that the efficiency of the scheme can be increased. Instead of choosing $\mathcal{H}'$ to be exponentially large, any super-polynomial cardinality will be sufficient.

Taken together, the results from Sections 5.1 and 5.1.1 establish the following theorem.

Theorem 5.1.1. *For any domain $\mathcal{X}$ and codomain $\mathcal{H}$, there unconditionally exits a protocol π in the $\mathcal{F}_{RO}$-hybrid model which UC-realizes $\mathcal{F}_{VRO}$.* □

By applying the Universal-Composition Theorem (cf. Theorem 2.4.3), we are able to prove the following corollary which shows that $\mathcal{F}_{VRO}$ can truly be interpreted as a relaxation of $\mathcal{F}_{RO}$, which we set out to do.

Corollary 5.1.2. *Let ξ be a protocol UC-realizing a functionality $\mathcal{F}$ in the $\mathcal{F}_{VRO}, \mathcal{G}$-hybrid model where $\mathcal{G}$ is any set of ideal functionalities. Then $\xi^{\mathcal{F}_{VRO} \to \pi}$, where π is the protocol whose existence is asserted by Theorem 5.1.1, UC-realizes $\mathcal{F}$ in the $\mathcal{F}_{RO}, \mathcal{G}$-hybrid model.* □

5.2. Trusted Party Instantiation

In this section, we assume the existence of a single trusted party $\mathcal{T}$ and give protocols realizing $\mathcal{F}_{VRO}$ based on different cryptographic primitives. Note that this is different from the ROM instantiation in the sense that, due to the required non-interactiveness of $\mathcal{F}_{VRO}$ verification requests, any secure, i.e. interactive, instantiation of $\mathcal{F}_{RO}$ would not realize $\mathcal{F}_{VRO}$ while an instantiation using $\mathcal{T}$ may achieve truly non-interactive verification.

Simulatable VRF At first sight, it seems like $\mathcal{T}$ could run an instance of a VRF VRF. It would generate a key-pair $(\mathsf{ek}, \mathsf{vk}) \leftarrow \mathsf{VRF.Gen}(1^\lambda)$ and publish vk. Whenever a party $\mathcal{P}$ requested a proof for some input q, $\mathcal{T}$ would return $(h, \pi) \leftarrow \mathsf{VRF.Eval}(\mathsf{ek}, q)$. Verification of (q, h, π) with respect to the verification key vk would return $b = \mathsf{VRF.Verify}(\mathsf{vk}, q, h, \pi)$. This approach, however, does not work. By the properties of VRF, vk acts as a perfectly binding commitment to a PRF key, for every q there is exactly one h such that there exists a proof π which Verify will accept with respect to vk. And while a PRF can be replaced by a truly random function, as long as the key k remains unknown, the same is not possible for a VRF, as long as the verification key has been made public. Any simulator $\mathcal{S}$ has to be able to *program* the outputs of $\mathcal{T}$, i.e. generate valid proofs for any pair (q, h), as in the ideal world the h are chosen by $\mathcal{F}_{VRO}$ independently for each q.

Programmability can be achieved by instead using a simulatable VRF sVRF. Using SimProve, the simulator can generate valid proofs for any pair (q, h). One drawback of using sVRF lies in the fact that it requires a CRS to later give the simulator the advantage of knowing a backdoor to it. There are two ways of distributing it. Either we can use an ideal functionality $\mathcal{F}_{CRS}$ external to $\mathcal{T}$ which essentially makes the CRS available to every party without possible interference by the adversary, or we can let $\mathcal{T}$ itself generate and distribute it. Of course, in the latter case, even in the real world a simulated CRS might be used and this would be indistinguishable to users. But as we have assumed $\mathcal{T}$ to be trusted this is not an immediate issue. For simplicity we let $\mathcal{T}$ generate the CRS in the description that follows. As we will see below, it can give some additional security in the case of a corrupted $\mathcal{T}$ if we instead use $\mathcal{F}_{CRS}$ to distribute the CRS.

We can now describe the protocol, which we call π_{sVRF}, in detail by describing the actions of both $\mathcal{T}$ and any other parties $\mathcal{P}_i$ wishing to either hash inputs and generate proofs or verify them.

Whenever party $\mathcal{P}$ receives an `Init` message, if it is the first of its kind, $\mathcal{T}$ is contacted to retrieve the verification key vk' as well as the CRS σ. As verification key vk, the pair (vk', σ) is returned. When $\mathcal{P}$ is asked to generate a proof for input q, it sends q to $\mathcal{T}$ and receives a hash h and proof π which $\mathcal{T}$ computes using sVRF.Eval. Both are returned. A verification request for $(q, h, \pi, \mathsf{vk}')$ is answered by first parsing vk' as $(\mathsf{vk}^*, \sigma^*)$ and returning $\mathsf{sVRF.Verify}(\sigma^*, \mathsf{vk}^*, q, h, \pi)$. A detailed description is given in Figure 5.1. We assume that the identity of $\mathcal{T}$ is encoded in session identifier *sid*. For the communication network, we assume secure channels as modeled by $\mathcal{F}_{SMT}$, i.e. the adversary sees the length of the messages and is tasked with delivering them.

Having defined the protocol, we have to prove that it UC-realizes $\mathcal{F}_{VRO}$. We give a simulator $\mathcal{S}$ for the dummy adversary $\mathcal{D}$ and show that for each admissible environment

On input $(\texttt{Init}, sid)$

1 : $(\mathcal{T}, sid') = \texttt{parse}(sid)$
2 : **if** $\mathsf{vk}' = \bot \land \sigma = \bot$ **do**
3 : $\quad(\texttt{Init}, sid) \rightarrow \mathcal{T}$
4 : $\quad(\texttt{Key}, sid, \mathsf{vk}', \sigma) \leftarrow \mathcal{T}$
5 : **fi**
6 : $\mathsf{vk} = (\mathsf{vk}', \sigma)$
7 : **return** $(\texttt{Key}, sid, \mathsf{vk})$

On input $(\texttt{Hash}, sid, q)$

1 : $(\mathcal{T}, sid') = \texttt{parse}(sid)$
2 : $(\texttt{Hash}, sid, q) \longrightarrow \mathcal{T}$
3 : $(\texttt{HashProof}, q, h, \pi) \longleftarrow \mathcal{T}$
4 : **return** $(\texttt{HashProof}, sid, q, h, \pi)$

On input $(\texttt{Verify}, sid, q, h, \pi, \mathsf{vk}')$

1 : $(\mathsf{vk}^*, \sigma^*) = \texttt{parse}(\mathsf{vk}')$
2 : $b = \mathsf{sVRF.Verify}(\sigma^*, \mathsf{vk}^*, q, h, \pi)$
3 : **return** $(\texttt{Verified}, sid, q, h, \pi, b)$

Figure 5.1.: The algorithms run by a party $\mathcal{P}$.

On input $(\texttt{Init}, \texttt{sid})$

1 : **if** $\mathsf{vk} = \bot \land \sigma = \bot$ **do**
2 : $\quad\sigma \leftarrow \mathsf{sVRF.Setup}(1^\lambda)$
3 : $\quad(\mathsf{vk}, \mathsf{ek}) \leftarrow \mathsf{sVRF.Gen}(1^\lambda, \sigma)$
4 : **fi**
5 : **return** $(\texttt{Key}, sid, \mathsf{vk}, \sigma)$
6 :

On input $(\texttt{Hash}, sid, q)$

1 : / Assume an `Init` message occurred
2 : / Else do initialization now.
3 : $(h, \pi) \leftarrow \mathsf{sVRF.Eval}(\sigma, \mathsf{ek}, q)$
4 : **return** $(\texttt{HashProof}, sid, q, h, \pi)$

Figure 5.2.: The algorithms run by the trusted party $\mathcal{T}$.

$\mathcal{Z}$ the output distributions for interactions with either $\mathcal{D}$ and a session of the above protocol or $\mathcal{S}$ and an instance of $\mathcal{F}_{VRO}$ are indistinguishable.
The simulator $\mathcal{S}$ has to

- answer the $(\texttt{Init}, sid)$ message from $\mathcal{F}_{VRO}$.
- deliver public or private delayed outputs by $\mathcal{F}_{VRO}$.
- answer messages of the form $(\texttt{Hashing}, sid, \mathcal{P}, l)$, indicating a hash query by party $\mathcal{P}$.
- answer messages of the form $(\texttt{Verify}, sid, q, h, \pi, \mathsf{vk}')$, indicating a proof π for some correct pair (q, h), but which was not output by $\mathcal{F}_{VRO}$ itself.
- simulate $\mathcal{T}$ towards $\mathcal{D}/\mathcal{Z}$ during initialization and hash queries by honest and corrupted parties.

$\mathcal{S}$ behaves as follows:

- $\mathcal{S}$ runs a simulation of $\mathcal{D}$ and relays information between it and $\mathcal{Z}$. It also simulates $\mathcal{T}$ as well as any honest parties $\mathcal{P}$ participating in the protocol.
- When $\mathcal{S}$ receives a message $(\texttt{Init}, sid)$ from $\mathcal{F}_{VRO}$, if it has not yet had to run $\mathsf{sVRF.SimSetup}(1^\lambda)$ during the simulation of an initialization message, it does so and obtains a CRS σ with corresponding simulation backdoor τ. $\mathcal{S}$ in this case also runs $\mathsf{sVRF.SimGen}(1^\lambda, \sigma, \tau)$ to obtain an evaluation key ek and verification key vk. As Prove algorithm it sets $\mathsf{sVRF.SimEval}(\sigma, \tau, \mathsf{ek}, \cdot, \cdot)$, i.e. Prove upon input (q, h, s) (where s is ignored) returns a simulated proof π. The verification key vk is set to (vk, σ). $\mathcal{S}$ immediately[2] responds with the message $(\texttt{Init}, sid, \mathsf{Prove}, \mathsf{vk})$.
- When $\mathcal{S}$ is asked to deliver a `Init` response by $\mathcal{F}_{VRO}$ to some (honest) party $\mathcal{P}$, $\mathcal{S}$ lets the simulated copy of $\mathcal{P}$ initiate the honest initialization protocol between it and $\mathcal{T}$. Only when $\mathcal{P}$ generates output, indicating that $\mathcal{D}$ has delivered all necessary messages, does $\mathcal{S}$ allow the delivery.
- Upon receiving a message $(\texttt{Hashing}, sid, \mathcal{P}, l)$ for an honest party $\mathcal{P}$, indicating a `Hash` message by $\mathcal{P}$ to $\mathcal{F}_{VRO}$, $\mathcal{S}$ lets the simulated copy of $\mathcal{P}$ behave as follows. It executes the honest protocol for this case, but instead of including q in its message to $\mathcal{T}$, it includes 0^l. $\mathcal{S}$ immediately responds with a message $(\mathsf{SimInfo}, sid, \mathcal{P}, \bot)$, but when asked to deliver the corresponding `HashProof` message it only does so once the simulated $\mathcal{P}$ has generated its output.
- Upon receiving a message $(\texttt{Verify}, sid, q, h, \pi, \mathsf{vk}')$ from $\mathcal{F}_{VRO}$, $\mathcal{S}$ parses vk' as $(\mathsf{vk}^*, \sigma^*)$ and computes $b = \mathsf{sVRF.Verify}(\sigma^*, \mathsf{vk}^*, q, h, \pi)$. It then sends a message $(\texttt{Verified}, sid, q, h, \pi, \mathsf{vk}', b)$ back to $\mathcal{F}_{VRO}$.

[2] Recall that this means in the same activation.

- When the simulated $\mathcal{T}$ receives a message $(\texttt{Init}, sid)$ from some corrupted party $\mathcal{P}$, $\mathcal{S}$ runs sVRF.SimSetup(1^λ) and sVRF.SimGen$(1^\lambda, \sigma, \tau)$, if not already done previously, and responds with the obtained $(\mathsf{Key}, sid, \mathsf{vk}, \sigma)$.

- When the simulated $\mathcal{T}$ receives a message (Hash, sid, q) from some corrupted party $\mathcal{P}$, $\mathcal{S}$ forwards the message to $\mathcal{F}_{VRO}$ on behalf of the real $\mathcal{P}$. $\mathcal{S}$ answers the generated Hashing message with $s = \perp$ and instructs $\mathcal{F}_{VRO}$ to deliver the output. Upon receiving a response $(\mathsf{HashProof}, sid, q, h, \pi)$ from $\mathcal{F}_{VRO}$ it lets $\mathcal{T}$ forward this message to the simulated $\mathcal{P}$.

Indistinguishability This concludes the description of $\mathcal{S}$. It follows the analysis of the validity of $\mathcal{S}$ as a simulator for $\mathcal{D}$. We observe that the differences between the real and ideal interactions lie in the distribution of the CRS, whether proofs are generated by Eval or SimProve, and in the rejection of all forged proofs by $\mathcal{F}_{VRO}$. This allows us to reduce to the security of sVRF in two game-hops.

Unforgeability First, we use the perfect unforgeability of sVRF when an honestly generated σ is used. Formally, we move from the real interaction to an intermediate interaction where honest verifiers behave slightly differently. Upon receiving input $(\mathsf{Verify}, sid, q, h, \pi, \mathsf{vk})$ where vk is the correct verification key, reject π if either: (1) no hash query for q was previously received by $\mathcal{T}$ or (2) $h \neq h'$ where h' is the first component of sVRF.Eval(σ, ek, q).

The second case is impossible as there only exist valid proofs for the correct h. For the first case, assume that a verifying proof π was submitted for a pair (q, h), but where q was not previously queried. If there existed an adversary which was able to produce such proofs with non-negligible probability, then it could break the pseudo-randomness of the PRF portion of sVRF as finding a valid proof for (q, h) guarantees that h is the correct output for q again by the perfect unforgeability property.

Simulate Proofs In a second step, we move from the intermediate interaction to the ideal world and reduce to the simulatability of sVRF. Assume that $\mathcal{Z}$ is an environment that is able to distinguish between the intermediate and the ideal interaction. We build a successful adversary $\mathcal{B}$ on the security of sVRF from the successful distinguisher $\mathcal{A}$ for outputs of $\mathcal{Z}$ which we are assured exists.

$\mathcal{B}$ behaves as follows. It has black-box access to an instance of $\mathcal{Z}$ and also simulates an instance of $\mathcal{F}_{VRO}$ and the simulator $\mathcal{S}$ with some small modifications. Instead of generating a simulated CRS σ by running SimSetup and key-pair using SimGen, $\mathcal{S}$ uses the σ and vk which $\mathcal{B}$ receives from its challenger. Also, $\mathcal{F}_{VRO}$ answers Init queries by returning the verification key $\mathsf{vk}' = (\mathsf{vk}, \sigma)$. $\mathcal{F}_{VRO}$ generates its hashes and proofs for input q not by running the algorithm provided by $\mathcal{S}$, but by querying the Eval oracle provided to $\mathcal{B}$ on input q. When $\mathcal{Z}$ generates output it is given as input to $\mathcal{A}$. $\mathcal{B}$ outputs whatever $\mathcal{A}$ outputs.

When $\mathcal{B}$ is given simulated σ and vk and Eval is given as sVRF.SimProve it is clear that the view provided to $\mathcal{Z}$ is identical to an ideal interaction. On the other hand, if σ and vk were generated honestly and Eval is the honest sVRF.Eval, then $\mathcal{S}$ in its simulation lets

all honest parties including $\mathcal{T}$ behave as in the honest interaction and consistent with the outputs generated by $\mathcal{F}_{VRO}$. Hence, the view of $\mathcal{Z}$, in this case, is identical to the intermediate interaction.

The distinguishing advantage of $\mathcal{B}$ is identical to the advantage of $\mathcal{A}$. By the assumption on the security of sVRF the advantage of $\mathcal{B}$ is negligible which is in contradiction with the assumed success of $\mathcal{A}$ in distinguishing outputs of $\mathcal{Z}$. □

This proves the following theorem.

Theorem 5.2.1. *The protocol π_{sVRF} UC-realizes $\mathcal{F}_{VRO}$ in the $\mathcal{F}_{SMT}$-hybrid model.*

Using Signatures There is another simple way to instantiate $\mathcal{F}_{VRO}$ using a trusted party $\mathcal{T}$. $\mathcal{T}$ simply uses either a PRF or lazy sampling to assign hashes h to inputs q and then signs (q, h) using some $\mathsf{EUF\text{-}CMA}$-secure signature scheme SIG. Verification consists in verifying the signature making up the proof.

We will not prove the security of this instantiation in the same detail as above. Nevertheless, the following theorem holds if we call this protocol π_{SIG}.

Theorem 5.2.2. *The protocol π_{SIG} UC-realizes $\mathcal{F}_{VRO}$ in the $\mathcal{F}_{SMT}$-hybrid model.* □

Programming is possible as $\mathcal{T}$ is free to sign any hash h as long as its distribution is computationally indistinguishable from uniform. Unforgeability holds as a forged proof immediately yields a forged signature under the verification key published by $\mathcal{T}$.

Remark 5.2.3. Note that this is essentially using the additive and positive *signature accumulator* [9] to represent either the set

$$\left\{(q_i, \mathsf{PRF}(k, q_i)) \mid 1 \leq i \leq n_{\mathrm{q}}(t)\right\}$$

which grows over time t and where $n_{\mathrm{q}}(t)$ is the number of past queries at time t, or alternatively the static (but infinite) set

$$\{(q, \mathsf{PRF}(k, q)) \mid q \in \mathcal{X}\}\,.$$

Variants In the signature-based construction, (computationally) *unique proofs* could be achieved by using an $\mathsf{sEUF\text{-}CMA}$ secure signature scheme, caching the signature generated upon the first query for a value q, and returning it on all subsequent queries for the same q. While there then may exist many proofs which would lead to a successful verification, it would be infeasible for any polynomially bounded adversary to find a second valid proof. The scheme could again be made to require $O(\lambda)$ space instead of space proportional to the number of past queries by deriving the randomness used by the signing algorithm from the input q using a PRF with independent key k.

Differences There are some differences between the two protocols above which are not captured by the fact that they both UC-realize $\mathcal{F}_{VRO}$. In the signature-based construction, if $\mathcal{T}$ were to be corrupted instead of honest, it could completely break any guarantees provided by $\mathcal{F}_{VRO}$. In particular, the adversary could forge valid proofs for any pair (q, h).

The construction using sVRF, on the other hand, is better behaved in this scenario[3]. For every q, only for a single h can a valid proof be efficiently found. The adversary can let $\mathcal{T}$ not respond to any `Hash` messages, but this power was already possessed by the adversary before. The main security degradation lies in the fact that now the adversary learns all inputs and hashes. As we have seen, this is already enough to break the security of the Fischlin transform.

5.3. Allowing Corruption

In this section, we build up to an instantiation of $\mathcal{F}_{VRO}$ involving multiple servers, some of which may be statically and maliciously corrupted. We begin by investigating generic constructions to distribute protocols for a single trusted party. Then we review a construction from [35] which we show to be insufficient to UC-realize $\mathcal{F}_{VRO}$, but which serves as an intermediate step for the main contribution contained in this chapter.

5.3.1. Distributing Protocols for Trusted Parties

One way of constructing a protocol resilient under some amount of corruption is by taking a protocol π which is secure in the setting of a single trusted party $\mathcal{T}$ and distributing the functionality offered by $\mathcal{T}$ onto a set of n servers $\mathcal{S}_1$ up to $\mathcal{S}_n$ such that any subset of $t < n$ corrupted servers is unable to break the security guarantees of the underlying primitive. This approach of distributing a primitive is generally known under the name *threshold cryptography* [34].

Let Prim be a cryptographic primitive which receives as input a secret key $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda)$ as well as some other data d and outputs $o \leftarrow \mathsf{Prim}(\mathsf{sk}, d)$. To distribute Prim, an algorithm Share is introduced which takes in a key sk as generated by Gen and outputs n shares $\mathsf{sk}_1, \ldots, \mathsf{sk}_n$ which are given to the $\mathcal{S}_n$. If Gen also outputs a public key pk, then this key is given to all of the servers. Evaluating Prim in a distributed manner in its most general form involves an interactive computation among the servers and where server $\mathcal{S}_i$ has input consisting of its share sk_i of the secret key, the public key pk, and the common input d. At the end of the computation, each server is in possession of an output share o_i. For a protocol secure under t corruptions, at least $t + 1$ servers have to partake in the computation and combine their shares of the output using an algorithm Combine to obtain the final result.

Based on the protocols from Section 5.2, the primitives we would wish to distribute are (simulatable) VRFs or PRFs and signature schemes. For the former, the resulting primitive is then generally called a *distributed VRF* (DVRF) in the literature [40, 38, 50]. There are, however, various security definitions for threshold systems. These range from game-based over standalone simulatability to UC simulatability.[4] Given that we are working within

[3]If we for a moment assume that the CRS is given externally by some functionality $\mathcal{F}_{CRS}$. Otherwise, if $\mathcal{T}$ is already corrupted at the beginning of the session it can generate a CRS with a backdoor and again simulate proofs for arbitrary pairs (q, h).

[4]We note that a UC-secure DVRF would necessarily be simulatable. The simulator receives a randomly chosen output from the ideal functionality and has to simulate a view towards the corrupted parties that is consistent with their shares of the secret key and this output.

the UC framework, only the latter kind of definition is sufficient. Furthermore, as we have said above, d is generally assumed to be a *common* input to all servers. As $\mathcal{F}_{VRO}$ requires the inputs to remain largely hidden from the adversary, DVRFs do not immediately yield protocols realizing $\mathcal{F}_{VRO}$. We will, however, in Section 5.5 look at relaxed variants of $\mathcal{F}_{VRO}$ where we will be able to directly use tools from threshold cryptography.

5.3.2. The PRF Construction

As hinted at in the introduction, one step on the way to finding an instantiation if $\mathcal{F}_{VRO}$ allowing for some malicious corruption is the *PRF construction* introduced in [35]. For sake of being self-contained, we first repeat the protocol here while casting it as a protocol π_{PRF} in the UC framework. We have incorporated some of the efficiency measures mentioned in [35] as well as slightly altered the initialization procedure such that π_{PRF} has the same interface as $\mathcal{F}_{VRO}$. We then show how it fails to UC-realize $\mathcal{F}_{VRO}$. Later, in Section 5.5.1, we will show that π_{PRF} *does* UC-realize a relaxed formulation of $\mathcal{F}_{VRO}$.

The Protocol The protocol is formulated in a client and server setting. Let $\mathcal{X}$ be the domain and $\mathcal{H}$ the codomain and let $\mathsf{PRF} : \mathcal{K} \times \mathcal{X} \to \mathcal{H}$ be a PRF. In its most simple form, the basic idea of π_{PRF} is to evaluate functions of the form

$$\begin{aligned} \mathsf{RF} : \mathcal{X} &\to \mathcal{H} \\ q &\mapsto h = \bigoplus_{j=1}^{4} \mathsf{PRF}(k_j, q) \end{aligned}$$

for random keys $k_i \leftarrow\$ \mathcal{K}$ in a distributed manner. For this, each of four servers $\mathcal{S}_i$ is in possession of three keys $k_j, j \neq i$. A client C sends its input q to each of the servers which reply with partial hashes $h_i = \mathsf{PRF}(k_i, q)$ for all keys they possess. To accommodate for one of the servers potentially misbehaving due to being corrupted, each of the h_i is determined by computing the majority over the three received candidate values. As long as at most one server is corrupted, every honest client is guaranteed to be able to compute the correct result.

Verifiability is added by assigning to each server $\mathcal{S}_i$ a key-pair $(\mathsf{vk}_i, \mathsf{sk}_i)$ for an EUF-CMA-secure signature scheme SIG. The verification keys vk_i are made publicly available. The distributed computation of RF is then augmented as follows: In addition to computing $h_j = \mathsf{PRF}(k_j, q)$ for $j \neq i$, server $\mathcal{S}_i$ signs the message (q, h_j, j). The obtained signatures σ_j are returned to the client. The client's task is augmented by computing a proof π consisting of the four partial hashes h_i obtained as shown above as well as all signatures obtained from all servers.

The full protocol is shown in Figures 5.3 and 5.4. The initialization algorithm Initialize is executed by the servers upon their first invocation. $\mathcal{F}_{bboard}$ is an ideal bulletin-board functionality providing public storage. It is shown in Figure 5.6 and allows any party to post and retrieve messages. MPC is a UC-secure MPC protocol and SIGKeyGen and PRFKeyGen are protocols shown in Figure 5.8 We will not need exact semantics for it at this point. Intuitively, each server obtains a key-pair for SIG, the verification keys for the other servers as well as three PRF keys. It then posts the verification key to a public location.

On input (Hash, sid, q)

```
1 :  for i ∈ {1, 2, 3, 4} do
2 :     (Hash, sid, q) → S_i
3 :     (h_{i,j}, σ_{i,j})_{j∈[4]\{i}} ← S_i
4 :  endfor
5 :  for i ∈ {1, 2, 3, 4} do
6 :     h_i = argmax_{h∈H} |{h = h_{j,i} | j ∈ [4] \ {i}}|
7 :  endfor
8 :  h = ⊕_{i=1}^{4} h_i
9 :  Σ = (σ_{i,j})_{1≤i,j≤4,i≠j}
10 : π = (h_1, h_2, h_3, h_4, Σ)
11 : return (HashProof, sid, q, h, π)
```

On input (Init, sid)

```
1 :  if vk ≠ ⊥ do
2 :     return (Key, sid, vk)
3 :  fi
4 :  for i ∈ {1, 2, 3, 4} do
5 :     (Retrieve, sid, S_i) → F_bboard
6 :     (Stored, sid, S_i, vk_i) ← F_bboard
7 :     if vk_i = ⊥ do
8 :        abort
9 :     endfor
10 :    vk = (vk_1, vk_2, vk_3, vk_4)
11 :    return (Key, sid, vk)
```

On input (Verify, sid, q, h, π, vk′)

```
1 :  (vk_1, vk_2, vk_3, vk_4) = parse(vk′)
2 :  (h_1, h_2, h_3, h_3, Σ) = parse(π)
3 :  if h ≠ ⊕_{i=1}^{4} h_i do
4 :     return (Verified, sid, q, h, π, vk′, 0)
5 :  fi
6 :  for i ∈ {1, 2, 3, 4} do
7 :     m = (q, h_i, i)
8 :     if |{σ_{j,i} | j ∈ [4] \ {i},
9 :        SIG.Verify(vk_j, m, σ_{j,i} = 1)}| < 2 do
10 :       return (Verified, sid, q, h, π, vk′, 0)
11 :    fi
12 : endfor
13 : return (Verified, sid, q, h, π, vk′, 1)
```

Figure 5.3.: The client algorithms of π_{PRF}

On input (Hash, sid, q)	Initialize($\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4$)
1 : **for** $i \in \{1,2,3,4\} \setminus \{n\}$ **do**	1 : $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4\}$
2 : $h_i = \mathsf{PRF}(k_i, q)$	2 : $\mathsf{out} \leftarrow \mathsf{MPC}(\mathsf{SIGKeyGen}, \mathcal{S})$
3 : $\sigma_i = \mathsf{SIG.Sign}(\mathsf{sk}, (q, h_i, i))$	3 : $(\mathsf{sk}_n, \mathsf{vk}_1, \mathsf{vk}_2, \mathsf{vk}_2, \mathsf{vk}_3, \mathsf{vk}_4) = \mathtt{parse}(\mathsf{out})$
4 : **endfor**	4 : **for** $i \in \{1,2,3,4\} \setminus \{n\}$ **do**
5 : **return** $(h_i, \sigma_i)_{i \in [4], i \neq n}$	5 : $k_i \leftarrow \mathsf{MPC}(\mathsf{PRFKeyGen}, \mathcal{S} \setminus \{\mathcal{S}_i\})$
	6 : **endfor** $(\mathtt{Store}, sid, \mathsf{vk}_n) \rightarrow \mathcal{F}_{bboard}$
	7 :

Figure 5.4.: The server algorithms of π_{PRF} from the perspective of server $\mathcal{S}_n$

Remark 5.3.1. We have fixed a small error in the protocol. In their proof of *weak unforgeability*, [35] implicitly assume that elements of a proof can not be reordered or copied. To fix this problem we bind signatures σ, which in the original protocol would have been for a message (q, h_i), to the index i by also including i as the third component in each signed message. This way, reordering or copying will cause a mismatch between the index a verifying party will use to check the correctness of the signature and the index contained in the message which was actually signed.

Evaluation We evaluate π_{PRF} with respect to $\mathcal{F}_{VRO}$. As stated in [35], the goal of π_{PRF} is to allow the set of honest servers to collectively program the random function RF in such a way that the adversary controlling the remaining party is oblivious to this being the case. This lead to the requirement that no single party or collection of dishonest parties as a whole have to be able to compute the "true" hash h for some input q by themselves. The input, on the other hand, was allowed to be obtained by the adversary and is in fact given to every single server. This level of privacy guaranteed to honest clients is insufficient as we have seen in our application of $\mathcal{F}_{VRO}$ to the randomized Fischlin transform.

5.4. Adding Privacy Using FHE-Encryption

Building upon the protocol π_{PRF} defined in the previous section, in this section we augment π_{PRF} using FHE to achieve the strong hiding properties which are required by $\mathcal{F}_{VRO}$. We begin by stating the goals we are trying to achieve and motivating our use of the FHE primitive. Having done so, we describe all the building blocks we need. For definitions, we refer the reader back to Chapter 2. Then we give the actual protocol π_{FHE}, both a textual overview as well as a formal algorithmic description. We proceed with the proof that our protocol UC-realizes $\mathcal{F}_{VRO}$. After that, we give a more detailed discussion of some aspects of π_{FHE} and its security proof. We also discuss several variations as well as address the efficiency of the protocol.

5.4.1. Goals

Beside UC-realizing $\mathcal{F}_{VRO}$, we wish to achieve several goals. First, as $\mathcal{F}_{VRO}$ is intended to be a relaxation of $\mathcal{F}_{RO}$ and by the fact that we have already shown that $\mathcal{F}_{RO}$ alone can be used to realize $\mathcal{F}_{VRO}$, we want to avoid using $\mathcal{F}_{RO}$ within the protocol. Second, we wish to retain the non-interactivity among servers from π_{PRF}, except again during an initialization phase at the beginning of each session. Third, we wish to retain the security for a single statically and maliciously corrupted server. Fourth, we wish to protect honest servers from corrupted clients, a concern that did not come up in π_{PRF} as there correct behavior by the client could trivially be enforced by the servers by rejecting all inputs which do not lie in the domain. Lastly, we avoid the use of outright NIZK proofs of knowledge.

Remark 5.4.1. We note that with security for a single corrupted server we, as for π_{PRF}, mean that one of the servers must be able to respond with values that arbitrarily deviate from the correct protocol while still allowing honest parties to obtain correct proofs.

5.4.2. Rationale for FHE

There are multiple techniques that allow a client interacting with a set of n servers to give input q to the servers in such a way that no coalition of $t < n$ corrupted servers can jointly compute q, but still allows all n servers to execute a computation depending on q. The classical approach to this task is the use of (threshold) *secret sharing* schemes such as Shamir's polynomial-based scheme [83] with reconstruction parameter $r > t$. In the present context, to guard servers against receiving inconsistent shares from a corrupted client, an augmented primitive, a so-called *verifiable secret sharing*, e.g. Feldman's scheme [43], would have to be employed. This secret-sharing approach is often used in general MPC protocols. A major drawback, however, is that when using an information-theoretic scheme—which Shamir's scheme is—every single share q_i received by an individual server contains no information about q. As such, to be able to do some computation that functionally depends on q, the servers have to communicate.

As we have stated above, one of our goals is *minimizing* the interaction between the servers. By replacing the secret sharing of the input q with an encryption $c = \mathsf{FHE.Enc}(\mathsf{pk}, q)$ of q under some key pk not known to the adversary, we are able to eliminate *all* of the interaction between the servers by allowing each individual server to do computation on q in isolation.[5]

5.4.3. Building Blocks

To be able to describe our protocol π_{FHE} for $\mathcal{F}_{VRO}$ where $\mathcal{F}_{VRO}$ is parametrized with domain $\mathcal{X}$ and codomain $\mathcal{H}$, we require the following primitives:

- An FHE scheme FHE which is semi-honestly statistically circuit private and has a full ciphertext space.
- An EUF-CMA signature scheme SIG with message space $\mathcal{M} = \mathcal{X} \times \mathcal{H} \times \mathbb{Z}_4$ and arbitrary signature space.

[5] At least during hash queries, there may be interaction during an initialization phase.

- A pseudo-random function PRF : $\mathcal{X} \to \mathcal{H}$.
- UC-secure zero-knowledge proofs/arguments of knowledge in the form of an ideal functionality $\mathcal{F}_{ZK}$ (shown in Figure 5.5), i.e. we work in the $\mathcal{F}_{ZK}$-hybrid model.
- An extractable non-interactive witness-indistinguishable argument system NIWI in the CRS model and for a relation described below.[6]
- Public storage in the form of an ideal bulletin-board functionality $\mathcal{F}_{bboard}$ (shown in Figure 5.6).
- A protocol MPC allowing for UC-secure function evaluation.
- Secure (and authentic) channels in the form of the previously introduced secure message-transfer functionality $\mathcal{F}_{SMT}$.

We describe some of the primitives more closely.

FHE As defined in Section 2.5.3, we abstract away from the low-level representation of ciphertexts for FHE. Messages might be encrypted bit-by-bit or in larger chunks depending on the algebraic setting. In any case, we will write c for a ciphertext (vector) for message m, independent of the length of m.

As the size of circuits we will have to evaluate using FHE may be fixed depending on whether the domain $\mathcal{X}$ is finite or infinite, we may only require a *leveled fully-homomorphic encryption scheme* where *leveled* means that the key-generation algorithm expects to receive a level d and only circuits C of depth at most d can be evaluated using such a key. This would allow us to get rid of the *circular security assumption* going into the security of all currently known non-leveled FHE schemes, albeit at the cost of having the size of public keys depend on d.

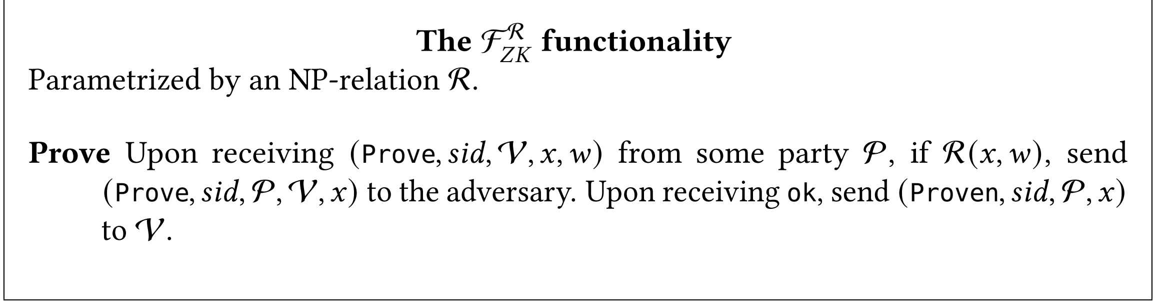
Zero-Knowledge Proofs For simplicity, we let $\mathcal{F}_{ZK}^{\mathcal{R}}$ be single-prover and single-proof, i.e. a single session of $\mathcal{F}_{ZK}^{\mathcal{R}}$ may only be used by a single prover $\mathcal{P}$ to prove a single statement x each to a single verifier $\mathcal{V}$ with respect to some parametrizing relation $\mathcal{R}$. To facilitate either multiple proofs between a fixed pair $(\mathcal{P}, \mathcal{V})$ of parties or between multiple, possibly disjoint, pairs of parties, different sessions of $\mathcal{F}_{ZK}^{\mathcal{R}}$ have to be used. To guarantee global uniqueness of session identifiers sid, these will generally be prefixed with both the prover and verifier identities, i.e. be of the form $sid = (\mathcal{P}, \mathcal{V}, sid')$ for some sid'. Only a different sid' has then to be chosen by a prover $\mathcal{P}$ for different proofs made towards the *same* verifier $\mathcal{V}$.

We use one instance of $\mathcal{F}_{ZK}^{\mathcal{R}}$ parametrized by a relation $\mathcal{R}_{\mathsf{FHE}}$ which is given as

$$\mathcal{R}_{\mathsf{FHE}} = \left\{ (x, w) \;\middle|\; \begin{array}{l} x = \mathsf{pk}, w = (\mathsf{sk}, r), \\ (\mathsf{pk}, \mathsf{sk}) = \mathsf{FHE.Gen}(1^\lambda; r) \end{array} \right\} \tag{5.1}$$

This means we may let parties prove knowledge of FHE secret keys.

[6]We will denote the CRS as crs instead of σ in the present context to avoid confusion with signatures.



The $\mathcal{F}_{ZK}^{\mathcal{R}}$ functionality

Parametrized by an NP-relation $\mathcal{R}$.

Prove Upon receiving $(\texttt{Prove}, sid, \mathcal{V}, x, w)$ from some party $\mathcal{P}$, if $\mathcal{R}(x, w)$, send $(\texttt{Prove}, sid, \mathcal{P}, \mathcal{V}, x)$ to the adversary. Upon receiving $\texttt{ok}$, send $(\texttt{Proven}, sid, \mathcal{P}, x)$ to $\mathcal{V}$.

Figure 5.5.: The UC Zero-Knowledge functionality.

Extractable NIWI We require an extractable non-interactive witness-indistinguishable argument NIWI. For convenience, we will speak of *proofs* instead of *arguments* even if only the latter are used. The language for which we require NIWI to prove statements consists of pairs (vk, m) where vk itself consists of three verification keys vk_i for SIG and $m \in \mathcal{M}$ is an arbitrary message. Witnesses consist of two signatures σ_1, σ_2 for the message m and valid under two different verification keys among the vk_i.

