Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Post-Quantum Building Blocks for Secure Computation – the Legendre OPRF

Master Thesis

Lucas Dodgson

September 13, 2023

Advisors: Prof. Dr. Kenny Paterson, Dr. Julia Hesse (IBM Research Zürich),
Sebastian Faller (IBM Research Zürich)

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

An Oblivious Pseudo-Random Function (OPRF) is a two-party protocol for jointly evaluating a Pseudo-Random Function (PRF), where a user has an input x and a server has an input k. At the end of the protocol, the user learns the evaluation of the PRF using key k at the value x, while the server learns nothing about the user's input or output. Despite OPRFs having numerous applications as building blocks of larger protocols, there are only a handful that are based on mathematical assumptions that cannot be easily broken by a quantum computer. These so-called *post-quantum* OPRF candidates use either a less strong security model, are practically inefficient, or are based on relatively new and potentially insecure assumptions.

We design and analyse the security and efficiency of one such post-quantum OPRF based on the Legendre symbols. Our construction relies on a hardness assumption that currently withstands quantum attacks. We model and prove the security of the OPRF in the Universal Composability framework, therefore showing the security of protocols such as OPAQUE when instantiated with the Legendre OPRF. We also find the Legendre OPRF to not be secure according to an alternative security notion and show how the achieved security notion does not suffice for certain applications.

Additionally, we consider the generalisation to higher-power residues, which brings a noticeable performance benefit in particular settings. The benchmarks for both OPRFs highlight how, when we allow for a preprocessing phase, the Legendre OPRF has significantly smaller bandwidth requirements than other post-quantum alternatives.

Contents

Contents **iii**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Contributions . . . 2
1.3 Organisation . . . 2

2 Background **5**
2.1 Oblivious Pseudo-Random Functions . . . 5
2.2 Legendre Symbols . . . 7
2.2.1 The Legendre PRF . . . 7
2.2.2 Cryptanalysis of the Legendre PRF . . . 10
2.2.3 Higher-power residues . . . 11
2.3 Universally Composable Security . . . 12
2.4 Related Work . . . 15

3 The Legendre OPRF **19**
3.1 The Legendre OPRF . . . 19
3.1.1 Ideal OPRF functionality . . . 19
3.1.2 Legendre OPRF . . . 21
3.2 Insecurity of the Legendre OPRF . . . 26
3.3 Correlated OPRF . . . 29
3.3.1 Ideal functionality . . . 29
3.3.2 Security analysis of the Legendre OPRF . . . 30
3.3.3 Strengthening of the notion . . . 47
3.4 Correlated OPRF with Prefixes . . . 47
3.5 Higher-Power Residue OPRF . . . 56

4 Alternative Uses of Correlated OPRFs **59**
4.1 Password-Protected Secret Sharing based on Correlated OPRF 59

5 Performance **67**
5.1 Overview 67
5.2 Legendre OPRF 69
5.2.1 128-bit prime 69
5.2.2 256-bit prime 71
5.2.3 Semi-honest security 71
5.3 Higher-Power Residue OPRF 73
5.3.1 Semi-honest performance 75
5.4 Ram Usage 75
5.5 Comparison 76

6 Conclusion **79**
6.1 Future Work 80

A Appendix **81**
A.1 Additional Benchmarks 81
A.1.1 Complete results for the semi-honest model 81
A.1.2 Complete results for the Power Residue OPRF 81

Bibliography **93**

Chapter 1

Introduction

With recent advances in quantum computing [4], the remaining life span of a significant part of in-use cryptography is slowly diminishing [49]. Applications and protocols that require long-term security should consider attackers with access to quantum computers. Due to results by Shor [55], efficient quantum algorithms are known that with a sufficiently advanced quantum computer can break most in-use asymmetric cryptography. This has led to a drive towards developing and standardising cryptosystems based on different hardness assumptions that do not have any known quantum vulnerabilities, commonly denoted as *post-quantum* schemes [8, 53].

In this thesis, we utilise one such post-quantum hardness assumption, the randomness of the Legendre symbols, to design and study a post-quantum scheme, the Legendre *Oblivious Pseudo-Random Function* (OPRF). We note here that, as commonly done in the literature, we operate in the model of *presumably* post-quantum schemes [27]. Meaning we consider schemes that can be instantiated from post-quantum assumptions but do not necessarily consider a quantum attacker for the security proofs.

1.1 Motivation

OPRFs have become ubiquitous in cryptographic protocols [16], finding applications ranging from private set intersection [47, 18] and secure cloud key management [38] to protecting WhatsApp backups [23]. Therefore, it is crucial to have OPRF protocols that rely on post-quantum hardness assumptions, which the majority of in-use OPRFs do not. While recent works have yielded massive improvements in post-quantum OPRFs [5, 33, 2, 27], their efficiency still pales in comparison to many of the in-use OPRFs [35]. Furthermore, they often are based on novel and not-well studied hardness assumptions [2, 5], require the server to be semi-honest [27, 33, 2], or only include security proofs in a non-composable security framework [33, 2]. As

formulated by Kampanakis and Lepoint in a recent paper, when talking about one OPRF application OPAQUE [39], *"but a fully-fledged post-quantum OPAQUE would also necessitate the OPRF to be quantum-resistant. Unfortunately, state-of-the-art post-quantum OPRFs are orders of magnitude away from being practical"* [42].

Therefore, due to the wide range of protocols that rely on OPRFs, it is crucial for further research into post-quantum OPRFs so that these protocols can be adjusted and implemented in a post-quantum way if the requirement arises. Furthermore, because OPRFs are used as building blocks for various protocols, they can be run in complex environments where multiple sessions may be run in parallel. Thus, we believe it is vital for OPRFs to be proven secure in a framework that provides security even in case of composition and execution in arbitrary environments. To address this, we use the Universal Composability (UC) framework [15], which provides us with strong composability guarantees that can be especially useful for OPRFs.

1.2 Contributions

We design an OPRF based on a post-quantum hardness assumption, the randomness of the Legendre symbols. We expand on previous works that introduced a Pseudo-Random Function (PRF) and a single output bit OPRF based on the Legendre symbols [30, 54]. We present an alternative OPRF that supports having multiple output bits and include a full security proof in the UC framework [15], whereas the single output bit Legendre OPRF did not have a security proof. We also introduce a second OPRF, based on the generalisation of Legendre symbols to higher-power residues, and discuss the necessary assumptions and adjustments for the security of that scheme.

Furthermore, we show that the Legendre OPRF is not secure according to an alternate security notion and that the achieved security notion does not suffice for a specific password-protected secret sharing scheme.

Finally, we implement and provide performance results of both our OPRFs using a Multi-Party Computation (MPC) framework [43]. With these results, we analyse the concrete performance trade-offs between the two OPRFs and compare them to other post-quantum OPRFs. We also evaluate the performance benefits of the two OPRFs when running multiple sessions in parallel.

1.3 Organisation

In Chapter 2 the necessary background on OPRFs, the Legendre symbol, and the universal composability framework are introduced. Furthermore, we provide an overview of post-quantum OPRFs in the literature.