Concretely, we define $\mathcal{R}_{\mathsf{SIG}}$ to be the relation

$$\mathcal{R}_{\mathsf{SIG}} = \left\{ (x, w) \;\middle|\; \begin{array}{l} x = (\mathsf{vk}_1, \mathsf{vk}_2, \mathsf{vk}_3, m), w = (\sigma_1, \sigma_2), \\ m = (q, h, i) \in \mathcal{X} \times \mathcal{H} \times \mathbb{Z}_4, \exists i, j \in \{1, 2, 3\}, i \neq j : \\ \mathsf{SIG.Verify}(\mathsf{vk}_i, m, \sigma_1) = 1, \mathsf{SIG.Verify}(\mathsf{vk}_j, m, \sigma_2) = 1 \end{array} \right\} \tag{5.2}$$

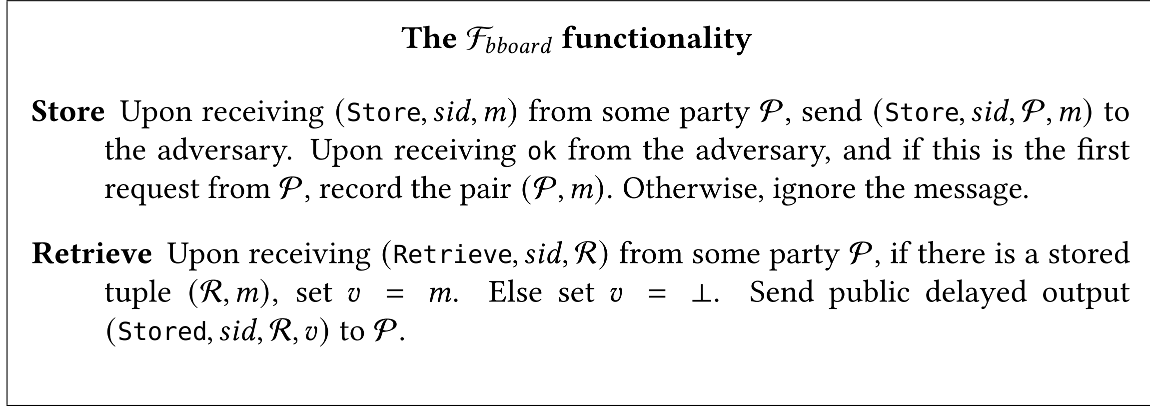
where $\mathsf{vk}_i, i \in [3]$, are any well-formed verification keys generated by $\mathsf{SIG.Gen}$.

The NIWI we employ makes use of a CRS which has to be available to all parties generating or verifying proofs. We will, however, not be working in a hybrid model where each party automatically has access to such a CRS. Instead, we will let a small number of servers compute the CRS using the (general) MPC protocol MPC. The CRS will then be distributed together with the $\mathcal{F}_{VRO}$ verification key and as such be available during verification.

Public Storage We make use of public storage represented by an ideal bulletin-board functionality $\mathcal{F}_{bboard}$ (taken from [23]) to which parties can post information using `Store` messages and other parties can retrieve this information using `Retrieve` messages. For simplicity, we assume that each party can only post a single message within a single session of $\mathcal{F}_{bboard}$ and that this information can not later be deleted, not even by the party which originally posted it.

5.4.4. The Protocol

Within any session of π_{FHE}, there are four distinguished servers $\mathcal{P}_1$ to $\mathcal{P}_4$ (the identities of which could for example be encoded in the session identifier) as well as other non-distinguished parties which we name callers. The number of callers does not have to be *a priori* bounded. We aim to provide security as long as at most one of the servers is



The $\mathcal{F}_{bboard}$ functionality

Store Upon receiving $(\texttt{Store}, sid, m)$ from some party $\mathcal{P}$, send $(\texttt{Store}, sid, \mathcal{P}, m)$ to the adversary. Upon receiving ok from the adversary, and if this is the first request from $\mathcal{P}$, record the pair $(\mathcal{P}, m)$. Otherwise, ignore the message.

Retrieve Upon receiving $(\texttt{Retrieve}, sid, \mathcal{R})$ from some party $\mathcal{P}$, if there is a stored tuple $(\mathcal{R}, m)$, set $v = m$. Else set $v = \perp$. Send public delayed output $(\texttt{Stored}, sid, \mathcal{R}, v)$ to $\mathcal{P}$.

Figure 5.6.: The UC Bulletin-Board functionality.

maliciously and statically corrupted, any number of callers may be corrupted. Without loss of generality, we will later let $\mathcal{P}_1$ be the corrupted server.

There are several sub-protocols executed by different parties over the course of a session of π_{FHE}. An initialization phase is executed by the servers upon their first invocation. At the end of this phase and if the adversary allowed delivery of the relevant messages, the servers will have computed their private keys as well as posted their share of the verification key to public storage. Other parties can then retrieve this verification key using another sub-protocol. To create a proof, i.e. execute a Hash query, a caller C and all four servers $\mathcal{P}_i$ are involved. Verification of proofs only involves the single party receiving the proof.

Server Initialization Server initialization is described by a function Initialize (see Figure 5.8) which is executed as soon as the servers $\mathcal{P}_i$ start running. The servers engage in several rounds of MPC. First, they execute the protocol SIGKeyGen which honestly samples a key-pair $(\mathsf{vk}_{Sig,i}, \mathsf{sk}_{Sig,i})$ for SIG for each server, and outputs sk_i as well as *all* verification keys $\mathsf{vk} = (\mathsf{vk}_{Sig,j})_{j\in[4]}$ to $\mathcal{P}_i$. Then, a CRS *crs* is computed using NIWI.Setup. Last, for each subset of three servers, a PRF key k is sampled from $\mathcal{K}$ and output to each server. Honest servers will, if and once all the previous protocols have terminated successfully, post the verification keys and the CRS, i.e. (crs, vk), to $\mathcal{F}_{bboard}$.

Remark 5.4.2. We can exploit the two-thirds honest majority setting we are working in and use MPC protocols that are specifically tailored to this purpose, see [49]. For example can the joint generation of PRF keys, which are usually represented by random bit-strings, be done efficiently using the protocol for realizing $\mathcal{F}_{coin}$ described in [49].

Caller Init Whenever a party receives input $(\texttt{Init}, sid)$ it retrieves the stored data for each of the $\mathcal{P}_i$, $i \in [4]$, from $\mathcal{F}_{bboard}$, obtaining (crs_i, vk_i). It constructs a verification key $\mathsf{vk} = ((\mathsf{vk}_{Sig,i})_{i\in[4]}, crs)$ from the vk_i as well as from the crs_i by taking the majority of the received values (only one of them may be wrong) and returns $(\texttt{Key}, sid, \mathsf{vk})$. Note that to facilitate the consistency of Init tasks required by $\mathcal{F}_{VRO}$, output is only generated if none of the retrieved values are equal to $\perp$. Notice that we could be slightly more lenient here

and only require that there are two equal values for each of the four signature verification keys as well as the CRS. At least one of them must be the honest value.

Caller Hash (1/2) Whenever a party $\mathcal{P}$ receives input $(\texttt{Hash}, sid, q)$ it proceeds as follows. If it does not yet know the verification key vk it retrieves the individual keys from $\mathcal{F}_{bboard}$ in the same way as if it had received input $(\texttt{Init}, sid)$. Once this has concluded[7], there are two possibilities. Either $\perp$ was returned when trying to retrieve the stored value for some server, in which case the current query aborts. Or vk has been successfully set, in which case the party proceeds as follows. It generates an FHE key-pair $(\mathsf{pk}_{FHE}, \mathsf{sk}_{FHE}) \leftarrow \mathsf{FHE.Gen}(1^\lambda; r)$ and encrypts q under pk_{FHE} obtaining a ciphertext c. For each of the servers $\mathcal{P}_i$, the party first proves the statement $x = \mathsf{pk}_{FHE}$. The witness w consists of sk_{FHE} as well as the randomness r. To do this, $\mathcal{P}$ sends the message $(\texttt{Prove}, (\mathcal{P}, \mathcal{P}_i, sid, i), \mathcal{P}_i, x, w)$ to $\mathcal{F}_{ZK}^{\mathcal{R}_{\mathsf{FHE}}}$ where i is the number of past queries executed by $\mathcal{P}$. For a definition of $\mathcal{R}_{\mathsf{FHE}}$ see (5.1). It then sends the message $(\texttt{Hash}, sid, c)$ to $\mathcal{P}_i$.

Server Hash To stay chronological, we interject with the actions of a server upon receiving c. A server only reacts to $(\texttt{Hash}, sid, c)$ messages if it was successfully initialized. This includes having posted a message to $\mathcal{F}_{bboard}$ (which in turn requires having received the output of SIGKeyGen and NIWI.Setup) and having received all PRF keys k_i for which it partook in the generation protocol. Then, if indeed a $\texttt{Proved}$ message containing some pk_{FHE} was received prior to receiving c, the server continues, otherwise, we ignore the request.[8] If all is well, the server for each of its known PRF keys k_i evaluates the function $x \mapsto \mathsf{PRF}(k_i, x)$ homomorphically on c using pk_{FHE}, obtaining ciphertexts c_i. It then homomorphically evaluates $x \mapsto \mathsf{SIG.Sign}(\mathsf{sk}_{Sig}, x)$ on each tuple (c, c_i, c_t) with c_t being an encryption of i. After obtaining the resulting encrypted signatures $c_{\sigma,i}$, all the c_i and $c_{\sigma,i}$ are returned to the caller.

Caller Hash (2/2) Once responses by each of the four servers are received, the caller decrypts all the received ciphertexts $c_{i,j}$ and $c_{\sigma,i,j}$ using sk_{FHE} to obtain partial hashes $h_{i,j}$ as well as purported signatures $\sigma_{i,j}$. Each of the signatures is checked for validity using the keys contained in vk. For each $i \in [4]$, h_i is computed as the majority over $h_{i,j}$ with $j \in [4]$ and $i \neq j$. Two different valid signatures σ_1 and σ_2 under two different verification keys and for the message (q, h_i, i) are selected as input to NIWI.Prove, yielding a proof π_i. The full proof π then consists of all the h_i and π_i.

Remark 5.4.3. Strictly speaking, Sig.Sign expects a message m as its second input and thus we have to provide a ciphertext c_m of m as input to $\mathsf{FHE.Eval}$, but we are providing (c, c_i, c_t). Thus c, c_i, and c_t would first have to be homomorphically evaluated to a ciphertext $\hat{c}$ containing the actual message we want to have signed. We note that mere concatenation is not sufficient as tuples have to be encoded in a way that allows them to be efficiently parseable.

[7]With this we mean once responses by $\mathcal{F}_{bboard}$ for all four servers have been received. Of course, this may hang indefinitely in which case the current task will also hang indefinitely.

[8]This may also occur for honest callers if the adversary chooses to deliver c before the $\texttt{Proved}$ message.

Verification To verify a proof $\pi = (h_1, \ldots, h_4, \pi_1, \ldots, \pi_4)$ for (q, h), a verifier checks that π contains four values h_i which sum to $h = \bigoplus_{i=0}^{4} h_i$ and that, for each $i \in [4]$, π_i is valid NIWI proof for the statement x_i containing the three verification keys $\{\mathsf{vk}_{Sig,j}\}_{j\in[4]\setminus\{i\}}$, and the message (q, h_i, i).

We give additional remarks about this protocol in Section A.3.1 of the appendix. There we discuss, among other things, the possibility of using key registration instead of letting callers prove knowledge of their secret key for each query and why we have to let callers wait for responses from all four servers before being allowed to output a proof.

5.4.5. Proof of Security

In this section, we prove the security of the above protocol.

5.4.5.1. The Simulator

Let $\mathcal{D}$ be the dummy adversary. We give a simulator $\mathcal{S}$ for it. Again let $\mathcal{P}_1$ be the corrupted server and C_1 to C_k be the identities of corrupted callers. The servers $\mathcal{P}_2$ to $\mathcal{P}_4$ as well as any other parties not among the C_i are honest.

The simulator has four main tasks:

- Answer the initialization message by $\mathcal{F}_{VRO}$.
- Simulate $\mathcal{D}$, all honest parties, as well as the functionalities $\mathcal{F}_{ZK}^{\mathcal{R}_{\mathsf{FHE}}}$ and $\mathcal{F}_{bboard}$.
- Handle hash queries by honest callers.
- Handle hash queries by corrupted callers.
- Handle (a subset of the) verification queries by honest callers.

Simulation $\mathcal{S}$ simulates a copy of $\mathcal{D}$ and relays all messages from $\mathcal{Z}$ to $\mathcal{D}$ and vice versa. It simulates all honest parties and gives $\mathcal{Z}$ (through $\mathcal{D}$) the control it expects over the corrupted parties as well as the length of all messages delivered through the network and control over their delivery.

Server Initialization The simulator behaves honestly on behalf of the honest servers during Initialization and also simulates $\mathcal{F}_{bboard}$ honestly. For the invocations of MPC for SIGKeyGen and PRFKeyGen, $\mathcal{S}$ samples the outputs honestly. For the computation of NIWI.Setup, NIWI.SimSetup is used instead. $\mathcal{S}$ delivers outputs to the honest parties once this has been allowed by the adversary. Independent of any actions of the adversary, $\mathcal{S}$ is in possession of the following data:

- SIG key-pairs $(\mathsf{vk}_{Sig,i}, \mathsf{sk}_{Sig,i})$ for all the servers
- all four PRF keys k_1 to k_4
- the simulated NIWI reference string crs and its extraction trapdoor τ.

On input (Hash, sid, q)

1 : **if** $\mathsf{vk} = \bot$ **do**
2 : $\circlearrowleft$ (Init, sid)
3 : **fi**
4 : $((\mathsf{vk}_{Sig,i})_{i\in[4]}, crs) = \mathtt{parse}(\mathsf{vk})$
5 : $(\mathsf{pk}_{FHE}, \mathsf{sk}_{FHE}) \leftarrow \mathsf{FHE.Gen}(1^\lambda; r)$
6 : $c \leftarrow \mathsf{FHE.Enc}(\mathsf{pk}, q)$
7 : **for** $i \in \{1,2,3,4\}$ **do**
8 : $sid' = (\mathcal{P}, \mathcal{P}_i, sid, p_i)$
9 : $p_i = p_i + 1$
10 : $(\mathtt{Prove}, sid', \mathcal{P}_i, \mathsf{pk}_{FHE}, (\mathsf{sk}_{FHE}, r)) \to \mathcal{F}_{ZK}^{\mathcal{R}_{\mathsf{FHE}}}$
11 : $(\mathtt{Hash}, sid, c) \to \mathcal{P}_i$
12 : $(c_{i,j}, c_{\sigma,i,j}) \leftarrow \mathcal{P}_i$
13 : **endfor**
14 : **for** $(i,j) \in \{1,2,3,4\}^2, i \neq j$ **do**
15 : $h_{i,j} = \mathsf{FHE.Dec}(\mathsf{sk}_{FHE}, c_{i,j})$
16 : $\sigma_{i,j} = \mathsf{FHE.Dec}(\mathsf{sk}_{FHE}, c_{\sigma,i,j})$
17 : **endfor**
18 : **for** $i \in \{1,2,3,4\}$ **do**
19 : $h_i = \arg\max_{h\in\mathcal{H}} \left|\{h = h_{j,i} \mid j \in 4 \setminus \{i\}\}\right|$
20 : **endfor**
21 : $h = \bigoplus_{i=1}^4 h_i$
22 : **for** $j \in \{1,2,3,4\}$ **do**
23 : $\Sigma = \{\sigma_{i,j} \mid i \in [4] \setminus \{j\}, \sigma_{i,j} \neq \bot$
24 : $\mathsf{SIG.Verify}(\mathsf{vk}_{Sig,i}, (q, h_j, j), \sigma_{i,j}) = 1\}$
25 : **if** $|\Sigma| < 2$ **do**
26 : **abort**
27 : **fi**
28 : $\{\sigma_1, \sigma_2\} \leftarrow\$ \Sigma$
29 : $\overline{\mathsf{vk}}_j = (\mathsf{vk}_i)_{i\in[4]\setminus\{j\}}$
30 : $x = (\overline{\mathsf{vk}}_j, (q, h_j, j))$
31 : $w = (\sigma_1, \sigma_2)$
32 : $\pi_j = \mathsf{NIWI.Prove}(crs, x, w)$
33 : **endfor**
34 : $\pi = (h_1, h_2, h_3, h_4, \pi_1, \pi_2, \pi_3, \pi_4)$
35 : **return** (HashProof, sid, q, h, π)
36 :

On input (Verify, $sid, q, h, \pi, \mathsf{vk}$)

1 : $(\mathsf{vk}_1, \mathsf{vk}_2, \mathsf{vk}_3, \mathsf{vk}_4, crs) = \mathtt{parse}(\mathsf{vk})$
2 : $(h_1, h_2, h_3, h_4, \pi_1, \pi_2, \pi_3, \pi_4) = \mathtt{parse}(\pi)$
3 : **if** $h \neq \bigoplus_{i=1}^4 h_i$ **do**
4 : **return** (Verified, $sid, q, h, \pi, \mathsf{vk}, 0$)
5 : **fi**
6 : **for** $i \in \{1,2,3,4\}$ **do**
7 : $\overline{\mathsf{vk}}_i = (\mathsf{vk}_j)_{j\in[4]\setminus\{i\}}$
8 : $x = (\overline{\mathsf{vk}}_i, (q, h_i, i))$
9 : **if** $\mathsf{NIWI.Verify}(crs, x, \pi_i) = 0$ **do**
10 : **return** (Verified, $sid, q, h, \pi, \mathsf{vk}, 0$)
11 : **fi**
12 : **endfor**
13 : **return** (Verified, $sid, q, h, \pi, \mathsf{vk}, 1$)

On input (Init, sid)

1 : **for** $i \in \{1,2,3,4\}$ **do**
2 : $(\mathtt{Retrieve}, sid, \mathcal{P}_i) \to \mathcal{F}_{bboard}$
3 : $(\mathtt{Stored}, sid, \mathcal{P}_i, (crs_i, \mathsf{vk}_i)) \leftarrow \mathcal{F}_{bboard}$
4 : **if** $(\mathsf{vk}_i, crs_i) = \bot$ **do**
5 : **abort**
6 : **fi**
7 : $(\mathsf{vk}_{i,j})_{j\in[4]} = \mathtt{parse}(\mathsf{vk}_i)$
8 : **endfor**
9 : **for** $i \in \{1,2,3,4\}$ **do**
10 : $\mathsf{vk}_{Sig,i} = \arg\max_{\mathsf{vk}'} |\{\mathsf{vk}' = \mathsf{vk}_{i,j} \mid j \in [4]\}|$
11 : **fi**
12 : $crs = \arg\max_{crs'} |\{crs' = crs_i \mid i \in [4]\}|$
13 : $\mathsf{vk} = (\mathsf{vk}_{Sig,1}, \mathsf{vk}_{Sig,2}, \mathsf{vk}_{Sig,3}, \mathsf{vk}_{Sig,4}, crs)$
14 : **return** (Key, sid, vk)
15 :

Figure 5.7.: The client algorithms for π_{FHE} from the perspective of a party $\mathcal{P}$.

On input $(\texttt{Hash}, sid, c)$

```
1 :  (Proved, P, pk_FHE) ← F_ZK^{R_FHE}
2 :  c_sk ← FHE.Enc(pk_FHE, sk_Sig)
3 :  C_1 = PRF(·, ·)
4 :  C_2 = SIG.Sign(·, ·; ·)
5 :  for i ∈ {1, 2, 3, 4} \ {n} do
6 :      c_{k,i}   ← FHE.Enc(pk_FHE, k_i)
7 :      c_i       ← FHE.Eval(pk_FHE, C_1, (c_{k,i}, c))
8 :      r         ← SIG.R
9 :      c_r       ← FHE.Enc(pk_FHE, r)
10 :     c_t       ← FHE.Enc(pk_FHE, i)
11 :     c_arg     = (c_sk, (c, c_i, c_t), c_r)
12 :     c_{σ,i}   ← FHE.Eval(pk_FHE, C_2, c_arg)
13 : endfor
14 : return {(c_i, c_{σ,i})}_{i∈[4]\{n}}
```

$\mathsf{Initialize}(\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_4)$

```
1 :  S = {P_1, P_2, P_3, P_4}
2 :  out ← MPC(SIGKeyGen, S)
3 :  (sk_Sig, vk) = parse(out)
4 :  crs ← MPC(NIWI.Setup, S)
5 :  for i ∈ {1, 2, 3, 4} \ {n} do
6 :      k_i ← MPC(PRFKeyGen, S \ {P_i})
7 :  endfor
8 :  (Store, sid, (crs, vk)) → F_bboard
9 :
```

$\mathsf{SIGKeyGen}$

```
1 :  for i ∈ {1, 2, 3, 4} do
2 :      (vk_i, sk_i) ← SIG.Gen(1^λ)
3 :  endfor
4 :  vk = (vk_i)_{i∈[4]}
5 :  for i ∈ {1, 2, 3, 4} do
6 :      out_i = (sk_i, vk)
7 :  endfor
8 :  return (out_i)_{i∈[4]}
```

$\mathsf{PRFKeyGen}$

```
1 :  k ←$ K
2 :  return (k, k, k)
```

Figure 5.8.: The server algorithms for π_{FHE} from the perspective of server $\mathcal{P}_n$.

VRO Initialization Upon receiving the message $(\texttt{Init}, sid)$ from $\mathcal{F}_{VRO}$, $\mathcal{S}$ collects the individual verification keys $\mathsf{vk}_{Sig,i}$ into a single key vk. Let then Prove be defined as shown in Figure 5.9. Essentially, the output of the PRF under the first key k_1 which is unknown to the corrupted server $\mathcal{P}_1$ is changed in such a way that the sum of all four PRF outputs matches the hash h chosen by $\mathcal{F}_{VRO}$. Prove has the verification keys $\mathsf{vk}_{Sig,i}$ of all servers as well as the PRF keys k_2 to k_4 hard-coded. The s provided by the simulator in its SimInfo messages does not contain any information for this instantiation and is thus ignored by Prove. Finally, but in the same activation as receiving the Init message, $\mathcal{S}$ sends the message $(\texttt{Init}, sid, \mathsf{Prove}, \mathsf{vk})$ to $\mathcal{F}_{VRO}$.

Caller Initialization Whenever $\mathcal{S}$ is asked to deliver a Init message to some honest party $\mathcal{P}$, it lets the simulated version of $\mathcal{P}$ initiate the honest protocol for obtaining the verification keys of SIG as well as the NIWI CRS crs by simulating queries to $\mathcal{F}_{bboard}$. Once the simulated $\mathcal{P}$ generates an output of the form $(\texttt{Key}, sid, \mathsf{vk})$, i.e. indicating that in the corresponding real interaction the initialization would have succeeded because the adversary has allowed the delivery of all responses by $\mathcal{F}_{bboard}$ and so forth, $\mathcal{S}$ allows $\mathcal{F}_{VRO}$ to deliver the response to the initialization request by the real $\mathcal{P}$.

Honest Hashing The occurrence of a hash query by an honest caller $\mathcal{P}$ is made known to $\mathcal{S}$ via a message $(\texttt{Hashing}, sid, \mathcal{P}, l)$ where l is the length of the input q. $\mathcal{S}$ now has to simulate the view of the corrupted server $\mathcal{P}_1$ for $\mathcal{D}$ as well as any network communication. The exact steps taken are shown in Figure 5.9. Intuitively, $\mathcal{S}$, instead of generating a key-pair for FHE, proving knowledge of the secret key using $\mathcal{F}_{ZK}^{\mathcal{R}_{\mathsf{FHE}}}$, encrypting q and sending the ciphertext c to the servers, does the following: It does the first step of generating a key-pair, but then $\mathcal{S}$ delivers a Proved message to $\mathcal{P}_1$ without receiving a secret key and encrypts a string of zeroes of the correct length l—it does not know q. The ciphertext containing only zeroes is sent to $\mathcal{P}_1$. $\mathcal{S}$ then sends the message $(\texttt{SimInfo}, sid, \mathcal{P}, \perp)$ back to $\mathcal{F}_{VRO}$. Note that this does not yet allow $\mathcal{F}_{VRO}$ to deliver any response to $\mathcal{P}$.

For the communication with the honest servers, $\mathcal{S}$ simulates messages of the correct length and lets $\mathcal{D}$ decide when to deliver them. Only once all messages to the servers as well as responses by the servers have been delivered in addition to the response by the corrupted server, $\mathcal{S}$ allows $\mathcal{F}_{VRO}$ to deliver the delayable HashProof response for this query to $\mathcal{P}$. This is consistent with the fact that an honest caller only outputs a proof once it receives responses from all four servers. We note that the honest servers only reply if they have been initialized at the time of this query.

Dishonest Hashing Whenever $\mathcal{D}$ initiates a hash query on behalf of a corrupted party C_j for some j, $\mathcal{S}$ has to extract the input q. This has to be possible, however, only when the real protocol would not have aborted the interaction given the same inputs. A real server aborts a query when it does not receive a Proved message including some FHE public key pk_{FHE}. As $\mathcal{S}$ is the party providing $\mathcal{F}_{ZK}^{\mathcal{R}_{\mathsf{FHE}}}$, it can either extract a corresponding secret key sk_{FHE} or it can let the simulated server abort the query. This happens independently for each honest server.

For each $\mathcal{P}_i$ $2 \leq i \leq 4$ and if no abort occurs for that server, $\mathcal{S}$ can use the respective extracted $\mathsf{sk}_{FHE,i}$ to decrypt c_i obtaining some q_i. It sends a message $(\texttt{Hash}, sid, q_i)$ to $\mathcal{F}_{VRO}$

On input $(\texttt{Hashing}, sid, \mathcal{P}, l)$

1 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{FHE.Gen}(1^\lambda)$
2 : $c \leftarrow \mathsf{FHE.Enc}(\mathsf{pk}, 0^l)$
3 : $\mathcal{F}_{ZK}^{\mathcal{R}_{\mathsf{FHE}}} \xrightarrow{(\mathsf{Proven}, sid, \mathcal{P}, \mathsf{pk})} \mathcal{P}_1$
4 : $\mathcal{P} \xrightarrow{(\mathsf{Hash}, sid, c)} \mathcal{P}_1$
5 : **Simulate messages**
6 : **from** $\mathcal{P}$ **to** $\mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_4$
7 : $\mathcal{P} \xleftarrow{\{c_i, c_{\sigma,i}\}_{i=2,3,4}} \mathcal{P}_1$
8 : $(\texttt{SimInfo}, sid, \mathcal{P}, \perp) \rightarrow \mathcal{F}_{VRO}$

$\mathsf{Prove}(q, h, s)$

1 : / Prove knows crs, all vk_j, and all k_i and sk_i for $i > 1$
2 : **for** $i \in \{2, 3, 4\}$ **do**
3 : $\quad h_i = \mathsf{PRF}(k_i, q)$
4 : **endfor**
5 : $h_1 = h \oplus \bigoplus_{i=2}^{4} h_i$
6 : **for** $i \in \{2, 3, 4\}$ **do**
7 : $\quad$ **for** $j \in \{1, 2, 3, 4\}, i \neq j$ **do**
8 : $\quad\quad \sigma_{i,j} \leftarrow \mathsf{SIG.Sign}(\mathsf{sk}_i, (q, h_j, j))$
9 : $\quad$ **endfor**
10 : **endfor**
11 : **for** $i \in \{1, 2, 3, 4\}$ **do**
12 : $\quad \{j_1, j_2\} \subseteq \{1, 2, 3, 4\} \setminus \{1, i\}$
13 : $\quad \overline{\mathsf{vk}}_i = (\mathsf{vk}_j)_{j \in [4] \setminus \{i\}}$
14 : $\quad \pi_i \leftarrow \mathsf{NIWI.Prove}(crs, (\overline{\mathsf{vk}}_i, (q, h_i, i)), (\sigma_{j_1,i}, \sigma_{j_2,i}))$
15 : **endfor**
16 : $\pi = (h_1, h_2, h_3, h_4, \pi_1, \pi_2, \pi_3, \pi_4)$
17 : **return** π

Figure 5.9.: The algorithms used by the simulator.

on behalf of C_j. After receiving the Hashing notification and answering with a SimInfo-message with $s = \perp$, it receives output $(\texttt{HashProof}, sid, q_i, h_i, \pi_i)$. Now $\mathcal{S}$ has to compute a response on behalf of $\mathcal{P}_i$ which makes it congruent with the full hash for q_i being h_i. To achieve this, it changes the output of $\mathsf{PRF}(k_1, q_i)$ such that the sum $\bigoplus_{n=1}^{4} \mathsf{PRF}(k_n, q_i)$ is equal to h_i, i.e. letting it equal $h_i \oplus \bigoplus_{n=2}^{4} \mathsf{PRF}(k_n, q_i)$. Observe that this is only possible because $\mathcal{S}$ knows all the PRF keys.

For $n \in \{2, 3, 4\} \setminus \{i\}$, all evaluations of $\mathsf{PRF}(k_n, \cdot)$ and subsequent signatures are done as before. For k_1, however, first $h_1^i = \mathsf{PRF}(k_1, q_i)$ is computed in the clear. Then, a simulated evaluated ciphertext $c_1^i \leftarrow \mathsf{Sim}(\mathsf{pk}, h_1^i)$ is computed. This c_1^i is then used in the homomorphic evaluation of SIG.Sign in stead of $\mathsf{FHE.Eval}(\mathsf{pk}, \mathsf{PRF}(k_1, \cdot), c_i)$.

Verification Upon receiving a message $(\texttt{Verify}, sid, q, h, \pi, \mathsf{vk}')$ from $\mathcal{F}_{VRO}$, $\mathcal{S}$ behaves like an honest party on the same input. Let b be the last component of the resulting output. $\mathcal{S}$ sends the message $(\texttt{Verified}, sid, b)$ back to $\mathcal{F}_{VRO}$.

5.4.5.2. Indistinguishability

Having given the simulation strategy $\mathcal{S}$ we now proceed to prove that $\mathcal{S}$ is indeed a valid simulator for the dummy adversary $\mathcal{D}$. To show this, we have to prove that for every PPT environment $\mathcal{Z}$ it holds that the ensembles of probability distributions $\mathrm{REAL}_{\pi,\mathcal{D},\mathcal{Z}}$ and

$\text{IDEAL}_{\mathcal{F}_{VRO},\mathcal{S},\mathcal{Z}}$ are computationally indistinguishable. First, we give informal justification for why indistinguishability holds. Afterward, we provide a formal proof.

Honest Hashing It has to be argued that proofs output by $\mathcal{F}_{VRO}$ to honest parties are distributed correctly. We begin with the NIWI proofs. In the real interaction, witness signatures by all four servers are used and chosen at random if more than the two which are required are available. On the other hand, `Prove` always uses signatures by the honest servers as witnesses. This difference will be reduced to the witness-indistinguishability property of NIWI.

Then to the partial hashes h_i for $i \in [4]$. For h_2 up to h_4, the distributions are identical in both cases. For h_1, in a real protocol execution they are produced according to the distribution of $h_1 = \mathsf{PRF}(k_1, q)$ for a randomly and independently chosen key k_1 and input q, while in the ideal protocol they are produced as $h_1 = h \oplus \bigoplus_{j=2}^{4} \mathsf{PRF}(k_j, q)$ for h chosen uniformly at random and independently for each q. By the properties of $\oplus$ this also implies that the h_1 are chosen according to the uniform distribution and are independent of each other. By the defining properties of pseudo-random functions, these two cases are indistinguishable when the probability is taken over the choice of k_1 and what can be described as the random function RF with which $\mathcal{F}_{VRO}$ assigns h's to q's.

At last, it has to be shown that the input q remains hidden from the adversary, i.e. that the view of a corrupted server is indistinguishable between the real and ideal interactions. This will follow from the semantic security of FHE.

Dishonest Hashing We have to show that the view of the simulated corrupted caller is indistinguishable from what it would have seen if it had behaved equally in a real protocol execution. As all interactions with honest servers are independent, we consider a single one.

First, by the soundness properties of $\mathcal{F}_{ZK}^{\mathcal{R}_{\mathsf{FHE}}}$, both the real and simulated honest server abort the interaction if the caller does not know and provide the correct sk_{FHE}. This means that, conditioned on not aborting, $\mathcal{S}$ can extract the correct secret key. Then $\mathcal{S}$ can decrypt q, give it to $\mathcal{F}_{VRO}$, and receive the correct hash h and a proof π.

As we assume that FHE is circuit private, the simulated responses are computationally indistinguishable from the responses in a real protocol execution with the exception of $\mathsf{PRF}(k_1, \cdot)$ being replaced with a truly random function. Because the adversary at no point has access to k_1, this is indistinguishable over the random choice of k_1.

Verification First proofs, output by honest parties in the real interaction are always accepted by the completeness of SIG and NIWI. Inputs whose validity is determined by $\mathcal{S}$ in the ideal interaction are answered identically as $\mathcal{S}$ uses the honest verification algorithm. This leaves proofs that trigger the unforgeability clause of $\mathcal{F}_{VRO}$. We will show that this case occurs with negligible probability by reducing to the extractability of NIWI and the EUF-CMA security of SIG.

After this informal reasoning for why the protocol realizes $\mathcal{F}_{VRO}$, let us now provide a formal proof. We proceed as in the proof in Section 4.2.5.1 by a sequence of interactions in which we gradually move from the real interaction involving an environment $\mathcal{Z}$, the dummy adversary $\mathcal{D}$ and a session of π_{FHE} to the ideal interaction with $\mathcal{Z}$, the simulator

$\mathcal{S}$ and the ideal functionality $\mathcal{F}_{VRO}$. We prove that each step changes the distribution of outputs of $\mathcal{Z}$ only in a computationally indistinguishable manner.

As in the proof in Section 4.2.5.1 we first introduce a challenger C which will be the entity acting on behalf of all honest parties and functionalities in the intermediate interactions. We again let $\text{INT}_{i,C,\mathcal{Z},\mathcal{D}}$ denote the ensemble of random variables representing the outputs of $\mathcal{Z}$ in interaction i. Note that for all ensembles we suppress the superscript indicating the hybrid model we are working within.

Interaction 1: C executes an instance of $\mathcal{Z}$ (with the appropriate auxiliary input) and $\mathcal{D}$. It executes π_{FHE} honestly on behalf of all honest parties which are activated by $\mathcal{Z}$ over the course of the interaction. All ideal functionalities are simulated honestly.

Lemma 5.4.4. *The distribution of outputs of $\mathcal{Z}$ is identically distributed in the real interaction and **Interaction 1**, formally*

$$\text{REAL}_{\pi_{FHE},\mathcal{Z},\mathcal{D}} = \text{INT}_{1,C,\mathcal{Z},\mathcal{D}} \quad \square$$

In the next step, we prepare the behavior of honest parties during hash queries for a later step. We do this by removing the proofs performed by honest parties using $\mathcal{F}_{ZK}^{\mathcal{R}_{\mathsf{FHE}}}$. This is necessary to later be able to reduce to the semantic security of FHE as there our reduction will not possess the necessary secret key.

Interaction 2: C behaves as in **Interaction 1**, except that honest parties no longer provide witnesses to $\mathcal{F}_{ZK}^{\mathcal{R}_{\mathsf{FHE}}}$, but C still delivers `Proved` messages as if witnesses had been received.

Lemma 5.4.5. *The distribution of outputs of $\mathcal{Z}$ is identically distributed between **Interaction 1** and **Interaction 2**, formally*

$$\text{INT}_{1,C,\mathcal{Z},\mathcal{D}} = \text{INT}_{2,C,\mathcal{Z},\mathcal{D}}$$

Proof. This follows by the unobservability of messages sent by honest parties to ideal functionalities. We are using the perfect simulatability provided by $\mathcal{F}_{ZK}^{\mathcal{R}_{\mathsf{FHE}}}$ which may degrade to computational simulatability when instantiating $\mathcal{F}_{ZK}^{\mathcal{R}_{\mathsf{FHE}}}$. $\square$

Next, we further reduce the reliance of honest callers to know the secret key sk for the public key pk which they use to encrypt the messages to the servers. Instead of computing proofs from the responses of the honest servers, callers recompute what is contained in the responses themselves.

Interaction 3: C behaves as in **Interaction 2**, except that it lets honest callers ignore the responses by honest servers and compute the values which are contained therein according to the honest protocol using the PRF keys k_i and signing keys $\text{sk}_{Sig,i}$ of the honest servers.

Lemma 5.4.6. *The distribution of outputs of $\mathcal{Z}$ is identically distributed between **Interaction 2** and **Interaction 3**, formally*

$$\mathrm{INT}_{2,C,\mathcal{Z},\mathcal{D}} = \mathrm{INT}_{3,C,\mathcal{Z},\mathcal{D}}$$

Proof. The h_i contained in the responses can be perfectly recomputed using k_i. For the signatures, the honest servers honestly compute signatures using fresh randomness. The callers in **Interaction 3** do the same and so the final proofs are identically distributed. □

Now we alter the interaction between corrupted callers and honest servers. To then be able to use the security of PRF we replace the fully-homomorphic evaluation of $\mathsf{PRF}(k_1, \cdot)$ by a computation in the clear followed by a use of the circuit privacy simulator Sim for FHE. This is necessary as (1) a truly random function does not have a sufficiently succinct representation that would allow for homomorphic evaluation, (2) even then, a non-circuit private FHE scheme may allow a corrupted caller to differentiate between a ciphertext which was the result of evaluating PRF as opposed to the random function.

Interaction 4: C behaves as in **Interaction 3**, except that it lets honest servers when interacting with a corrupted caller $\mathcal{P}$, instead of homomorphically evaluating $\mathsf{PRF}(k_1, \cdot)$ on the received ciphertext c, do the following. First, they use the sk which corresponds to the pk contained in the `Proved` message which they have received from $\mathcal{F}_{ZK}^{\mathcal{R}}$ and which is extracted from the message by $\mathcal{P}$ to $\mathcal{F}_{ZK}^{\mathcal{R}}$ to compute the plaintext $q = \mathsf{FHE.Dec}(\mathsf{sk}, c)$. Then they evaluate $\mathsf{PRF}(k_1, \cdot)$ on q in the clear to obtain h_1. Lastly, they then simulate an evaluated ciphertext c_1 as $c_1 \leftarrow \mathsf{Sim}(\mathsf{pk}, h_1)$. This c_1 replaces the result of $\mathsf{FHE.Eval}(\mathsf{pk}, \mathsf{PRF}(k_1, \cdot), c)$ in the subsequent computation and in the response to $\mathcal{P}$.

Lemma 5.4.7. *The distribution of outputs of $\mathcal{Z}$ in **Interaction 3** is statistically close to the distribution in **Interaction 4**, formally*

$$\Delta\left(\mathrm{INT}_{3,C,\mathcal{Z},\mathcal{D}}, \mathrm{INT}_{4,C,\mathcal{Z},\mathcal{D}}\right) \leq \mathsf{negl}(\lambda)$$

for some negligible function $\mathsf{negl}(\lambda)$.

Proof. We use the statistical semi-honest circuit privacy we have assumed of FHE. For this, let p be a polynomial upper bound on the number of hash queries initiated by $\mathcal{Z}$ on behalf of corrupted callers. Then $3p$ is an upper bound on the number of executions of Sim in **Interaction 4**. The factor 3 stems from the fact that each of the three honest servers runs Sim once during each query.[9]

If we were working with computational circuit privacy, we would be stuck now as we are using indistinguishability not of a single pair of distributions, but of p distributions of the form

$$\{\mathsf{Sim}(\mathsf{pk}, \mathsf{PRF}(k_1, q_i))\} \overset{c}{\approx} \{\mathsf{FHE.Eval}(\mathsf{pk}, \mathsf{PRF}(k_1, \cdot), c_i)\}$$

[9] While all of them are on the same input, running Sim only once and using the same result for each honest server would trivially be detectable.

for $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{FHE.Gen}(1^\lambda)$ some inputs q_i and $c_i \leftarrow \mathsf{FHE.Enc}(\mathsf{pk}, q_i)$. This would require us to make a polynomial number of hybrid-steps without being able to reduce to the indistinguishability of a single pair of distributions as required for the standard hybrid-argument. Luckily, our definition of circuit privacy is statistical and so we do not encounter any such complications. Going from **Interaction 3** to **Interaction 4**, we gradually replace invocations of Eval by those of Sim.

For each block of three evaluations of $\mathsf{PRF}(k_1, \cdot)$ in **Interaction 3**, let the first be executed by $\mathcal{P}_2$, the second by $\mathcal{P}_3$ and the third by $\mathcal{P}_4$. For $j \in [3p]$ let G_j be the intermediate interaction where the first j evaluations

$$\mathsf{Eval}\left(\mathsf{pk}_j, \mathsf{PRF}(k_1, \cdot), c_j\right) \tag{5.3}$$

where pk_j is the public key sent by the caller in the j'th hash query by a corrupted caller and c_j is the sent ciphertext are replaced by

$$\mathsf{Sim}\left(1^\lambda, \mathsf{pk}_j, \mathsf{PRF}(k_1, q_j)\right) \tag{5.4}$$

where $q_j = \mathsf{Dec}(\mathsf{sk}_j, c_j)$ for the extracted secret key sk_j. Note that we can assume $q_j \neq \perp$ due to our assumption regarding FHE having a full ciphertext space.

Clearly, G_0 is identical to **Interaction 3** while G_{3p} is identical to **Interaction 4**. Furthermore, using a sample which has either been generated according to (5.3) or (5.4) allows simulating an interaction of $\mathcal{Z}$ which is either distributed according to G_k (if (5.4) is used) or G_{k-1} (if (5.3) is used). By the properties of statistical distances, the statistical distance between the output of $\mathcal{Z}$ between G_k and G_{k-1} is then bounded by the statistical distance between the two distributions (5.3) and (5.4), which is negligible with bound $\mathsf{negl}(\lambda)$ by the statistical circuit privacy of FHE. We can thus upper bound the statistical distance between G_0 and G_{3p} as

$$\Delta(\mathrm{G}_0, \mathrm{G}_{3p} \leq 3p\,\mathsf{negl}(\lambda) \leq \mathsf{negl}(\lambda)'$$

for some additional negligible function $\mathsf{negl}(\lambda)'$.

What remains to be argued is why it was justified to use the *semi-honest* circuit privacy in our malicious context. The reason for this is two-fold. First, by letting callers prove knowledge of the secret key for the public key they send to servers ensures that the public key is well-formed. The second difference between semi-honest and malicious circuit privacy concerns well-formed ciphertexts. As per the discussion in Chapter 2, forego this problem by assuming that the ciphertext space is *full*, valid ciphertexts are efficiently recognizable and so invalid ciphertexts will be rejected also in the real interaction. □

Remark 5.4.8. If the protocol is instantiated with FHE having the property that evaluated ciphertexts are statistically close to fresh encryptions of the contained value, then the present step becomes trivial. In particular, no simulator has to be used.

We can now finally use the security of PRF by replacing the instance using the key k_1 not known to the adversary by a random function RF.

Interaction 5: C behaves as in **Interaction 4**, except that all occurrences of $\mathsf{PRF}(k_1, \cdot)$, i.e. as arguments to Sim, are replaced with $\mathsf{RF}(\cdot)$ for a common random function RF.

Lemma 5.4.9. *The distribution of outputs of $\mathcal{Z}$ in **Interaction 4** is computationally indistinguishable from the distribution in **Interaction 5**, formally*

$$\mathrm{INT}_{4,C,\mathcal{Z},\mathcal{D}} \overset{c}{\approx} \mathrm{INT}_{5,C,\mathcal{Z},\mathcal{D}}$$

Proof. We reduce to the security of PRF by constructing an adversary $\mathcal{B}$ on the PRF security of PRF from any distinguisher $\mathcal{A}$ for the output distributions of $\mathcal{Z}$ produced in **Interaction 4** and **Interaction 5**.

$\mathcal{B}$ runs a simulation of $\mathcal{Z}$ and $\mathcal{D}$. It uses its oracle O, which is either given by a random function RF or by $\mathsf{PRF}(k, \cdot)$ for k drawn uniformly at random from the key-space of PRF, to replace all occurrences of $\mathsf{PRF}(k_1, \cdot)$. All other actions of honest parties are according to **Interaction 4**. When $\mathcal{Z}$ produces its output and halts, the output is given as input to the distinguisher $\mathcal{A}$. When $\mathcal{A}$ produces its decision b, $\mathcal{B}$ gives it to its own challenger and halts.

When O is given by $\mathsf{PRF}(k, \cdot)$ the simulation by $\mathcal{B}$ is of **Interaction 4** and when O is given by RF it is a simulation of **Interaction 5**. For the advantages $\mathsf{Adv}^{\mathrm{prf}}_{\mathcal{B},\mathsf{PRF}}(\lambda)$ of $\mathcal{B}$ and $\mathsf{Adv}^{\mathrm{dist}}_{\mathcal{A},\mathrm{INT}_{4,C,\mathcal{Z},\mathcal{D}},\mathrm{INT}_{5,C,\mathcal{Z},\mathcal{D}}}(\lambda)$ of $\mathcal{A}$ it holds that

$$\mathsf{Adv}^{\mathrm{dist}}_{\mathcal{A},\mathrm{INT}_{4,C,\mathcal{Z},\mathcal{D}},\mathrm{INT}_{5,C,\mathcal{Z},\mathcal{D}}}(\lambda) \leq \mathsf{Adv}^{\mathrm{prf}}_{\mathcal{B},\mathsf{PRF}}(\lambda) \tag{5.5}$$

But by assumption on the security of PRF

$$\mathsf{Adv}^{\mathrm{prf}}_{\mathcal{B},\mathsf{PRF}}(\lambda) = \mathsf{negl}(\lambda) \tag{5.6}$$

for some negligible function $\mathsf{negl}(\lambda)$. Taken together, Equation (5.5) and Equation (5.6) as well as the fact that the above construction works for any PPT machine $\mathcal{A}$, imply that for any PPT distinguisher $\mathcal{E}$ there exists a negligible function $\mathsf{negl}(\lambda)'$ such that

$$\mathsf{Adv}^{\mathrm{dist}}_{\mathcal{E},\mathrm{INT}_{4,C,\mathcal{Z},\mathcal{D}},\mathrm{INT}_{5,C,\mathcal{Z},\mathcal{D}}}(\lambda) \leq \mathsf{negl}(\lambda)' \tag{5.7}$$

This shows the lemma. □

Next, we make the proof produced by honest parties fully independent of the responses they receive from the corrupted caller. While so far the selection of which valid signatures to use to compute NIWI proofs was made randomly (or equivalently using some fixed deterministic process), we now only consider signatures returned by the honest servers. Indistinguishability will follow from the witness-indistinguishability of NIWI where we have to use a hybrid-argument as there may be polynomially many proofs requested from honest parties. As WI is an indistinguishability notion, this easily follows from the definition for a single proof.

Interaction 6: C behaves as in **Interaction 5**, except that honest parties, upon receiving input $(\texttt{Hash}, sid, q)$, execute the protocol as in **Interaction 5** until they are supposed to compute proofs using NIWI.Prove. They exclude the signatures they have received from $\mathcal{P}_1$ from the sets Σ of valid signatures, i.e. only use signatures under the verification keys belonging to honest servers (and which they already compute by themselves). As there are three signatures valid under such keys for $(q, h_1, 1)$ which may be used as witnesses, but only two are required, those valid under $\mathsf{vk}_{Sig,2}$ and $\mathsf{vk}_{Sig,3}$ are used.

Lemma 5.4.10. *The distribution of outputs of $\mathcal{Z}$ in **Interaction 5** is computationally indistinguishable from the distribution in **Interaction 6**, formally*

$$\mathrm{INT}_{5,C,\mathcal{Z},\mathcal{D}} \overset{c}{\approx} \mathrm{INT}_{6,C,\mathcal{Z},\mathcal{D}}$$

Proof. We reduce to the witness-indistinguishability of NIWI by constructing a successful adversary $\mathcal{B}$ on the witness-indistinguishability of NIWI from any successful distinguisher $\mathcal{A}$ for the output distributions of $\mathcal{Z}$ produced in **Interaction 5** or **Interaction 6**. As stated above, there are potentially polynomially many proofs by honest callers which have to be simulated, but an attacker on the witness-indistinguishability of NIWI is only allowed to request a single proof. We employ a standard hybrid-argument.

$\mathcal{B}$ runs a simulation of $\mathcal{Z}$ and $\mathcal{D}$. It chooses uniformly at random an index $i \leq 4\,p(\lambda)$ where p is a polynomial upper bound on the number of hash queries for honest callers made by $\mathcal{Z}$. The factor 4 is required because each proof contains four NIWI proofs which have to be replaced incrementally.

When $\mathcal{B}$ is asked to simulate the j'th NIWI proof which occurs in the $\lfloor j/4 \rfloor$'th hash query by an honest caller on some input q, $\mathcal{B}$ simulates an honest caller according to **Interaction 5**, but whenever a NIWI proof π for some statement x is required, it computes it as follows:

- If $j < i$: Run NIWI.Prove with a witness according to **Interaction 5**
- If $j = i$: Let w_1 be a witness according to **Interaction 5** and w_2 a witness according to **Interaction 6**. Give (x, w_1, w_2) to the WI challenger and use the received proof π.
- If $j > i$: Run NIWI.Prove with a witness according to **Interaction 6**.

At the end of the simulation, any output produced by $\mathcal{Z}$ is given as input to the distinguisher $\mathcal{A}$. Finally, $\mathcal{B}$ gives the output bit of $\mathcal{A}$ to its challenger.

Let G_i be the game where the first i NIWI proofs contained in proofs by honest callers are generated according to **Interaction 5** and the remaining proofs are generated according to **Interaction 6**. Clearly G_0 is identical to **Interaction 5** and $\mathrm{G}_{4\,p(\lambda}$ is identical to

Interaction 6. Also, depending on the challenge bit b in the witness-indistinguishability game and the index i randomly chosen by $\mathcal{B}$, an instance of G_{i-b} is simulated.

We may now apply the standard hybrid-argument to show that

$$\mathsf{Adv}^{\mathrm{dist}}_{\mathcal{A},\mathrm{INT}_{5,C,\mathcal{Z},\mathcal{D}},\mathrm{INT}_{6,C,\mathcal{Z},\mathcal{D}}}(\lambda) \leq 4\, p(\lambda)\ \mathsf{Adv}^{\mathrm{wit-ind}}_{\mathcal{B},\mathsf{NIWI}}(\lambda) \tag{5.8}$$

But by the assumption on the security of NIWI

$$\mathsf{Adv}^{\mathrm{wit-ind}}_{\mathcal{B},\mathsf{NIWI}}(\lambda) = \mathsf{negl}(\lambda) \tag{5.9}$$

for some negligible function $\mathsf{negl}(\lambda)$. Taken together, Equation (5.8) and Equation (5.9) as well as the fact that the above construction works for any PPT machine $\mathcal{A}$, imply that for any PPT distinguisher $\mathcal{E}$ there exist negligible functions $\mathsf{negl}'(\lambda)$ and $\mathsf{negl}''(\lambda)$ such that

$$\mathsf{Adv}^{\mathrm{dist}}_{\mathcal{E},\mathrm{INT}_{5,C,\mathcal{Z},\mathcal{D}},\mathrm{INT}_{6,C,\mathcal{Z},\mathcal{D}}}(\lambda) \leq 4\, p(\lambda)\ \mathsf{negl}'(\lambda) \leq \mathsf{negl}''(\lambda) \tag{5.10}$$

This proves the lemma. □

The next step again concerns how honest callers behave. This time we wish to show that the corrupted server has no information on q except its length $\|q\|$ as is required by the definition of $\mathcal{F}_{VRO}$. We achieve this by using the semantic security of FHE, i.e. instead of sending an encryption of q we send an encryption of the fixed value $0^{\|q\|}$, and this will go undetected with overwhelming probability. As the simulator only has to simulate the view of the corrupted server when it is interacting with an honest caller, only their code has to be changed.

Interaction 7: C behaves as in **Interaction 6**, except that honest callers on input q compute the ciphertext c which they send to the corrupted server $\mathcal{P}_1$ as $c \leftarrow \mathsf{FHE.Enc}(\mathsf{pk}, 0^{\|q\|})$ instead of $c \leftarrow \mathsf{FHE.Enc}(\mathsf{pk}, q)$.

Lemma 5.4.11. *The distribution of outputs of $\mathcal{Z}$ in **Interaction 6** is computationally indistinguishable from the distribution in **Interaction 7**, formally*

$$\mathrm{INT}_{6,C,\mathcal{Z},\mathcal{D}} \overset{c}{\approx} \mathrm{INT}_{7,C,\mathcal{Z},\mathcal{D}}$$

Proof. We reduce to the IND-CPA security of FHE.

Again we are given a PPT distinguisher $\mathcal{A}$ for the distributions of the outputs of $\mathcal{Z}$ produced when engaged either in a random execution of **Interaction 6** or **Interaction 7** and construct from it an adversary $\mathcal{B}$ for the IND-CPA game for FHE. We use the "real-or-zero" formulation of the notion where the adversary is given an oracle $\mathcal{O}$, which on input a message m returns either an encryption of m or an encryption of $0^{\|m\|}$.

$\mathcal{B}$ is given a public key pk by its challenger C. It then runs a simulation of $\mathcal{Z}$ and $\mathcal{D}$ according to **Interaction 6**. As $\mathcal{B}$ may be asked to simulate up to polynomially many

hash queries by honest callers, we again have to invoke a hybrid-argument and thus let $\mathcal{B}$ choose an index $1 \leq i \leq p(\lambda)$ where p is a polynomial upper bound on the number of hash queries for honest callers made by $\mathcal{Z}$.

When $\mathcal{B}$ is asked to simulate the j'th hash query by an honest caller on input q, $\mathcal{B}$ creates the ciphertext c and the FHE public key pk with which it engages with $\mathcal{P}_1$ as follows:

- If $j < i$: $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{FHE.Gen}(1^\lambda)$, $c \leftarrow \mathsf{FHE.Enc}(\mathsf{pk}, 0^{\|q\|})$
- If $j = i$: Use the pk from the IND-CPA challenger and request an encryption c of q from the encryption oracle
- If $j > i$: $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{FHE.Gen}(1^\lambda)$, $c \leftarrow \mathsf{FHE.Enc}(\mathsf{pk}, q)$

The proof π is produced as in **Interaction 6**, i.e. only based on signatures by honest servers and the correct PRF and RF outputs, and hence independently of any ciphertext contained in the response by $\mathcal{P}_1$. In particular, knowledge of sk is not required for any index j and is indeed unknown to $\mathcal{B}$ for $j = i$.

In any case, the output produced by $\mathcal{Z}$ is given as input to the distinguisher $\mathcal{A}$. Finally, $\mathcal{B}$ gives the output bit of $\mathcal{A}$ to its challenger.

Let G_i be the game where for the first i queries an encryption of $0^{\|q\|}$ is used and the rest of the queries use an encryption of q. Clearly G_0 is identical to **Interaction 6** and $G_{p(\lambda}$ is identical to **Interaction 7**. Also, depending on the challenge bit b in the IND-CPA game and for index i chosen randomly by $\mathcal{B}$, an instance of G_{i-b} is simulated.

We may now apply the standard hybrid-argument to show that

$$\mathsf{Adv}^{\mathrm{dist}}_{\mathcal{A},\mathrm{INT}_{6,C,\mathcal{Z},\mathcal{D}},\mathrm{INT}_{7,C,\mathcal{Z},\mathcal{D}}}(\lambda) \leq p(\lambda)\ \mathsf{Adv}^{\mathrm{roz}}_{\mathcal{B},\mathsf{FHE}}(\lambda) \tag{5.11}$$

But by assumption on the security of FHE

$$\mathsf{Adv}^{\mathrm{roz}}_{\mathcal{B},\mathsf{FHE}}(\lambda) = \mathsf{negl}(\lambda) \tag{5.12}$$

for some negligible function $\mathsf{negl}(\lambda)$. Taken together, Equation (5.11) and Equation (5.12) as well as the fact that the above construction works for any PPT machine $\mathcal{A}$, imply that for any PPT distinguisher $\mathcal{E}$ there exist negligible functions $\mathsf{negl}'(\lambda)$ and $\mathsf{negl}''(\lambda)$ such that

$$\mathsf{Adv}^{\mathrm{dist}}_{\mathcal{E},\mathrm{INT}_{6,C,\mathcal{Z},\mathcal{D}},\mathrm{INT}_{7,C,\mathcal{Z},\mathcal{D}}}(\lambda) \leq p(\lambda)\ \mathsf{negl}'(\lambda) = \mathsf{negl}''(\lambda) \tag{5.13}$$

This proves the lemma. □

So far, we have not altered the initialization phase of the servers, i.e. it has remained totally honest as in the real interaction. However, to be able to use the extractability of NIWI, we now exchange the honestly generated CRS *crs* using NIWI.Setup for a CRS *crs′* with accompanying backdoor τ as produced by NIWI.ExtSetup.

Interaction 8: C behaves as in **Interaction 7**, except that during the initialization phase by the servers, instead of sampling the CRS *crs* output to all servers using NIWI.Setup computes $(crs, \tau) \leftarrow \mathsf{NIWI.ExtSetup}(1^\lambda)$ and outputs *crs* to all servers while keeping τ hidden.

Lemma 5.4.12. *The distribution of outputs of $\mathcal{Z}$ in **Interaction 7** is computationally indistinguishable from the distribution in **Interaction 8**, formally*

$$\mathrm{INT}_{7,C,\mathcal{Z},\mathcal{D}} \stackrel{c}{\approx} \mathrm{INT}_{8,C,\mathcal{Z},\mathcal{D}}$$

Proof. This immediately follows from the first requirement of the *extractability* condition as stated in Definition 2.5.13. □

Having given C access to the extraction backdoor τ in the last step, we are now able to make use of it to reduce to the unforgeability of SIG. Intuitively, we wish to show that to produce a forged proof a signature had to be forged. Showing this is complicated by the fact that proofs do not contain plain signatures, but NIWI proofs instead. Supposedly it is difficult for the adversary to produce such a proof without forging a signature for one of the honest servers, but a reduction has to actually reconstruct this forgery in full. We show that this can be done using τ.

Interaction 9: C behaves as in **Interaction 8**, except that honest parties, upon receiving input $(\texttt{Verify}, sid, q, h, \pi, \mathsf{vk})$ (i.e. for the correct vk), reject this proof if either q was never before queried or $h \neq \mathsf{RF}(q) \oplus \bigoplus_{i=2}^{4} \mathsf{PRF}(k_i, q)$ by responding with $(\texttt{Verified}, sid, q, h, \pi, \mathsf{vk}, 0)$. The set Q of queried inputs contains all those q such that either an honest caller received input $(\texttt{Hash}, sid, q)$ from the environment one of the honest servers received a ciphertext c decrypting to q under the secret key sk sent to $\mathcal{F}_{ZK}^{\mathcal{R}}$ as input from some corrupted caller and generated a response (i.e. the server was initialized, the caller performed a valid proof using $\mathcal{F}_{ZK}^{\mathcal{R}}$).

Lemma 5.4.13. *The distribution of outputs of $\mathcal{Z}$ in **Interaction 8** is computationally indistinguishable from the distribution in **Interaction 9**, formally*

$$\mathrm{INT}_{8,C,\mathcal{Z},\mathcal{D}} \stackrel{c}{\approx} \mathrm{INT}_{9,C,\mathcal{Z},\mathcal{D}}$$

Proof. We reduce to the EUF-CMA security of SIG by constructing an adversary $\mathcal{B}$ on the EUF-CMA-security from $\mathcal{Z}$.

$\mathcal{B}$ receives a verification key vk from its challenger and is provided with an oracle Sign which on input a message m responds with a signature σ for m. It runs a simulation

of $\mathcal{Z}$ and $\mathcal{D}$. As $\mathcal{B}$ is faced with having to generate signatures for three different key-pairs, it randomly chooses $i \leftarrow\!\!\$ \{2,3,4\}$ and sets vk as the verification key for party $\mathcal{P}_i$ during initialization. It generates two SIG key-pairs for the remaining servers. Any time a signature for party $\mathcal{P}_i$ on message m is required, $\mathcal{B}$ obtains it from its oracle. Otherwise, the servers and honest callers are simulated consistently with **Interaction 8**.

Now, whenever $\mathcal{Z}$ submits a valid proof (q, h, π) for verification to an honest party (together with the correct verification key vk, other verification queries are answered as before), $\mathcal{B}$ does the following, using its knowledge of the trapdoor τ:

- $(h_1, h_2, h_3, h_4, \pi_1, \pi_2, \pi_3, \pi_4) = \mathtt{parse}(\pi)$
- For $j \in \{1,2,3,4\}$ do:
  - $\overline{\mathsf{vk}_j} = (\mathsf{vk}_k)_{k \in [4] \setminus \{i\}}$
  - $x = (\overline{\mathsf{vk}_j}, (q, h_j, j))$
  - $(\sigma_1, \sigma_2) = \mathsf{NIWI.Extract}(\tau, x, \pi)$
  - For $m \in \{1,2\}$, if $\mathsf{SIG.Verify}(\mathsf{vk}_i, \sigma_m, (q, h_j, j)) = 1$ and $\mathcal{B}$ never submitted (q, h_j, j) to its signing oracle, return $((q, h_j, j), \sigma_m)$ to the EUF-CMA challenger.
  - For $m \in \{1,2\}$ and $n \in \{2,3,4\}$, if $\mathsf{SIG.Verify}(\mathsf{vk}_n, \sigma_m, (q, h_j, j)) = 1$ and $\mathcal{B}$ never signed (q, h_j, j) using $\mathsf{sk}_{Sig,n}$, $\mathcal{B}$ aborts the simulation and halts without output.

In short, $\mathcal{B}$ extracts a witness from every proof submitted for verification and checks whether it contains a signature valid under $\mathsf{vk} = \mathsf{vk}_{Sig,i}$ and for a message which was never submitted to the signing oracle. Should such a signature be found it is returned to the challenger and $\mathcal{B}$ wins the EUF-CMA game and halts, thereby also halting the simulation. Each extracted signature is also checked for validity under the honest servers verification keys apart from $\mathsf{vk}_{Sig,i}$ and $\mathcal{B}$ halts without output in that case. As long as no such signature is found the simulation continues.

When $\mathcal{Z}$ produces output and halts, $\mathcal{B}$ halts without any output. Conditioned on the fact that the simulation ends with $\mathcal{Z}$ producing output and is not prematurely aborted by $\mathcal{B}$, the view of the interaction for $\mathcal{Z}$ is identical to **Interaction 8**. We also claim that, as long as $\mathcal{B}$ does not abort the simulation, the change made to the protocol when going from **Interaction 8** to **Interaction 9** is not visible to $\mathcal{Z}$ and so the view is also consistent with **Interaction 9**.

To prove (the contrapositive of) this claim we have to show that when some proof is rejected according to **Interaction 0** when it would have been valid in **Interaction 8** (i.e. when **Interaction 8** and **Interaction 9** differ), $\mathcal{B}$ does abort the simulation, i.e. finds a forgery under either the verification key it received from its challenger or under one of the other honest servers keys.

A proof (q, h, π) is rejected when either (1) h is the wrong hash for q with respect to the $\{k_i\}_{2\leq i\leq 4}$, and $\mathsf{RF}(q)$ or (2) q was never queried before we have defined q as having been queried if it was either queried through an honest party or the ciphertext c provided by a corrupted caller to one of the servers decrypts to q using the decryption key provided to $\mathcal{F}_{ZK}^{\mathcal{R}_{\mathsf{FHE}}}$.

We begin with (1). If h is the "wrong" hash, then at least one of the h_i contained in π is not equal to either $\mathsf{RF}(q)$, if $i = 1$, or $\mathsf{PRF}(k_i, q)$, if $i > 1$. In either case, no honest party ever signed (q, h_i, i) using some $\mathsf{sk}_{Sig,j}$ for $j > 1$. On the other hand, any valid witness w for the statement $x = ((\mathsf{vk}_j)_{j\in[4]\setminus\{i\}}, (q, h_i, i))$ corresponding to π_i, i.e. such that $(x, w) \in \mathcal{R}_{\mathsf{SIG}}$ (see (5.2)), will have to contain at least one valid signature for (q, h_i, i) under such a signing key (at most one can come from the corrupted server). Thus, $\mathcal{B}$ will extract this signature and halt the simulation in this case.

The way we define when an input q was queried, (2) means that no honest party ever signed a message with the first part equal to q and so for any $i \in \{1, 2, 3, 4\}$, from π_i a signature valid under an honest parties verification key for message (q, h_i, i) can be extracted. Again this will lead to $\mathcal{B}$ halting the simulation. Together this proves the claim.

By the EUF-CMA-security of SIG it holds that

$$\mathsf{Adv}^{\mathrm{euf-cma}}_{\mathcal{B},\mathsf{SIG}}(\lambda) = \mathsf{negl}(\lambda) \tag{5.14}$$

for a negligible function $\mathsf{negl}(\lambda)$. As $\mathcal{B}$ also terminates the simulation whenever a signature for another honest server is forged, but the choice of server for which vk is used is information-theoretically hidden from $\mathcal{Z}$, the probability of a simulation being terminated is at most $3 \cdot \mathsf{negl}(\lambda)$. Together with the above reasoning we see that the view of $\mathcal{Z}$ in **Interaction 8** and **Interaction 9** only differs with negligible probability which provides an upper bound for the distinguishing advantage of any potential distinguisher $\mathcal{A}$ for the output distributions of $\mathcal{Z}$ in these two interactions by the difference lemma. □

At this point, we could in theory switch back from a simulated CRS to an honestly generated CRS. The reason for this is that only the reduction from **Interaction 8** to **Interaction 9** was required to make use of the extraction trapdoor, but C can execute either interaction without knowledge of it. This would have allowed us to let $\mathcal{S}$ behave totally honestly with respect to the initialization of the servers. As it saves us a step and still yields a valid simulator, we do not do this. Instead, we observe the remaining differences between **Interaction 8** and the ideal interaction. These are mostly cosmetic if we observe that C in **Interaction 9** essentially executes Prove as provided by $\mathcal{S}$ in the ideal interaction to generate proofs and lets honest parties verify proofs as done by $\mathcal{F}_{VRO}$.

Lemma 5.4.14. *The distribution of outputs of $\mathcal{Z}$ in **Interaction 9** is computationally indistinguishable from the ideal interaction, formally*

$$\mathrm{INT}_{9,C,\mathcal{Z},\mathcal{D}} \overset{c}{\approx} \mathrm{IDEAL}_{\mathcal{F}_{Vkern-1ptRO},\mathcal{Z},\mathcal{S}}$$

Proof. We first consider inputs $(\texttt{Verify}, sid, q, h, \pi, \mathsf{vk}')$. If $\mathsf{vk}' \neq \mathsf{vk}$, in both **Interaction 9** and the ideal interaction the same honest verification algorithm as in the real interaction is used to determine the result. Let us thus from now on assume $\mathsf{vk}' = \mathsf{vk}$. In the ideal

interaction, if π was previously output by $\mathcal{F}_{VRO}$, it will always verify later. The same is true in **Interaction 9** by the perfect completeness of SIG and NIWI. For proofs not output by $\mathcal{F}_{VRO}$, in both cases these are rejected if either q was never hashed before or h is not the correct hash for q. The set Q of previously hashed q in the ideal interaction is identical to how we have defined it in **Interaction 9** for the following reason. On the one hand, hash queries to honest parties in **Interaction 9** immediately map onto hash queries made to $\mathcal{F}_{VRO}$ in the ideal interaction. On the other hand, interactions by honest servers with corrupted callers on input c decrypting to q lead to a hash query for q made by $\mathcal{S}$ on behalf of the corrupted caller. Hence these sets are equal and proofs for inputs outside of Q or in Q but the proof is for the wrong hash are rejected in both cases. In the remaining cases, the honest verification algorithm is used in both interactions.

Switching to hash queries and considering corrupted callers first, we observe that the only difference consists in how h_1 is computed. In **Interaction 9** it is computed as $\mathsf{RF}(q)$ for a truly random function RF while in the ideal interaction it is computed by first choosing an independent random h and then setting $h_1 = h \oplus \bigoplus_{i=2}^{4}$, but these two distributions are identical. Thus the views of corrupted callers are identical in both games.

The distributions of proofs are also identical in both cases. Having already established that the h_i are identically distributed, we consider the NIWI proofs π_i. First, the selection of servers from which witness signatures are selected is deterministic and identical in both games, and Prove as well as an honest caller in **Interaction 9** both generate fresh signatures using uniform randomness using the respective signing keys of the honest servers. Thus also these distributions are identical.

This leaves the view of the corrupted server when interacting with an honest caller. In both cases, it first receives a Proved message from $\mathsf{F}_{ZK}^{\mathcal{R}_{\mathsf{FHE}}}$ containing some public FHE key pk as well as an encryption of zeroes of equal length. These distributions are hence identical also. □

Remark 5.4.15. We have silently ignored the case where $\mathcal{F}_{VRO}$ is unable to produce a proof π for input q with hash h, because π has already been marked as invalid for (q, h). In the present situation, for this to be the case the environment would have to have predicted π and tried to verify (q, h, π) before making the first hash query for q. By the extractability of NIWI, this requires forging a signature and so occurs with negligible probability. We could have incorporated this into the proof by doing another step where we reduce to the security of SIG.

Collecting these lemmas, we have shown that all of these finitely many steps result in computationally indistinguishable output distributions for an arbitrary environment $\mathcal{Z}$. By the properties of computational indistinguishability, this means that also the extremal distributions are indistinguishable. This proves the following theorem.

Theorem 5.4.16. *The protocol π_{FHE} UC-realizes $\mathcal{F}_{VRO}$ under static corruption of a single server and static corruption of an arbitrary number of callers in the $(\mathcal{F}_{bboard}, \mathcal{F}_{ZK}^{\mathcal{R}_{\mathsf{FHE}}}, \mathcal{F}_{SMT})$-hybrid model.*

5.4.6. Reducing to Semantic Security

At first sight, the use of NIWI seems to be unnecessary and expensive. Why is it not enough to include the raw signatures in proofs? In the following, we explain why the use of NIWI was required and what problem it solves. We also give an alternative solution that does allow proofs to consist of signatures themselves. We begin by motivating the problem. Then we give two possible categories of solutions. In the end, we also describe how our approach using NIWI can be implemented efficiently.

Problem and Solution To keep things brief, the full rationale with all the technical details can be found in Appendix A.6. Let π^*_{FHE} be the protocol which is obtained from π_{FHE} by removing NIWI and instead including the witnesses w_i used to compute the π_i. In π^*_{FHE} (and π_{FHE}), the corrupted server is free to return any ciphertext c^* which may not have been evaluated correctly. The caller will then decrypt c^*. By observing the subsequent behavior of the caller, the adversary may obtain some (or possibly all) information about the plaintext to which c^* decrypts. Now, if during the security proof of π^*_{FHE} we wish to reduce to the semantic security of FHE to argue that the adversary does not learn anything about the input q, the reduction does not have a decryption oracle. Hence, this step of the proof must fail.

We have identified two possible solutions:

- Let the servers prove that they executed their evaluation correctly using NIZKPoKs.
- Make the distribution of proofs independent of the behavior of the corrupted server. This way we can during the security proof remove the dependence of proofs on the response by the corrupted server and subsequently reduce to the semantic security of FHE as we no longer have to decrypt.

We analyze both solutions in detail in the appendix.

Efficiency As using NIWI proofs is the option we have finally opted for, it is worth discussing how this approach can be made efficient as the efficiency of the resulting VRO instantiation crucially hinges on the efficiency of the NIWI scheme.

One of the most efficient (extractable) NIWI proof systems which are currently known are Groth-Sahai (GS) proofs [64]. These work well in conjunction with *structure preserving signatures* (SPS) [1] or [79]. GS proofs themselves are efficient NIWI protocols for propositions about so-called *pairing-product equations* (PPE) in the setting of bilinear groups, i.e. groups with an efficiently computable, non-degenerate, bilinear form [69]. SPS are signature schemes specifically tailored to be used with GS proofs. Some of their distinguishing properties are that all data structures, i.e. messages, signatures, and verification keys, are themselves group elements, as well as the fact that verifying a signature involves checking a system of PPE. This allows to efficiently proof knowledge of (1) a signature for some message under some verification key, (2) a message for which some signature is valid under some key, and other such relations.

5.4.7. Using Singly-Homomorphic Encryption

We may potentially get away without FHE and instead use an encryption scheme that is only homomorphic with respect to a single operation. There are signature schemes such as the one described in [14] where signing of encrypted messages only requires one homomorphic operation. If the same was possible also for the PRF evaluation then this could substantially increase the efficiency of the scheme. The important fact to note is that even with singly-homomorphic encryption schemes, a second operation can be applied *if only one operand is only known in encrypted form.* If one of the operands is known to the evaluator, then the operation can be performed.

The authors of [14] use the SPS scheme in [62] together with a variant of El'Gamal encryption [42] to homomorphically sign encrypted messages. As we have seen above, these are the same kind of signatures that are amenable to GS NIWI proofs. We are, however, currently unaware of the existence of any PRF which allows homomorphic evaluation through the same encryption scheme.

5.4.8. Relying on Preprocessing

Instead of distributing the verification key by letting the servers post their shares of it to $\mathcal{F}_{bboard}$, we could instead work in a *preprocessing model* in which some information is initially provided to all parties and the delivery of which can not be prevented by the adversary. In the simplest case, all parties receive the same output, i.e. this is exactly the common reference string model, but the received output may also depend on the identity of the receiving party.

In our case, we could let the preprocessing stage generate the four key-pairs for SIG and the CRS for NIWI. Each server could then be given its signing key and all parties receive the CRS as well as all four verification keys.

By working in such a model, the resulting protocol π'_{FHE} would realize a version of $\mathcal{F}_{VRO}$ for which either retrieval of the verification key was guaranteed to succeed or there would not be any verification key altogether. Looking at applications, this would allow key generation of FDH-VROM to remain non-interactive and similarly for verification of proofs produced by $\mathcal{F}_{TZK}$ (or rather it would allow us to realize $\mathcal{F}_{NIZK}$ where there exist no verification keys instead).

5.4.9. Eliminating Secure Channels

As all transmitted protocol messages in π_{FHE} are encrypted under the caller's key, we can indeed only require authenticated channels. The simulator has to be slightly adapted to produce (simulated) messages which are correctly distributed. For the communication between honest callers and honest servers, encryptions of zeroes are used just as in the message which is sent to the corrupted server and which the adversary was able to see already when using secure channels. As in the honest protocol, the same message is sent to all servers.

The indistinguishability proof is adapted accordingly. In the step where we reduce to the semantic security of FHE, the ciphertext obtained from the single query to the encryption oracle is now used as a message to all servers.

5.4.10. Analyzing Efficiency

In this section, we gauge the efficiency of the protocol π_{FHE}. We go over the different building blocks in turn and describe the costs for modern ways of instantiating them.

FHE Due to its relative novelty, this primitive is quite expensive. In the original scheme due to Gentry [53], evaluating each multiplication gate required executing a bootstrapping step, i.e. a homomorphic evaluation of the scheme's own decryption circuit, and was therefore computationally very demanding. Subsequent works such as [54, 13, 41] have reduced the overhead by, among others, eliminating the need for bootstrapping, moving to other algebraic settings, or allowing evaluation of the same circuit on multiple data items. In our setting, the latter would allow each server to evaluate all three PRF instances at the same time and similarly for the signature generation.

Even after these advances, FHE is still a very active area of research. This includes the formation of the *Homomorphic Encryption Standardization consortium*[10] which contains members from industry, government, and academia such as Microsoft, Intel, IBM, and NIST. To make the homomorphic evaluation more efficient, there is an ongoing effort to develop hardware accelerators and improvements at the software level for this task as part of the *Data Protection in Virtual Environments*[11] (DPRIVE) program by DARPA.

NIWI and Signatures Expanding on what we have already stated at the end of Section 5.4.6, we describe some further efficiency optimizations. First, by using *randomizable SPS* [28] the necessary GS proofs can be simplified. Randomizable SPS allow publicly transforming a concrete signature σ for a message m under a key vk into a signature σ' which has the same distribution as fresh signatures for m. In [28] it is described how this can be used in the context of proving knowledge of a signature. Concretely, it allows including a message-independent portion R of the signature in the clear as part of the statement to be proved. This in turn may allow reducing the number of PPE which have to be verified.