In Chapter 3 we present our variant of the Legendre OPRF. Furthermore, we consider the security of the Legendre OPRF using two different security notions and present the proof of the Legendre OPRF in one of these security notions. We also consider a variant of this security notion in which prefixes are output. These are additional messages the parties output during the OPRF evaluation that are required and used by certain higher-level applications, such as OPAQUE [39]. Finally, we discuss the required modifications to the OPRF and security proof to adapt them to the generalised Power Residue OPRF.

In Chapter 4 we analyse the security of a password-protected secret sharing scheme when the utilised OPRF has the correlate OPRF security notion we have shown the Legendre OPRF to have.

In Chapter 5 we benchmark the concrete efficiency of our schemes. We do so considering both a malicious and a semi-honest security model and provide timing benchmarks in a simulated Wide-Area Network (WAN) environment. We also include comparisons to existing post-quantum OPRFs.

Finally, in Chapter 6 we draw conclusions on the properties and performance of the Legendre and Power Residue OPRFs and propose ideas for future work.

Chapter 2

Background

We begin by introducing OPRFs and their various properties in Section 2.1. We then present the necessary background on Legendre symbols, the Legendre PRF, and the single output bit Legendre OPRF in Section 2.2. Finally, in Section 2.3 we give a short overview of the universal composability framework and the various functionalities we will make use of.

2.1 Oblivious Pseudo-Random Functions

Pseudo-Random Functions (PRFs), first introduced by Goldreich, Goldwasser, and Micali in 1986 [29], are amongst the most widely used cryptographic primitives. They denote a class of keyed functions $\mathrm{PRF}_k(\cdot)$. Given a randomly chosen and secret value k, the output of an oracle that given an element x returns the value $\mathrm{PRF}_k(x)$ is indistinguishable from a truly random value for any probabilistic polynomial-time algorithm [52]. That means that the PRF for any value x needs to output something indistinguishable from a truly random value.

Naor and Reingold [51, 52] realised that certain PRFs could be evaluated *obliviously*, meaning that if there are two parties U and S, where S holds the key k and U holds the value x, they can run a protocol together in which U learns $\mathrm{PRF}_k(x)$, but nothing else about k or any of the other PRF output values, and S learns nothing about x or about the output value. This idea was then formalised by Freedman et al. [28] and was given the name *Oblivious Pseudo-Random Function* (OPRF). On a high level, the requirements during the evaluation of an OPRF, $\mathrm{OPRF}_k(x)$, are the following:

- The client does not learn anything about k
- The client learns the pseudo-random output $\mathrm{OPRF}_k(x)$
- The server does not learn anything about x

- The server does not learn anything about the output

There are various ways to model these security requirements formally [36, 39, 40, 14]. In this work, we consider three variants of OPRF security notions, which we introduce in Chapter 3.

In the years since, there is a vast literature on different OPRFs [3, 2, 40, 5, 12, 27, 33, 36] and their numerous applications. These applications include secure messenger backups [23], password-protected secret sharing [35, 36, 37, 1], private set interaction [24, 47, 57], oblivious keyword search [28], password-authenticated key exchange [39], single-sign on [50, 7], and various others [16].

There is a large variety of properties that OPRFs can have [16]. While an exact description of all these properties is beyond the scope of this thesis, we include a short overview of some of the more common properties.

- *Partially-oblivious PRFs.* These reveal part of the client's input to the server during the execution, which can be useful for establishing some form of rate limiting on client requests.
- *Verifiable OPRFs.* These assure the client that it has received the correct output, i.e., an assurance that the server did not change the key.
- *Updateable OPRFs.* These aim to enable the client to efficiently update previously computed OPRF values to a new key.
- *Programmable OPRFs.* These allow the server to program the OPRF output on a limited number of inputs.
- *Distributed & Threshold OPRFs.* These distribute the server role to several entities, requiring a certain number of them to partake in the protocol in order for the client to receive the output.
- *Single-Point OPRFs.* These do not take a key as an input but instead give a server the used key as an output. This results in the OPRF, for any given key, to be evaluated only once. OPRFs for which this is not the case can also be called *multi-point*.
- *Batched OPRFs.* These can be evaluated in parallel, either with different inputs and the same key or with the same inputs and different keys. The main goal is to provide a performance benefit when requiring multiple protocol evaluations.

One way of creating an OPRF is to evaluate a PRF in a *Multi-Party Computation* (MPC) protocol [27, 54, 25, 17]. These protocols enable the joint evaluation of functions on different parties' inputs without requiring the parties to reveal their inputs to each other. However, one downside is that computation performed inside MPC is costly because it requires the par-

ties to exchange data. Therefore, it is crucial to minimise the computation performed as part of the MPC protocol.

2.2 Legendre Symbols

An element x is a quadratic residue modulo p if an element y exists such that $y^2 = x \mod p$. For a prime number p and natural number $x < p$, the *Legendre symbol* $L_p(x) := \left(\frac{x}{p}\right)$ is defined as 1 if x is a quadratic residue modulo p, -1 if x is a quadratic non-residue modulo p, and 0 if x is 0 modulo p, i.e. we have:

$$\left(\frac{x}{p}\right) \equiv x^{\frac{p-1}{2}} \mod p = \begin{cases} 1 & \text{if } x \neq 0 \text{ is a quadratic residue modulo } p \\ -1 & \text{if } x \text{ is a quadratic non-residue modulo } p \\ 0 & \text{if } x \equiv 0 \text{ modulo } p \end{cases}$$

The Legendre symbol can also be mapped to $\{0,1\}$ by computing

$$\left\lfloor \frac{1}{2}\left(1 - \left(\frac{x}{p}\right)\right)\right\rfloor$$

then quadratic residues and 0 have a value of 0 and quadratic non-residues have a value of 1.

One important property of the Legendre symbol is its *multiplicativity* which states that given elements a, b that:

$$\left(\frac{a \cdot b}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right)$$

The idea of using the Legendre symbols as a source of randomness goes back to a paper published in 1988 by Damgård, who postulated the pseudo-randomness of the Legendre symbol [20]. In particular, that without knowing a key k, the sequence of Legendre symbols

$$\left(\frac{k}{p}\right), \left(\frac{k+1}{p}\right), \left(\frac{k+2}{p}\right), \ldots$$

is indistinguishable for a polynomial-time adversary from a sequence of random elements in $\{\pm 1\}$. It is this underlying assumption on which the Legendre PRF and OPRF are based.

2.2.1 The Legendre PRF

From this randomness assumption, a PRF with a single-bit output can be defined as $\text{PRF}_k(x) = \left(\frac{k+x}{p}\right)$. The security of this PRF is based on the following two problems: the *Shifted Legendre Symbol* (SLS) problem and the *Decision SLS* (DSLS) problem [30].