Furthermore, we observe that the message m for which knowledge of two signatures has to be proven is public at the time of the verification. This allows us to rely only on a subset of the full privacy-enabling capabilities provided by SPS. In particular, the requirement for SPS message spaces to be of the form $\mathcal{M} = \mathbb{G}^n$ for some group $\mathbb{G}$ which is the first source group of the associated pairing

$$e : \mathbb{G} \times \mathbb{G}' \to \mathbb{G}_t$$

enables the following.[12] It for example allows some party to either prove knowledge of a message m such that a public signature σ is valid for m under some public verification key vk, or to prove knowledge of a signature for a message m contained in a public

[10] `https://www.homomorphicencryption.org` (accessed: 20.09.2022)

[11] `https://www.darpa.mil/program/data-protection-in-virtual-environments` (accessed: 20.09.2022)

[12] We remark that some schemes also allow message spaces of the form $\mathcal{M} = \mathbb{G}^{n_1} \times \mathbb{G}'^{n_2}$.

commitment $\mathsf{COM}(m; r)$. The standard techniques of first applying a collision-resistant hash-function (CRHF) $H : \mathcal{M} \rightarrow \mathbb{G}^n$ is insufficient in particular in the second case as the prover would have to both prove knowledge of a signature for the hash $H(m)$ contained in the commitments as well as knowledge of the pre-image under H of this value. For a general domain $\mathcal{M}$ of H we can not expect to do so efficiently, i.e. using GS proofs.

In our case, however, we do not require to keep the message hidden or within a commitment. As such we can employ an SPS scheme SPS with the simple message space $\mathbb{G}$ and transform it into a signature scheme SPS* with message space equal to the domain $\mathcal{X}$ of $\mathcal{F}_{VRO}$ by using the hash-then-sign paradigm. SPS* then still allows efficient proofs of knowledge of a signature σ for $q \in \mathcal{X}$ by giving a proof of knowledge of a signature for $H(q) \in \mathbb{G}$.

We may then incorporate H into π_{FHE} in the following way. While we can not let the client apply H to its input q before encrypting it as we have to retain the ability for the simulator to extract q and not merely $H(q)$ from corrupted callers, we can allow the servers to do this. Before evaluating PRF and SIG, servers would first compute

$$c^* \leftarrow \mathsf{FHE.Eval}(\mathsf{pk}, H(\cdot), c)$$

where pk is the current FHE public key and c is the encryption of q under pk. All further evaluations are then applied to c^* instead of c. A proof $\pi = (h_i, \pi_i)_{i \in [4]}$ for a pair (q, h) now contains NIWI proofs for the relation

$$\mathcal{R} = \left\{ (x, w) \;\middle|\; \begin{array}{l} x = ((\mathsf{vk}_{Sig,j})_{j \in [4] \setminus \{i\}}, g), w = (\sigma_1, \sigma_2), \\ \exists i_1, i_2 \in [4] \setminus \{i\}, i_1 \neq i_2 \, \forall k \in \{1, 2\} : \\ \mathsf{SPS.Verify}(\mathsf{vk}_{Sig,i_k}, g, \sigma_k) = 1 \end{array} \right\}$$

where the verifier first computes the g for which the π_i must be valid proofs as $H(q)$.

The security of this scheme can be reduced to the collision-resistance of H. The structure of the proof has to be changed as follows. As a first step, one moves to an interaction where the adversary loses if it is able to produce a collision for H. All subsequent steps are essentially identical to those in Section 5.4.5.2.

Remark 5.4.17. We have ignored the fact that in π_{FHE} not q itself is signed, but (q, h, i) for $h \in \mathcal{H}$ and $i \in [4]$. This can be solved by letting SPS have the message space $\mathbb{G}^3$ and hashing h into a second group element and using the third group element as a tag by having it contain one of four elements $g_i, i \in [4]$ which are fixed at the start of the protocol.

5.4.11. Analyzing Scalability

As π_{FHE} has the same underlying structure as the PRF construction from [35], the number of servers can be increased in the same way as described therein. Concretely, to allow t corrupted servers at least $2t + 1$ total servers are required. Instead of being given as

$$h = \bigoplus_{i=0}^{4} \mathsf{PRF}(k_i, q)$$

for four PRF keys k_i, hashes are computed as

$$h = \bigoplus_{i=0}^{m} \mathsf{PRF}(k_i, q)$$

for some number m of PRF keys. To prevent the adversary from being able to predict hash values and to allow the simulator to replace some $\mathsf{PRF}(k_j, \cdot)$ with a random function, at least one of the k_i has to be unknown to the adversary. As the adversary may control any t-element subset of servers, there must always exist a key that is not known to any server in such a set. Unfortunately and as shown in [35], this requires choosing m to be exponential in t.

5.5. Relaxing the VRO

In this section, we look at relaxed formulations of $\mathcal{F}_{VRO}$ where the adversary is allowed to see the input q and/or the output h for each hash query. We then investigate whether the original PRF construction UC-realizes this relaxed functionality. In particular, this would show that hiding the input is at the heart of what makes it difficult to realize the full version of $\mathcal{F}_{VRO}$.

To this end, we define three functionalities $\mathcal{F}_{VRO}^{q,h}$, $\mathcal{F}_{VRO}^{q}$, and $\mathcal{F}_{VRO}^{h}$ where the superscript q means the input of each hash query is leaked to the adversary and correspondingly for h and the sampled hash. In more detail, superscript q means the `Hashing` messages sent to the adversary are augmented by q and similarly for h. Of course, if we include q, $\|q\|$ can be excluded.

First, we observe that as soon as the input q is leaked to the adversary, the adversary can just execute a hash query for q to obtain the hash h. The same is possible for the simulator. As such, $\mathcal{F}_{VRO}^{q}$ and $\mathcal{F}_{VRO}^{q,h}$ are largely equivalent functionalities. Furthermore, we note that $\mathcal{F}_{VRO}^{q,h}$ is equivalent to a functionality $\mathcal{F}$ where the adversary is not asked to initially provide an algorithm Prove but is instead allowed to fully determine all proofs. The reason for this is that in this case the adversary receives all inputs to Prove. It can then let Prove always output its third input s which is fully controlled by the adversary and can thus contain the selected proof.

5.5.1. Revisiting the PRF Construction

We claim that the PRF construction (as defined in Section 5.3.2) UC-realizes both $\mathcal{F}_{VRO}^{q,h}$ and $\mathcal{F}_{VRO}^{q}$. To this end, we give a simulator $\mathcal{S}$ for the dummy adversary $\mathcal{D}$. Clearly, it suffices to show that $\mathcal{F}_{VRO}^{q}$ is realized. We note that we do not make use of the "equivalence" of $\mathcal{F}_{VRO}^{q}$ and $\mathcal{F}_{VRO}^{q,h}$ as it is unclear to us whether this equivalence formally holds in the UC framework with respect to the capabilities given to the environment to gather knowledge about currently corrupted parties.[13]

[13] The problem we see is the following. For the equivalence to hold, the simulator has to query $\mathcal{F}_{VRO}$ in the name of some corrupted party. Depending on the exact information available to the environment, it may not be able to do so without being detected in certain circumstances.

The Simulator $\mathcal{S}$ behaves as follows:

- The only difference between the initialization procedures in π_{PRF} and π_{FHE} is that in π_{FHE}, a CRS for NIWI is computed among the servers. As such, $\mathcal{S}$ handles initialization as before. For SIGKeyGen, $\mathcal{S}$ samples four key-pairs $(\mathsf{vk}_i, \mathsf{sk}_i) \leftarrow \mathsf{SIG.Gen}(1^\lambda)$ and allows the adversary to deliver the outputs to the $\mathcal{S}_i$. It behaves analogously during PRFKeyGen. $\mathcal{F}_{bboard}$ is provided honestly.
- As $\mathcal{S}$ can pre-compute all the values in its very first activation, $\mathcal{S}$ is able to answer the `Init` message it at some point receives from $\mathcal{F}^q_{VRO}$ as follows. First, $\mathcal{S}$ sets $\mathsf{vk} = (\mathsf{vk}_i)_{i\in[4]}$. It further sets Prove to the algorithm shown in Figure 5.10. Prove, on input (q, h, s), Prove parses s into three signatures $(\sigma_{1,2}, \sigma_{1,3}, \sigma_{1,3})$. It then completes s into a proof π for (q, h) by computing signatures under the signing keys of the honest servers as in the honest protocol, except that $\mathsf{PRF}(k_1, q)$ is exchanged for the value $h \oplus \bigoplus_{i=2}^{4} h_i$.
- Whenever $\mathcal{F}^q_{VRO}$ asks $\mathcal{S}$ to deliver a response to a `Init` query to party $\mathcal{P}$, $\mathcal{S}$ lets the simulated $\mathcal{P}$ execute the honest protocol and allows the delivery once the simulated $\mathcal{P}$ generates output.
- Upon receiving a message $(\texttt{Hashing}, sid, \mathcal{P}, q)$, $\mathcal{S}$ executes the following steps. First, it lets the simulated copy of $\mathcal{P}$ execute the honest protocol on input $(\texttt{Hash}, sid, q)$. This includes letting the honest servers execute the honest protocol as well (note that they will use k_1 to compute h_1 instead of computing it as Prove does, but this is unobservable by the adversary). Once $\mathcal{P}$ obtains a response $(h_2, h_3, h_4, \sigma_2, \sigma_3, \sigma_4)$ from $\mathcal{S}_1$, $\mathcal{S}$ sets $s = (\sigma_2, \sigma_3, \sigma_4)$ and sends the message $(\texttt{SimInfo}, sid, \mathcal{P}, s)$ to $\mathcal{F}^q_{VRO}$. Only once the simulated $\mathcal{P}$ generates output, $\mathcal{S}$ allows the delivery of the proof.
- Whenever $\mathcal{D}$ initiates a hash query on behalf of a corrupted caller C_k by sending one of the honest servers $\mathcal{P}_j$ a message of the form $(\texttt{Hash}, sid, q)$, $\mathcal{S}$ sends the message $(\texttt{Hash}, sid, q)$ to $\mathcal{F}^q_{VRO}$ on behalf of C_k and answers the `Hashing` message with $s = \perp$. After receiving the response $(\texttt{HashProof}, sid, q, h, \pi)$, $\mathcal{S}$ extracts from π the signatures belonging to $\mathcal{S}_j$ as well as the h_l for $l \in [4] \setminus \{i\}$ and sends them to C_k.
- Upon receiving a message $(\texttt{Verify}, sid, q, h, \pi, \mathsf{vk}')$ from $\mathcal{F}^g_{VRO}$, $\mathcal{S}$ executed the honest verification protocol on the same input, obtaining a bit b. It immediately responds with the message $(\texttt{Verified}, sid, q, h, \pi, \mathsf{vk}', b)$.

Indistinguishability To show indistinguishability, we proceed by defining several hybrids as follows:

- G_0: The real interaction between $\mathcal{Z}$, $\mathcal{D}$ and a session of π_{PRF}.
- G_1: Like G_0, but whenever an honest server is supposed to evaluate the function $\mathsf{PRF}(k_1, \cdot)$, a truly random function RF is used instead.

```
Prove(q, h, s)
1 : (σ_{1,2}, σ_{1,3}, σ_{1,4}) = parse(s)
2 : for i ∈ {2, 3, 4} do
3 :     h_i = PRF(k_i, q)
4 : endfor
5 : h_1 = h ⊕ ⨁_{i=2}^{4} h_i
6 : for 2 ≤ i ≤ 4, 1 ≤ j ≤ 4, i ≠ j do
7 :     σ_{i,j} ← SIG.Sign(sk_i, (q, h_i, i))
8 : endfor
9 : π = (h_1, h_2, h_3, h_4, {σ_{i,j} | 1 ≤ i, j ≤ 4, i ≠ j})
10 : return π
```

Figure 5.10.: The Prove algorithm sent by the simulator.

- G_2: Like G_1, but whenever an honest verifier receives a valid proof (q, h, π, vk) such that either no honest server ever received a message $(\texttt{Hash}, sid, q)$ or $h \neq \mathsf{RF}(q) \oplus \bigoplus_{i=2}^{4} \mathsf{PRF}(k_i, q)$, it rejects it by returning $(\texttt{Verified}, sid, q, h, \pi, \mathsf{vk}, 0)$.
- G_3: The ideal interaction between $\mathcal{Z}$, $\mathcal{S}$ and $\mathcal{F}_{VRO}^{q}$.

Remark 5.5.1. As the proof is relatively short compared to the proof for π_{FHE} we have chosen to deviate from presenting the different games incrementally and motivating them as they are essentially simpler versions of steps in the proof for π_{FHE}.

We step through the different games and prove that the outputs of any environment $\mathcal{Z}$ remain computationally indistinguishable in each step.

Step 1: $G_0 \rightarrow G_1$

We reduce to the security of PRF by constructing an adversary $\mathcal{B}$ on the PRF security of PRF from any distinguisher $\mathcal{A}$ for the output distributions of $\mathcal{Z}$ produced in games G_0 and G_1.

This step is identical to the one from G_3 to G_4 in Section 5.4.5.2 and so we omit it here for brevity.

Step 2: $G_1 \rightarrow G_2$

We reduce to the EUF-CMA-security of SIG by showing that a proof leading to rejection in G_2 when it would have been accepted in G_1 necessarily contains a forgery for SIG.

The reduction consists in constructing an adversary $\mathcal{B}$ playing the EUF-CMA-game for SIG from any distinguisher $\mathcal{A}$ for the output distributions of $\mathcal{Z}$ in G_1 and G_2 respectively.

$\mathcal{B}$ receives a SIG verification key vk. It runs a simulation of $\mathcal{Z}$ and $\mathcal{D}$ and provides simulations of all honest parties according to G_1. As there are three honest parties for

which signatures have to be produced, $\mathcal{B}$ samples an index $i \leftarrow\$ \{2,3,4\}$ and uses vk as the verification key for party $\mathcal{P}_i$. $\mathcal{B}$ generates SIG key-pairs for the other two parties (and the corrupted server) as usual. Whenever $\mathcal{P}_i$ is supposed to sign a message m, $\mathcal{B}$ makes a Sign query for m to its oracle. Verification queries to honest parties are handled as in G_1, except for the fact that valid proofs are searched for valid signatures (m, σ) under an honest party $\mathcal{P}_j$'s key, but where this party never signed m. In that case $\mathcal{B}$ halts the simulation and, if $i = j$, outputs (m, σ) to its challenger.

Conditioned on the fact that the simulation is not prematurely aborted, the view of $\mathcal{Z}$ in the above simulation is consistent with G_1. Below we will argue that it is also identical to G_2 and that abortion occurs only with negligible probability.

We start by arguing the former and have to show that a rejection as introduced by G_2 leads to $\mathcal{B}$ aborting the simulation as this is the only case where G_1 and G_2 differ.[14] Let thus (q, h, π) with $\pi = (h_1, h_2, h_3, h_4, \ldots)$ be a query which is rejected and we proceed by analyzing both cases.

If q was never the input contained in any query to an honest server, then no honest server ever signed a triple with the first component equal to q. Together with the fact that a valid proof has to contain valid signatures by at least two different parties, this means that a forgery has to be contained in the proof.

In the other case, i.e. when $h \neq \mathsf{RF}(q) \oplus \bigoplus_{j=2}^{4} \mathsf{PRF}(k_j, q)$, then either $h_1 \neq \mathsf{RF}(q)$ or $h_j \neq \mathsf{PRF}(k_j, q)$ for $2 \leq j \leq 4$. Let k be one of the differing indices. Then no honest party ever signed (q, h_k, k), but an accepting proof must contain at least one signature by an honest party for this message and so again a forgery has to be contained.

This shows that as long as the simulation is running, the view of $\mathcal{Z}$ is identical to a view in both G_1 and G_2. It remains to be shown that the simulation is aborted only with negligible probability. By the EUF-CMA-security of SIG it holds that

$$\mathsf{Adv}^{\mathrm{euf-cma}}_{\mathcal{B},\mathsf{SIG}}(\lambda) = \mathsf{negl}(\lambda) \tag{5.15}$$

for a negligible function $\mathsf{negl}(\lambda)$. As $\mathcal{B}$ terminates the simulation whenever a rejection occurs for *any* of the three honest parties, but only a forgery for $\mathcal{P}_i$ leads to $\mathcal{B}$ winning, and the fact that i is information-theoretically hidden from $\mathcal{Z}$, imply that the probability of a simulation being terminated is at most $3 \cdot \mathsf{negl}(\lambda)$. Together with the above reasoning, we see that the view of $\mathcal{Z}$ in games G_1 and G_2 only differs with negligible probability. This provides an upper bound for the distinguishing advantage of any potential distinguisher $\mathcal{A}$ for the output distributions of $\mathcal{Z}$ in these two games by the difference lemma.

Step 3: $G_2 \rightarrow G_3$

We observe the remaining differences between G_2 and G_3. First, we note that the distribution of hashes is uniform in both cases. Furthermore, in both games h_2 to h_4 are computed by applying PRF to the input for uniformly chosen keys and h_1 is chosen uniformly at random. This shows that proofs output by $\mathcal{F}^{q}_{VRO}$ and those output by honest parties in G_2 are identically distributed. Then, verification of proofs for wrong verification keys $\mathsf{vk}' \neq \mathsf{vk}$ has not been changed with respect to G_0 and also the simulator in G_3 uses the honest verification procedure in this case. For the correct verification key vk, honest parties in G_2

[14]That is to say we argue by contraposition.

use the honest verification algorithm, but in addition reject a proof (q, h, π) if either no honest server ever received a message $(\texttt{Hash}, q)$ or $h \neq \mathsf{RF}(q) \oplus \bigoplus_{i=2}^{4} \mathsf{PRF}(k_i, q)$. We show that the latter corresponds to the *unforgeability* clause in the verification code of $\mathcal{F}_{VRO}^{q}$.

We first show that the set Q containing inputs q for which there has been a previous hash query is determined in both games. In G_3, the set of Q of past queries to $\mathcal{F}_{VRO}^{q}$ contains two types of elements. On the one hand, elements that were added because an honest party $\mathcal{P}$ executed a hash query. On the other hand, queries by $\mathcal{S}$ on behalf of some corrupted party C which, by the description of $\mathcal{S}$, occur because the simulation of C sent a message $(\texttt{Hash}, sid, q)$ to the simulation of one of the honest servers. But this is identical to how Q is determined in G_2 as honest parties will always contact the honest servers. As such, in both games verification queries for inputs not in Q are rejected in the same manner. For elements in $q \in Q$ and with h the correct hash for q, proofs π previously output by $\mathcal{F}_{VRO}^{q}$ are always accepted. The same is true for G_2 and proofs output by honest parties by the completeness of SIG. For proofs that were not previously output by $\mathcal{F}_{VRO}$, $\mathcal{S}$ is asked to decide whether to accept or reject. For that, it uses the honest verification algorithm. The same is true in G_2, for inputs in q the honest verification algorithm is always used. This leaves the case where h is not the correct hash for q, but all proofs for such inputs are rejected in both games.

One small thing we are left to consider is the case where $\mathcal{F}_{VRO}^{q}$ aborts during a hash query for q with hash h, because the proof π output by Prove is already marked as invalid. Such an event can only occur, if $\mathcal{S}$ at some point answered a $\texttt{Verify}$ message by $\mathcal{F}_{VRO}^{q}$ for (q, h, π, vk) with rejecting π. As $\mathcal{S}$ uses the honest verification algorithm to answer such messages, this would contradict the completeness of Prove. □

This shows the following theorem.

Theorem 5.5.2. *The protocol π_{PRF} UC-realizes $\mathcal{F}_{VRO}^{q}$ and thus also $\mathcal{F}_{VRO}^{q,h}$ in the $\mathcal{F}_{SMT}$-hybrid model.*

Due to space concerns we have moved the remaining portion of this section into the Appendix A.4. There we go into more depth about how the additional information afforded to the simulator in the different relaxations for $\mathcal{F}_{VRO}$ allows for more efficient protocols.

5.6. Strengthening the VRO

We also investigate some stronger variants of $\mathcal{F}_{VRO}$. We evaluate the instantiations we have seen so far with respect to these variants as well as in some cases propose changes with which they could be made to achieve them if they do not yet do so.

5.6.1. Stronger Proofs

The first additional property we are interested in is with regards to *proof strength* or *malleability* of proofs. In $\mathcal{F}_{VRO}$ and all of the above variants we only forbid the adversary to forge valid proofs for new q or for a wrong hash h for a given q. This means an adversary may alter a proof π for (q, h) into a proof $\pi' \neq \pi$ which is also valid for (q, h). Some

applications, however, require stronger non-malleability properties. We have for example seen this in Section 4.2.6 where the inclusion of $\mathcal{F}_{VRO}$ proofs in protocol messages destroy the required non-malleability.

Two potential strengthenings are

- **Unique Proofs:** For every $(q, h) \in \mathcal{X} \times \mathcal{H}$ there exists at most one valid proof π.
- **Strong Proofs:** For a computationally bounded adversary $\mathcal{A}$ it is infeasible to produce a valid proof π for $(q, h) \in \mathcal{D} \times \mathcal{H}$ that was not produced by the functionality.

The second notion on its own seems to be less useful as any party can just request fresh proofs at any time. Of course, this requires the party holding a proof π to also hold the corresponding input q and there may be applications where a proof is not necessarily always accompanied by the input to which it belongs. For example, a pair (h, π) could be employed in conjunction with a proof of knowledge of some q such that π is correct with respect to (q, h). For this to be useful it may be necessary to require that proofs are hiding the input (and/or hash) for which they were generated as that is not guaranteed by $\mathcal{F}_{VRO}$. Prove receives both q and h as input and is not prevented from leaking them in its output, see Section 5.6.2.

These complications could be circumvented wholly by requiring that $\mathcal{F}_{VRO}$ always output the same proof for some input q. If this was guaranteed, then the only difference between strong and unique proofs is the class of adversaries, efficient or unbounded, against which security holds. In a setting with a single party, always outputting the same proof seems feasible. As soon as multiple servers are involved, however, it seems much more difficult to guarantee. A corrupted server can not be made to restrain itself voluntarily and so the inability to create multiple different proofs for the same input has to be inherent to the information known to the adversary.

Evaluating Instantiations In the trusted party case, both of these notions are (easily) attainable by using the protocol in Section 5.2 with signature schemes which are either *unique* [72], or *strong*. For both variants of the ROM instantiation presented in Section 5.1, unique proofs are produced. In the protocol for large codomains, this holds as proofs are empty and non-empty proofs are rejected. In the protocol for polynomial codomains, a proof π for (q, h) is the unique suffix of the output of $\mathcal{F}_{RO}$ on input q after removing h.

While the PRF construction from Section 5.3.2 does not UC-realize $\mathcal{F}_{VRO}$ itself, we can nonetheless analyze the strength of its proofs. Due to the fact that the set of signatures which has to be contained in a valid proof is not fixed, the inclusion or exclusion of signatures by the adversary trivially allows it to modify a proof in such a way that it remains valid. At first sight, the changes to the proof structure we have made within our FHE construction in Section 5.4 are more amenable towards stronger proofs. A proof π always included four NIWI proofs π_1 up to π_4, independent of the behavior of the corrupted server. Nonetheless, by using a corrupted caller, the adversary can gain access to valid

signatures by the honest servers and can thus compute fresh valid proofs. We note that neither requiring stronger signatures nor some notion of non-malleability for the NIWIs is of any help.

5.6.2. Hiding Proofs

As hinted at above, our formulation of $\mathcal{F}_{VRO}$ does not prevent proofs π from leaking both the input q and hash h. Again this may be undesirable in some applications. Using techniques from [19], we can formalize independence of π from q and h as follows. Instead of providing the real q and h as input to Prove, we would give random aliases r_q and r_h instead. These aliases are only known to $\mathcal{F}_{VRO}$ and their association with inputs and hash values is fixed within a single session. The verification procedure remains unchanged.

Evaluating Instantiations As in the previous section, both instantiations based on $\mathcal{F}_{RO}$ possess this property. In the protocol for large codomains, empty proofs clearly do not contain any information about either q or h. For small codomains, the proof consists of a uniformly random value that is independent of the hash h and also of q by the properties of $\mathcal{F}_{RO}$. Again, no information is thus leaked by the proof. Continuing to the trusted party instantiations, proofs by the sVRF-based protocol are generated by running the SimProve algorithm which receives both q and h as input. As such there can not be a generic argument for any provided hiding properties. Specific schemes may, however, provide it. In the signature-based protocol, requiring the signature scheme to be *confidential* [33] or *indistinguishable* [46] in the sense that an adversary receiving a signature σ without the corresponding message m is unable to compute m,[15] hides both q and h. The protocol in Section 5.4 does not hide h as each proof contains shares h_i summing to h, but using a signature scheme that is again confidential would hide q.

5.7. Hybrid Instantiations

In this section, we want to investigate a type of instantiation which possess a special kind of corruption resistance and which make use of a hash-function H. On the surface, we still consider protocols realizing $\mathcal{F}_{VRO}$. Their security, however, involves two cases. In the first, H is assumed to be a function having some standard model properties, e.g. collision resistance, and the protocol has to realize $\mathcal{F}_{VRO}$ given that the number of corrupted servers is below some threshold t. In the other case, H is modeled as a random oracle and we allow *all* servers to be corrupted. We wish to retain most of the properties of $\mathcal{F}_{VRO}$, but allow the adversary to obtain the full input.

Using a hybrid instantiation in the real world where necessarily H is not a random oracle can be seen as a kind of insurance. The protocol is fully secure and UC-realizes $\mathcal{F}_{VRO}$ as long as the associated corruption threshold is not reached, and remains at least heuristically secure even if more than the threshold up to all parties are corrupted with the slight degradation in security of leaking all inputs.

[15] We remark that this is only intended to provide some intuition. The actual definitions are more subtle. In particular, messages are required to have high entropy. Otherwise, a full search over the message space using the verification algorithm reveals the message.

In the full corruption case, it is clear that something like a random oracle that is outside of the control of the adversary is necessary. Otherwise, the combined private state of the servers necessarily allows at least the prediction of hashes when given to the environment while the random oracle has to be explicitly queried by the adversary and this can be programmed by the simulator and thus be made consistent with $\mathcal{F}_{VRO}$.

Remark 5.7.1. Hiding also the input all but necessitates the use of (fully-homomorphic) encryption as secret sharing can not help when all receivers of shares are corrupted. Additionally, we have to require proof of correct evaluation from the server as in the case of full corruption the technique using witness-indistinguishable proofs or other ways of making proofs independent of answers by corrupted parties are not applicable. As hybrid instantiations are intended as a *last resort*, i.e. when the corruption is larger than intended, we think that slightly reducing the afforded security is appropriate.

The Idea Consider the instantiation using a sVRF described in Section 5.2. If the reference string generation can be guaranteed to be independent and honest, then, in the face of the server being corrupted, this instantiation remains secure in the sense that no proofs can be forged and hashes remain pseudo-random, but it would be predictable using the secret key. It also leaks inputs and hashes. The idea is to thwart predictability by computing hashes as

$$h = \mathsf{H}(q) \oplus \mathsf{sVRF.Eval}(\sigma, \mathsf{ek}, q)$$

instead of

$$h = \mathsf{sVRF.Eval}(\sigma, \mathsf{ek}, q).$$

This works *if* the simulator can gain access to ek, i.e. we would have to let the party prove knowledge of ek corresponding to the public vk to some authority or have it be provided by some honest party at the start of the protocol.

The Protocol We define a protocol π_{Hyb}. First, recall that sVRF consists of the algorithms $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Setup},)$ and $(\mathsf{SimGen}, \mathsf{SimSetup}, \mathsf{SimProve})$. Let $\mathcal{T}$ be the trusted party. It executes the same protocol as in π_{sVRF} except that both the CRS σ and key-pair $(\mathsf{vk}, \mathsf{ek}) \leftarrow \mathsf{sVRF.Gen}(1^\lambda)$ are provided to it by some authority. The algorithms executed by the clients are altered as shown in Figure 5.11 and where H is either a local function evaluation or a call to a random oracle. The correct verification key vk can be retrieved from the authority which initially provided it to $\mathcal{T}$.

The changes made to clients with respect to π_{sVRF} are:

1. Clients check whether the proof π which they obtain from $\mathcal{T}$ is valid. This is necessary to prevent clients from outputting non-verifying proofs in the case where $\mathcal{T}$ is corrupted.

2. Clients add $\mathsf{H}(q)$ to the partial hash h_1 they obtain from $\mathcal{T}$ to compute the final hash h.

Let us now take a closer look at the simulators $\mathcal{S}_1$ for (1), the case where H is a standard function and $\mathcal{T}$ is a trusted honest party, and $\mathcal{S}_2$ for (2) where it is replaced by a random oracle and $\mathcal{T}$ is corrupted.

On input $(\texttt{Hash}, sid, q)$	**On input** $(\texttt{Verify}, sid, q, h, \pi, \mathsf{vk})$
1 : / Assume the sVRF verification key vk is known	1 : $(\pi', h_1) \leftarrow \texttt{parse}(\pi)$
2 : / as well as the CRS σ	2 : **if** $\mathsf{sVRF.Verify}(\sigma, \mathsf{vk}, q, h_1, \pi') = 0$ **do**
3 : $(h_1, \pi') \leftarrow \mathcal{T}(\texttt{Hash}, sid, q)$	3 : **return** $(\texttt{Verified}, sid, q, h, \pi, 0)$
4 : **if** $\mathsf{sVRF.Verify}(\sigma, \mathsf{vk}, q, h_1, \pi) = 0$ **do**	4 : **fi**
5 : **abort**	5 : **if** $\mathsf{H}(q) \neq h \oplus h_1$ **do**
6 : **fi**	6 : **return** $(\texttt{Verified}, sid, q, h, \pi, 0)$
7 : $h_2 = \mathsf{H}(q)$	7 : **fi**
8 : $h = h_1 \oplus h_2$	8 : **return** $(\texttt{Verified}, sid, q, h, \pi, 1)$
9 : $\pi = (\pi', h_1)$	
10 : **return** $(\texttt{HashProof}, sid, q, h, \pi)$	

Figure 5.11.: The client algorithms for π_{Hyb}.

The First Simulator $\mathcal{S}_1$ behaves as follows:

- $\mathcal{S}$ simulates a reference string σ with backdoor τ by running $\mathsf{SimSetup}(1^\lambda)$. It also generates a simulated key-pair $(\mathsf{vk}, \mathsf{ek})$ by running $\mathsf{SimGen}(1^\lambda, \sigma, \tau)$. Upon receiving a message $(\texttt{Init}, sid)$ from $\mathcal{F}_{VRO}$, $\mathcal{S}_1$ sets $\texttt{Prove}$ to the following algorithm. On input (q, h, s), Prove computes $h' = H(q)$ and then simulates a proof $\pi \leftarrow \mathsf{sVRF.SimProve}(\sigma, \tau, \mathsf{ek}, q, h \oplus h')$. The last input s is ignored by Prove. $\mathcal{S}_1$ sends the message $(\texttt{Init}, sid, \mathsf{Prove}, \mathsf{vk})$ back to $\mathcal{F}_{VRO}$.

- Upon being asked to deliver the verification key to an honest party $\mathcal{P}$, $\mathcal{S}_1$ lets the simulated $\mathcal{P}$ try and retrieve the key and allows $\mathcal{F}_{VRO}$ to deliver its response upon this succeeding.

- Upon receiving $(\texttt{Hashing}, sid, \mathcal{P}, l)$ from $\mathcal{F}_{VRO}$, $\mathcal{S}_1$ simulates an honest hash query by $\mathcal{P}$ and allows $\mathcal{F}_{VRO}$ to deliver the proof once $\mathcal{P}$ outputs a proof. The $\texttt{SimInfo}$ message is sent with $s = \perp$ immediately.

- Whenever $\mathcal{T}$ receives a message $(\texttt{Hash}, sid, q)$ from some corrupted party C, $\mathcal{S}_1$ makes a hash query for q to $\mathcal{F}_{VRO}$ on behalf of C and receives output (h, π). It computes $h' = q \oplus h$ and sends h' and π to C.

- Upon receiving $(\texttt{Verify}, sid, q, h, \pi, \mathsf{vk}')$ from $\mathcal{F}_{VRO}$, $\mathcal{S}_1$ computes the response bit as b using the honest verification algorithm.

Indistinguishability Indistinguishability immediately follows from the security of π_{sVRF} and the collision resistance of H. The strategy of $\mathcal{S}_1$ is essentially the same as for the π_{sVRF}-simulator. The only changes are the addition of a number of evaluations of H at appropriate places. This shows that the simulation itself is still valid, we only have to argue that the introduction of H did not make forging proofs noticeably easier.

This follows by a standard reduction. In detail, we first switch from the real interaction to an interaction where honest verifiers reject all (otherwise valid) queries for some (q, h, π, vk) such that q has the same hash under H as some previous hash query made to $\mathcal{T}$. A distinguishing environment $\mathcal{Z}$ for the real and intermediate interaction immediately yields a successful adversary on the collision-resistance of H in an obvious manner.

All that is left to argue is that distinguishing the intermediate from the ideal interaction reduces to the security of π_{sVRF}. But this follows from the fact that we can efficiently switch between the two protocols by adding $\mathsf{H}(q)$ at appropriate places. This allows us to build a distinguishing environment $\mathcal{Z}^*$ for π_{FHE} from a distinguishing environment $\mathcal{Z}'$ between the intermediate and real interaction in the present protocol. This crucially relies on having excluded the case where $\mathcal{Z}'$ wins by finding a $\mathcal{H}$ collision.

Remark 5.7.2. In the above, we have silently ignored that H has to remain secure in the face of non-uniform adversaries. Such hash functions are necessarily keyed. This can easily be remedied by making the key for H part of the verification key.

The Second Simulator We now switch to the case where H is given as a random oracle. As we no longer try to hide the inputs from the adversary, we show that $\mathcal{S}_2$ is a simulator for π_{Hyb} and $\mathcal{F}_{VRO}^q$ instead of for $\mathcal{F}_{VRO}$. $\mathcal{S}_2$ behaves as follows:[16]

- $\mathcal{S}_2$ generates the CRS σ honestly using sVRF.Setup and also generates an honest key-pair $(\mathsf{vk}, \mathsf{ek})$ using sVRF.Gen. All data is given to $\mathcal{T}$ when requested.
- Upon receiving a message $(\mathtt{Init}, sid)$ from $\mathcal{F}_{VRO}$, $\mathcal{S}_2$ responds with $(\mathtt{Init}, sid, \text{Prove}, \mathsf{vk})$ where vk is the sVRF verification key and Prove simply outputs its third input s.[17]
- Upon receiving $(\mathtt{Hashing}, sid, \mathcal{P}, q)$ from $\mathcal{F}_{VRO}^q$, $\mathcal{S}_2$ simulates an honest hash query by $\mathcal{P}$ for q. Let $(\pi.h^*)$ be the value received from $\mathcal{T}$. If $\mathsf{sVRF.Verify}(\sigma, \mathsf{vk}, q, h^*, \pi) = 0$, do not consider this request further. Else, send the message $(\mathtt{SimInfo}, sid, \mathcal{P}, (\pi, h^*))$ back to $\mathcal{F}_{VRO}^q$.
- Upon receiving $(\mathtt{Verify}, sid, q, h, \pi, \mathsf{vk}')$ from $\mathcal{F}_{VRO}^q$, $\mathcal{S}_2$ executes the honest verification algorithm to obtain a bit b and returns $(\mathtt{Verified}, sid, b)$ to $\mathcal{F}_{VRO}^q$.
- This may entail queries to H. All such queries as well as any other random oracle queries for some q are answered as follows. If this is not the first query for q, answer consistently. Else, $\mathcal{S}_2$ makes a hash query for q to $\mathcal{F}_{VRO}^q$ to obtain hash h (the proof is ignored). It then computes h' as the first component of $\mathsf{sVRF.Eval}(\sigma, \mathsf{ek}, q)$, sets $h^* = h \oplus h'$ and returns h^*.

Indistinguishability The analysis of this protocol is essentially a combination of the analysis of the pure random oracle instantiation π_{RO} as well as π_{sVRF}. The simulation of H is perfect by adding a constant to the hash h obtained from $\mathcal{F}_{VRO}^q$. Proofs output by honest parties are also identically distributed by $\mathcal{S}_2$ executing the honest querying algorithm

[16]We only consider the case where $\mathcal{T}$ is corrupted, in the other case $\mathcal{S}_2$ simply lets $\mathcal{T}$ behave honestly and programs H using queries to $\mathcal{F}_{VRO}$ in an obvious way.

[17]As noted at the start of Section 5.5, this strategy is always possible when realizing $\mathcal{F}_{VRO}^q$.

inside the simulation and forwarding the obtained proof. Unforgeabilty follows by the same arguments as for π_{sVRF} because for some input $(q, h, (h', \pi))$ and by the programming of H

$$\mathsf{H}(q) = h \oplus h' \iff \pi_1(\mathsf{sVRF.Eval}(\sigma, \mathsf{ek}, q)) = h'$$

where π_1 is the projection to the first component. This forces π to be a valid proof for the correct evaluation of sVRF at input q yielding unforgeability.

A General Transformation Let us call verification algorithms where for each q there is at most one h such that there *exists* a valid proof π as having *unique verification.*[18] The same procedure of adding the random oracle output seems to also work for any protocol which has unique verification. The addition of $\mathsf{H}(q)$ reestablishes programmability—given that the simulator can extract all the private keying material from the corrupted parties to program correctly. As a slight relaxation, requiring that for well-formed verification keys it is computationally infeasible to produce valid proofs for wrong hashes, even knowing the secret key, seems to suffice as well.

For the PRF construction, nothing prevents fully corrupted servers from signing arbitrary values and thus creating valid proofs for any hash value, i.e. it does not have unique verification, so this transformation is not directly applicable. Replacing PRF and signature scheme with an sVRF makes it applicable, again at the cost of requiring honest reference string generation.