Definition 2.1 (Shifted Legendre Symbol (SLS) Problem) *Let p be some prime and k randomly chosen in $\mathbb{Z}_p$. Given an oracle $\mathcal{O}_{Leg}$ that given an x outputs $\left(\frac{k+x}{p}\right)$, recover k with non-negligible probability.*

Definition 2.2 (Decisional Shifted Legendre Symbol (DSLS) Problem) *Let k be chosen randomly, the oracle $\mathcal{O}_{Leg}$ be defined as above and $\mathcal{O}_R$ a random oracle that maps elements from $\mathbb{Z}_p$ to $\{-1, 1\}$. Distinguish between $\mathcal{O}_{Leg}$ and $\mathcal{O}_R$ with non-negligible advantage.*

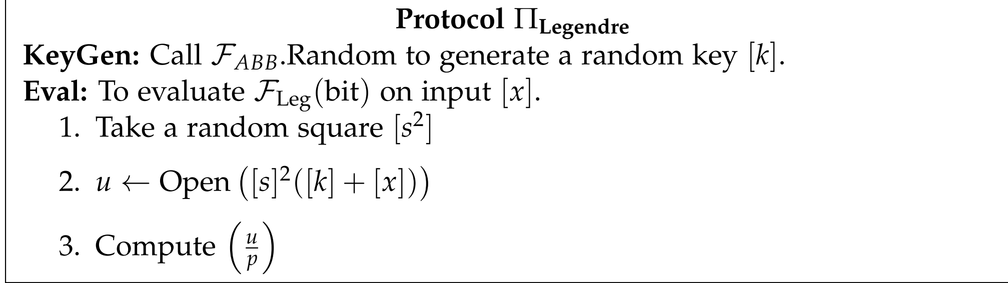
The *DSLS assumption* is then the assumption that there is no efficient polynomial-time algorithm that solves the DSLS problem.

Constructions based on this idea remained mostly unstudied until Grassi et al. [30] showed that assuming the hardness of the decisional SLS problem, a PRF could be constructed from the Legendre symbol with desirable properties in the MPC setting. This is due to the resulting PRF being very efficient, with an evaluation having the cost of just two multiplications in three rounds of communication.

The Legendre PRF scheme operates as follows with the fundamental idea being that, due to the multiplicativity property of the Legendre symbol, the group element can be masked by some random value s^2. After this masking, the only information that remains in the masked value is its Legendre symbol. Therefore, the value can be revealed to the parties, who can perform the more expensive part of computing the Legendre symbol offline. The protocol description relies on an "Arithmetic Black Box" $\mathcal{F}_{ABB}$, also called an arithmetic MPC functionality, which models the operations supported by the underlying MPC protocol. This allows parties to input and reveal values, generate random elements, and compute basic arithmetic operations on these elements in a secure fashion [22, 30]. The notation used in this description is that if an element is written in braces [], it is an element inside the MPC functionality that is secret-shared between the participating parties. The scheme, as presented in [30], can be seen in Fig. 2.1. They also present a version in which the output is not revealed to either party but is again a secret-shared value.

Additionally, they show the PRF's security and correctness. The requirement for this is that $u \neq 0$, which occurs with probability bounded by $1/p + \epsilon$, where ϵ is the probability of solving the SLS problem. They also present an expansion to a probabilistic algorithm that repeatedly samples $[s^2]$ until it is not zero. To check if it is zero, they further sample some value y, reveal the value $[v] = [ys^2]$, and verify that $v \neq 0$. This requires, in expectation, one round and removes the $1/p$ factor from the bound.

Using this method, one can create an OPRF with a single output bit by outputting the result to only one of the parties. In a subsequent work, published in 2022 [54], Seres, Horváth, and Burcsi modelled the Legendre PRF



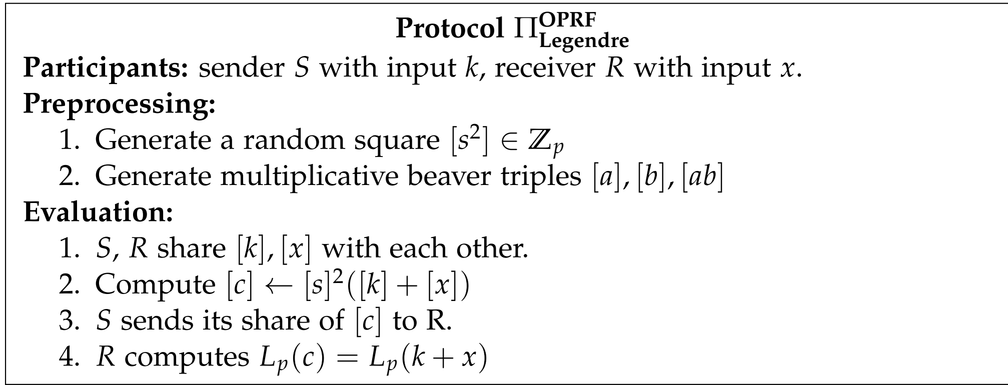
Protocol $\Pi_{\textbf{Legendre}}$

KeyGen: Call $\mathcal{F}_{ABB}$.Random to generate a random key $[k]$.

Eval: To evaluate $\mathcal{F}_{\text{Leg}}(\text{bit})$ on input $[x]$.

1. Take a random square $[s^2]$
2. $u \leftarrow \text{Open}\left([s]^2([k]+[x])\right)$
3. Compute $\left(\frac{u}{p}\right)$

Figure 2.1: The Legendre PRF protocol [30].



Protocol $\Pi^{\textbf{OPRF}}_{\textbf{Legendre}}$

Participants: sender S with input k, receiver R with input x.

Preprocessing:

1. Generate a random square $[s^2] \in \mathbb{Z}_p$
2. Generate multiplicative beaver triples $[a], [b], [ab]$

Evaluation:

1. S, R share $[k], [x]$ with each other.
2. Compute $[c] \leftarrow [s]^2([k]+[x])$
3. S sends its share of $[c]$ to R.
4. R computes $L_p(c) = L_p(k+x)$

Figure 2.2: The basic Legendre OPRF protocol [54].

as a multivariate quadratic cryptosystem and did precisely this, introducing an OPRF based on the Legendre PRF. The OPRF uses a slightly different functionality to model the underlying MPC behaviour and is presented in Fig. 2.2. It works in two phases, an input-independent *preprocessing* phase followed by an *evaluation* phase. In the preprocessing phase beaver triples, used by the secret-sharing MPC protocol to securely do multiplication, and a random secret-shared square are generated. In the evaluation phase, both parties input their shares, perform the computation, and then the masked output is revealed to only the user.

Again, the scheme can be modified to ensure that the value s is picked such that $s^2 \neq 0$, adding in expectation one more round. A programmable and a verifiable OPRF based on the Legendre symbol were also introduced in that paper, but we do not further discuss those constructions here.

In this thesis, we will further study and analyse a variant of this construction. Furthermore, we will include a security proof for our scheme, which was omitted for the single output bit Legendre OPRF of [54].

2.2.2 Cryptanalysis of the Legendre PRF

In this section, we will briefly present an overview of the various known attacks against the DSLS and SLS assumptions.

In the case of quantum query access to the Legendre oracle, an attack is known that recovers the key k in a single query and in quantum-polynomial time [56]. On the other hand, if there is no quantum query access to the oracle, there is currently no known sub-exponential attack against the assumptions, both for classical and quantum adversaries. Therefore, the hardness of the two problems is seen as *post-quantum* if there is no quantum oracle access.

An important factor in determining the efficiency of attacks is the number of available oracle queries, or alternatively the number of known output bits. Spurred on by the Ethereum Challenge[1], a series of bounties for both arithmetic improvements and concrete instances to be broken, there have been various works in recent years analysing the security of the assumptions [9, 46, 41]. The challenge provides rewards for improved complexity key recovery and PRF distinguishing attacks, as well as rewards for key recovery of certain publicly provided instances. So far, for the publicly provided instances, the largest broken size is for an 84-bit prime where 2^{20} bits of output are provided.

In 2019, Khovratovich presented a collision-based attack, which recovers the key with $\mathcal{O}(\sqrt{p}\log p)$ queries and computational cost of $\mathcal{O}\left(\sqrt{p}\log p\right)$ [46]. This was followed by Beullens et al. [9], who found a more efficient attack in the low-data case by using the multiplicative property of the Legendre symbol. Concretely for the case where there are M available queries, the attack works with a runtime of $\mathcal{O}\left(M + \frac{p\log p}{M}\right)$ and $M \cdot \log p$ memory, or a runtime of $\mathcal{O}\left(M^2 + \frac{p\log^2 p}{M^2}\right)$ and $\frac{M^2}{\log p}$ memory. A concurrent work by Kaluđerović et al. [41] further improved on these bounds, presenting an attack with complexity $\mathcal{O}\left(\frac{M^2}{p} + \frac{p\log p\log\log p}{M^2}\right)$ with a memory requirement of M^2. For the case $M = \sqrt[4]{p\log^2 p\log\log p}$ oracle queries are available this results in an attack where the expected number of operations is $\mathcal{O}(\sqrt{p\log\log p})$.

A different approach, used in [54], modelled the Legendre PRF as a multivariate quadratic cryptosystem, but this did not result in any improvements on the above attacks.

Therefore, we conclude that under the current status of cryptanalysis of the Legendre PRF, for a security parameter of ℓ, using a prime p such that

[1] https://legendreprf.org/bounties

$\log_2(p)$ is greater than $2 \cdot \ell$ suffices, as the best-known attack has complexity bounded by $\mathcal{O}(\sqrt{p \log\log p})$, regardless of the number of queries.

2.2.3 Higher-power residues

Using the Legendre symbol, one can recover one bit of output from a group element. As an alternative, we also consider the generalisation to *higher-power residues*, which enable getting multiple bits of outputs from any group element, thus resulting in more efficient schemes [10].

Using the a-th power residue, each element now provides us with $\log_2(a)$ many output bits, coming at the cost of increased local computation. We will use $\mathcal{L}_p^a(x)$ to denote the a-th power residue of the term x. If we have some prime p and generator g of $\mathbb{Z}_p^*$ this is defined as:

$$\mathcal{L}_p^k(x) = \begin{cases} i & \text{if } x/g^i \equiv h^a \mod p \text{ for some } h \in \mathbb{Z}_p^* \\ 0 & \text{if } x \equiv 0 \mod p \end{cases}$$

Concretely, the a-th power residue of an element x is the value i such that $x = g^i h^a = g^{i+j\cdot a}$ for some j, so it is the value of the power of g that x is, modulo a. We also observe that if we utilise $a = 2$ we recover the definition of the mapped Legendre symbol.

A hardness assumption about the randomness of the generalised power residues can be formulated analogously to the SLS and DSLS problems and assumptions, now comparing to a random oracle that returns values from $\{0, \ldots, a-1\}$.

For an attacker that can query the oracle M times, the best-known attack against the higher-power residues has a computational cost of $\mathcal{O}\left(\frac{p \log^2 p}{a \cdot M \cdot \log^2 a}\right)$ [9]. Still, it is important to note that this pseudo-randomness is less well studied than that of the Legendre symbol, which presents a potential downside of using these.

Now that we have introduced the power residues, we introduce one further property of the power residues, which we use for the proof. We will state this as a lemma that was proven by Beullens et al. in the paper in which they introduced LegRoast, a signature scheme based on the Legendre symbol [10]. We note that the lemma uses K for the key and k for the power residue being used, as opposed to k and a in our notation.

Lemma 2.3 *Let p be a prime and $k | p-1$. For any $K \neq K' \in \mathbb{F}_p$ and $a \in \mathbb{Z}_k$, let $I_{K,K',a}$ be the set of indices i such that $\mathcal{L}_p^k(K+i) = \mathcal{L}_p^k(K'+i) + a$. Then we have*

$$\frac{p}{k} - \sqrt{p} - 1 \leq |I_{K,K',a}| \leq \frac{p}{k} + \sqrt{p} + 2$$

For the proof of the lemma, we direct the reader to the LegRoast paper [10]. The lemma provides a lower bound on the number of indexes where the power residues offset by two different keys have a constant difference a.

2.3 Universally Composable Security

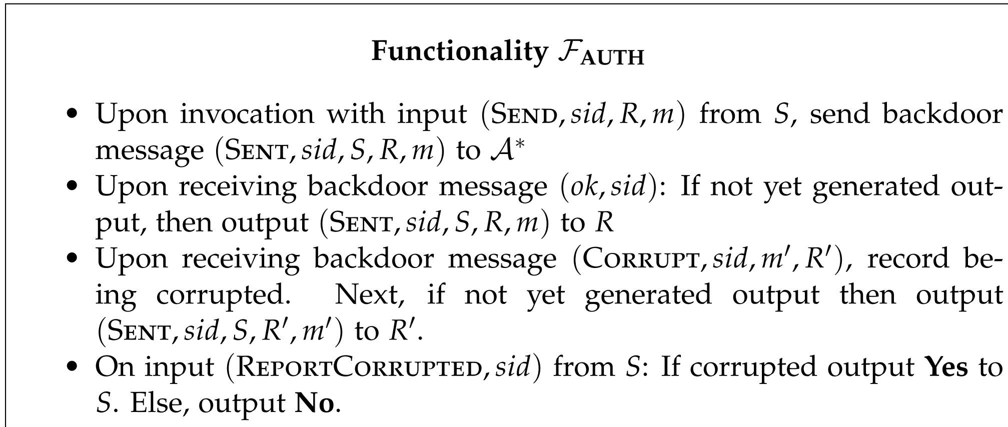
Universally Composable (UC) security [15] is a framework that aims to prove and guarantee the security of schemes in arbitrary polynomial runtime environments, including having multiple sessions executed in parallel and concurrently. As such, it provides very strong guarantees as any scheme building on some UC secure protocol can, for the sake of the proof, replace the utilised UC secure protocol with an idealised functionality.

Since its introduction in 2000, the framework has undergone numerous revisions and adjustments, becoming widely used for security proofs. In this thesis, we will use this framework to formalise the desired security properties and perform the security proofs.

We begin here by very briefly outlining the concepts required for a security proof in the UC framework. A more thorough description is omitted for the sake of brevity, and we refer the reader to the Canetti paper for a complete introduction [15].

The way the UC framework works is that the security and functionality requirements of a system are specified by the formal description of an ideal functionality. This functionality specifies how the system functions, what the parties input, what the adversary, denoted as $\mathcal{A}^*$, learns, and what the adversary can control. A simulation-based indistinguishably proof is performed to then show the security of some protocol, where the distinguisher is the so-called *environment*. The environment determines the inputs of all the parties, learns their outputs, and controls the adversary [15]. The simulation aims to ensure that any polynomial-time environment cannot distinguish between the real-world protocol execution and an *ideal world* execution in which the parties are so-called dummy parties that just forward their inputs to the ideal functionality, and the simulator plays in place of the adversary.