Other Constructions Apart from adding $\mathsf{H}(q)$ to compute the final hash, we wonder how else a random oracle could be useful. First, it is clear that oracle queries have to be part of the verification process, else the adversary could just "lie" about oracle values during proof generation, effectively nullifying any security gains. They also have to be involved in the derivation of hashes.

One way for H to influence hash values we have already seen. Another way would be to let h itself be the output under H of some input x which is a function of q. To be able to program when H is a random oracle, the simulator would have to be able to derive x from q using its knowledge of the adversaries' keys. When H is a function, on the other hand, outputs have to also be programmable, but this time without being able to program the oracle. This seems to require inverting H and being able to bias whatever computation occurs before the application of H to a matching pre-image. We have not investigated this further.

5.8. Multiple Sessions

In the UC framework, if some protocol π wants to make use of multiple instances of some ideal functionality $\mathcal{F}$, then these copies of $\mathcal{F}$ have to be totally independent of each other. This means that if some protocol ξ UC-realizing $\mathcal{F}$, in turn, makes use of some expensive resource, such as a common reference string, every session of ξ used in a session of π has to receive a fresh string. One way to make more efficient use of the expensive resource is

[18] Note that this notion differs from unique proofs as defined in Section 5.6.1. Unique verification allows multiple proofs to exist, but only for the correct hash.

by defining some protocol ξ' of which a single session realizes multiple sessions of ξ, but using fewer resources than independent copies of ξ.

The expensive resources in our case may be servers or the aforementioned common reference strings. A simple way to let one instance of $\mathcal{F}_{VRO}$ (with domain $\{0,1\}^*$) realize multiple sessions of $\mathcal{F}_{VRO}$ (with domains $\{\mathcal{D}_i\}_{i\in I}$ for some index set I) is by employing *domain separation.* To query q in some sub-session with identifier *ssid* and main session with identifier *sid*, an instance of $\mathcal{F}_{VRO}$ within session *sid* is queried with input $\mathsf{Enc}(ssid, q)$ where

$$\mathsf{Enc} : \{0,1\}^* \times \cup_{i\in I}\mathcal{D}_i \to \{0,1\}^*$$

is an (efficiently computable) encoding function such that for all $ssid_1, ssid_2 \in \{0,1\}^*$, it holds that

$$ssid_1 \neq ssid_2 \Rightarrow \mathrm{Im}(\mathsf{Enc}(ssid_1, \cup_{i\in I}\mathcal{D}_i)) \cap \mathrm{Im}(\mathsf{Enc}(ssid_2, \cup_{i\in I}\mathcal{D}_i))) = \emptyset,$$

that is, different sub-sessions have disjoint domains. Depending on the space from which session identifiers are taken, the domain of the single instance of $\mathcal{F}_{VRO}$ may be chosen smaller. Regarding different codomains $\{\mathcal{H}_j\}_{j\in J}$, if we assume that they are all efficiently (and possibly invertibly) samplable, the single instance of $\mathcal{F}_{VRO}$ may produce enough random bits to sample an element from the $\mathcal{H}_j$ requiring the most random bits to sample. For the others, using a coherent prefix of the bits would suffice.

5.9. Semi-Honest Adversaries

So far we have only considered security against malicious adversaries. Restricting adversaries to follow the honest protocol greatly simplifies the design of secure protocols. If we can be sure that each caller behaves honestly, we can for example be sure that a purported ciphertext c contains the same value as some UC-commitment *com* without having to require some proof of this fact. On the server, side we can also replace primitives, e.g. actively secure MPC protocols used during initialization, with variants that are only passively secure. Assuming that callers are semi-honest seems to be overly optimistic. For this reason, we will restrict ourselves to considering protocols where the servers act according to their specification.

Properties that still have to hold even when restricting to semi-honest servers are the following:

- **Unpredictability:** Using information known to the corrupted servers, the adversary can not predict outputs.
- **Hiding:** Using information sent to the corrupted servers by honest callers during a query, the adversary can not compute the input/output of the query.
- **Unforgeability:** Using information known to the corrupted server, the adversary can not forge proofs for never queried values or modify existing proofs for previously queried inputs while changing the hash value.

It is clear that for the third point, the servers being semi-honest does not make a difference as in both cases the environment has the same information. For this reason, in our setting the difference between actively secure protocols and protocols secure against semi-honest adversaries may not differ as greatly. In Appendix A.7 we detail how π_{FHE} can be simplified if we assume semi-honest servers.

5.10. General MPC

One generic way to realize $\mathcal{F}_{VRO}$ is by utilizing the wide feasibility results for UC-realizing essentially any PPT functionality by a non-trivial protocol. The downside of this approach is the usually high communication complexity which for most protocols is proportional to parameters such as the number n_{mult} of multiplication gates within the evaluated circuit C or the depth d of the circuit.

Restrictions apply, however. Most results assume that the functionality $\mathcal{F}$ is *well-formed* for some notion of this term. [25] define the term as follows:

Definition 5.10.1 (Well-Formed Functionalities)**.** A *well-formed* functionality $\mathcal{F}$ consists of a shell and a core. The core is an arbitrary PPT algorithm. The shell acts as an intermediary between incoming messages and the core. Messages informing the functionality of corruptions are held back by the shell and not forwarded to the core. Messages from the core are delivered as intended. In addition, well-formed functionalities are required to allow the simulator to individually deliver all responses by $\mathcal{F}$ to honest (dummy) parties and vice versa.

The first restriction of the set of allowed functionalities is necessary as otherwise there exist functionalities that use their knowledge of the set of corrupted parties in such a way as to prevent any real protocol from realizing them. In [25] an example of such a functionality is the functionality that sends to all parties the set of corrupted parties. A successful attack on any protocol trying to realize this functionality lets the adversary corrupt a single randomly chosen party but lets the party follow the protocol. In the real world, the honest parties have no way of finding out the identity of the corrupted party, but in the ideal world they will gain this knowledge from the functionality.

Regarding the second restriction, if the functionality was allowed to accept input x from a party $\mathcal{P}_1$ and immediately generate and deliver some output $y = f(x)$ to second party $\mathcal{P}_2$, then again such a functionality would not be realizable by any protocol.

On the positive side, feasibility results guarantee the existence of *non-trivial* protocols for realizing any functionality for which these two restrictions hold. By requiring $\mathcal{F}$ to allow the simulator to not deliver any messages, the trivial protocol which ignores all inputs and never generates any outputs UC-realizes any well-formed functionality. A non-trivial protocol π for $\mathcal{F}$, however, still UC-realizes $\mathcal{F}$ if the real adversary delivers all messages and does not corrupt any party, then the simulator does the same.

Having defined these two terms, we can state a version of the feasibility theorem from [25].

Theorem 5.10.2. *Assuming that enhanced trapdoor permutations exist. Then, for any well-formed multi-party ideal functionality $\mathcal{F}$, there exists a non-trivial protocol that UC-realizes $\mathcal{F}$ in the $\mathcal{F}_{CRS}$-hybrid model in the presence of malicious, static adversaries.*

5.10.1. Client-Server Protocols

One general pattern would then be the following. We can select any well-formed (allowed to be reactive) functionality $\mathcal{F}$ and have it be executed by some set of n servers $\mathsf{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$. A subset $\mathsf{Corr} \subset \mathsf{S}$ with $|\mathsf{Corr}| < n$ is corrupted, the remaining servers, i.e. those in the set $\mathsf{Hon} = \mathsf{S} \setminus \mathsf{Corr}$ follow the protocol honestly. A honest caller C, receiving some input q, first does some local computation Split on q and generates a vector $\mathbf{m}$ of n messages as

$$\mathbf{m} = (m_1, \ldots, m_n) \leftarrow \mathsf{Split}(q).$$

The vector $\mathbf{m}$ is distributed among the servers in $\mathcal{S}$ via the assignment $m_i \rightarrow \mathcal{S}_i$. The servers engage in a multi-party computation by sending values $\{\hat{m}_i\}_{i=0}^{n}$ to $\mathcal{F}$. All honest servers $\mathcal{S}_i \in \mathsf{Hon}$ will provide $\hat{m}_i = m_i$ whereas corrupted servers $\mathcal{S}_j \in \mathsf{Corr}$ are free to provide any value $\hat{m}_j \neq m_j$. Depending on the form of $\mathcal{F}$, an output o_i is returned to $\mathcal{S}_i$. In the most simple case, each server receives output once all servers have provided their input. A subset $\mathsf{R} \subseteq \mathsf{S}$ then returns values $\{\hat{o}_i\}_{i \in \mathsf{R}}$ to C where again corrupted servers may return a different value than the one they received from $\mathcal{F}$. Finally, the caller assembles from these responses its output (h, π) using a combination algorithm $\mathsf{Combine}(o_1, o_2, \ldots, o_n)$. Combine may either be defined only for fully-defined inputs, i.e. $o_i \neq \perp$ for all $i \in [n]$, or it may require more than a threshold $t \leq n$ of inputs to be $\neq \perp$.[19]

As C will not receive the $\hat{o}_i$ at the same time, a rule has to be established for when to run Combine on the response received so far. Which rule to use will depend on the structure of Combine. In case Combine is only defined for full inputs, C waits until all n servers have replied (and hangs indefinitely until this occurs). If Combine is able to be run once t responses have been received, there are several sensible rules. C may for example wait for the first t responses it receives and runs Combine on these. The execution of a single hash query is visualized in Figure 5.12. More general communication patterns involving multiple rounds are possible, but this two-message exchange between the caller and each server suffices for our purposes.

A Simple Realization There are several natural candidates for $\mathcal{F}$ and the surrounding protocol executed between caller and servers. We show one of them which seems to use particularly simple as it essentially distributes the protocol π_{SIG} from Section 5.2. The caller first shares its input q additively as $q = \bigoplus_{i=1}^{n} q_i$ and sends q_i to server $\mathcal{S}_i$. $\mathcal{F}$ has the following form: During initialization, a PRF key k is generated and additive shares k_i are distributed among the servers. The same is done for the signing key sk of a signature scheme SIG while the verification key vk is sent in one piece to all the servers. During a hash query, after having received input from each of the servers, first, q is reassembled. Then $h = \mathsf{PRF}(k, q)$ is computed and finally (q, h) is signed using sk, obtaining a signature

[19] We allow Combine to output $\perp$ in which case C either aborts the current query or runs Combine on a different set of inputs in the hope of generating valid output.

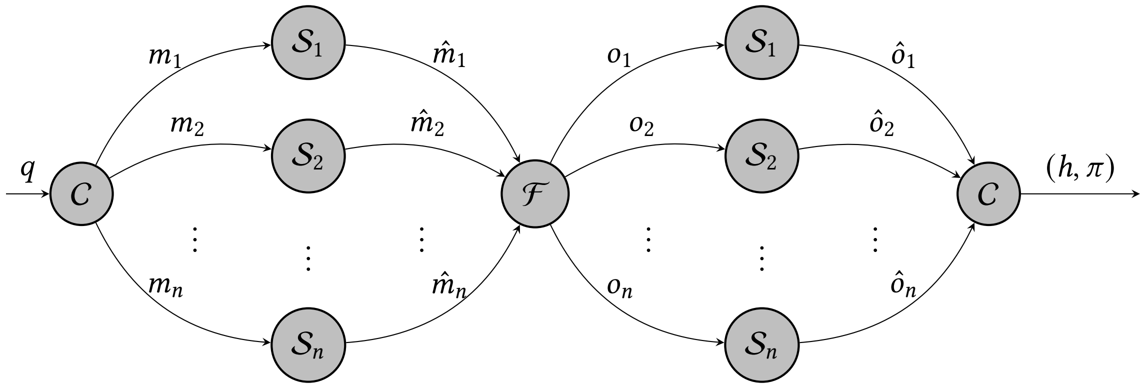


Figure 5.12.: Visualization of the typical flow of messages for a hash query by a caller C on input q with servers $\mathcal{S}_1, \ldots, \mathcal{S}_n$ and ideal functionality $\mathcal{F}$. Time advances from left to right.

σ. For both h and σ, only additive shares $h = \bigoplus_{i=1}^{n} h_i$ and $\sigma = \bigoplus_{i=1}^{n} \sigma_i$ are returned to the servers which are then returned to the caller. The caller re-assembles h and σ. The hash is given by h and $\pi = \sigma$ is the proof. To verify a proof π for (q, h) using vk, it is simply checked that π is a valid signature for (q, h) under vk. The key vk has been previously obtained during an `Init` request by contacting each of the servers and outputting anything if and only if the same value has been received from all servers (in particular, a message has to be received from all servers). This has to be done to ensure that even $n - 1$ corrupted servers can not make an honest party obtain a wrong verification key.

Remark 5.10.3. Of course, the corrupted servers may not partake in the multi-party computation using the share q_i which they have initially received. But doing so will lead to the servers computing $h' = \mathsf{PRF}(k, q')$ for some reconstructed $q' \neq q$ and also a signature for (q', h') instead. Having clients output such signatures σ is prevented by letting clients check that σ is valid for the message (q, h') and only outputting it when it is. Another potential problem is introduced by the possibility of having corrupted servers input different shares k_i' than the one they have received initially. $\mathcal{F}$ would thus further check that the employed shares are indeed the correct ones.

Remark 5.10.4. The fact that the adversary is allowed to make some honest servers obtain output and others not is non-problematic. If any honest party does not receive its output during initialization, no honest party will ever obtain output trying to obtain the verification key or execute a hash query as there will be *some* server that does not respond. This is allowed by the definition of $\mathcal{F}_{VRO}$ as only having an honest party output some wrong verification key or invalid proof has to be prevented.

It is then not difficult to show that this protocol does indeed UC-realize $\mathcal{F}_{VRO}$, albeit requiring extensive interaction between the servers when compared to our FHE-based protocol and being more reliant on the correct behavior of all of the corrupted servers. The randomness guarantees follow by replacing the PRF with a random function. Unforgeability follows by the security of the signature scheme. Extraction of the input is possible by reconstructing it from the shares. If we assume that causing a server to misbehave is easier

for the adversary to achieve than stopping the delivery of a network message between honest parties, then this protocol is more susceptible to denial-of-service attacks.

We can allow some corrupted servers to deviate from correctly following the protocol by using other forms of secret sharing, i.e. (t, n) verifiable secret sharing. We then require that at most t servers are corrupted, but can cope with the fact that these may arbitrarily deviate from the protocol while still allowing an honest caller to generate a proof (assuming enough messages have been delivered of course).

5.10.2. General Multi-Party Protocols

If we are willing to fix the set of parties $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$ participating in a session of $\mathcal{F}_{VRO}$ from the start of the execution, then we could also let these parties compute arbitrary n-party functionalities between themselves and without having to interact with some fixed set of servers. If n is small this may be an acceptable, but note that as n grows the total amount of interaction generally grows much faster when compared to client-server protocols.

First, each client-server protocol immediately yields a protocol of this type by treating all parties as servers and assuming the protocol can be efficiently scaled to contain n servers. Going in the other direction, each protocol for n parties can be made into a protocol for n servers and additional clients by letting clients execute the code which was previously executed by any party wishing to query $\mathcal{F}_{VRO}$ and with these n servers.

6. Related Primitives

In this chapter, we compare our definition of $\mathcal{F}_{VRO}$ to some other functionalities, not necessarily in the UC framework, which try to achieve similar goals. We begin comparing our definition of a verifiable random oracle to the definition thereof in the previous master's thesis [35] in which they were originally defined. While we have already proved that the relaxed variant $\mathcal{F}_{VRO}^{q}$ of $\mathcal{F}_{VRO}$ in which we allow the simulator to learn the input q to each hash query is UC-realized by the main construction in [35], in this section we compare our two definitions directly. In the second part of the chapter, we compare $\mathcal{F}_{VRO}$ to a primitive *oblivious pseudo-random function evaluation* and augmentations thereof.

6.1. Comparing VRO Definitions

In this section, we compare our definition of $\mathcal{F}_{VRO}$ to the VRO definition contained in [35]. We begin by recalling the syntax used and then restate the game-based security offered by the definition in [35] (call this definition *game-based VRO*, or in short, gVRO). Then, we show that our definition is stronger, i.e. any protocol π UC-realizing $\mathcal{F}_{VRO}$ is a gVRO.

6.1.1. Syntax

According to [35], an n-party VRO VRO for domain D and codomain H is a tuple $(\mathscr{s}, \mathscr{q}, \mathscr{H}, \mathscr{v})$ where

$$\mathscr{s} : \mathbb{N} \to \mathsf{PK} \times \mathsf{SK}^n$$

is setup algorithm which, on input the security parameter λ, outputs a public key $\mathsf{pk} \in \mathsf{PK}$ and n secret keys $\mathsf{sk} = (\mathsf{sk}_i)_{i\in[n]} \in \mathsf{SK}^n$. Keys come from arbitrary sets PK and SK (which may depend on λ). After running $\mathscr{s}$ at the onset of the protocol, pk is made publicly known to all protocol parties while the sk_i are given to distinguished parties $\mathcal{P}_i$ which will later implement the VRO.

$$\mathscr{q} : \mathsf{PK} \times \mathsf{D} \to \mathsf{H} \times \mathsf{PROOF}$$

is a public interactive PPT algorithm which, on input a public key pk and an element q from the domain D, outputs a hash h from H as well as a proof π in PROOF. The component $\mathcal{H} = \mathcal{H}_{\mathsf{pk},\mathsf{sk}}$ is itself is a tuple $(\mathscr{h}^1_{\mathsf{pk},\mathsf{sk}_1}, \mathscr{h}^2_{\mathsf{pk},\mathsf{sk}_2}, \dots, \mathscr{h}^n_{\mathsf{pk},\mathsf{sk}_n})$ of interactive PPT algorithms with $\mathscr{h}^i_{\mathsf{pk},\mathsf{sk}_i}$ being run by party $\mathcal{P}_i$.[1] Finally,

$$\mathscr{v} : \mathsf{PK} \times \mathsf{D} \times \mathsf{H} \times \mathsf{PROOF} \to \{0, 1\}$$

[1] Think of each $\mathscr{h}^i_{\mathsf{pk},\mathsf{sk}_i}$ as an instance of an ITM in the UC framework.

is the verification procedure that receives as input a tuple (pk, q, h, π) from its domain and decides whether π is a valid proof under pk attesting to the fact that h is the correct hash for q. We write

$$(h, \pi) \leftarrow \langle \mathcal{q}, \mathcal{H}_{\mathsf{pk},\mathsf{sk}} \rangle(\mathsf{pk}', q)$$

for the result of $\mathcal{q}$ running with input (pk', q) while interacting with $\mathcal{H}_{\mathsf{pk},\mathsf{sk}}$. Corruption is modeled as giving the adversary control over a subset $\mathsf{Corr} \subseteq \{1, 2, \ldots, n\}$ of the $\mathcal{h}^i_{\mathsf{pk},\mathsf{sk}_i}$, thereby also gaining access to the sk_i as well as all queries q.

6.1.2. Security

Having defined the syntax, let $\mathsf{VRO} = (\mathcal{s}, \mathcal{q}, \mathcal{H}, \mathcal{v})$ now be a VRO with domain D, codomain H and other associated spaces as in Section 6.1.1. We recall the security definitions from [35].

Definition 6.1.1. VRO is called a game-based VRO (gVRO), if and only if the following conditions hold.

- **Completeness:** Proofs output by $\mathcal{q}$ are always accepted by $\mathcal{v}$. Formally, for all $\lambda \in \mathbb{N}$ and all $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathcal{s}(1^\lambda)$ it holds that

$$\forall (h, \pi) \leftarrow \langle \mathcal{q}, \mathcal{H}_{\mathsf{pk},\mathsf{sk}} \rangle(\mathsf{pk}, h) : \mathcal{v}(q, h, \pi) = 1$$

- **Pseudo-Determinism:** Hash values output by $\mathcal{q}$ for the same input q are consistent across invocations. Formally, for all $\lambda \in \mathbb{N}$ and all $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathcal{s}(1^\lambda)$, it holds that

$$\forall (h_1, \pi_1) \leftarrow \langle \mathcal{q}, \mathcal{H}_{\mathsf{pk},\mathsf{sk}} \rangle(\mathsf{pk}, h)\ \forall (h_2, \pi_2) \leftarrow \langle \mathcal{q}, \mathcal{H}_{\mathsf{pk},\mathsf{sk}} \rangle(\mathsf{pk}, h) : h_1 = h_2$$

- **Pseudorandomness:** Hash values output by $\mathcal{q}$ are pseudo-random. For any PPT adversary $\mathcal{A}$, the probability

$$\mathbf{Pr}[\mathsf{PR}(\mathcal{A}, \lambda) = 1]$$

is negligible in λ where the game PR is the one shown in Figure 6.1.

- **Weak Unforgeability:** Proofs are unforgeable in a sense similar to the EUF-CMA definition for signature schemes. For any PPT adversary $\mathcal{A}$, the probability

$$\mathbf{Pr}[\mathsf{wUF}(\mathcal{A}, \lambda) = 1]$$

is negligible in λ where the game wUF is the one shown in Figure 6.2.

Programming An addition to Definition 6.1.1 is made in the form of a *programming interface* to allow reductions to choose the output of a gVRO at some specified input. In [35], this is done by introducing $n + 1$ further algorithms/interfaces $\mathsf{prgm}_{\mathsf{pk}}$ and $\mathsf{PRGM}_{\mathsf{pk},\mathsf{sk}} = (\mathsf{prgm}^i_{\mathsf{pk},\mathsf{sk}_i})_{i \in [n]}$ where $\mathsf{prgm}_{\mathsf{pk}}$ is run by another party $\mathcal{P}$ and $\mathsf{prgm}^i_{\mathsf{pk},\mathsf{sk}_i}$ is an interface provided to $\mathcal{P}$ by $\mathcal{P}_i$. Informally, these interfaces allow $\mathcal{P}$ to alter hash values. This may, however, lead to the properties of Definition 6.1.1 no longer holding. For this reason, the concept of a *suitable programming interface* is introduced. It is defined in a context-dependent way, i.e. in [35] it is used in the context of the simulation-sound extractability of the Fiat-Shamir transform, but it roughly means that programming can not be recognized by the adversary.

$\mathsf{PR}(\mathcal{A}, \lambda)$	$\mathsf{Hash}(q)$
1 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathcal{g}(1^\lambda)$	1 : $M = M \cup \{q\}$
2 : $M = \emptyset$	2 : $(h, \pi) \leftarrow \langle \mathcal{q}, \mathcal{H}_{\mathsf{pk},\mathsf{sk}} \rangle (q)$
3 : $(q, \mathtt{state}) \leftarrow \mathcal{A}^{\mathsf{Hash}(\cdot)}(1^\lambda)$	3 : **return** (h, π)
4 : $(h_0, \pi_0) \leftarrow \langle \mathcal{q}, \mathcal{H}_{\mathsf{pk},\mathsf{sk}} \rangle (\mathsf{pk}, q)$	
5 : $h_1 \leftarrow \mathsf{H}$	$\mathsf{Verify}(q, h, \pi)$
6 : $b \leftarrow\$ \{0, 1\}$	1 : $b = v(\mathsf{pk}, q, h, \pi)$
7 : $b' \leftarrow \mathcal{A}(1^\lambda, \mathtt{state}, h_b)$	2 : **return** b
8 : **return** $b' = b \wedge x \notin M$	

Figure 6.1.: The Pseudorandomness game for a gVRO.

$\mathsf{wUF}(\mathcal{A}, \lambda)$	$\mathsf{Hash}(q)$
1 : $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathcal{g}(1^\lambda)$	1 : $M = M \cup \{q\}$
2 : $M = \emptyset$	2 : $(h, \pi) \leftarrow \langle \mathcal{q}, \mathcal{H}_{\mathsf{pk},\mathsf{sk}} \rangle (\mathsf{pk}, q)$
3 : $(q, h, \pi) \leftarrow \mathcal{A}^{\mathsf{Hash}(\cdot)}(1^\lambda, \mathsf{pk})$	3 : **return** (h, π)
4 : **return** $v(\mathsf{pk}, q, h, \pi) \stackrel{?}{=} 1 \wedge (q, h) \notin M$	

Figure 6.2.: The Weak Unforgeability game for a gVRO.

6.1.3. Comparison

We compare our definition of $\mathcal{F}_{VRO}$ to the above definition for a gVRO. To be able to do this we assume that in the setting with $\mathcal{F}_{VRO}$ the adversary is given the adversary interface and in particular provides vk and Prove.

Completeness A gVRO is required to be perfectly complete. The same is true if we consider proofs π which were output by $\mathcal{F}_{VRO}$, these will always verify. But even if we ignore proofs that are not delivered by the adversary (which seems sensible as gVRO does not consider network adversaries), there is one circumstance in which no proof is produced. This situation occurs for some input q with hash h, if the proof π output by Prove, is already marked as invalid by an existing record $(\mathtt{ver}, sid, q, h, \pi, \mathsf{vk}, 0)$ where vk is the correct verification key. Hence, depending on whether we allow the adversary not only to initialize $\mathcal{F}_{VRO}$, the completeness provided by $\mathcal{F}_{VRO}$ may be slightly weaker.[2]

Pseudo-Determinism Both definitions guarantee perfect consistency. After having been randomly selected upon the first hash query for q, the same h is returned by $\mathcal{F}_{VRO}$ in all subsequent queries. For $\mathcal{F}_{VRO}$, we again have to restrict to the case where output is indeed generated.

[2] If we do not allow the adversary to make verification queries and require proofs to be delivered, then a valid proof will always be received.

Pseudorandomness First, the ideal functionality $\mathcal{F}_{VRO}$ associates hashes h to inputs q uniformly using real randomness while PR only guarantees pseudo-randomness. This, however, only makes a difference when interacting with super-polynomial adversaries. Further, in the game PR, the adversary is not given the public key pk and so can not use it to break pseudo-randomness. An entity interacting with $\mathcal{F}_{VRO}$ can freely retrieve the verification key vk and use it within its attack.

Weak Unforgeability The unforgeability guarantees provided by $\mathcal{F}_{VRO}$ are perfect. No adversary is able to produce a valid proof π under the correct key vk and for some pair (q, h) such that either h is not the correct hash for q or q has not been hashed before. The same is true in a computational sense for any adversary in the wUF game. The adversary only wins by providing a valid proof π for some (q, h) such that (q, h) is not in the set M of input-hash pairs given and received by $\mathcal{A}$.

Programming A simulator for a protocol ξ in the $\mathcal{F}_{VRO}$-hybrid model is able to freely program the hashes generated by $\mathcal{F}_{VRO}$ as long as the resulting distribution is indistinguishable from uniform to the environment, i.e. either truly random, pseudo-random, or otherwise. As long as this holds, programming is undetectable to the adversary in any context.

6.1.4. Other Differences

In this section, we discuss some more differences between a gVRO and $\mathcal{F}_{VRO}$. These mostly arise due to differences between the *ad hoc* definitions of interactive protocols made in [35] and the UC framework.

Wrong Key From Definition 6.1.1 it is not clear what should happen upon executing $\mathscr{v}$ with input some $\mathsf{pk}' \neq \mathsf{pk}$. This seems to implicitly mean that no guarantees are made in such a case. Essentially, this kind of behavior is made explicit by having $\mathcal{F}_{VRO}$ let the simulator choose which proofs to accept or reject in such cases. One small difference, however, is that even for wrong keys $\mathcal{F}_{VRO}$ guarantees consistency. There is some justification for ignoring this issue as all honest parties receive the correct public key pk as output by $\mathscr{s}$ at the onset of the protocol and so are under little threat of executing $\mathscr{v}$ on a different key.

Leakage In [35], it is not clearly defined what information the adversary is supposed to maximally learn. The adversary is the entity providing the subset Corr of the $\mathscr{h}^i_{\mathsf{pk},\mathsf{sk}_i}$, but the information exchanged by $\mathscr{q}$ and the algorithms in $\mathscr{H}$ is not defined and depends on the specific protocol. Considering the PRF construction, the adversary is able to learn all of q, but only an additive share of h (if we disregard the adversary itself querying q).[3] With $\mathcal{F}_{VRO}$, the information leaked to the adversary is clearly defined as being the identity of the party making the request as well as the length $\|q\|$ of the input q.

[3] It supposedly also obtains the identity of the party making the query.

Corruption The definition of a gVRO does not consider the possibility of having corrupted callers, i.e. only the correct execution of $\mathcal{q}$ is considered in Definition 6.1.1. Depending on the protocol, this can make a large difference. While in the PRF construction, the correct behavior of the caller is easily checkable, i.e. that the sent value belongs to the domain D, the same is not true in e.g. the π_{FHE} protocol. If callers were assumed to be only semi-honest, the proof of knowledge of the secret FHE key could be removed among other simplifications.

Input Hashing In [35], it is noted that most instantiations can be optimized by first computing a collision-resistant hash $h = \mathsf{CRHF}(q)$ from the input q before sending it to the servers. As the input q has to be straight-line extractable for the simulator for any protocol UC-realizing $\mathcal{F}_{VRO}$, this technique can in general not be used.[4] It *could* be used if for example h was accompanied by some UC-commitment com on q and a proof of consistency π, but the added cost seems to greatly outweigh the potential benefits of being able to do all subsequent computations on the shorter h.

We note that if it was possible for a protocol for $\mathcal{F}_{VRO}$ to hash all inputs before sending them to the servers, then this would almost achieve the necessary input-hiding property. Only almost as it would leak the hash of the input which can of course not be computed by the simulator from the length of the input alone.

Simulatability Building on the last point, there are also general differences regarding the level of *simulatability* and hence composability. While game-based definitions such as Definition 6.1.1 do not in general compose, any protocol UC-realizing $\mathcal{F}_{VRO}$ will compose with itself and arbitrary other protocols in any manner. It was this UC simulatability that for example necessitated our use of NIWI in the protocol π_{FHE}.

6.2. Comparing VRO and OPRF Variations

In this section, we compare our definition $\mathcal{F}_{VRO}$ to another related primitive called *oblivious pseudo-random function evaluation* (OPRF). We begin by introducing the many variants and definitions for OPRF. Afterward, we investigate how certain strong variants of these can be used to generically instantiate $\mathcal{F}_{VRO}$. Last, we show the close relation between OPRF variants and $\mathcal{F}_{VRO}$ by constructing OPRF protocols using $\mathcal{F}_{VRO}$. In particular, we show that a special class of protocols UC-realizing $\mathcal{F}_{VRO}$ are in direct correspondence with protocols for a particular OPRF ideal functionality.

6.2.1. OPRF Variations

Informally, an OPRF protocol for a PRF $\mathsf{PRF} : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ allows two parties, a client holding some input $x \in \mathcal{X}$ and a server holding some key $k \in \mathcal{K}$, to compute the functionality

$$f_{\mathsf{PRF}}(x, k) = (\mathsf{PRF}(x, k), \bot).$$

[4] One might ask whether the simulator must make hash queries to $\mathcal{F}_{VRO}$ on behalf of corrupted parties, but this is indeed true to ensure that the environment is able to later verify the obtained proof by giving it to an honest verifier.

This means the client ends up with the result of applying PRF to (x, k) and the server learns nothing (this property is called *obliviousness*).

As this is a form of MPC, security of an OPRF protocol can be required to hold in either the stand-alone or the UC setting and with either party being allowed to be either semi-honest or maliciously corrupted. The security guarantee of the client involves the server learning nothing about its input and output and likewise, the guarantee for the server consists of the client learning nothing but the output, i.e. it gains no information on the key. Security might also be defined in a game-based fashion. For concrete definitions, see for example [26].

It is not quite sensible to directly compare variants of OPRF protocols to an ideal functionality like $\mathcal{F}_{VRO}$. The former is explicitly a two-party protocol with mutual security requirements and the latter involves only the honest or corrupted clients as physical parties interacting with the ideal $\mathcal{F}_{VRO}$. A protocol for realizing $\mathcal{F}_{VRO}$ might, however, involve clients and servers. As such, we *can* draw comparisons on the level of potential protocols for $\mathcal{F}_{VRO}$ and investigate which properties commonly found in OPRF protocols are achieved by them.

Motivation As a motivating example, consider a protocol π where a distinguished (trusted) party $\mathcal{S}$ acts as the server in a (UC-secure) OPRF protocol with other parties $\mathcal{P}_i$ and always inputs a fixed key k. This protocol realizes $\mathcal{F}_{VRO}$ (with interactive verification, assume for a moment that we allow this), but as the server is already guaranteed to be honest this is no different from $\mathcal{S}$ just being sent the inputs in the clear, evaluating PRF and returning the results. The only required property, i.e. that no (corrupted) client learns k, does still hold in this case.

If one considers a notion of security for a protocol realizing $\mathcal{F}_{VRO}$ where we allow full corruption of all servers and still want to retain some form of security, then using an OPRF protocol could be useful. It could prevent the corrupted server from learning the inputs of honest clients. On the other hand, the clients can no longer be assured that what they receive was indeed computed using the same key k, a guarantee like this lies outside what an (even UC-secure) OPRF protocol provides.

Randomness Guarantees Another difference in OPRF definitions is exemplified by the two UC-functionalities shown in Figures 6.3 and 6.4. For simplicity, both of these are for the case of a single client C and a single server $\mathcal{S}$, but a single session *sid* allows multiple evaluations. In Figure 6.3, the functionality evaluates a different random function for each key k input by the server. The functionality in Figure 6.4, on the other hand, evaluates an underlying PRF F in an oblivious manner. Using the former in particular does not even allow the (honest) server itself to predict outputs. To take care of the fact that a corrupted server can simply run the protocol for an honest client, the simulator receives the additional ability to freely evaluate the function whenever the server is corrupted. Think for example of a protocol where the final output is set as the output of $\mathcal{F}_{RO}$ (which is commonly the case for protocols realizing this kind of functionality). Then the adversary can freely query $\mathcal{F}_{VRO}$ and outputs to these queries have to be made consistent with whatever is output by $\mathcal{F}_{OPRF}$.

Parametrized by a domain $\mathcal{X}$ and codomain $\mathcal{Y}$. Let $F : \mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$ be a family of random functions.

Client Eval Upon receiving $(\texttt{Eval}, sid, eid, x)$ from some party C, check that $sid = (C, \mathcal{S}, sid')$ for some sid'. If it is not, ignore the message. Else, store $(\texttt{Eval}, eid, x)$ and send $(\texttt{Eval}, sid, eid)$ to the adversary. If there is a record $(\texttt{Key}, eid, k)$, send private delayed output $(\texttt{Answer}, sid, eid, x, F(k, y))$ to C.

Server Eval Upon receiving $(\texttt{Key}, sid, eid, k)$ from some party $\mathcal{S}$, check that $sid = (C, \mathcal{S}, sid')$ for some sid'. If it is not, ignore the message. Else, store $(\texttt{Key}, eid, k)$ and send $(\texttt{Key}, sid, eid)$ to the adversary. If there is a record $(\texttt{Eval}, eid, x)$, send private delayed output $(\texttt{Answer}, sid, eid, x, F(k, x))$ to C.

Corrupted Server Upon receiving $(\texttt{Predict}, sid, k, x)$ from the adversary and $\mathcal{S}$ is corrupted, send a message $(\texttt{Predicted}, sid, k, x, F(k, y))$ back to the adversary.

Figure 6.3.: An ideal OPRF functionality for evaluating a random function.

Parametrized by a PRF $F : \mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$.

Client Eval Upon receiving $(\texttt{Eval}, sid, eid, x)$ from some party C, check that $sid = (C, \mathcal{S}, sid')$ for some sid'. If it is not, ignore the message. Else, store $(\texttt{Eval}, eid, x)$ and send $(\texttt{Eval}, sid, eid)$ to the adversary. If there is a record $(\texttt{Key}, eid, k)$, send private delayed output $(\texttt{Answer}, sid, eid, x, F(k, y))$ to C.

Server Eval Upon receiving $(\texttt{Key}, sid, eid, k)$ from some party $\mathcal{S}$, check that $sid = (C, \mathcal{S}, sid')$ for some sid'. If it is not, ignore the message. Else, store $(\texttt{Key}, eid, k)$ and send $(\texttt{Key}, sid, eid)$ to the adversary. If there is a record $(\texttt{Eval}, eid, x)$, send private delayed output $(\texttt{Answer}, sid, eid, x, F(k, x))$ to C.

Figure 6.4.: An ideal OPRF functionality parametrized by a PRF F which it evaluates.

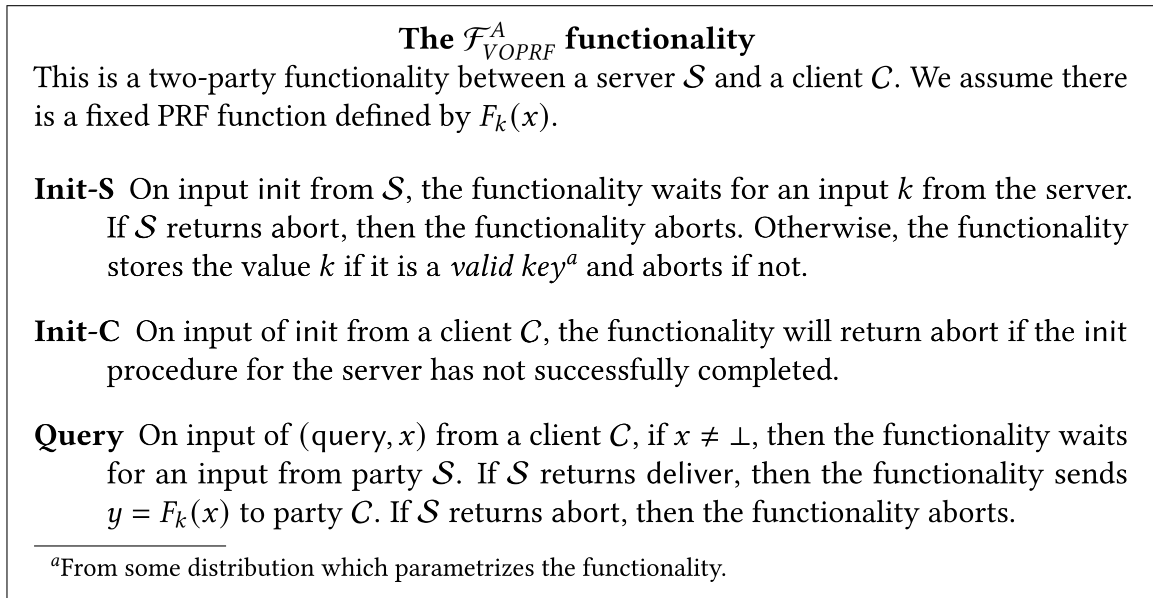
Augmented OPRF Notions There exist augmentations and additional properties which OPRF protocols may possess. For an overview see again [26]. The at first sight most promising properties for our application are *verifiability* and being *distributed* or *threshold* protocols. The latter two essentially mean that the key, instead of being held by a single server, is shared among a set of n servers. In the distributed case, to successfully complete an evaluation, all n servers need to act honestly with respect to their share of the key for output to be produced. In the threshold case, on the other hand, there is a threshold parameter t and at least $t + 1$ servers are required to behave correctly during evaluation. There are several differing formulations in terms of UC-functionalities for threshold OPRF protocols, e.g. in [68]. In most of these functionalities and also in the one contained in [68], the correctness of the result is only guaranteed if all $t + 1$ contacted servers are honest. Otherwise, the adversary may select a random function (a label identifying a function, but whose values are sampled randomly by the functionality) using which the result is computed.