Once such a security proof has been performed, this framework provides security in any context and under general composition. Furthermore, when proving the security of any protocol that uses the secure protocol as a sub-protocol, the sub-protocol can be replaced by the ideal functionality that models the desired security. In doing so, the sub-protocol can be arbitrarily replaced with any other protocol secure according to the same functionality without requiring any adjustments to the proof.



Functionality $\mathcal{F}_{\text{AUTH}}$

- Upon invocation with input $(\textsc{Send}, sid, R, m)$ from S, send backdoor message $(\textsc{Sent}, sid, S, R, m)$ to $\mathcal{A}^*$
- Upon receiving backdoor message (ok, sid): If not yet generated output, then output $(\textsc{Sent}, sid, S, R, m)$ to R
- Upon receiving backdoor message $(\textsc{Corrupt}, sid, m', R')$, record being corrupted. Next, if not yet generated output then output $(\textsc{Sent}, sid, S, R', m')$ to R'.
- On input $(\textsc{ReportCorrupted}, sid)$ from S: If corrupted output **Yes** to S. Else, output **No**.

Figure 2.3: Ideal authenticated channel functionality $\mathcal{F}_{\text{AUTH}}$ [15].

We also specify the following convention we will use in this thesis. Whenever a message is sent to an ideal functionality, but the behaviour of the functionality in response to that message is not explicitly defined, we assume that the message is ignored.

We now present some common functionalities, which we will use in our constructions.

Authenticated Channels $\mathcal{F}_{\text{AUTH}}$, as seen in Fig. 2.3, is a functionality that models the behaviour of authenticated channels between two parties [15]. Any party can use the Send interface to cause a message to be sent to another party. Once they do so, the adversary is notified of the sender, the content, and the recipient of the message. The content is revealed to the adversary because we do not have any confidentiality as part of these channels. The message is only forwarded to the recipient once the adversary permits it by sending the *ok* message. This models the adversary as a network-level adversary that can arbitrarily delay or drop messages sent between the parties. Furthermore, the functionality can be corrupted, allowing the adversary to make it appear like the sender sent a message of the adversary's choosing to an arbitrary recipient. The sender can detect this behaviour using the ReportCorrupted interface. While all messages sent to/from the adversary are labelled as *backdoor* messages in this functionality, the backdoor word is often omitted in functionality descriptions, and we do so as well in this thesis.

Common Reference String Another commonly used functionality is $\mathcal{F}_{CRS}$, which models the idealised behaviour of a common reference string [15, 34]. As can be seen in Fig. 2.4, this functionality only has a single interface: Value. On the first invocation, the functionality samples a random value

Functionality $\mathcal{F}_{CRS}$

$\mathcal{F}_{CRS}$ proceeds as follows, when parameterised by a distribution D.

When activated for the first time on input (Value, sid) choose a value $d \leftarrow\$ D$ and send d back to the activating party. In each other activation return the value d to the activating party.

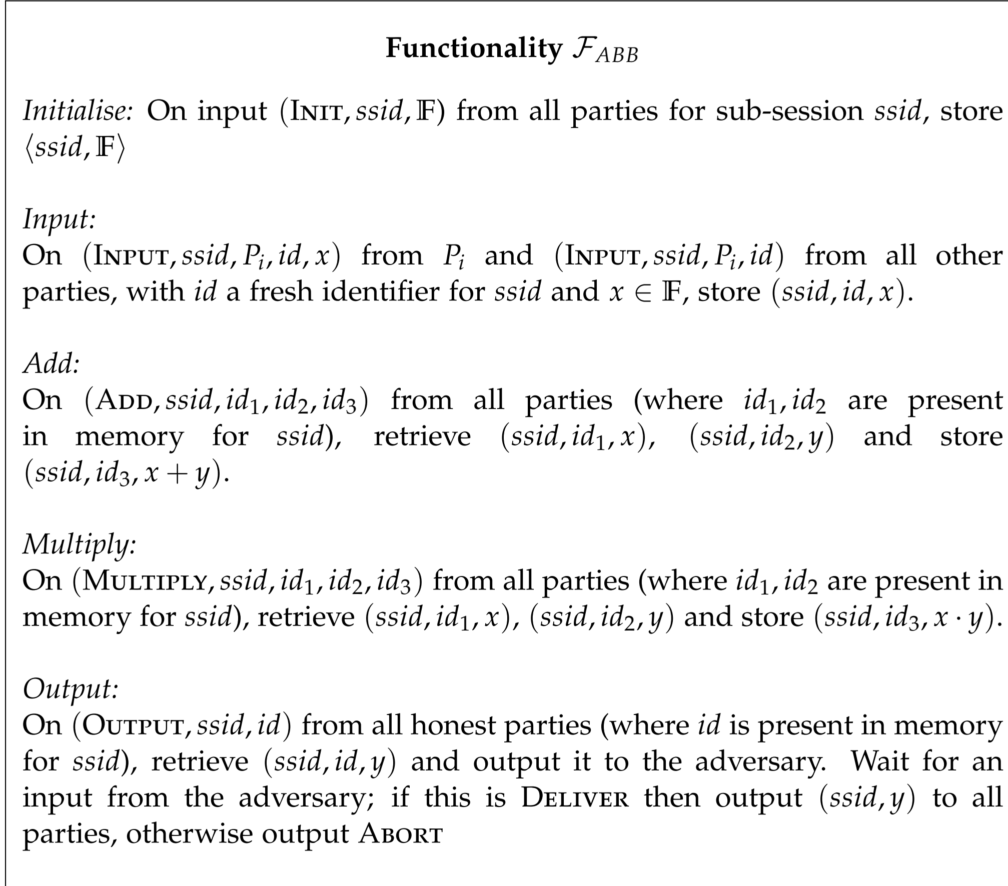
Figure 2.4: Ideal Common Reference String Functionality $\mathcal{F}_{CRS}$ [15, 34].

according to some distribution and responds with that value. In all later invocations, it responds with the same value.

Arithmetic Black Box (ABB) $\mathcal{F}_{ABB}$ is the functionality we will be using to model the functionalities provided by the used MPC protocol [44, 19, 32, 22, 26, 21].[2] As seen in Fig. 2.5, this functionality has five interfaces. For any command to be executed, it requires all the parties to send the command. The Init interface allows the parties to initialise it with the field the arithmetic operations will be performed over. Then there is the Input interface, which models a party secret-sharing a value between the other parties. There are then two permitted operations Add and Multiply, which allow for operations on secret shared values. Finally, there is the Output interface, which will output the value to the adversary and, upon the adversary's choosing, will be output to all the parties. While these interfaces only expose fairly simple operations, they can be used for more complex computations, such as sampling random values, adding and multiplying by constants, or outputting to only a single party. We briefly outline how each of these operations can be performed, as our OPRF makes use of them.