Adding Verifiability Verifiability is a similarly broad term. In the literature on OPRFs, the modifier *verifiable* usually means that the client is able to verify that it received the correct output corresponding to its input x and *some* key k input by the server, i.e. that the client receives $\mathsf{PRF}(k, x)$. It may also be guaranteed that the same k is used for successive interactions, see $\mathcal{F}^{A}_{VOPRF}$ shown in Figure 6.5 and taken from [2]. This usually involves some initial commitment by the server on its key and with respect to which correct evaluation is proven by the server. This commitment might be transferable between parties and thus allow multiple parties or the same party to be sure that all outputs are correct with respect to a single fixed key. We, on the other hand, require *public verifiability* where a client inputting x receives both a purported output y as well as a proof of correct evaluation π. Using x, y, and π as well as some verification key vk, any other party can then later be convinced that indeed $y = \mathsf{PRF}(k, y)$ for some key k (to which vk acts as a commitment). Note that this is strictly stronger than the intermediate notion. Proofs of correct evaluation by the server in the intermediate notion might involve interaction while public verifiability requires non-interactive verifiability long after the evaluation has concluded.

The weakest kind of verifiability is already achieved by using an actively secure OPRF protocol of which there exist multiple for different underlying PRFs. For the intermediate notion there are multiple UC-functionalities in the literature. For example does the functionality $\mathcal{F}^{J}_{VOPRF}$ in [66] (shown in Figure 6.6) allow for the explicit transfer of commitments to PRF keys while $\mathcal{F}^{A}_{VOPRF}$ in [2] only allows a single client to verify that the same key is used in sequential evaluations. Note that $\mathcal{F}^{A}_{VOPRF}$ is formulated with instantiations not relying on authenticated channels in mind which complicates the description.

Publicly verifiable OPRF protocols are essentially protocols for obliviously evaluating VRFs and can sometimes be obtained from the intermediate kind by making the server's zero-knowledge proofs non-interactive (of course this comes at the cost of requiring the ROM or some other setup, but many UC-secure OPRF protocols already use PRFs such as $\mathsf{2HashDH}$[5] which use random oracles). Alternatively, any VRF can generically be evaluated obliviously using FHE and letting the server prove correct evaluation. Depending

[5]This function is essentially given by $H(x, H'(x)^k)$ for H and H' modeled as random oracles.



The $\mathcal{F}^{A}_{VOPRF}$ functionality

This is a two-party functionality between a server $\mathcal{S}$ and a client C. We assume there is a fixed PRF function defined by $F_k(x)$.

Init-S On input init from $\mathcal{S}$, the functionality waits for an input k from the server. If $\mathcal{S}$ returns abort, then the functionality aborts. Otherwise, the functionality stores the value k if it is a *valid key*[a] and aborts if not.

Init-C On input of init from a client C, the functionality will return abort if the init procedure for the server has not successfully completed.

Query On input of (query, x) from a client C, if $x \neq \bot$, then the functionality waits for an input from party $\mathcal{S}$. If $\mathcal{S}$ returns deliver, then the functionality sends $y = F_k(x)$ to party C. If $\mathcal{S}$ returns abort, then the functionality aborts.

[a] From some distribution which parametrizes the functionality.

Figure 6.5.: An ideal VOPRF functionality (taken from [2], figure 1).

on the randomness guarantees one wishes to achieve, care has to be taken that the VRF is simulatable. Otherwise, the VRF verification key already commits the server to a fixed function and even the ability of the simulator to simulate the proofs of correct evaluation (of the server's computation) does not allow it to program.

Explicit descriptions of publicly verifiable OPRF functionalities as we have described them are quite rare in the literature. The reason for this is that instantiating them always relies on some form of authentication to ensure that honest parties are able to retrieve an authentic copy of the correct verification key vk associated with some server $\mathcal{S}$ with which they wish to engage in an evaluation. Assuming infrastructure to provide this authentication, i.e. a public-key infrastructure, is oftentimes unnecessary for the applications in which verifiable OPRF protocols are used. Most protocols realizing a functionality such as $\mathcal{F}^{J}_{VOPRF}$ can, however, be adapted to be publicly verifiable in our stronger sense by adding authentication and non-interactive proofs.

In a first step, we may upgrade $\mathcal{F}^{J}_{VOPRF}$ to be publicly verifiable with interactive verification. Concretely, servers execute the **Key Generation** task of $\mathcal{F}^{J}_{VOPRF}$ and publicize their received token π during an initialization step. A client wishing to evaluate the function associated with $\mathcal{S}$ first retrieves the token π. It then engages in the **V-OPRF Evaluation** task, thereby obtaining output as well as a second token π'. Output is generated by the client if and only if $\pi = \pi'$. Public verification is possible by executing an evaluation and comparing the resulting output and tokens as before. All communication in this protocol is assumed to be authenticated. Depending on the specific protocol, verifiability may then be upgraded to be non-interactive by making the zero-knowledge proofs performed by the server non-interactive in an appropriate manner.

The $\mathcal{F}^{J}_{VOPRF}$ functionality

Key Generation On message $(\texttt{KeyGen}, sid)$ from $\mathcal{S}$, forward $(\texttt{KeyGen}, sid, \mathcal{S})$ to the adversary. On message $(\texttt{Parameter}, sid, \mathcal{S}, \pi, M)$ from the adversary, ignore this call if $\mathsf{param}(\mathcal{S})$ is already defined. Otherwise, set $\mathsf{param}(\mathcal{S}) = \langle \pi \rangle$ and initialize $\mathsf{tickets}(\pi) = 0$, and $\mathsf{hist}(\pi)$ to the empty string. If $\mathcal{S}$ is honest send $(\texttt{Parameter}, sid, \pi)$ to party $\mathcal{S}$, else parse M as a circuit with l-bit output and insert (π, M) in CorParams.

V-OPRF Evaluation On message $(\texttt{Eval}, sid, \mathcal{S}, x)$ from party $\mathcal{U}$ for sender $\mathcal{S}$, record $(\mathcal{U}, x)$ and forward $(\texttt{Eval}, sid, \mathcal{U}, \mathcal{S})$ to the adversary. On message $(\texttt{SenderComplete}, sid, \mathcal{S})$ from the adversary for some honest $\mathcal{S}$, output $(\texttt{SenderComplete}, sid)$ to party $\mathcal{S}$ and set $\mathsf{tickets}(\pi) = \mathsf{tickets}(\pi) + 1$ for π s.t. $\langle \pi \rangle = \mathsf{param}(\mathcal{S})$. On message $(\texttt{UserComplete}, sid, \mathcal{U}, \pi, \mathsf{flag})$ from the adversary, recover $(\mathcal{U}, x)$ and:

- If $\mathsf{flag} = \top$ and $\langle \pi \rangle = \mathsf{param}(\mathcal{S})$ for an honest $\mathcal{S}$ then: If $\mathsf{tickets}(\pi) \leq 0$ ignore the UserComplete request of the adversary. Otherwise: (1) if $\mathsf{hist}(\pi)$ includes a pair $\langle x, \rho' \rangle$, set $\rho = \rho'$, else sample ρ at random from $\{0,1\}^l$ and enter $\langle x, \rho \rangle$ into $\mathsf{hist}(\pi)$; (2) Set $\mathsf{tickets}(\pi) = \mathsf{tickets}(\pi) - 1$ and output $(\texttt{Eval}, sid, \rho)$ to party $\mathcal{U}$.
- Else, if $\mathsf{flag} = \perp$ then return $(\texttt{Eval}, \pi, \perp)$ to $\mathcal{U}$.
- Else, if $\mathsf{flag} = \top$ and π is such that $(\pi, M) \in$ CorParams for some circuit M, compute $\rho = M(x)$, enter $\langle x, \rho \rangle$ in $\mathsf{hist}(\pi)$, output $(\texttt{Eval}, sid, \rho)$ to party $\mathcal{U}$.

Figure 6.6.: A second ideal VOPRF functionality (taken from [66], figure 2).

Adding Commitments Committed inputs and/or outputs are a modification of OPRFs such that the server, instead of obtaining no output, obtains commitments on the input and/or the output of the client. These commitments can later be used to ensure that the client uses the output in some specified way. We might for example use this to first let the client obtain the hash h for input q and then blindly signs the message (q, h). Enforcing that the correct message is being signed could be ensured using the commitments on q and h; although at that point it might be easier to use a publicly verifiable scheme.

Some further remarks and explanations can be found in Section A.3.2 of the appendix.

6.2.2. Comparison

Armed with the knowledge on OPRF variants and ideal functionalities, we now compare them to $\mathcal{F}_{VRO}$. As we have pointed out there is a multitude of definitions and so this comparison can only be broad in nature.

- All OPRF functionalities explicitly involve one or more (in the threshold setting) servers while $\mathcal{F}_{VRO}$ does not. This also means that the OPRF functionality may behave differently based on whether the server is corrupted or not, i.e. some guarantees may be broken[6], while protocols for $\mathcal{F}_{VRO}$ have to be able to handle corruption without compromising security.
- Relating to the last point, one major goal of ours was to find protocols for $\mathcal{F}_{VRO}$ which allow the maximal number of corrupted servers. The same question does not arise when considering protocols realizing OPRF functionalities. $\mathcal{F}_{VRO}$ thus differs from most OPRF functionalities in terms of the type of protocols we wish to construct for them.
- Most OPRF functionalities fully hide the client's input from the server and the adversary while $\mathcal{F}_{VRO}$ leaks the length of the input. This minor difference mostly stems from the fact that we did not outright prohibit the possibility of $\mathcal{F}_{VRO}$ having $\{0, 1\}^*$ as the domain. One could easily define a functionality $\mathcal{F}'_{VRO}$ which does not leak the length and all of our protocols from Chapter 5 immediately UC-realize also this functionality for finite domains.[7]
- Some OPRF functionalities are parametrized by and evaluate an underlying function for which the server inputs a key k while others have the functionality evaluate a random function. $\mathcal{F}_{VRO}$ uses the second mechanism to generate hashes.
- There are many notions of verifiability for OPRF protocols with public verifiability and non-interactive verification being the strongest notion. This latter notion is the one employed by $\mathcal{F}_{VRO}$, but is very rare among OPRF protocols. There, the most

[6] Observe for example that a corrupted server interacting with $\mathcal{F}^{J}_{VOPRF}$ may register an arbitrary M using which outputs are computed.

[7] If the length of inputs is non-constant, some padding to a common length may have to be applied by the client.

common notion is the one of having keys vk committing the server to some key k, but using it to only ensure that the same key is used across different evaluations (possibly by different users by transferring vk).

- In relation to the last point as well as the first and second points, allowing multiple servers in protocols not only allows protocols for $\mathcal{F}_{VRO}$ to potentially cope with a number of servers being corrupted, it also may allow using weaker primitives to achieve public verifiability.
- Some OPRF functionalities do not require inputs by corrupted parties to be efficiently extractable by relying on ticketing mechanisms while $\mathcal{F}_{VRO}$ does require this to be possible.

6.2.3. Evaluating the FHE Construction

We briefly evaluate the protocol π_{FHE} from Chapter 5 with respect to it achieving properties that are characteristic of OPRF protocols.

In π_{FHE}, one might think that we were constructing a publicly verifiable OPRF using an FHE scheme, signatures, and a PRF, and running three instances of it between the caller and any of the four servers. This is not the case. In fact, what we constructed is not verifiable in the sense that even a malicious server holding a key can not produce a valid proof for a wrong pair (q, h), i.e. is committed to *some* function. A server could just encrypt and sign wrong PRF outputs. We achieved verifiability by relying on the responses from honest servers to cross-check and discard potentially wrong results by the corrupted server. In that sense, we do not require full public verifiability against malicious servers, but only against semi-honest servers.

In fact, we explicitly refrained from constructing a full publicly verifiable OPRF protocol for efficiency reasons. As stated previously, as an alternative to NIWI proofs we could have let servers prove the correctness of their FHE evaluations of the PRF and the signing algorithm (after also having the servers publish commitments to their PRF keys with respect to which this proof is then constructed). This would essentially mean that we would be using the generic protocol for obliviously evaluating a PRF using FHE, but even in that case, nothing prevents the corrupted server from signing wrong pairs (q, h).

To summarize, the sub-protocol of π_{FHE} between the client and a single server possesses only some of the properties which are required of a publicly verifiable OPRF protocol and overall public verifiability is achieved only by running multiple copies of this protocol with a collection of servers where three out of four of the servers are known to be honest.

6.3. Generic Constructions from OPRF

In this section, we sketch some generic constructions of protocols for $\mathcal{F}_{VRO}$ based on OPRF protocols. We chose to present these here and not in Chapter 5 for the following reason. Many protocols for the kind of OPRF variants which we use below are either in the $\mathcal{F}_{RO}$-hybrid model and/or employ NIZKPoKs. As our protocol π_{FHE} does require neither of these primitives and it was a stated goal not to use them, we consider π_{FHE} to be the

main protocol and present the current section merely to show that certain kinds of OPRF protocols are themselves sufficient (but not necessary) to construct instantiations of $\mathcal{F}_{VRO}$.

Single Party Another kind of primitive that essentially immediately gives a single party instantiation (where this party is allowed to be corrupted) are the publicly verifiable OPRFs mentioned above and where the functionality evaluates a random function as in Figure 6.3. Obliviousness ensures that not even a corrupted server learns the input q and hash h. The fact that a random function is evaluated ensures that the simulator can bias results to the h it receives from $\mathcal{F}_{VRO}$. Lastly, public verifiability provides publicly, non-interactively verifiable proofs of correct evaluation.

Multiple Parties To distribute this simple protocol there are several ways, each of them achieving a different kind of *distributedness* and *robustness.*

Notions of Distributedness Call a protocol for computing some (non-interactive) primitive P (or UC-functionality representing an instance of P) a *threshold protocol* if it allows setting parameters (t, n) where n is the total number of servers and t is the corruption threshold. Call it a *distributed protocol* if it is a threshold protocol, but only allows setting $t = n - 1$, i.e. all but one party may be corrupted before the adversary is able to evaluate P using the obtained secret state. Observe that for threshold protocols it is possible to set $2t + 1 < n$, i.e. have $> t + 1$ honest servers which in principle can evaluate P on their own. In that case (and assuming a fully robust protocol allowing verification of individual shares returned by the servers) an honest client is able to evaluate P by querying all n servers and assuming that the adversary delivers at least $t + 1$ of the honest server's responses. The same is not true for a distributed protocol. For that reason, it is often much easier to come up with distributed protocols as opposed to ones allowing arbitrary thresholds. As all shares have to be valid, it is enough to have them be an additive sharing of the result as opposed to some (t, n) secret sharing. The former does not have to involve any coordination between the servers.

For example, it is relatively easy to distribute a PRF using a VRF (or another way of proving to the client that the server behaved honestly, this does not necessarily require public verifiability) than to construct a distributed threshold PRF. The client simply interacts with n servers, each holding a different key, on the same input x and obtains both outputs y_i and proofs π_i for $i \in [n]$, and aborts if one of the results is incorrect. The output is then the sum $y = \bigoplus_{i=1}^{n} y_i$ of the individual outputs. On the other hand, it is much more involved to give a threshold protocol for a PRF.

The Constructions We give both a fully distributed as well as a threshold protocol. For both of them, it is sufficient for the OPRF protocol to evaluate a PRF instead of a random function. They are, however, required to be fully input-extractable and so can for example not be of the same type as $\mathcal{F}_{VOPRF}^{J}$.

- **Distributed:** Each server is associated with a single session of a publicly verifiable OPRF. The total verification key vk is the collection of the n individual keys vk_i. Proofs for input q are generated by having the caller engage in protocol executions

with each of the n servers, thereby obtaining outputs $(h_i, \pi_i)_{i \in [n]}$. Only once all n of these values have been obtained, the final hash $h = \bigoplus_{i=1}^{n} h_i$ is computed and the proof π is constructed as

$$\pi = (h_1, \pi_1, h_2, \pi_2, \ldots, h_n, \pi_n).$$

To verify some (q, h, π), it is checked that the h_i contained in π sum to h, and the π_i are valid under the respective key vk_i. Security follows by the fact that the simulator will be able to evaluate the functions associated with the corrupted servers. It can then change the outputs corresponding to one of the honest parties such that the sum of this output and the predicted outputs by the other servers matches the value obtained from $\mathcal{F}_{VRO}$. Security follows by replacing the PRF used by this honest parties' ideal functionality with a random function and observing that the result of a sum is uniform as long as at least one summand is uniform. Security crucially hinges on the fact that even corrupted servers are bound to either produce consistent values or not respond at all.

- **Threshold:** By distributing equal keys among parties as in π_{FHE}, strong robustness can be achieved. Each subset of three of the four servers is assigned a key k_1. On input q, a caller engages in three oblivious evaluations with each server. An interaction with one of the servers is considered to have succeeded only once all three evaluations have successfully concluded. As soon as at least three of the four interactions have succeeded, the hash h is computed in the obvious way (all partial hashes obtained from successful interactions are guaranteed to be correct, so the caller may select and sum one of them for each k_i). All partial proofs are included in the final proof π. Valid proofs must contain at least two valid partial proofs for each of the four keys as well as partial hashes summing to h. Security follows by arguments similar to those proving the security of π_{FHE}. In contrast to π_{FHE}, however, this protocol can cope with having a client receive responses from only three of the four servers whereas a caller in π_{FHE} always had to wait until a message had been received from all four servers.

Starting from Weaker OPRF Variants So far, we have constructed protocols for $\mathcal{F}_{VRO}$ from one of the strongest forms of OPRF protocol, i.e. publicly verifiable ones. It seems difficult to make black-box use of weaker forms of OPRF protocols for the following reasons. As $\mathcal{F}_{VRO}$ itself is publicly verifiable, we somehow have to add this property to the OPRF protocol. If we were to use an ordinary OPRF protocol, we would have to overcome the following problems. First, if we begin by letting the client evaluate the OPRF on its input q, thereby obtaining the hash h, how do we force it to keep using h in the subsequent computation of the proof? If we do not do this, then a corrupted client can replace h with some different value h' and potentially obtain a valid proof for the pair (q, h'). This breaks unforgeability. Second, how do we enforce servers using the same key k across different evaluations and make this verifiable not only to the party doing the evaluation but also to any verifier? Again it seems like we have to let servers publish commitments to their keys and require them to prove that they keep using the same key across different evaluations in a publicly verifiable manner. The latter strategy essentially leads us back to publicly verifiable OPRF protocols.

6.3.1. Relation between OPRF and Hybrid Instantiations

We can also generically construct a hybrid instantiation (see Section 5.7 for what this means) generically using a publicly verifiable OPRF. These will have even better properties than the construction given in Section 5.7 as they also hide the input in the full corruption case. The core idea is identical. Instead of directly outputting the output h' generated by the OPRF protocol, the client first hashes the input q and adds the result $h^* = \mathsf{H}$ to h' to obtain the final hash h.

In case the server is not corrupted and H is only collision resistant, security follows by reducing to the collision-resistance of H. If the server is corrupted, on the other hand, the simulator can evaluate the function to which the server has committed itself (and which may not be pseudo-random) and can program H correctly.

Again, the reason why we have chosen not to present this construction in Section 5.7 is the fact that it uses the very strong primitive of a publicly verifiable OPRF protocol. While this yields hiding of inputs even in the full corruption case, doing so comes at a cost (which can be seen by the fact that all efficient such protocols make use of random oracles and we explicitly wish to eliminate them in the case where H is modeled as a standard-model function).

6.4. Using a Concrete OPRF

Having looked at generic ways of constructing protocols UC-realizing $\mathcal{F}_{VRO}$ using any publicly verifiable OPRF protocol, we will now be more concrete. As stated before, one motivation for not presenting the generic construction above in Chapter 5 was the fact that many protocols of this type use random oracles and NIZKPoKs. In [66], a protocol for evaluating the Naor-Reingold (NR) PRF [75] is given which does not use random oracles. We will be evaluating the generic threshold construction above when instantiated with this protocol and compare it to π_{FHE}.

The Naor-Reingold PRF The setting for the NR PRF PRF_{NR} are cyclic groups $\mathbb{G}$ of prime order q. Let $\mathcal{X} = \{0,1\}^l$ be the desired domain. The key-space $\mathcal{K}$ then consists of $l+1$-tuples in $\mathbb{Z}_q$, i.e. $\mathcal{K} = \mathbb{Z}_q^{l+1}$. To evaluate PRF_{NR} on some input $x = (x_1, x_2, \ldots, x_l)$ and key $a = (a_0, a_1, \ldots, a_l)$, one computes

$$\mathsf{PRF}_{NR}(a, x) = g^{a_0 \prod_{j=1}^{l} a_j^{x_j}}.$$

To show the pseudo-randomness of this family of functions one relies on the *Decisional Diffie-Hellman* (DDH) assumption.

Oblivious Evaluation of the NR PRF We briefly describe the protocol for evaluating PRF_{NR} in an oblivious and verifiable manner. As it is given in [66], this protocol is not fully publicly verifiable. It is only verifiable in the intermediate notion defined in Section 6.2.1 where parties holding a verification key vk can be sure that the correct function is used in subsequent evaluations. However, it can be made publicly verifiable by making the interactive proofs used in the protocol non-interactive and forming a proof π them.

Intuitively, the protocol is based on the following protocol for obliviously evaluating PRF_{NR} (missing verifiability and with $a_0 = 1$ being fixed). On input $x = (x_1, \ldots, x_l)$, the client and server engage in l oblivious-transfers (OT) where in the i'th invocation the client inputs x_i and the server inputs $a_i r_i$ and r_i where $r_1, \ldots, r_l$ are random exponents. The server also sends $A = g^{1/r_1 \ldots r_k}$. Clearly, the obtained information allows the client to compute $\mathsf{PRF}_{NR}(a, x)$.

Making this protocol verifiable, the verification key vk consists of the $l+1$ group elements $g^{a_0}, \ldots, g^{a_l}$. The generic OT protocol is replaced by a specific protocol. Concretely, the client constructs l ciphertexts containing either $(0, 1)$ or $(1, 0)$, depending on the l'th bit of the input, under a key-pair $(\mathsf{pk}, \mathsf{sk})$ generated by the client. The encryption scheme is required to be (singly) homomorphic.[8] The server transforms these ciphertexts homomorphically into ciphertexts of $(r_i, 0)$ and $(0, a_i r_i)$. By decrypting these evaluated ciphertexts, the client is able to compute the output.

During all of the interaction, both parties use various Σ-protocols which have been strengthened using techniques from [51] to prove that they are behaving correctly.[9] These *could* be made non-interactive in an efficient manner by using the Fiat-Shamir transform. We, however, wish to evaluate the viability of a construction for $\mathcal{F}_{VRO}$ using this protocol. By working in the ROM, this evaluation would become meaningless as we have a much more efficient protocol at our disposal in this case. Thus, making the proofs non-interactive may incur expensive reductions to NP-complete languages.

Conclusion We compare π_{FHE} and the generic threshold construction from Section 6.3 using the protocol just described.

- **Proof size:** Instantiating π_{FHE} with SPS and GS-proofs as laid out in Section 5.4.10, proofs consist of four NIWI proofs π_i. Each of these can be represented by a small constant number of group elements. On the other hand, the type of statements involved in the protocol for evaluating PRF_{NR} seem to be much larger yielding larger proofs.

- **Assumptions:** As we have already noted, making the above protocol non-interactive requires the full capabilities of NIZK while π_{FHE} requires the (slightly) weaker notion of NIWI proofs.

- **Interaction:** Both protocols do not require any interaction between the servers during hash queries. Similar amounts of interaction are required during the initialization phase.

- **Computational Overhead:** It is quite difficult to compare this aspect due to the genericity of π_{FHE} and us not having specified how to make the protocol for evaluating PRF_{NR} publicly verifiable. If we are only concerned with the efficiency of the verifier (in practice, these will most likely be the entities with the least computational resources), then again due to the smaller proof sizes, π_{FHE} seems to have an edge in this regard.

[8] In [66], Paillier encryption is used.

[9] This requires using a common reference string.

6.5. VOPRF from VRO

In this section, we investigate whether $\mathcal{F}_{VRO}$ may be used to UC-realize some (publicly verifiable) $\mathcal{F}_{VOPRF}$ with domain $\mathcal{X}$ and codomain $\mathcal{Y}$.

6.5.1. Naive Approach

We begin by showing that a naive protocol does not work. At first sight, $\mathcal{F}_{VRO}$ *is* already a publicly verifiable $\mathcal{F}_{VOPRF}$. A client, on input q, makes a hash query for q to $\mathcal{F}_{VRO}$ and obtains a hash h and proof π. To later verify π, a verification query is made (after having obtained the verification key vk). The problem with this approach is that all OPRF variants guarantee that the client on its own can not evaluate the function without the server agreeing for this to occur. If we consider the corresponding protocol in the $\mathcal{F}_{RO}$-hybrid model obtained by applying Theorem 5.1.1, we see that this would lead to a protocol where the client simply evaluates $\mathcal{F}_{RO}$, i.e. a local function, and this clearly does not involve any server.[10] If, on the other hand, we think about our protocol π_{FHE} from Section 5.4, we see that the insufficiency of this protocol can also be *partially* explained by our modeling choice of not explicitly including the servers as physical parties in the description of $\mathcal{F}_{VRO}$.[11] In π_{FHE}, no client can successfully complete a hash query without interacting with at least some of the servers. But letting the $\mathcal{F}_{VOPRF}$ server be the entity providing the functionality of all the π_{FHE} servers neither solves all of the problems. In this new situation, a corrupted server would be free to choose arbitrary hashes as the consistency property for π_{FHE} requires three of the four servers to be honest. Indeed, this is what we mentioned above regarding us *not* having constructed a publicly verifiable OPRF using FHE and signatures, but relying on the honest servers to correct potential deviations by the corrupted server.

The full situation is thus the following. There are protocols ξ which UC-realize $\mathcal{F}_{VRO}$ in a client-server model with a single server and which at the same time UC-realize a publicly verifiable variant of $\mathcal{F}_{VOPRF}$ with the same distribution of roles. These protocols ξ are essentially the protocols for single parties described in Section 6.3. When they are considered as protocols for $\mathcal{F}_{VRO}$, while they only UC-realize $\mathcal{F}_{VRO}$ when the server is honest, some properties such as obliviousness and unforgeability of proofs are retained even if the server is corrupted. Taken together, these two results establish a kind of equivalence between protocols for a publicly verifiable $\mathcal{F}_{VOPRF}$ and for $\mathcal{F}_{VRO}$ in the specific model of a single server and where obliviousness and unforgeability of proofs hold also for a corrupted server.

There is not quite an exact equivalence between protocols for $\mathcal{F}_{VRO}$ with a single server and publicly verifiable OPRF protocols due to the fact that, as we have seen, there are other types of protocols for $\mathcal{F}_{VRO}$ which make use of a trusted server and which thus do not yield OPRF protocols as they do not have to hide inputs.

[10] It would indeed be publicly verifiable, however.

[11] Which was deliberate as we are modeling $\mathcal{F}_{VRO}$ as providing a public random function, i.e. an abstract *service*, to the other protocol parties and wish to abstract from the physical form this may take in any concrete instantiation.

6.5.2. Arguments Against an Unconditional Construction

In the last paragraph, we have only shown that the protocol which simply evaluates $\mathcal{F}_{VRO}$ does not work. We now show that it is unlikely that there unconditionally exists a protocol ξ in the $\mathcal{F}_{VRO}$-hybrid model which yields a publicly verifiable $\mathcal{F}_{VOPRF}$. Again by the fact that $\mathcal{F}_{RO}$ can be seen as an idealized version of $\mathcal{F}_{RO}$, ξ would immediately yield a protocol ξ' for $\mathcal{F}_{VOPRF}$ in the $\mathcal{F}_{RO}$-hybrid model. Translating this into simpler terms, it would yield a protocol where both client and server have access to a hash function modeled as a random oracle for verifiably and obliviously evaluating a PRF. It is thus sufficient to argue that even in the $\mathcal{F}_{RO}$-hybrid model there can not exist such a ξ'.

The arguments against such a protocol are the following:

- If ξ' has non-trivial verification keys $\mathsf{vk} \neq \perp$, then computing from these the secret key k held by the server would have to be infeasible, i.e. computing vk from k would have to be a one-way function.
- If ξ', on the other hand, has trivial verification keys $\mathsf{vk} = \perp$, then the correct evaluation of the function represented by a session of $\mathcal{F}_{VOPRF}$ must be verifiable using only access to $\mathcal{F}_{RO}$ and making queries which are efficiently derivable from the pair (q, h) to be verified.

6.5.3. Relying on Computational Assumptions

One possibility for circumventing the results of the last section is to rely on computational intractability assumptions in constructing a protocol. Similarly to the applications, we considered in Chapter 4, this would entail adapting an existing (or new) protocol in the $\mathcal{F}_{RO}$-model to the $\mathcal{F}_{VRO}$-hybrid model and showing that the resulting protocol remains sound. We argue that this is generally the case in Section A.8 of the appendix.

7. Future Work

In this chapter, we take a look at what may be interesting venues for future research on the topic of VROs.

7.1. Adaptive Adversaries

Within this thesis, we have only considered static adversaries where the set of corrupted parties is fixed at the start and known to the simulator. As a stronger notion of corruption, adaptive adversaries choose the parties which they corrupt dynamically over the course of protocol execution. Depending on whether erasures are allowed, the adversary learns either only the current internal state of the party at the time of corruption or is given access to all past states.

Briefly considering an adaptive adversary attacking our protocol π_{FHE}, we see that the simulation strategy crucially hinges on the fact that the simulator is able to choose the PRF key k for which to replace the PRF with a random function such that the corrupted server does not know k. It thus seems reasonable to assume that there is *no* valid simulator for an adaptive adversary and so finding a protocol secure against adaptive adversaries, either with erasures or without, is left for future work.

7.2. Standalone Security

It might be interesting to consider instantiations of $\mathcal{F}_{VRO}$ when considering it as a functionality in the stand-alone model of security. The main difference is that there environment and adversary are not allowed to communicate during the execution of the protocol. Only after the execution has ended, which for reactive functionalities happens by choice of the environment, the view of the adversary is given to the environment which based on this information makes its decision between real and ideal interaction.

In particular, this allows the simulator to rewind its black-box simulation of the adversary. Only the final view has to be correctly distributed. One way this manifests itself is a simplification of π_{FHE} where the UC-secure ideal functionalities for zero-knowledge as well as the MPC protocol used during initialization of the servers are replaced by their stand-alone counterparts. Apart from these changes, there may exist other protocols which are overall more simple.

7.3. More Tasks

Some more variations and extensions of $\mathcal{F}_{VRO}$ may be investigated. One may for example add further tasks to $\mathcal{F}_{VRO}$ which allow things such as *batch queries*, i.e. hash queries containing multiple inputs $(q_1, q_2, \ldots, q_n)$, *batch verification*, or *proof aggregation*, i.e. proofs π_i attesting to the correctness of pairs (q_i, h_i) may be combined into a single proof π which can then be verified using a separate task.

One may also investigate other types of proofs than those of correct evaluation. An example for this are *range-type* proofs which assert that the hash h for some input q lies in a subset $\mathcal{S} \subset \mathcal{H}$ of the full codomain $\mathcal{H}$. A special kind of such proofs arise when $\mathcal{S}$ is equal to $\mathcal{H} \setminus \{h\}$ for some element $h \in \mathcal{H}$. Using such proofs, a party would be able to show that the hash for some input q is not equal to h, but is otherwise unconstrained.

7.4. Weaker Randomness Guarantees

The current formulation of $\mathcal{F}_{VRO}$ requires hash values to be computationally indistinguishable from uniform as well as independent for every single input. It does not matter whether the queried inputs are themselves related. A common relaxation of this requirement, e.g. seen in weak PRFs [31] or deterministic PKE [4], is that the entropy of hashes is related to the entropy of inputs, i.e. highly correlated inputs are allowed to have similar hashes associated to them.

This would at the same time be a modeling task of expressing such a requirement in the UC framework. As one can no longer allow the environment to freely and adaptively choose inputs, the hash query task would have to be appropriately adapted.

7.5. Global VRO

The *generalized UC* framework (GUC) [24] allows the definition of global functionalities which are able to be used not only by parties within a single session of some protocol ξ, but are globally available. It may be interesting to define a global version $\mathcal{G}_{VRO}$ of $\mathcal{F}_{VRO}$ and consider its limitations. In particular, a comparison with variants of the global random oracle functionality $\mathcal{G}_{RO}$, e.g. as defined in [15], may be done.

7.6. More Efficient Instantiations

As we have noted before, the protocol we defined in Section 5.4 is quite inefficient based on our use of both FHE and NIWI. Eliminating both of these primitives and thus coming up with a practically efficient instantiation is an open question. Another aspect is the scalability with respect to the number of corrupted parties. By being based on the PRF construction from [35] we inherited the same scalability issues mentioned there.

8. Conclusion

In this thesis, we have explored the concept of verifiable random oracles as a way to replace the heuristic security provided by the random oracle methodology with provable security. In this chapter, we summarize our results.

Verifiable Random Oracles We began by motivating and defining an ideal functionality $\mathcal{F}_{VRO}$ in the UC framework. Compared to the random oracle functionality $\mathcal{F}_{RO}$, this functionality allows the creation and verification of proofs of correct evaluation. This was motivated by an analysis of how random oracles are commonly used within protocols to check that some other party correctly computed a hash value. The definition we made allows protocols realizing $\mathcal{F}_{VRO}$ to have interactive generation of proofs, but forces verification of proofs to be non-interactive.

Applications We then applied our model to two ROM applications, FDH signatures and the Fischlin transform. We showed that both of these can be transferred soundly into the VROM while utilizing the ability to non-interactively verify proofs to retain the non-interactive verifiability of FDH signatures and proofs of Fischlin-transformed zero-knowledge proofs.

Instantiations Having shown the applicability of our definition, we proceeded to show that there are protocols that satisfy it. First, we showed an instantiation based on random oracles, thereby proving that our definition is a relaxation of random oracles. Second, we gave two instantiations where a single trusted party provides the VRO. Last, we gave an instantiation where multiple servers jointly provide a VRO and where a subset of the servers may be statically and maliciously corrupted. It is based on fully-homomorphic encryption.

Related Definitions In Chapter 6, we first contrasted our definition in the UC framework with the game-based definition from [35]. We showed that our definition is stronger once we take the necessary relaxations by working in the UC framework, such as having to give the adversary the ability to delay the delivery of proofs, into account. We also compared it to (variations of) definitions for OPRF protocols.

Bibliography

[1] Masayuki Abe et al. “Structure-preserving signatures and commitments to group elements”. In: *Annual Cryptology Conference*. Springer. 2010, pp. 209–236.

[2] Martin R. Albrecht et al. “Round-optimal verifiable oblivious pseudorandom functions from ideal lattices”. In: *IACR International Conference on Public-Key Cryptography*. Springer. 2021, pp. 261–289.

[3] Matilda Backendal et al. “The Fiat-Shamir zoo: relating the security of different signature variants”. In: *Nordic Conference on Secure IT Systems*. Springer. 2018, pp. 154–170.

[4] Mihir Bellare, Alexandra Boldyreva, and Adam O’Neill. “Deterministic and efficiently searchable encryption”. In: *Annual International Cryptology Conference*. Springer. 2007, pp. 535–552.

[5] Mihir Bellare, Viet Tung Hoang, and Sriram Keelveedhi. “Instantiating random oracles via UCEs”. In: *Annual Cryptology Conference*. Springer. 2013, pp. 398–415.

[6] Mihir Bellare and Phillip Rogaway. “Optimal asymmetric encryption”. In: *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer. 1994, pp. 92–111.

[7] Mihir Bellare and Phillip Rogaway. “Random oracles are practical: A paradigm for designing efficient protocols”. In: *Proceedings of the 1st ACM Conference on Computer and Communications Security*. 1993, pp. 62–73.

[8] Mihir Bellare and Phillip Rogaway. “The exact security of digital signatures-How to sign with RSA and Rabin”. In: *International conference on the theory and applications of cryptographic techniques*. Springer. 1996, pp. 399–416.

[9] Josh Benaloh and Michael de Mare. “One-way accumulators: A decentralized alternative to digital signatures”. In: *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer. 1993, pp. 274–285.

[10] M BLUM. “How to generate cryptographically strong sequences of pseudo-random bits”. In: *SIAM J. Comput.* 13 (1984), pp. 850–864.

[11] Dan Boneh and Xavier Boyen. “Efficient selective-ID secure identity-based encryption without random oracles”. In: *International conference on the theory and applications of cryptographic techniques*. Springer. 2004, pp. 223–238.

[12] Dan Boneh, Ben Lynn, and Hovav Shacham. “Short signatures from the Weil pairing”. In: *International conference on the theory and application of cryptology and information security*. Springer. 2001, pp. 514–532.

[13] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) fully homomorphic encryption without bootstrapping". In: *ACM Transactions on Computation Theory (TOCT)* 6.3 (2014), pp. 1–36.