- To get a random value: Have all the parties input a random value and add them together. As long as one party is honest, then the others cannot influence or learn the random value.
- To multiply by a constant: Repeatedly add the value to itself.
- To add by a constant: If the value is outputted, locally add the constant after the fact. If the value is multiplied with some other value, do the multiplication combined with the above steps to multiply by a constant and add the two parts together.
- To output to only one party: have that party input a random value and add that random value to the output before performing the output operation. This hides the true value from the adversary and the other parties.

[2] This is also sometimes called $\mathcal{F}_{\text{Online}}$ or $\mathcal{F}_{\text{AMPC}}$.



Functionality $\mathcal{F}_{ABB}$

Initialise: On input $(\text{Init}, ssid, \mathbb{F})$ from all parties for sub-session $ssid$, store $\langle ssid, \mathbb{F} \rangle$

Input:
On $(\text{Input}, ssid, P_i, id, x)$ from P_i and $(\text{Input}, ssid, P_i, id)$ from all other parties, with id a fresh identifier for $ssid$ and $x \in \mathbb{F}$, store $(ssid, id, x)$.

Add:
On $(\text{Add}, ssid, id_1, id_2, id_3)$ from all parties (where id_1, id_2 are present in memory for $ssid$), retrieve $(ssid, id_1, x)$, $(ssid, id_2, y)$ and store $(ssid, id_3, x + y)$.

Multiply:
On $(\text{Multiply}, ssid, id_1, id_2, id_3)$ from all parties (where id_1, id_2 are present in memory for $ssid$), retrieve $(ssid, id_1, x)$, $(ssid, id_2, y)$ and store $(ssid, id_3, x \cdot y)$.

Output:
On $(\text{Output}, ssid, id)$ from all honest parties (where id is present in memory for $ssid$), retrieve $(ssid, id, y)$ and output it to the adversary. Wait for an input from the adversary; if this is Deliver then output $(ssid, y)$ to all parties, otherwise output Abort

Figure 2.5: Ideal MPC arithmetic black box functionality $\mathcal{F}_{ABB}$

We note that there are many variations of this functionality found in the literature. The presented ABB functionality is slightly modified from one of the variants commonly found in the literature. We modified it by augmenting it with subsession identifiers (ssids). We assume that these contain the session identifier as a substring. The variant without the subsession identifiers can be seen in Fig. 2.6 [44, 19].

2.4 Related Work

There have been various other proposals for post-quantum OPRF schemes. In this section we aim to highlight some of them. In the benchmark section, we will also compare the performance of our OPRF to some of these.

Some of the first post-quantum OPRFs were presented by Boneh et al. [12], who presented two isogenies-based OPRFs. The first relies on a hardness assumption which has since been shown to be insecure (SIDH) [6, 48]. The

Functionality $\mathcal{F}_{ABB}$

Initialise: On input $(\textsc{Init}, \mathbb{F})$ from all parties, store $\mathbb{F}$.

Input:
On $(\textsc{Input}, P_i, id, x)$ from P_i and $(\textsc{Input}, P_i, id)$ from all other parties, with id a fresh identifier and $x \in \mathbb{F}$, store (id, x).

Add:
On $(\textsc{Add}, id_1, id_2, id_3)$ from all parties (where id_1, id_2 are present in memory), retrieve (id_1, x), (id_2, y) and store $(id_3, x + y)$.

Multiply:
On $(\textsc{Multiply}, id_1, id_2, id_3)$ from all parties (where id_1, id_2 are present in memory), retrieve (id_1, x), (id_2, y) and store $(id_3, x \cdot y)$.

Output:
On $(\textsc{Output}, id)$ from all honest parties (where id is present in memory), retrieve (id, y) and output it to the adversary. Wait for an input from the adversary; if this is Deliver then output y to all parties, otherwise output Abort

Figure 2.6: Ideal MPC arithmetic black box functionality $\mathcal{F}_{ABB}$, as is commonly seen in the literature [44, 19]. This does not include multiple sub-sessions.

second OPRF is based on the CSIDH assumption but no security proof in the Universal Composability framework for it was included and it requires the server to be semi-honest. The CSIDH-based construction has communication costs of around 424KB in 3 rounds of communication. Another CSIDH-based OPRF, called OPUS, was introduced in a recent work [33]. OPUS has reduced bandwidth requirements but is secure only with semi-honest client and server.

Two lattice-based OPRFs, including one with malicious security, were introduced by Albrecht et al. [3]. However, while the variant with malicious security is practically instantiable, it requires communication of over 120GB, limiting its potential usability.

There have also been various OPRFs based on the idea of computing a PRF in an MPC framework. Faller et al. [27] proposed an OPRF scheme based on garbled circuits. This is shown to be UC secure in the case of a semi-honest server but not in the case of a malicious server. Furthermore, multiple works have investigated using the Dark Matter weak PRF [11] in an MPC setting as

an OPRF. Dinur et al. proposed doing so using secret-sharing sharing [25], resulting in an OPRF secure in the semi-honest model which requires pre-processing. Albrecht et al. instead used torus fully homomorphic encryption [2], resulting in a scheme that requires communication of around 70 MB for an evaluation, whereby only 3MB are required for the online phase. However, the server utilised for their evaluation was a server with 768GB of RAM, which may not be realistic in many real-world applications. Again, the OPRF was only shown to be secure in the case of a semi-honest server, although a potential extension based on heuristics to a verifiable OPRF with malicious security was also discussed.

Finally, one of the most efficient post-quantum OPRFs offering malicious security was introduced in a recent work by Basso [5]. It is another isogeny-based OPRF, relying on a trusted setup and the SIDH assumption but with countermeasures against the known attacks built in. It is round-optimal, requiring only two rounds of communication for an evaluation. The required bandwidth for it is 3MB when the verifiability property of the OPRF is not needed and 8.7MB for the verifiable variant that offers security in case of a malicious server, although we note that no implementation for this OPRF is available.

Chapter 3

The Legendre OPRF

In Section 3.1 we introduce the Legendre OPRF and motivate the design decisions. We then analyse our OPRF with two different security definitions $\mathcal{F}_{OPRF}$ in Section 3.2 and $\mathcal{F}_{corOPRF}$ in Section 3.3. We also consider the security based on a variant of $\mathcal{F}_{corOPRF}$ that includes prefixes and discuss the required changes in Section 3.4. Finally, we introduce and discuss the Power Residue OPRF in Section 3.5.

3.1 The Legendre OPRF

3.1.1 Ideal OPRF functionality

Before we introduce the Legendre OPRF, we first introduce and discuss $\mathcal{F}_{OPRF}$, the UC security notion for OPRFs, which we use to analyse the security of our scheme in Section 3.2. The presented functionality $\mathcal{F}_{OPRF}$ is a slight modification of the one introduced by Jarecki et al. [39], which was used to show the security of the OPAQUE protocol and was a revision of a similar OPRF functionality, also introduced by Jarecki et al. [36]. For the sake of simplicity, we modify it by removing the prefixes. The other security notion we use, $\mathcal{F}_{corOPRF}$, is also based on a slight modification of this one.