[14] Jan Camenisch and Anja Lehmann. "Privacy-preserving user-auditable pseudonym systems". In: *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2017, pp. 269–284.

[15] Jan Camenisch et al. "The wonderful world of global random oracles". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2018, pp. 280–312.

[16] Jan Camenisch et al. "Universal composition with responsive environments". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2016, pp. 807–840.

[17] Ran Canetti. "Towards realizing random oracles: Hash functions that hide all partial information". In: *Annual International Cryptology Conference*. Springer. 1997, pp. 455–469.

[18] Ran Canetti. "Universally composable security: A new paradigm for cryptographic protocols". In: *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE. 2001, pp. 136–145.

[19] Ran Canetti. "Universally composable signature, certification, and authentication". In: *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004*. IEEE. 2004, pp. 219–233.

[20] Ran Canetti, Yilei Chen, and Leonid Reyzin. "On the correlation intractability of obfuscated pseudorandom functions". In: *Theory of cryptography conference*. Springer. 2016, pp. 389–415.

[21] Ran Canetti and Marc Fischlin. "Universally composable commitments". In: *Annual International Cryptology Conference*. Springer. 2001, pp. 19–40.

[22] Ran Canetti, Oded Goldreich, and Shai Halevi. "The random oracle methodology, revisited". In: *Journal of the ACM (JACM)* 51.4 (2004), pp. 557–594.

[23] Ran Canetti, Daniel Shahaf, and Margarita Vald. "Universally composable authentication and key-exchange with global PKI". In: *Public-Key Cryptography–PKC 2016*. Springer. 2016, pp. 265–296.

[24] Ran Canetti et al. "Universally composable security with global setup". In: *Theory of Cryptography Conference*. Springer. 2007, pp. 61–85.

[25] Ran Canetti et al. "Universally composable two-party and multi-party secure computation". In: *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. 2002, pp. 494–503.

[26] Sílvia Casacuberta, Julia Hesse, and Anja Lehmann. "SoK: Oblivious Pseudorandom Functions". In: *Cryptology ePrint Archive* (2022).

[27] Melissa Chase and Anna Lysyanskaya. "Simulatable VRFs with applications to multi-theorem NIZK". In: *Annual International Cryptology Conference*. Springer. 2007, pp. 303–322.

[28] Sanjit Chatterjee and Alfred Menezes. “Type 2 structure-preserving signature schemes revisited”. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2015, pp. 286–310.

[29] Ran Cohen, Abhi Shelat, and Daniel Wichs. “Adaptively secure MPC with sublinear communication complexity”. In: *Annual International Cryptology Conference*. Springer. 2019, pp. 30–60.

[30] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. “Proofs of partial knowledge and simplified design of witness hiding protocols”. In: *Annual International Cryptology Conference*. Springer. 1994, pp. 174–187.

[31] Ivan Damgåard and Jesper Buus Nielsen. “Expanding pseudorandom functions; or: From known-plaintext security to chosen-plaintext security”. In: *Annual International Cryptology Conference*. Springer. 2002, pp. 449–464.

[32] I Damgård. *On Sigma-Protocols*. `http://www.daimi.au.dk/~ivan/Sigma.pdf`.

[33] Alexander W Dent et al. “Confidential signatures and deterministic signcryption”. In: *International Workshop on Public Key Cryptography*. Springer. 2010, pp. 462–479.

[34] Yvo G Desmedt. “Threshold cryptography”. In: *European Transactions on Telecommunications* 5.4 (1994), pp. 449–458.

[35] Karsten Diekhoff. “Verifiable Random Oracles”. MA thesis. Karlsruher Institut für Technologie (KIT), 2021. DOI: `10.5445/IR/1000135426`.

[36] Whitfield Diffie and Martin E. Hellman. “New directions in cryptography”. In: *IEEE Trans. Information Theory* 22.6 (1976), pp. 644–654. DOI: `10.1109/TIT.1976.1055638`. URL: `https://doi.org/10.1109/TIT.1976.1055638`.

[37] Marten van Dijk et al. “Fully homomorphic encryption over the integers”. In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2010, pp. 24–43.

[38] Yevgeniy Dodis. “Efficient construction of (distributed) verifiable random functions”. In: *International Workshop on Public Key Cryptography*. Springer. 2003, pp. 1–17.

[39] Yevgeniy Dodis, Roberto Oliveira, and Krzysztof Pietrzak. “On the generic insecurity of the full domain hash”. In: *Annual International Cryptology Conference*. Springer. 2005, pp. 449–466.

[40] Yevgeniy Dodis and Aleksandr Yampolskiy. “A verifiable random function with short proofs and keys”. In: *International Workshop on Public Key Cryptography*. Springer. 2005, pp. 416–431.

[41] Léo Ducas and Daniele Micciancio. “FHEW: bootstrapping homomorphic encryption in less than a second”. In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2015, pp. 617–640.

[42] Taher ElGamal. “A public key cryptosystem and a signature scheme based on discrete logarithms”. In: *IEEE transactions on information theory* 31.4 (1985), pp. 469–472.

[43] Paul Feldman. “A practical scheme for non-interactive verifiable secret sharing”. In: *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*. IEEE. 1987, pp. 427–438.

[44] Amos Fiat and Adi Shamir. “How to prove yourself: Practical solutions to identification and signature problems”. In: *Conference on the theory and application of cryptographic techniques*. Springer. 1986, pp. 186–194.

[45] Marc Fischlin. “Communication-efficient non-interactive proofs of knowledge with online extractors”. In: *Annual International Cryptology Conference*. Springer. 2005, pp. 152–168.

[46] Nils Fleischhacker et al. “Pseudorandom signatures”. In: *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. 2013, pp. 107–118.

[47] Eduarda SV Freire, Julia Hesse, and Dennis Hofheinz. “Universally composable non-interactive key exchange”. In: *International Conference on Security and Cryptography for Networks*. Springer. 2014, pp. 1–20.

[48] Georg Fuchsbauer and Michele Orrù. “Non-interactive zaps of knowledge”. In: *International Conference on Applied Cryptography and Network Security*. Springer. 2018, pp. 44–62.

[49] Jun Furukawa and Yehuda Lindell. “Two-thirds honest-majority MPC for malicious adversaries at almost the cost of semi-honest”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 1557–1571.

[50] David Galindo et al. “Fully distributed verifiable random functions and their application to decentralised random beacons”. In: *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2021, pp. 88–102.

[51] Juan A Garay, Philip MacKenzie, and Ke Yang. “Strengthening zero-knowledge protocols using signatures”. In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2003, pp. 177–194.

[52] Rosario Gennaro et al. “Secure distributed key generation for discrete-log based cryptosystems”. In: *Journal of Cryptology* 20.1 (2007), pp. 51–83.

[53] Craig Gentry. “Fully homomorphic encryption using ideal lattices”. In: *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 2009, pp. 169–178.

[54] Craig Gentry, Shai Halevi, and Nigel P Smart. “Fully homomorphic encryption with polylog overhead”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2012, pp. 465–482.

[55] Craig Gentry, Amit Sahai, and Brent Waters. “Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based”. In: *Annual Cryptology Conference*. Springer. 2013, pp. 75–92.

[56] Craig Gentry and Alice Silverberg. “Hierarchical ID-based cryptography”. In: *International conference on the theory and application of cryptology and information security*. Springer. 2002, pp. 548–566.

[57] Oded Goldreich. *Foundations of Cryptography, Volume 2.* Cambridge university press Cambridge, 2004.

[58] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. "How to construct random functions". In: *25th Annual Symposium onFoundations of Computer Science, 1984.* IEEE. 1984, pp. 464–479.

[59] Oded Goldreich and Yair Oren. "Definitions and properties of zero-knowledge proof systems". In: *Journal of Cryptology* 7.1 (1994), pp. 1–32.

[60] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. "The knowledge complexity of interactive proof systems". In: *SIAM Journal on computing* 18.1 (1989), pp. 186–208.

[61] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. "A digital signature scheme secure against adaptive chosen-message attacks". In: *SIAM Journal on computing* 17.2 (1988), pp. 281–308.

[62] Jens Groth. "Efficient fully structure-preserving signatures for large messages". In: *International Conference on the Theory and Application of Cryptology and Information Security.* Springer. 2015, pp. 239–259.

[63] Jens Groth. "Simulation-sound NIZK proofs for a practical language and constant size group signatures". In: *International Conference on the Theory and Application of Cryptology and Information Security.* Springer. 2006, pp. 444–459.

[64] Jens Groth, Rafail Ostrovsky, and Amit Sahai. "New techniques for noninteractive zero-knowledge". In: *Journal of the ACM (JACM)* 59.3 (2012), pp. 1–35.

[65] Susan Hohenberger and Brent Waters. "Short and stateless signatures from the RSA assumption". In: *Annual International Cryptology Conference.* Springer. 2009, pp. 654–670.

[66] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. "Round-optimal password-protected secret sharing and T-PAKE in the password-only model". In: *International Conference on the Theory and Application of Cryptology and Information Security.* Springer. 2014, pp. 233–253.

[67] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. "OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques.* Springer. 2018, pp. 456–486.

[68] Stanisław Jarecki et al. "TOPPSS: cost-minimal password-protected secret sharing based on threshold OPRF". In: *International Conference on Applied Cryptography and Network Security.* Springer. 2017, pp. 39–58.

[69] Antoine Joux. "A one round protocol for tripartite Diffie–Hellman". In: *International algorithmic number theory symposium.* Springer. 2000, pp. 385–393.

[70] Neal Koblitz and Alfred J Menezes. "The random oracle model: a twenty-year retrospective". In: *Designs, Codes and Cryptography* 77.2 (2015), pp. 587–610.

[71] Yashvanth Kondi et al. "Improved Straight-Line Extraction in the Random Oracle Model With Applications to Signature Aggregation". In: *Cryptology ePrint Archive* (2022).

[72] Anna Lysyanskaya. "Unique signatures and verifiable random functions from the DH-DDH separation". In: *Annual International Cryptology Conference*. Springer. 2002, pp. 597–612.

[73] Blum Manuel. "Coin flipping by telephone". In: *IEEE Workshop on Communications Securit, A Report on CRYPTO'81*. 1981, pp. 11–15.

[74] Silvio Micali, Michael Rabin, and Salil Vadhan. "Verifiable random functions". In: *40th annual symposium on foundations of computer science (cat. No. 99CB37039)*. IEEE. 1999, pp. 120–130.

[75] Moni Naor and Omer Reingold. "Number-theoretic constructions of efficient pseudo-random functions". In: *Journal of the ACM (JACM)* 51.2 (2004), pp. 231–262.

[76] Moni Naor and Moti Yung. "Universal one-way hash functions and their cryptographic applications". In: *Proceedings of the twenty-first annual ACM symposium on Theory of computing*. 1989, pp. 33–43.

[77] Rafail Ostrovsky, Anat Paskin-Cherniavsky, and Beni Paskin-Cherniavsky. "Maliciously circuit-private FHE". In: *Annual Cryptology Conference*. Springer. 2014, pp. 536–553.

[78] Pascal Paillier. "Public-key cryptosystems based on composite degree residuosity classes". In: *International conference on the theory and applications of cryptographic techniques*. Springer. 1999, pp. 223–238.

[79] David Pointcheval and Olivier Sanders. "Short randomizable signatures". In: *Cryptographers' Track at the RSA Conference*. Springer. 2016, pp. 111–126.

[80] David Pointcheval and Jacques Stern. "Security arguments for digital signatures and blind signatures". In: *Journal of cryptology* 13.3 (2000), pp. 361–396.

[81] Alfredo De Santis et al. "Robust non-interactive zero knowledge". In: *Annual International Cryptology Conference*. Springer. 2001, pp. 566–598.

[82] Claus-Peter Schnorr. "Efficient identification and signatures for smart cards". In: *Conference on the Theory and Application of Cryptology*. Springer. 1989, pp. 239–252.

[83] Adi Shamir. "How to share a secret". In: *Communications of the ACM* 22.11 (1979), pp. 612–613.

[84] Nigel P Smart and Frederik Vercauteren. "Fully homomorphic encryption with relatively small key and ciphertext sizes". In: *International Workshop on Public Key Cryptography*. Springer. 2010, pp. 420–443.

A. Appendix

A.1. Standard Definitions

We recall standard definitions for cryptographic primitives.

A.1.1. Pseudo-Random Functions

One of the classical cryptographic primitives are *pseudo-random functions* (PRF) [58]. A PRF PRF with key-space $\mathcal{K}$, domain $\mathcal{X}$, and codomain $\mathcal{Y}$ is a function PRF : $\mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$ such that no PPT algorithm $\mathcal{A}$ can distinguish between $\mathsf{G}^{\mathrm{prf}}_{\mathcal{A},\mathsf{PRF}}(\lambda, 0)$ and $\mathsf{G}^{\mathrm{prf}}_{\mathcal{A},\mathsf{PRF}}(\lambda, 1)$ shown in Figure A.1, except with negligible probability. $\mathcal{A}$ is given access to an oracle $\mathcal{O}_b$ which is either given as $\mathsf{PRF}(k, \cdot)$ for a randomly sampled key k or by a truly random function RF, implemented below via lazy sampling.
We define the advantage $\mathsf{Adv}^{\mathrm{prf}}_{\mathcal{A},\mathsf{PRF}}(\lambda)$ of $\mathcal{A}$ to be

$$\mathsf{Adv}^{\mathrm{prf}}_{\mathcal{A},\mathsf{PRF}}(\lambda) = \left| \mathbf{Pr}\left[\mathsf{G}^{\mathrm{prf}}_{\mathcal{A},\mathsf{PRF}}(\lambda, 0) = 1\right] - \mathbf{Pr}\left[\mathsf{G}^{\mathrm{prf}}_{\mathcal{A},\mathsf{PRF}}(\lambda, 1) = 1\right]\right| \tag{A.1}$$

Definition A.1.1 (Pseudo-Random Function)**.** PRF is a secure PRF iff for all PPT algorithms $\mathcal{A}$, $\mathsf{Adv}^{\mathrm{prf}}_{\mathcal{A},\mathsf{PRF}}(\lambda)$ is a negligible function in λ.

A.1.2. Trapdoor One-Way Permutations

A *trapdoor one-way permutation* (TDOWP) consists of a PPT algorithm Gen which, on input the security parameter 1^λ, outputs descriptions of two functions f, f^{-1}. Both f and f^{-1} are permutations on domain $\mathcal{X}_f$. For all $x \in \mathcal{X}_f$, it holds that

$$f^{-1} \circ f(x) = f \circ f^{-1}(x) = x$$

i.e. f and f^{-1} are inverses of each other. As indicated by the subscript, we allow $\mathcal{X}_f$ to depend on the specific permutation but require $\mathcal{X}_f$ to be efficiently recognizable and samplable when given f. We will later drop the dependence on f and only write $\mathcal{X}$.

Security is defined by the usual definition for one-way functions where we fix the distribution from which pre-images are sampled to be the uniform distribution. The challenger generates some function key f using Gen and samples a random element x from the domain. It hands f and x to the adversary $\mathcal{A}$ and the adversary wins if it outputs x. Note that due to the fact that f is a permutation, there are no other elements x' with $f(x') = f(x)$ and so the exact element sampled by the challenger has to be found.

$G^{prf}_{\mathcal{A},\mathsf{PRF}}(\lambda, 0)$	$G^{prf}_{\mathcal{A},\mathsf{PRF}}(\lambda, 1)$
$k \leftarrow\$ \mathcal{K}$	$L[\cdot] \leftarrow \bot$
return $\mathcal{A}^{O}(1^\lambda)$	**return** $\mathcal{A}^{O}(1^\lambda)$
$O_0(x)$	$O_1(x)$
return $\mathsf{PRF}(k, x)$	**if** $L[x] \neq \bot$ **do**
	return $L[x]$
	fi
	$y \leftarrow\$ \mathcal{Y}$
	$L[x] = y$
	return y

Figure A.1.: The PRF security game.

$G^{td-owp}_{\mathcal{A},\mathsf{Gen}}(\lambda)$
$(f, f^{-1}) \leftarrow \mathsf{Gen}(1^\lambda)$
$x \leftarrow \mathcal{X}_f$
$y = f(x)$
$x' \leftarrow \mathcal{A}(1^\lambda, f, y)$
return $x \stackrel{?}{=} x'$

Figure A.2.: The TDOWP security game.

Definition A.1.2 (Trapdoor One-Way Permutation)**.** A PPT algorithm Gen defines a TDOWP, if for every PPT adversary $\mathcal{A}$ and the game $G^{td-owp}_{\mathcal{A},\mathsf{Gen}}$ shown in Figure A.2

$$\mathbf{Pr}\left[G^{td-owp}_{\mathcal{A},\mathsf{Gen}}(\lambda) = 1\right]$$

is negligible is λ.

Example One example of a trapdoor permutation is the RSA function. There, Gen on input 1^λ first samples two random λ-bit primes P and Q. It then computes $N = PQ$ and $\phi(N) = (P-1)(Q-1)$. An element e with $\gcd(\phi(N), e)$ is selected and d with $de \equiv 1 \pmod N$ is computed using the *extended euclidean algorithm.* The function key f then consists of (N, e) and f^{-1} contains (N, d). To apply f to an element x from the domain $\mathcal{X}_f = \mathbb{Z}_N$, $y = x^e \mod N$ is computed. To apply f^{-1} to y, compute x as $x = y^d \mod N$. The RSA function is a TDOWP under the *RSA assumption.*

A.1.3. Digital Signature Schemes

Like ordinary signatures, *digital signatures* [36] can be used to prove both the integrity and authenticity of messages originating from some sender. A digital signature scheme SIG with message space $\mathcal{M}$ is a three-tuple of algorithms $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$. We will now describe the purpose of each algorithm in turn and then define the correctness and security of signature schemes.

The algorithm Gen, on input the unary security parameter, outputs a pair $(\mathsf{vk}, \mathsf{sk})$ consisting of a verification key vk and a signing key sk. The second algorithm Sign receives the signing key sk as well as a message $m \in \mathcal{M}$ and outputs a signature string σ. Finally, the verification algorithm Verify takes in the verification key vk, a message m and the purported signature σ for m, and outputs a bit $b \in \{0, 1\}$ which represents the decision as to whether σ is a valid signature for m. A signature scheme SIG is called *correct*, if

$\mathsf{sEUF\text{-}CMA}(\mathcal{A}, \lambda)$	$\mathsf{Sign}(m)$
$Q \leftarrow \emptyset$	$\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$
$(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Gen}(1^\lambda)$	$Q = Q \cup \{(m, \sigma)\}$
$(m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(\cdot)}(1^\lambda, \mathsf{vk})$	**return** σ
return $\mathsf{Verify}(\mathsf{vk}, \sigma^*, m^*) \stackrel{?}{=} 1$ $\wedge\ (m^*, \sigma^*) \notin Q$	

Figure A.3.: The sEUF-CMA security game.

$$\forall\, (\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Gen}(1^\lambda)\ \forall m \in \mathcal{M}\ \forall\, \sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m) : \mathsf{Verify}(\mathsf{vk}, \sigma, m) = 1 \tag{A.2}$$

Security For the definition of security of a signature scheme SIG we employ the notion(s) of *(strong) existential unforgeability under chosen message attacks* ((s)EUF-CMA) [61]. The definition is via a game, shown in Figure A.3, where an adversary $\mathcal{A}$ interacts with a challenger C. C begins by generating a key-pair $(\mathsf{sk}, \mathsf{vk})$ and giving vk to $\mathcal{A}$. An oracle Sign is made accessible to $\mathcal{A}$ and which on input a message m returns an honestly generated signature σ using sk. Oracle queries are allowed to be adaptive, i.e. they may depend on previously received signatures by the oracle. When $\mathcal{A}$ decides to end the game it outputs a pair (m^*, σ^*) and wins if σ^* is a valid signature for m^*.

Definition A.1.3 (Digital Signature Scheme)**.** SIG is sEUF-CMA-secure iff for all PPT adversaries $\mathcal{A}$ the probability

$$\mathsf{Adv}^{\mathrm{euf-cma}}_{\mathcal{A},\mathsf{SIG}}(\lambda) := \mathbf{Pr}[\mathsf{sEUF\text{-}CMA}(\mathcal{A}, \lambda) = 1]$$

is negligible in λ. SIG is EUF-CMA-secure iff the probability is negligible in the altered game where the check $(m^*, \sigma^*) \notin Q$ is replaced by the check that m^* is not among the first components of elements in Q.

A.2. An Ideal VRO Functionality with Algorithmic Verification

In this section, we define an alternative version of $\mathcal{F}_{VRO}$ where the adversary not only supplies an algorithm Prove for the generation of proofs, but also a second algorithm Verify to verify proofs without in some cases having to contact the adversary to do so.

We first compare the differences between the two versions. Then we show some problems which exist for this conceptually simpler definition and how they can be solved. Finally, we argue why we chose to define $\mathcal{F}_{VRO}$ the way we did.

Differences Let us call the alternative functionality $\mathcal{F}_{VRO}^{alt}$. It is shown fully in Figure A.4 and is based on the 2005 version of Canetti's signature functionality [18][1]. The differences between $\mathcal{F}_{VRO}$ and $\mathcal{F}_{VRO}^{alt}$ are as follows.

- During initialization, instead of providing one algorithm Prove and an arbitrary string vk, two algorithms (Prove, Verify) are asked to be provided. Verify is required to be stateless and deterministic. The code of Verify is returned instead of vk to any party sending an `Init` message.
- There are no longer any `ver` tuples that have to be stored to ensure consistency (of verification queries). Only pairs of the form (q, h) have to be stored to ensure the consistency (of hashing queries).
- Whenever a proof is generated using Prove it is checked using Verify. If the proof is rejected then $\mathcal{F}_{VRO}^{alt}$ halts and outputs a (Completeness error).
- Verification queries have to include an algorithm Verify′ the decision whether to accept or reject a verification query $(q, h, \pi, \text{Verify}')$ is always determined by simply running Verify′ on (q, h, π), conditioned on the fact that a no forgery occurs. In case of a forgery, which is determined to have occurred when the provided Verify′ is equal to the correct Verify and 1 is returned despite h not being set as the hash of q, a (Unforgeability error) is output and $\mathcal{F}_{VRO}^{alt}$ halts.

Applying these changes arguably makes the functionality simpler to state, more symmetric between the different types of queries as well as more private with respect to verification queries. There is, however, also a cost that has to be paid and which we describe next.

Problems In this alternative formulation of $\mathcal{F}_{VRO}$, Verify has to be a plain algorithm such that it can be executed within $\mathcal{F}_{VRO}$. This in particular means that there can be no accesses to functionalities such as a random oracle functionality $\mathcal{F}_{RO}$ within Verify although we can easily imagine protocols where this is the case. It is important to note that such calls to ideal functionalities do not conflict with the non-interactivity of verification requests in $\mathcal{F}_{VRO}$, i.e. that the simulator can not delay answers to such requests, as calls to ideal functionalities are only *interactive* if they, in turn, allow the adversary to delay the delivery of responses.

[1] The 2005 version can be found here: `https://eprint.iacr.org/archive/2000/067/20051214:064128`

Solutions There are two solutions to this problem, see also the discussion in Section 3.1 of [15]. One is the solution we have opted for in $\mathcal{F}_{VRO}$, i.e. letting the adversary only choose a verification key vk and having verification of proofs partially involve interaction with the adversary. Another is to let Verify contain special instructions denoting a call to some external functionality, e.g. a random oracle. $\mathcal{F}_{VRO}$ would then implement these instructions by supplying the correct result after making a query to obtain it. For $\mathcal{F}_{VRO}$ to be able to obtain these results it is necessary to restrict to *global functionalities*. Such functionalities exist in both the real as well as the ideal world and are directly accessible to the environment[2]. This solution is quite unsatisfactory as it is relatively non-standard and requires the use of global functionalities which may not always suffice.

Conclusion To conclude, we have opted not to choose this conceptually simpler version and thereby give up the total obliviousness of the adversary to verification requests. The reason for this choice lies in the main reason which led us to define $\mathcal{F}_{VRO}$ in the first place. As a relaxation of a random oracle. Only by allowing the verification procedure in a protocol for $\mathcal{F}_{VRO}$ to access ideal functionalities do we allow the protocol where both hash and verification queries are implemented as calls to the ideal random oracle functionality $\mathcal{F}_{RO}$ to UC-realize $\mathcal{F}_{VRO}$. Only in that case do we feel like we are able to claim that we have achieved our goal.

[2]Technically this is the case only within the EUC framework as opposed to the GUC framework [24], but both frameworks are ultimately equivalent as proven in [24] (in the sense that the granted security is equivalent, there exist protocols which can only be expressed in the GUC framework).

The $\mathcal{F}_{VRO}^{alt}$ functionality

Initialization Upon receiving a value ($\texttt{Init}, sid$) from party $\mathcal{P}$, if this is the first time that ($\texttt{Init}, sid$) was received, send ($\texttt{Init}, sid$) to the adversary. Wait for an answer ($\texttt{Init}, sid, \mathsf{Prove}, \mathsf{Verify}$) where Prove is the description of a stateless PPT TM and Verify is the description of a stateless deterministic TM, store these machines. Send public delayed output with either the just received or stored Verify as ($\texttt{Key}, sid, \mathsf{Verify}$) to $\mathcal{P}$.

Hashing Upon receiving a value ($\texttt{Hash}, sid, q$) from party $\mathcal{P}$, if there is no stored (q, h) for some h, let $h \leftarrow \mathcal{H}$. In either case proceed to send the adversary a message ($\texttt{Hashing}, sid, \mathcal{P}, \|q\|$) and wait for an answer ($\texttt{SimInfo}, sid, \mathcal{P}, s$). Let $\pi \leftarrow \mathsf{Prove}(q, h, s)$ and verify that $\mathsf{Verify}(q, h, \pi) = 1$. If so, then send private delayed output ($\texttt{HashProof}, sid, q, h, \pi$) to $\mathcal{P}$ and store (q, h). Else, output an error message (Completeness error) to $\mathcal{P}$ and halt.

Verification Upon receiving a value ($\texttt{Verify}, sid, q, h, \pi, \mathsf{Verify}'$) from some party $\mathcal{V}$, do the following: If $\mathsf{Verify} = \mathsf{Verify}'$, $\mathsf{Verify}(q, h, \pi) = 1$ and no entry (q, h) is recorded, then output an error message (Unforgeability error) to $\mathcal{V}$ and halt. Else output ($\texttt{Verified}, sid, q, h, \pi, \mathsf{Verify}'(q, h, \pi)$) to $\mathcal{V}$.

Figure A.4.: An alternative variant of the $\mathcal{F}_{VRO}$ functionality where proofs are verified algorithmically.

A.3. Additional Remarks

This section contains further remarks and technical asides referenced in the main part of the thesis.

A.3.1. Remarks about the FHE Protocol

Remarks about the protocol π_{FHE} defined in Section 5.4.4.

Infinite Domains If the domain in which the q lie is finite, e.g. $\{0,1\}^{p(\lambda)}$ for some polynomial p, fixed (in the security parameter λ) circuits for computing $\mathsf{PRF}(k_i, \cdot)$ and $\mathsf{SIG.Sign}(\mathsf{sk}, \cdot)$ may be generated after initialization and used for each query. If this is not the case, generating new circuits upon encountering an input q which has a new size may be necessary. Depending on the specifics of the FHE scheme it may also be possible to evaluate a circuit C_R representing the round-function of a hash-function repeatedly on parts of some large ciphertext c, finally obtaining a ciphertext c' containing the same plaintext as would have been the result of evaluating some larger circuit representing the whole hash-function (suitably unrolled) on c.

Partial Hashes and Proof Structure Instead of including the four partial hashes $\{h_i\}_{i\in[4]}$ and letting the verifier check whether $h = \bigoplus_{i=1}^{4}$ we may safe some space by only including $\{h_i\}_{i\in[3]}$ (say) and letting verifiers compute $h_4 = h \oplus \bigoplus_{i=1}^{3}$. The rest of the verification process would remain unchanged. It is easy to see that these two formulations are essentially equivalent, but including all four h_j yields a slightly nicer presentation.

Letting Callers Register Keys Proving knowledge of the secret key by the caller could be replaced by using a key-registration functionality $\mathcal{F}_{KR}$ (shown in Figure A.5 and taken in slightly adapted form from [47]) which allows parties to register keys, but requires proving knowledge of the corresponding secret key to do so successfully. The caller would then only send a ciphertext c of q under its personal and static secret key and a server would look up the public key for the calling party and from then on proceed as before. This change increases efficiency by reducing the number of proofs each party has to perform and verify (as well as the number of keys that have to be generated). It would further simplify the communication pattern between the caller and one of the servers towards a simple two-message interaction with one message being sent by each party. The downside is of course the introduction of an additional trusted party.

Responses by All Four Servers It is unfortunate that we have to rely on receiving responses by all four servers. Intuitively, as long as the responses contain at least two valid signatures for each (q, h_i, i), a proof could be created. Later during simulation, however, we can not rely on the response by the corrupted server (see Section 5.4.6)[3] To see why this requirement implies that also honest parties in the real world have to wait until they have received *some* response by all four servers, consider the following scenario. Let C be an honest

[3]Briefly, the simulator is not allowed to decrypt the response by the corrupted server as this would provide something akin to a decryption oracle to the adversary and would thus require a form of active security from FHE. As FHE is inherently malleable, this can not be added as an additional assumption.

party wishing to create a proof. Consider the case where $\mathcal{D}$ delivers the responses by the honest servers, but not by the corrupted server; a proof has to be returned. Also consider a second query where $\mathcal{D}$ lets the corrupted server either behave honestly or dishonestly and in addition delivers two responses by honest servers. As long as the identity of the corrupted server is unknown, and this is the case in the real interaction, and $\mathcal{D}$ lets it behave honestly, this interaction is indistinguishable from the first and so a valid proof has to be produced. On the other hand, if $\mathcal{D}$ lets the server behave dishonestly, no valid proof could and would be produced from the remaining two honest responses. A real party can distinguish these two cases by decrypting all received ciphertexts, but the simulator can not do so and hence can not distinguish. Thus, this case has to be explicitly excluded by the protocol.

Letting Servers Generate Their Own Keys We would wish to allow the servers to generate and publish their own SIG key-pairs, especially since we do not require the simulator to know the signing key of the corrupted server in order to simulate successfully. Unfortunately, this is prohibited by our modeling of $\mathcal{F}_{VRO}$. Concretely, $\mathcal{F}_{VRO}$ requires the simulator to provide the correct verification key vk on the `Init` query by any party.[4] Furthermore, the simulator has to answer immediately, i.e. in the same activation in which it receives the request to do so. For this reason, the simulator can not wait until the adversary lets the corrupted server select and post some SIG verification key to $\mathcal{F}_{bboard}$ before answering.

Due to the fact that verifying a proof is a non-interactive task, we can not simply relax the requirement on the simulator to answer immediately. We can do so, however, if we further change the behavior of $\mathcal{F}_{VRO}$ upon being asked to verify a proof. The solution is to make the behavior of this task depend on whether vk has already been initialized. If not, all proofs (that means for any contained verification key) are rejected as invalid. Having made this change, we can change π_{FHE} such that each party only posts its own verification key to $\mathcal{F}_{bboard}$. If the simulator is then asked to provide Prove and vk, it waits until all servers have successfully posted their keys to $\mathcal{F}_{bboard}$ and constructs vk (and Prove, which depends on vk) from these.

We have to make a small additional change to π_{FHE} for it to realize this altered version of $\mathcal{F}_{VRO}$. The problem we have to solve lies in the fact that the adversary may execute the following distinguishing attack. First, the adversary waits until all honest servers have posted their SIG verification keys to $\mathcal{F}_{bboard}$ and retrieves them. Second, it selects some SIG verification key which it will later post to $\mathcal{F}_{bboard}$, but does not yet do so. The adversary is not able to form the key vk which will later become the public correct verification key. Using this key, it computes a proof π for some input q by contacting the honest servers as prescribed by π_{FHE}. In the real world, this proof will be verifiable, but in the ideal world it will be rejected as vk is still undefined within $\mathcal{F}_{VRO}$. To prevent this attack, we let the honest servers themselves retrieve the portions of vk by the other servers from $\mathcal{F}_{bboard}$ and refuse to communicate with any caller until they have been able to do so. By doing so, the adversary will (with overwhelming probability) not be able to compute a proof that is valid in the real world before having let the corrupted server publicize its portion of vk. And once it does so, the simulator is able to initialize $\mathcal{F}_{VRO}$ properly.

[4]We note that such a query is also triggered if the first query made to $\mathcal{F}_{VRO}$ is a hash or proof verification query as these refer to the correct verification key.

The $\mathcal{F}_{KR}$ functionality

Registration Upon receiving ($\texttt{Register}, sid, \mathsf{pk}, \mathsf{sk}$) from some party $\mathcal{P}$, send ($\texttt{Register}, sid, \mathcal{P}$) to the adversary. Upon receiving ($\texttt{Output}, sid, \mathcal{P}$) from the adversary, check that pk is the public key for the secret key sk. If the check succeeds, store ($\mathcal{P}, \mathsf{pk}$) and output ($\texttt{Registered}, sid, \mathcal{P}, \mathsf{pk}$) to $\mathcal{P}$.

Retrieval Upon receiving ($\texttt{Retrieve}, sid, \mathcal{P}_i$) from some party $\mathcal{P}_j$, if there is a pair ($\mathcal{P}_i, \mathsf{pk}$) stored, return ($\texttt{Key}, sid, \mathcal{P}_i, \mathsf{pk}$). Else return ($\texttt{Key}, sid, \mathcal{P}_i, \perp$).

Figure A.5.: The UC Key-Registration functionality.

Weaker Guarantees with Corruption The protocol in π_{FHE} has a property akin to *guaranteed output delivery* (GOD). In the stand-alone definition of simulation-based security, GOD means that the honest parties are guaranteed to generate some output at the end of a computation which is consistent with the true inputs of all honest parties as well as *some* input by the corrupted parties (there may not be a well-defined *true* input of a corrupted party).

In the UC framework, the real-world adversary is generally given control over message delivery and can thus cause a protocol that involves communication between parties to execute to fail. Thus the ability to prevent the ideal functionality from generating output also has to be given to the simulator (for tasks that are not required to consist only of local computation such as verifying a signature). To prevent any protocol which does not react to inputs from realizing any functionality for which the simulator can prevent delivery of all outputs, [25] introduce *non-trivial* protocols for which the simulator does not corrupt any parties and allows all output delivery to happen if the real-world adversary does not corrupt any parties and delivers all messages.

Remark A.3.1. In previous versions of the UC framework, the simulator was automatically granted the ability to deliver all messages between dummy parties and the ideal functionality while observing some *public header* information but keeping the *private content* of each message hidden.

This would allow our protocol to abort without output once it observes misbehavior by one of the servers or if one of the servers otherwise shows signs of deviation from the protocol, e.g. during the initialization phase to the honest servers by deviating from the MPC protocol (if we assume *identifiable abort*, i.e. that the identity of some party deviating from the protocol is known to all honest parties upon aborting). Our protocol, however, even generates a valid proof if the corrupted server misbehaves (in the sense of replying with garbage values) but all honest responses are delivered. Both of these are clear signs of interference by the adversary and thus would allow aborting the current task or the whole protocol.

It may be possible to find a simpler protocol that does not have these “unnecessary” additional guarantees with respect to generating output for honest parties.

A.3.2. Remarks About OPRF Variants

Further remarks about OPRF variants investigated in Section 6.2.1.

Evaluating a Random Function There seems to be a general necessity to use random oracles or similar assumptions to realize functionalities in the spirit of Figure 6.3 where a random function is evaluated, i.e. that not even the honest server can evaluate the function on its own. We have already seen a similar requirement for protocols realizing $\mathcal{F}_{VRO}$. There, it has to be impossible for the environment to *offline* predict hash values. And also in that case we have seen that predictions by way of random oracle queries are allowed, e.g. when instantiating $\mathcal{F}_{VRO}$ using a single random oracle and with empty proofs, as then the simulator can make the responses congruent with those by $\mathcal{F}_{VRO}$. Our FHE-based protocol circumvented this by guaranteeing that some portion of the key determining the function for the current session is kept hidden from the adversary and can thus be used by the simulator to (undetectably) program outputs as required by the outputs randomly sampled within $\mathcal{F}_{VRO}$. As noted in [67], achieving this in the setting where there is only a single server (which is allowed to be corrupted), seems to require the use of non-black-box assumptions such as assuming the ROM.

The Use of Ticketing Mechanisms A quick note about the tickets used by $\mathcal{F}^{J}_{VOPRF}$. These are introduced to allow for more efficient instantiations by alleviating the simulator from the task of having to extract the true input from corrupted clients. Such a requirement can often only be satisfied by adding expensive zero-knowledge proofs by the client. Tickets circumvent this by letting the simulator act in the following way upon noticing an attempted evaluation by some corrupted party C with server $\mathcal{S}$ in its simulation: Instead of extracting the input and sending an `Eval` message to $\mathcal{F}^{J}_{VOPRF}$, the simulator sends a `SenderComplete` message for $\mathcal{S}$, thereby increasing the tickets counter of $\mathcal{S}$ by one and letting $\mathcal{S}$ acknowledge that an evaluation has occurred by outputting a receipt. Note that this does not mean that the simulator never has to extract the input from a corrupted client, only that this is shifted to a later part of the evaluation procedure. For example, the final output might be produced from a query to a random oracle where the original input is part of the oracle input. The simulator can then at this point extract the input, make an `Eval` query for it on behalf of the corrupted client, and use up the ticket it created earlier to obtain the output and use it to program the random oracle.

We note that, generally, protocols that only realize such a ticketing functionality can be upgraded to be fully input-extractable by adding appropriate proofs of knowledge to the messages sent to the server.

A.4. Relaxing the VRO (Continued)

This section is a continuation of Section 5.5 in the main body.

A.4.1. Leaking Only the Hash

This is the weakest form of additional leakage. As hashes are random values from the codomain, this knowledge essentially only allows the adversary to link queries by honest callers (with overwhelming probability if we assume a super-polynomial codomain). It seems difficult to use this information to simplify π_{FHE}. As we still have to hide q, how might this hidden q be used by the servers in a computation revealing h?