The ideal functionality $\mathcal{F}_{OPRF}$ can be seen in Fig. 3.1. The functionality models the interaction of multiple users with a single server. The functionality is initialised by a server sending an Init message. This is then the unique server used for this session and the functionality stores a table F_S of truly random values representing the server's OPRF output values. The functionality models adaptive compromise, which allows the adversary (with the environment's permission) to send a Compromise message and then arbitrarily query the server's table F_S. This is modelled using the OfflineEval interface, which either allows the server to query its own function table or

Ideal OPRF functionality $\mathcal{F}_{OPRF}$

The OPRF function is parameterised by a public PRF output length ℓ. For every i and x the value $F_i(x)$ is initially undefined. The first time an undefined value $F_i(x)$ is referenced $\mathcal{F}_{OPRF}$ sets $F_i(x) \leftarrow\$ \{0,1\}^{\ell}$.

Initialisation:
On message (Init, sid) from party S, if this is the first Init message for sid set $tx = 0$ and send (Init, sid, S) to $\mathcal{A}^*$. From now on use the tag S to denote the unique entity which sent the Init message for the session identifier sid. (Ignore all subsequent Init messages for sid.)

Server Compromise:
On message (Compromise, sid) from $\mathcal{A}^*$, declare S as Compromised.
Note: Message (Compromise, sid) requires permission from the environment.
//If S is corrupted, then it is declared Compromised as well.

Offline Evaluation:
On (OfflineEval, sid, S^*, x) from $P \in \{S, \mathcal{A}^*\}$ do:

- Send (OfflineEval, $sid, F_{S^*}(x)$) to P if (i) $P = S$ and $S^* = S$ or (ii) $P = \mathcal{A}^*$ and either $S^* \neq S$ or S Compromised

Online Evaluation:

- On (Eval, $sid, ssid, S', x$) from $P \in \{U, \mathcal{A}^*\}$, send (Eval, $sid, ssid, P, S'$) to $\mathcal{A}^*$. Record $\langle ssid, P, x \rangle$
- On (SndrComplete, $sid, ssid'$) from S, send (SndrComplete, $sid, ssid', S$) to $\mathcal{A}^*$, set tx++
- On (RcvComplete, $sid, ssid, P, S^*$) from $\mathcal{A}^*$, ignore this message if there is no record $\langle ssid, P, x \rangle$ stored. Else:
  - If S is not Compromised and $S^* = S$:
    If $tx = 0$ ignore this message. Else decrement tx
  - Send (Eval, $sid, ssid, F_{S^*}(x)$) to P

Figure 3.1: Ideal OPRF Functionality, based on the ideal OPRF functionality introduced in [39].

the adversary to query an arbitrary table F_i if $i \neq S$ or if $i = S$ and the server is compromised.

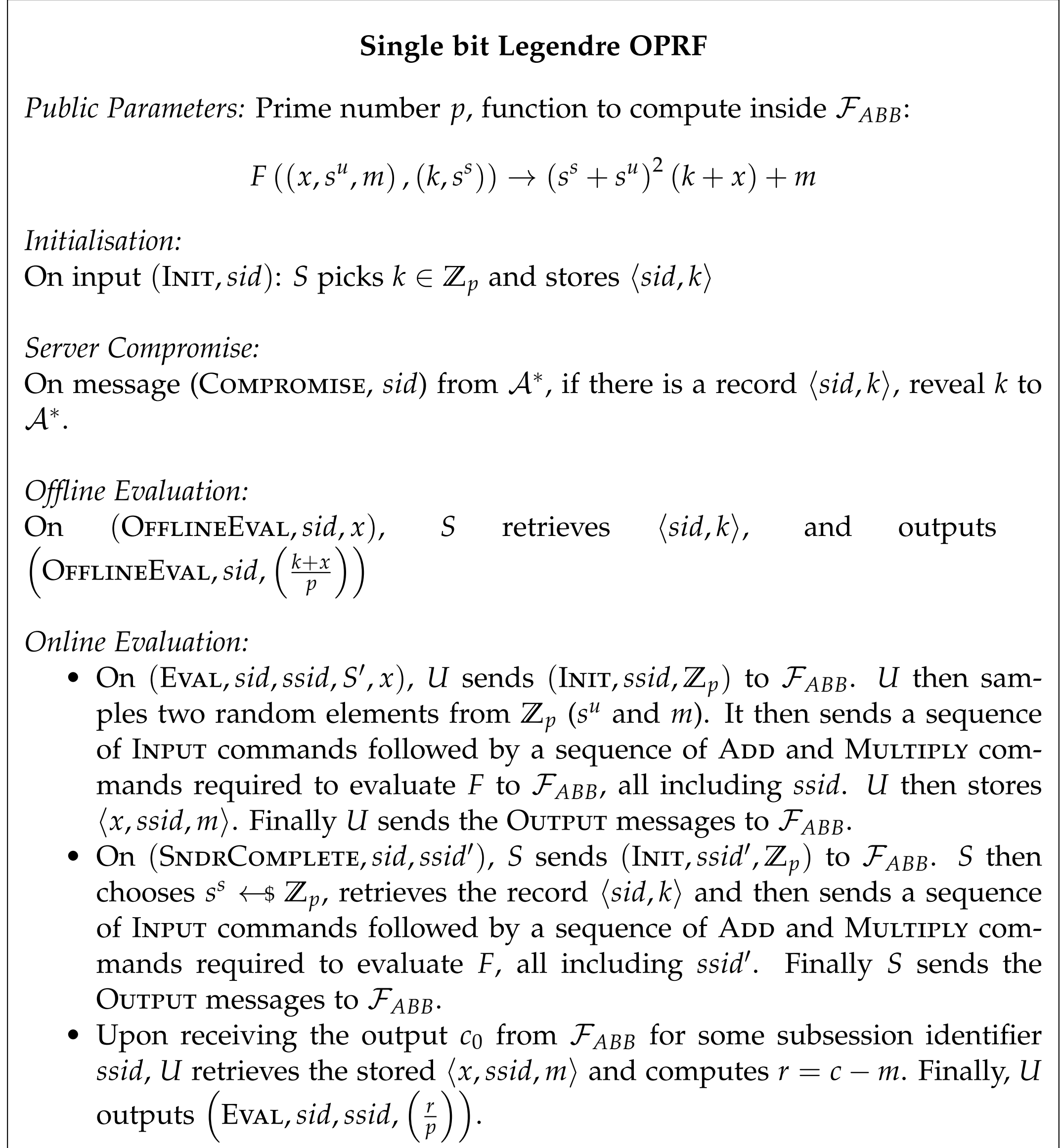
Finally, there is the *Online Evaluation* interface, which allows the user and server together to run the OPRF protocol in some subsession *ssid*. It requires the user to send an EVAL message with its input and the server to send a SNDRCOMPLETE message. The adversary can then send a RCVCOMPLETE message to cause the session to complete. The adversary learns when the parties send their messages and, in the RCVCOMPLETE message, can specify an alternative table S^* from which the user will receive its output. This models the network adversary participating in the role of the server but using some different key.

The counter for the server *tx* ensures that the number of times the server's table is evaluated is limited by the number of times the server participated in the protocol. Each time the server sends a SNDRCOMPLETE message, it is incremented, and each time the OPRF of the server is evaluated, it is again decremented. Furthermore, the functionality guarantees random outputs even in the case of malicious servers and requires that the user's input and output remain hidden from both the server and the adversary.