The servers *could* operate on a secret sharing of q, engage in a protocol to recognize whether the currently shared value has been queried already, and, if necessary, jointly generate a fresh hash h (this can be done through multi-party coin-flipping as we do not have to hide the hash). Each server could then sign (q_i, h) where q_i is its share of the input and a valid proof for (q, h) would have to contain shares q_i which reconstruct to q and signatures σ_i for (q_i, h) from different servers.

This naive construction does not yet guarantee unforgeability, shares q_i and signatures σ_i can be taken from different honest proofs and puzzled together in a different way which is still valid for the above verification procedure. To prevent this, the signed messages for a single query have to either be bound to q or to each other. Keeping track of a counter c of answered queries and always including c in the signed messages may already prevent this kind of mix-and-match attack. Keeping such a counter is possible for a simulator.

Another way might be to use a verifiable secret sharing, letting servers agree that they all received the same verification information, and including this verification information in the signed message. As this commits to the input, no mixing of different proofs would be possible.

If Shamir's secret sharing scheme is used to share q then recognizing whether two sharings $\{\hat{q}_i\}_{i=1}^n, \{\bar{q}_i\}_{i=1}^n$ are for the same secret may be possible by using the homomorphic properties and check whether $\{\hat{q}_i - \bar{q}_i\}_{i=1}^n$ is a sharing of 0.

One drawback of this scheme is that it requires storage as well as communication proportional to the number of past queries. Also, the protocol sketched above leaks exactly which past query is equal to the current one and this may not be simulatable for a simulator that only receives the hash, at least if the codomain is polynomial. Depending on the number of expected queries, this may, however, still yield a more efficient protocol than π_{FHE}. Furthermore, the resilience to corruption only depends on the chosen secret sharing and will thus be generally larger than for π_{FHE}.

A.4.2. Leaking Both Input and Hash

This is the strongest form of additional leakage. If we allow the adversary to learn both every query input q as well as the corresponding hash value h, then one way to reduce trust in single servers is to distribute our instantiations from Section 5.2 using a single trusted server by using tools from threshold cryptography. We previously attempted to

do this in Section 5.3.1 but were stumped by requiring to hide the input from corrupted servers. This is no longer the case in the current setting.

Threshold Signing We first describe protocols based on a threshold signature protocol for n parties with threshold t and for a signature scheme SIG. For these, to compute a signature σ on a message q, at least $t + 1$ servers must honestly participate in the protocol. Each server holds a share sk_i of a signing key sk as well as the full verification key vk. These are obtained during some initialization phase, e.g. by a distributed key-generation algorithm in the discrete logarithm setting [52]. If we assume a non-interactive scheme, to sign q, each server executes a partial signing algorithm $\mathsf{PartSign}(\mathsf{sk}_i, q)$ and obtains a partial signature σ_i. We assume a robust combination algorithm Combine which receives as input at least $t+1$ partial signatures $\sigma_1, \sigma_2, \ldots, \sigma_l$, $l \geq t+1$, and outputs a valid signature σ for q under vk.

As we wish to output a signature for (q, h), the servers first have to compute the hash h. The simplest way to achieve this is by using a general MPC protocol for the function $q \mapsto \mathsf{PRF}(k, q)$. Again we can not allow the adversary to possess knowledge of all of k and thus have to secret share it.

The full protocol ξ is thus as follows. The caller C, receiving input $(\texttt{Hash}, sid, q)$ simply forwards q to all n servers. On receiving q, each of the servers behaves as follows:

- Let $\hat{k}$ be the share of the PRF key k and $\hat{\mathsf{sk}}$ the share of the SIG signing key sk.
- Compute $h = \mathsf{PRF}(k, q)$ in a distributed manner.
- Compute $\hat{\sigma} \leftarrow \mathsf{PartSign}(\hat{\mathsf{sk}}, q)$.
- Return $(h, \hat{\sigma})$ to C.

Upon receiving $((h_1, \sigma_1), \ldots, (h_n, \sigma_n))$ from the servers and if at least $t + 1$ of these are not equal to $\perp$, C computes h as majority value over the h_i and uses Combine to compute either a signature σ for (q, h) which is valid under vk or $\perp$. Only in the former case does C output $(\texttt{HashProof}, sid, q, h, \sigma)$.

The security of this protocol is clear if we assume that both the distributed evaluation of PRF as well as the threshold signing protocol are UC-secure. In this case, the simulator can replace PRF by a random function RF and thus appropriately bias hashes. The security of the threshold signing protocol ensures that the adversary will be unable to forge signatures even after observing the views of the corrupted servers as well as the views of corrupted clients in polynomially many interactions.

One advantage of this construction over the PRF construction is that it has better scalability, the number of PRF instances remains one, independent of the total number of servers. The size of proofs also remains at a single signature σ (although signature aggregation can be used to reduce the number of signatures included in each proof for the PRF construction as well).

If we do not want to employ threshold signatures we can also allow each server to sign (q, h) directly using a separate signing key and a valid proof has to contain more valid signatures than there may be corrupted servers. This theoretically supports arbitrarily

many corrupted servers, but at the cost of having to allow "trivial" denial-of-service attacks/aborts by having some corrupted server not respond. Aggregation may help in keeping proof sizes relatively small, although aggregation would have to be possible for different public keys.[5]

Instead of doing general MPC, there may exist more simple protocols for distributed evaluation of a random function. Such a protocol essentially would realize a variant of the ideal random oracle functionality $\mathcal{F}_{RO}$ where the delivery of outputs can be delayed by the adversary and the input q and hash h are leaked to the adversary upon each hash query. Looking through the literature on distributed PRF protocols, very few of them are fully UC-simulatable.

Distributed VRF Similarly, the servers may evaluate a (simulatable) VRF in a distributed manner. Again, this protocol has to be UC-secure to allow the simulator to determine hashes as prescribed by $\mathcal{F}_{VRO}$. We do not go into details.

Stateful Version There is a much simpler protocol if we allow storage proportional to the number of past queries as well as interaction. The basic task on input q is to

- Recognize whether q has been queried before.
- If yes, look up what the hash h for q is and give the same answer as before.
- Else, use a multi-party coin-tossing functionality $\mathcal{F}_{MPCT}$ to generate an independent h and store (q, h).
- Sign (q, h) and return it.

A valid proof for (q, h) again consists either of a threshold signature for (q, h) or of a majority of valid signatures for (q, h) under independent keys associated with each server. Programmability is given by programming $\mathcal{F}_{MPCT}$.

One way to implement $\mathcal{F}_{MPCT}$ is by having each party broadcast a UC-commitment [21] to a random value and then have all parties de-commit (essentially [73]) although care has to be taken with respect to parties which don't de-commit. As such behavior clearly identifies the offending server as corrupted, repeating the protocol may be a solution. If we assume all parties de-commit for a given execution, the simulator can bias the result to a specific value r by extracting the values committed to by the corrupted parties and later opening the commitment by one of the honest parties such that the sum of all values is r using the equivocality of the UC-commitment scheme.

This is essentially using the *lazy sampling* interpretation of random oracles to build a random function instead of requiring a physical key that already determines all hashes.

[5] This is for example possible for BLS signatures, but they require the ROM. We are unaware of any such schemes in the standard model.

A.4.3. Leaking Only the Input

This form of leakage is considerably stronger than only leaking the hash, but only slightly weaker than leaking both input and hash as in the last section. Knowing q, the servers could still execute the first part of the proposed protocol ξ in the next section. They, however, would not be allowed to publicly reconstruct the PRF output and create signatures for (q, h) directly. Signing shares (q, h_i) and including a counter c may again facilitate unforgeability.

A.5. An Attack on the Randomized Fischlin Transform

In this section we describe an attack on the zero-knowledge property of the randomized Fischlin transform as defined in [71]. Note that this means we are for the moment not using our definition of zero-knowledge but the notion defined in [45] which only allows a single proof to be requested by the simulator. The attack crucially depends on restricting the length l of challenges to be in $O(\log \lambda)$.

To state the attack, we first make the following observation regarding Σ-protocols. Let $\mathcal{A}$ be in possession of some $(x, w) \in \mathcal{R}$. Let $(\mathsf{com}, \mathsf{ch}, \mathsf{resp})$ be a valid transcript for x, but which was generated by some party $\mathcal{B} \neq \mathcal{A}$. Then there exists a Σ-protocol Σ which satisfies the requirements of the randomized Fischlin transform and for which $\mathcal{A}$ can compute valid transcripts $(\mathsf{com}, \mathsf{ch}_i, \mathsf{resp}_i)$ for all challenges ch_i from the challenge space of Σ. Note that this insofar intuitively surprising as the name *commitment* for the first message sent by the prover suggests that the contained value should remain hidden from any other party not in possession of the randomness used to generate it.[6]

We show that this is generally not the case when the party receiving a valid transcript containing the independently generated commitment has access to a witness. Recall the structure of Schnorr's Σ-protocol for proving knowledge of a discrete logarithm. The prover has access to $(g, X = g^x, x)$ with g the generator of some group $\mathbb{G}$ of prime order q and $x \in \mathbb{Z}_q$ while the verifier only knows g and X. The first message by the prover consists of a random element of the group $\mathbb{G}$, generated as $z \leftarrow\$ \mathbb{Z}_q$, $\mathsf{com} = g^r$. The verifier responds with a random challenge $\mathsf{ch} \leftarrow\$ \mathbb{Z}_q$. From this challenge ch, the randomness z, as well as the witness x, the prover computes a response $\mathsf{res} = z + x\mathsf{ch}$. Finally, the verifier checks that

$$g^{\mathsf{resp}} = \mathsf{com}\, X^{\mathsf{ch}} \tag{A.3}$$

and rejects or accepts accordingly.

Passing to the compiled protocol, consider a single accepting transcript

$$(\mathsf{com}, \mathsf{ch}, \mathsf{resp}) = (g^z, \mathsf{ch}, \mathsf{resp})$$

for the statement $X = g^x$, i.e. such that Equation (A.3) holds. As we have stated above, we assume that the adversary $\mathcal{A}$ has access to x, but not to z. Nonetheless, if we consider a challenge $\mathsf{ch}^* \in \mathbb{Z}_q$ with $\mathsf{ch}^* \neq \mathsf{ch}$, an accepting transcript

$$(\mathsf{com}, \mathsf{ch}^*, \mathsf{resp}^*)$$

can be computed as follows. First, define

$$\Delta = \mathsf{ch}^* - \mathsf{ch} \bmod q$$

and notice that

[6] In other words, the commitment should be *hiding*.

$$\begin{aligned}\mathsf{resp}^* &\equiv_q z + \mathsf{ch}^* x \\ &\equiv_q z + (\mathsf{ch} + \Delta)x \\ &\equiv_q z + \mathsf{ch}\, x + \Delta x \\ &\equiv_q \mathsf{resp} + \Delta x.\end{aligned}$$

Looking at the first and last term in this chain of equalities and observing that $\mathcal{A}$ has access to all of the information to evaluate the last term, we see that resp* can be computed by $\mathcal{A}$. That this leads to a valid transcript can be seen by the chain

$$\begin{aligned}g^{\mathsf{resp}^*} &\equiv_q g^{\mathsf{resp}+\Delta x} \\ &\equiv_q g^{\mathsf{resp}} + g^{\Delta x} \\ &\equiv_q \mathsf{com}\, X^{\mathsf{ch}} + g^{\Delta x} \\ &\equiv_q \mathsf{com}\, X^{\mathsf{ch}} + X^{\Delta} \\ &\equiv_q \mathsf{com}\, X^{\mathsf{ch}^*}\end{aligned}$$

as the last term shows by comparing it to Equation (A.3).

We notice that this is immediately applicable to the wider class of Σ-protocols where knowledge of a pre-image under a group-homomorphism

$$\Phi : \mathbb{G}' \to \mathbb{G}$$

such that

$$\forall g_1, g_2 \in \mathbb{G}' : \Phi(g_1 *' g_2) = \Phi(g_1) * \Phi(g_2).$$

The Attack Let A be the algorithm which, on input a transcript $t = (\mathsf{com}, \mathsf{ch}, \mathsf{resp})$ for statement (g, g^x), the witness x, and another challenge ch^*, outputs the transcript $t^* = (\mathsf{com}, \mathsf{ch}^*, \mathsf{resp}^*)$ where resp^* was computed as above. Let further $\Pi = (\mathcal{P}, \mathcal{V})$ be the Schnorr protocol after being transformed using the randomized Fischlin transform as originally described in [71] and let $\mathcal{S}$ be the zero-knowledge simulator also described therein. We describe an adversary $\mathcal{B}$ on the $\mathcal{S}$, i.e. a distinguisher between interacting with an oracle for $\mathcal{P}$ and an honest random oracle H or an oracle for $\mathcal{S}$ and a simulated random oracle $H_{\mathcal{S}}$.

Before describing $\mathcal{B}$, let X be a random variable distributed according to the binomial distribution $\mathrm{Bin}(q, 2^{-b})$, i.e. with $n = q$ samples and $p = 2^{-b}$ being the associated probability, and $Y = 1 + Y'$ with Y' distributed according to $\mathrm{Bin}(q - 1, 2^{-b})$. We need that for some $k \in [q]$, the probabilities $\mathbf{Pr}[X = k]$ and $\mathbf{Pr}[Y = k]$ can be efficiently computed as long as $q \in \mathcal{O}(\mathsf{poly}(\lambda))$, which is what we assume.

$\mathcal{B}$ behaves as follows:

- Generate a random statement by sampling $x \leftarrow\!\$\ \mathbb{Z}_q$ and computing $X = g^x$.
- Give (X, x) to the proof generation oracle and receive a proof $\pi = (\mathsf{com}_i, \mathsf{ch}_i, \mathsf{resp}_i)_{i=1,2,\dots,r}$.

- If $\pi = \perp$, output 0.
- Initialize a variable count $= 0$ and set $\vec{\mathsf{com}} = (\mathsf{com}_1, \mathsf{com}_2, \ldots, \mathsf{com}_r)$.
- For $\mathsf{ch}^* \in \mathbb{Z}_q$
  - Let $t = (\mathsf{com}_1, \mathsf{ch}_1, \mathsf{resp}_1)$.
  - Let $(\mathsf{com}_1, \mathsf{ch}^*, \mathsf{resp}^*) = \mathsf{A}(t, x, \mathsf{ch}^*)$.
  - Query $(X, \vec{\mathsf{com}}, 1, \mathsf{ch}^*, \mathsf{resp}^*)$ to the random oracle and let h be the result.
  - If $h = 0^b$, increment count.
- Compute $p_1 = \mathbf{Pr}[X = \text{count}]$ and $p_2 = \mathbf{Pr}[Y = \text{count}]$.
- If $p_1 > p_2$, output 0.
- Else, if $p_1 < p_2$, output 1.
- Else, if $p_1 = p_2$, output 0 or 1, each with probability $1/2$.

We analyze the distinguishing advantage of $\mathcal{B}$. First, we observe that if interacting with the honest H, count has the same distribution as X while in the simulated case it has the distribution of Y. By the test we are employing, the distinguishing advantage of $\mathcal{B}$ is equal to the statistical distance $\Delta(X, Y)$. All that is left to do is analyze $\Delta(X, Y)$ and show that it is non-negligible.

We begin by observing that for $0 < k \leq q$

$$\mathbf{Pr}[Y = k] = \frac{k}{q}\,\mathbf{Pr}[X = k] + \frac{q - (k-1)}{q}\,\mathbf{Pr}[X = k-1]$$

while of course

$$\mathbf{Pr}[Y = 0] = 0.$$

This can be seen as follows. Y can be seen to be the number of heads that are observed in a sequence of q coin tosses with a coin with $p = 2^{-b}$ and where after sampling the q coins a random coin is selected and set to head, independent of whether it was already heads before. As such, the probability for observing $k \geq 1$ heads in the final sequence is composed of two cases. In the first, k heads were already contained in the initial sequence and the randomly selected coin which was switched to head was among these k coins. This occurs with probability $\frac{k}{q}\,\mathbf{Pr}[X = k]$. In the second, $k - 1$ heads were in the initial sequence and one of the remaining $q - (k - 1)$ coins was flipped to head which gives the second term.

Using this alternate formulation, the statistical distance $\Delta(X, Y)$ can be written as

$$\begin{aligned}\Delta(X,Y) &= \frac{1}{2}\sum_{i=0}^{q}|\mathbf{Pr}[X=i]-\mathbf{Pr}[Y=i]| \\ &= \frac{1}{2}\left(\mathbf{Pr}[X=0]+\sum_{i=1}^{q}|\mathbf{Pr}[X=i]-\mathbf{Pr}[Y=i]|\right) \\ &= \frac{1}{2}\left(\mathbf{Pr}[X=0]+\sum_{i=1}^{q}\left|\mathbf{Pr}[X=i]-\frac{i}{q}\mathbf{Pr}[X=i]-\frac{q-(i-1)}{q}\mathbf{Pr}[X=i-1]\right|\right) \\ &= \frac{1}{2}\left(\mathbf{Pr}[X=0]+\sum_{i=1}^{q}\left|\frac{q-i}{q}\mathbf{Pr}[X=i]-\frac{q-(i-1)}{q}\mathbf{Pr}[X=i-1]\right|\right).\end{aligned}$$

Now, for any $1 \leq m \leq q$ (which we will determine later), we have that

$$\begin{aligned}\Delta(X,Y) &\geq \frac{1}{2}\left(\mathbf{Pr}[X=0]+\sum_{i=1}^{m}\left|\frac{q-i}{q}\mathbf{Pr}[X=i]-\frac{q-(i-1)}{q}\mathbf{Pr}[X=i-1]\right|\right) \\ &\geq \frac{1}{2}\left(\mathbf{Pr}[X=0]+\sum_{i=1}^{m}\frac{q-i}{q}\mathbf{Pr}[X=i]-\frac{q-(i-1)}{q}\mathbf{Pr}[X=i-1]\right) \\ &\geq \frac{1}{2}\left(\mathbf{Pr}[X=0]+\frac{q-m}{q}\mathbf{Pr}[X=m]-\mathbf{Pr}[X=0]\right) \\ &= \frac{1}{2}\frac{q-m}{q}\mathbf{Pr}[X=m] \end{aligned} \tag{A.4}$$

by $|a - b| \geq a - b$ for any $a, b \in \mathbb{R}$ and by telescoping the resulting sum.

Next, as we only have to show that the distinguisher works for some infinite sequence $(q_i)_{i\in\mathbb{N}}$, we may restrict ourselves to only working with such parameters that

$$\mathbb{E}[X] = q2^{-b} \in \mathbb{N}$$

for infinitely many values of λ. Such parameters exist. We may for example set[7]

$$\begin{aligned} b(\lambda) &= \lfloor \log \lambda \rfloor \\ q(\lambda) &= 2^b q'(\lambda) \end{aligned}$$

for some function $q' : \mathbb{N} \to \mathbb{N}$ such that

$$q \in \mathcal{O}(\mathsf{poly}(\lambda))$$

and simultaneously

$$2^{\log(q)-b} \in \omega(\log \lambda)$$

[7]The number r of repetitions can be set to any admissible value.

as is required for the transform to be applicable. This is clearly possible. We may then even assume for *all* $\lambda \in \mathbb{N}$ that $q2^{-b} \in \mathbb{N}$. All that is left to do to derive a lower bound for $\Delta(X, Y)$ is to observe the following.

For the expected value $\mathbb{E}[X] = \mu$, it holds that

$$\mathbf{Pr}[X = \mu] \geq \frac{1}{q+1} \tag{A.5}$$

by the fact that the support $\mathrm{Supp}(X)$ has cardinality $q + 1$ and μ is an integer and in our case always equal to the mode, i.e. most likely expression, of the distribution. As q is polynomial this is noticeable and in particular non-negligible. Then, by choosing $m = \mu$ in (A.4) (which we are allowed to do as q grows much faster than 2^{-b} and thus $\mu \geq 1$ eventually) we get

$$\begin{aligned}
\Delta(X, Y) &\geq \frac{q-\mu}{2q}\mathbf{Pr}[X = \mu] \\
&\geq \frac{q-\mu}{2q}\frac{1}{q+1} \\
&\geq \frac{q(1-2^{-b})}{2q(q+1)} \\
&= \frac{1-2^{-b}}{2(q+1)} \\
&\geq \frac{1}{4(q+1)}
\end{aligned}$$

which is non-negligible (even noticeable) as q is polynomial. □

To summarize, by restricting the challenge space to be polynomial, the transformation described in [71] succumbs to an exhaustive search that exploits the non-negligible statistical distance introduced by having the simulator program using a fixed value instead of a pseudo-random one.

A.6. Reducing to Semantic Security

This section contains the technical details for why the protocol π_{FHE} from Section 5.4 makes use of the NIWI proof system NIWI.

Motivation Consider a variation of π_{FHE} where callers, instead of computing four NIWI proofs π_i, include the (valid) signatures used to compute them. This protocol suffers from a problem related to the security guarantees afforded by the semantic security of FHE. As we have seen, a common use case for FHE is the following:

1. A client C generates a FHE key-pair $(\mathsf{pk}, \mathsf{sk})$
2. uses pk to encrypt some data d, obtaining a ciphertext c
3. sends c to some server $\mathcal{S}$
4. and receives a response c' which it expects to be equal to $\mathsf{FHE.Eval}(\mathsf{pk}, C, c, c^*)$ where C is a circuit known to the client and c^* is some encrypted, secret data belonging to $\mathcal{S}$.

The insecurity of this plain protocol for outsourcing some computation on a known circuit C, but involving private data by the party doing the actual computation, stems from the fact that $\mathcal{S}$ does not have to respond with a correctly evaluated ciphertext c'. It may respond with some arbitrary ciphertext c'' instead.[8] $\mathcal{S}$ may have received c'' from some arbitrary source. As such, $\mathcal{S}$ may not be aware of the plaintext contained in c''. By observing the subsequent behavior of the client after receiving c'', the server may then acquire information about this plaintext. In the worst case, the client leaks the complete plaintext, i.e. the server may use the client as a decryption oracle. As we know, such a decryption oracle is not afforded to an adversary on the semantic security of an encryption scheme. The situation is even worse in the FHE case as all FHE schemes are inherently malleable and thus can not be IND-CCA2-secure; which makes adding this as an additional assumption not possible.

In our case, however, things seem to be slightly different at first sight. The response by the server is not decrypted and then indiscriminately output to the environment. Instead, it is checked for correctness by comparing it to the responses by the other servers which guarantees correct evaluation of the PRF as well as the correctness of the signature with respect to the server's verification key. Hence, the only information that an adversary should be able to gain by replying with some dishonestly generated ciphertext c'' is whether it for example decrypts to $\mathsf{PRF}(k_2, q)$ or contains a valid signature for the message $(q, \mathsf{PRF}(k_2, q))$ under its own signing key. Note that all of these plaintexts are already computable by the corrupted server itself using for example the known k_2 and its own signing key.

Unfortunately, this information is already enough to render the protocol insecure (if only in the strong sense of UC). To see that the simulator may not at all rely on the information obtained by decrypting the response of the corrupted server—although it

[8]With the restriction of having the same length.

knows the decryption key of every honest party—we observe what happens if we try to reduce to the semantic security of the FHE scheme. In the reduction for such a game-hop we obtain a public key pk as well as an oracle which either returns an encryption of a message m or an encryption of $0^{\|m\|}$. This key is then used as the key of some honest caller C. It is then unclear how the reduction is supposed to produce the proof in the interaction where pk is used. Either the corrupted server behaved honestly, in which case a proof could contain signatures from all four servers, or the corrupted server sent some bogus response, in which case the proof has to contain only signatures by the three honest servers. Without access to the decryption key or a decryption oracle, the reduction has no way to produce correctly distributed proofs. Hence the reduction does not succeed.

The general problem is that the simulator can not detect whether the simulated corrupted server behaved honestly (and include or exclude signatures by the corrupted server from the produced proof based on this information) if it is not allowed to decrypt the response (and in the general case not even then as there may be private input by the server), and it is not allowed to decrypt the response in order to be able to argue that the server does not learn the input q. The environment, on the other hand, knows whether it instructed the adversary to execute either a correct or an incorrect evaluation and can compare this to the signatures contained in the produced proof.

Two solutions for this problem come to mind:

1. Make the servers prove that they evaluated correctly.
2. Make proofs independent of the response by the corrupted server.

We investigate both approaches in more detail.

Proving Correct Evaluation If servers were required to prove correct evaluation and honest callers were to discard responses accompanied by incorrect proofs, then the proof strategy would be as follows: In the game-hop before we want to utilize semantic security, the proof generation by an honest caller is made independent of the ciphertext returned by the corrupted server. If the proof of correct evaluation verifies, the witness is extracted from the message to the zero-knowledge functionality itself. This witness contains the randomness the server used when performing the homomorphic signature generation (on message $0^{\|q\|}$). The caller then uses this randomness r_i as well as the signing key sk to recreate the signature

$$\sigma_i = \mathsf{SIG.Sign}(\mathsf{sk}, (q, h_i, i); r_i)$$

which the corrupted server *would have* created if it had received an encryption of q.

This approach would be highly impractical, however. Fully-homomorphic computations themselves are already computationally expensive and the involved statements to be proven are represented by large circuits. It also requires the three honest servers to do this much work *just* to catch misbehavior by the corrupted server *even though we could cope with wrong answers by a single server* by way of cross-checking the answers between the servers.

If we were to use this approach we could also use a much simpler basic protocol. Each server $\mathcal{P}_i$ would publicly commit to a PRF key k_i and on input a ciphertext c and public key

pk would execute the current server protocol (with a single PRF instance) augmented with a proof of correctness of the resulting ciphertexts (c_1^*, c_2^*) with respect to the commitment as well as the evaluation randomness used. The caller would only output a proof once it received valid proofs from all servers. Note that this requires all servers to do their evaluations correctly, but in principle allows for all but one server to be statically and maliciously corrupted. Programmability is possible if the commitments are extractable by the simulator, e.g. by using UC-commitments, such that one of the uncorrupted PRF instances can be programmed appropriately.

Independent Proofs If proofs were independent of the response by the corrupted server, then, before the game-hop utilizing the semantic security of FHE, we may move to a hybrid where the proofs returned by honest callers are generated solely based on the responses by honest servers. We may then subsequently move to a hybrid where the corrupted server receives an encryption of zeroes because we no longer require the decryption key to simulate the final proof and thus *can* successfully reduce to the semantic security of FHE.

But achieving this kind of independence seems to come at the price of some primitive such as *non-interactive witness-indistinguishable arguments* (NIWI) which, while not as strong as NIZK, nonetheless is more expensive than including a number of signatures. The specific NIWI relations which would be required are those used in π_{FHE} where knowledge of at least two valid signatures for each PRF instance is proved.

Sole witness-indistinguishability is, however, not enough. We also require (an intermediate version of) the simulator to be able to *extract* witnesses from proofs that come from the adversary/environment. This means we can not rely on rewinding but have to be able to extract in a *straight-line* manner. To see why this is (was) necessary, imagine the following. At some point during the proof, we want to reduce unforgeability of $\mathcal{F}_{VRO}$ (i.e. that the *unforgeability* clause is only hit with negligible probability) to the security of the signature scheme. In essence, this means that we have to be able to build an adversary on the EUF-CMA-security of the signature scheme which produces a forged signature from a forged $\mathcal{F}_{VRO}$ proof with some non-negligible probability.

As long as a proof consists of signatures themselves, there is no obstacle to extracting forged signatures. But by replacing signatures with NIWI proofs, this easy extractability of forged signatures from forged proofs is lost. Hence we have (had) to reintroduce it by using some form of extractable NIWI proof. Now, because when using a NIWI in the plain model there is no advantage for the simulator, i.e. by knowing the extraction trapdoor to some CRS, straight-line extraction can not exist (otherwise we might use the scheme in [48]). We thus have to proceed differently than just having parties use a NIWI proof system formulated in the plain model.

Instead, a CRS would have to be either

- globally available, i.e. we work in a model where *all* parties have access to some functionality $\mathcal{F}_{CRS}$ which distributes the CRS.
- constructed by the servers utilizing some form of MPC and the fact that we have a 3/4 super-majority and then (1) given to each caller making a query and (2) also hard-coded into the verification algorithm Verify and distributed with it.

From a practical point of view, the latter seems to be more appealing which is why ended up choosing this option. It means that verifiers only have to trust the verification key whose integrity is already guaranteed by any protocol realizing $\mathcal{F}_{VRO}$.[9] Especially the fact that we have a two-thirds honest majority allows for very efficient computation involving only the servers, see [49], while any instantiation of a global CRS functionality involving all parties can not rely on this fact (and in general is not possible without any other setup if there is no honest majority).

[9] If it was retrieved using the intended mechanism.

A.7. Simplifying the FHE Construction for Semi-Honest Adversaries

The protocol in Section 5.4 can be simplified in several ways if one is willing to assume that the corrupted server behaves in a semi-honest manner. We briefly describe each potential change.

Eliminating Public Storage We are able to eliminate the use of public storage as provided by $\mathcal{F}_{bboard}$. Using it was required to ensure consistency of verification keys by honest parties. Letting partial verification keys be obtained directly from the servers, a malicious server may answer with different keys.

Eliminating NIWI In Section 5.4.6 we gave the rationale for our use of NIWI proofs. This rationale is directly related to the fact that the corrupted server may deviate from its prescribed protocol and respond with an arbitrary ciphertext. As an alternative solution, we showed that we could have required servers to provide proofs of correct evaluation. In the semi-honest setting, we get these proofs essentially for free and can therefore go back to including signatures in the clear.

To still enable Prove to simulate proofs, it then has to produce valid signatures under the verification key of the corrupted servers. Including the signatures obtained from the corrupted server in the simulation in the `SimInfo` message is insufficient as it is a signature on a string of zeroes instead of the correct input. Luckily, even in its current version the simulator for π_{FHE} has access to the corrupted server's signing key and may include it in the description of Prove.[10]

A slight quirk is the following. Even if the server can be assured to do the homomorphic evaluation correctly, it may still (although now honestly) sample randomness for use in the evaluation of the signature scheme, if this is probabilistic, which would remain hidden from callers. As in general the used randomness is not extractable from a signature, we have to use a deterministic scheme. This is a minor restriction.

Optimizing Initialization As we have seen in the last paragraph, the simulator now requires access to the signing key of the corrupted server. In consequence, we can not switch to a simplified initialization protocol where each server chooses its own key. A slight optimization of this portion of the protocol is possible by switching from a maliciously secure MPC protocol to one that is secure only against semi-honest adversaries.

This concludes our optimizations for semi-honest servers.

[10]This key is currently unused. We could, however, not allow the corrupted server to generate its own key-pair due to the simulator requiring immediate access to the full verification key vk to initialize $\mathcal{F}_{VRO}$.

A.8. Using VROs in OPRF Protocols

In this section, we argue that, in general, the use of $\mathcal{F}_{RO}$ within protocols for $\mathcal{F}_{VOPRF}$ can be replaced by $\mathcal{F}_{VRO}$. The reasoning goes as follows:

- If we assume that retrieving the $\mathcal{F}_{VOPRF}$ verification key is delayable, we can augment it with the key obtained from $\mathcal{F}_{VRO}$.
- As OPRF evaluations always involve interaction between the client and the server, we may replace queries to $\mathcal{F}_{RO}$ during it with (delayable) hash queries to $\mathcal{F}_{VRO}$.
- The adversary already learns the identity of honest clients doing OPRF evaluations.
- Depending on the domain of the evaluated PRF, leaking the length of hash queries to the adversary can be simulated.
- Queries to $\mathcal{F}_{RO}$ during the verification process can generally be replaced with verification queries to $\mathcal{F}_{VRO}$ and letting clients include the necessary $\mathcal{F}_{VRO}$-proofs within $\mathcal{F}_{VOPRF}$-proofs.
- Soundness of the last step follows by the perfect unforgeability of $\mathcal{F}_{VRO}$ and if we assume that $\mathcal{F}_{VOPRF}$-proofs possess the same malleability properties as $\mathcal{F}_{VRO}$.

We briefly describe the transformation by looking at a protocol ξ for publicly and verifiably evaluating the aforementioned function 2HashDH. ξ is essentially the protocol 2HashDH-NIZK from [66], except that we have made it fully publicly verifiable (instead of only UC-realizing $\mathcal{F}^{J}_{VOPRF}$) according to our notion thereof and also upgraded it to possess full input-extractability. The function 2HashDH to be evaluated is given as

$$\begin{aligned} \mathsf{2HashDH} : \mathcal{X} &\to \mathcal{Y} \\ x &\mapsto H(x, H'(x)^k) \end{aligned}$$

where H and H' are hash functions

$$\begin{aligned} H &: \mathcal{X} \times \mathbb{G} \to \mathcal{Y} \\ H' &: \mathcal{X} \to \mathbb{G} \end{aligned}$$

and where $\mathbb{G}$ is a group of prime order q having λ bits and which are modeled as random oracles via calls to $\mathcal{F}_{RO}$.[11] The security of the usual blinding-based protocol for evaluating 2HashDH relies on the *One-More Diffie-Hellman assumption* (OMDH).[12] In it, the client, on input q, samples $r \leftarrow \mathbb{Z}_q$, computes $x = H'(q)^r$ and sends x to the server. The server, using

[11] A single session suffices by a domain separation argument.

[12] Briefly, this requires that no adversary having access to a challenge oracle outputting random elements $g \leftarrow\$ \mathbb{G}$ as well as a DLog oracle $x = \mathcal{O}(g^x)$ is able to answer all q of its challenges by making $< q$ queries to the DLog oracle.

his key $k \in \mathbb{Z}_q$, computes $y = x^k$ and returns it to the client. Finally, the client unblinds y as $h' = y^{\frac{1}{r}}$ and computes the final output $h = H(x, h')$. It is easy to see that

$$\begin{aligned}
h &= H(q, h') \\
&= H(q, y^{\frac{1}{r}}) \\
&= H(q, x^{k\frac{1}{r}}) \\
&= H(q, H'(q)^{kr\frac{1}{r}}) \\
&= H(q, H'(q)^k) \\
&= \mathsf{2HashDH}(k, q)
\end{aligned}$$

We will not prove the security of this protocol, but observe that every response by the server determines *some* effective key k^* and so no additional measures to protect from a misbehaving server have to be taken.[13] We first make this protocol publicly verifiable. For this, we let the server publish a verification key $\mathsf{vk} = g^k$ at the start of the protocol to a bulletin-board $\mathcal{F}_{bboard}$. The first message by the client additionally contains a proof of knowledge for r. The server checks this proof and, if it is correct, augments its response with a NIZKPoK $\tilde{\pi}$, generated by the functionality $\mathcal{F}_{NIZK}$, attesting to the equality of discrete logarithms[14]

$$\log_g(\mathsf{vk}) = \log_g(y) = \log_g(x^k) = k.$$

The client's output is augmented to in addition include

$$\pi = (\tilde{\pi}, r, y).$$

To verify some input (q, h, π) with $\pi = (\tilde{\pi}, r, y)$, a verifier computes and checks the following

$$\begin{aligned}
x &= H'(q)^r \\
\mathsf{Verify}(\tilde{\pi}, g, \mathsf{vk}, x, y) &\stackrel{?}{=} 1 \\
H(q, y^{\frac{1}{r}}) &\stackrel{?}{=} h
\end{aligned}$$

where we denote by Verify the proof verification interface by $\mathcal{F}_{NIZK}$. By the properties of $\tilde{\pi}$, if the verification succeeds, it holds that

$$y = x^k = H'(q)^{rk}$$

and hence that

$$y^{\frac{1}{r}} = H'(q)^k.$$

[13]Recall that the server may use different keys in different evaluations, but consistency has to hold for the same key.

[14]We note that there exists a Σ-protocol for this relation to which then the Fischlin transform from Chapter 4 can be applied to obtain non-interactive proofs.

Thus, the last check will only succeed if $h = 2\mathsf{HashDH}(k, q)$. This sketches public verifiability under the OMDH assumption. For a detailed proof see [66].

To argue input extractability, we observe that the simulator can extract r from a corrupted client's first message. Given the fact that H' has super-polynomial codomain, the corrupted client will not be able to compute a valid proof unless it knows a pre-image of $x^{\frac{1}{r}}$ under H', i.e. has indeed obtained $x^{\frac{1}{r}}$ by querying q. This q can thus be extracted by the simulator and input into $\mathcal{F}_{VOPRF}$ which thus does not have to use the ticketing mechanism from $\mathcal{F}^{J}_{VOPRF}$.

Moving to the VROM We argue that this protocol remains secure when $\mathcal{F}_{RO}$ is replaced by $\mathcal{F}_{VRO}$, $\mathcal{F}_{NIZK}$ is replaced by $\mathcal{F}_{TZK}$ and proofs π are augmented with the proof π^* which was returned to the client upon computing $H(q, H'(q)^k)$. The proof π^* is then instead checked by verifiers in the last check above. Lastly, the verification key vk, which so far consisted of g^k, is augmented by the verification key vk_{TZK} output by $\mathcal{F}_{TZK}$ and vk_{VRO} output by $\mathcal{F}_{VRO}$.[15]

We again argue informally that this is sound. First, the fact that both answers to hash queries made to $\mathcal{F}_{VRO}$ and proof generation requests to $\mathcal{F}_{TZK}$ are delayable is allowed by the fact that $\mathcal{F}_{VOPRF}$ evaluation queries are delayable. The same is true for the retrieval of vk_{TZK} and vk_{VRO}. Similarly, verification of proofs remains non-interactive by the non-interactiveness of these tasks in both $\mathcal{F}_{TZK}$ and $\mathcal{F}_{VRO}$. Furthermore, if we allow the same malleability for proofs by $\mathcal{F}_{VOPRF}$ as for $\mathcal{F}_{VRO}$, then the inclusion π^* in π is sound. Last, the perfect unforgeability of $\mathcal{F}_{VRO}$ shows that $\mathcal{F}_{VOPRF}$ remains as unforgeable as before. This concludes the sketch.

[15] A single instance of $\mathcal{F}_{VRO}$ is again sufficient by employing domain separation.