3.1.2 Legendre OPRF

In Fig. 3.2 we consider taking the single output bit Legendre OPRF, as introduced in Fig. 2.2, from [54] and formulate it in the universal composability framework with the above interfaces. We note that due to the limited interfaces available for $\mathcal{F}_{ABB}$, to get a random value s^2 we combine two shares s^S and s^U. Furthermore, to have an output that we reveal to only one party, we mask the output with a randomly chosen user value m. We do not include the whole sequences of commands that the parties send to $\mathcal{F}_{ABB}$, but they can be summarised as follows:

1. Send sequence of INPUT messages to load the user and server input into ABB.
2. Recreate the multiplicative masking value s from the two shares using ADD
3. Compute s^2 using MULTIPLY
4. Compute $x + k$ using ADD
5. Compute $s^2 \cdot (x + k)$ using MULTIPLY
6. Add m to the result of the above step to mask the output value using ADD
7. Send the OUTPUT message to return the value $m + s^2(x + k)$.



Single bit Legendre OPRF

Public Parameters: Prime number p, function to compute inside $\mathcal{F}_{ABB}$:

$$F\left((x, s^u, m), (k, s^s)\right) \rightarrow (s^s + s^u)^2 (k + x) + m$$

Initialisation:
On input $(\textsc{Init}, sid)$: S picks $k \in \mathbb{Z}_p$ and stores $\langle sid, k \rangle$

Server Compromise:
On message ($\textsc{Compromise}$, sid) from $\mathcal{A}^*$, if there is a record $\langle sid, k \rangle$, reveal k to $\mathcal{A}^*$.

Offline Evaluation:
On $(\textsc{OfflineEval}, sid, x)$, S retrieves $\langle sid, k \rangle$, and outputs $\left(\textsc{OfflineEval}, sid, \left(\frac{k+x}{p}\right)\right)$

Online Evaluation:

- On $(\textsc{Eval}, sid, ssid, S', x)$, U sends $(\textsc{Init}, ssid, \mathbb{Z}_p)$ to $\mathcal{F}_{ABB}$. U then samples two random elements from $\mathbb{Z}_p$ (s^u and m). It then sends a sequence of Input commands followed by a sequence of Add and Multiply commands required to evaluate F to $\mathcal{F}_{ABB}$, all including $ssid$. U then stores $\langle x, ssid, m \rangle$. Finally U sends the Output messages to $\mathcal{F}_{ABB}$.
- On $(\textsc{SndrComplete}, sid, ssid')$, S sends $(\textsc{Init}, ssid', \mathbb{Z}_p)$ to $\mathcal{F}_{ABB}$. S then chooses $s^s \leftarrow\!\!\$\ \mathbb{Z}_p$, retrieves the record $\langle sid, k \rangle$ and then sends a sequence of Input commands followed by a sequence of Add and Multiply commands required to evaluate F, all including $ssid'$. Finally S sends the Output messages to $\mathcal{F}_{ABB}$.
- Upon receiving the output c_0 from $\mathcal{F}_{ABB}$ for some subsession identifier $ssid$, U retrieves the stored $\langle x, ssid, m \rangle$ and computes $r = c - m$. Finally, U outputs $\left(\textsc{Eval}, sid, ssid, \left(\frac{r}{p}\right)\right)$.

Figure 3.2: The Legendre OPRF scheme with a single output bit. This is an adaption of the single output bit OPRF shown in Fig. 2.2 from [54], adapted to the UC functionality $\mathcal{F}_{OPRF}$.

Unfortunately, this scheme has multiple fundamental issues if one tries to prove its security in the UC framework.

Firstly, in case the server is compromised and the user is honest, the environment can calculate the expected output $(\frac{x+k}{p})$ and compare that to the user's output. On the other hand, while the simulator knows k it does not learn any information about the user's input x during the evaluation. Due to this, the simulator cannot reliably influence the output of the user to be the correct value. This problem arises in many such UC security notions. We will solve this by using the *Random Oracle Model* (ROM) and making the user output a hash value depending on x and the Legendre symbol, as is commonly done in the literature for this security notion [5, 27, 36].

We now discuss various possibilities for the hash inputs and their problems.

Single bit hash function. The first option we consider is to take some hash function with an output length of one bit. The output value in this case would be $H\left(x, \left(\frac{x+k}{p}\right)\right)$.
Under this modification, for the environment to be able to compute the expected output, it needs to query the random oracle with a pair (x, b). The simulator can then verify if $b = \left(\frac{x+k}{p}\right)$ and, if so, set the hash function output to the output of $\mathcal{F}_{OPRF}$. Otherwise, the simulator can set the hash output to a random bit or the output of some other table.

Two issues remain with this construction. Firstly, even without knowing the key, the hash output can be queried in advance, as there are only two possibilities for b and the simulator would need to set the value correctly. Secondly, there will exist k_1, k_2 such that the OPRF evaluated with k_1 at value x is equal to the OPRF evaluated with k_2 at value x, whereas for some value x' they are not equal. The simulator does not learn the user input, though and thus, the environment can distinguish between these cases. This follows directly from the fact that given some random keys k, k' and value x, under the DSLS assumption, we have $\Pr\left[\left(\frac{k+x}{p}\right) = -1\right] \approx \Pr\left[\left(\frac{k'+x}{p}\right) = -1\right] \approx \frac{1}{2}$.

Multi-key. One attempt to deal with these issues is to have an ℓ-bit hash value output and model the server as having ℓ keys. So the protocol would take $k_1, \ldots, k_\ell$ as input from the server, for $H : \{0,1\}^* \times \{0,1\}^\ell \to \{0,1\}^\ell$ the final output would be

$$H\left(x, \left(\frac{x+k_1}{p}\right), \left(\frac{x+k_2}{p}\right), \ldots, \left(\frac{x+k_\ell}{p}\right)\right)$$

While this remedies the first problem, the second problem remains unsolved by this. Because only a single output bit is retrieved for each key, the following becomes possible: the environment chooses two keys such that for

some values x they have the same PRF value and for other values x' they do not, for example, by replacing only the last key. Since it can then pass such an x to the user without the simulator learning x and tell the server to use one of these keys, the simulator cannot set the PRF values to maintain consistency. Thus, the environment would be able to distinguish between the two worlds and the scheme does not fulfil the desired security notion in the UC framework.

Single key – multi-bit output. The third method we discuss here relies on the DSLS assumption. Since we assume that under any key k, $\left(\frac{x+k}{p}\right)$ is distributed randomly, we can use the same key and instead offset x by one instead of using ℓ different keys. Thus, the final output would be

$$H\left(x, \left(\frac{x+k+1}{p}\right), \left(\frac{x+k+2}{p}\right), \ldots, \left(\frac{x+k+\ell}{p}\right)\right)$$

Since we assume that without knowing k, the Legendre symbol output is pseudo-random, the sequence is indistinguishable from a random sequence in isolation. The issue now arises from the fact that a malicious user could evaluate the OPRF at some value x and then have only to guess a single bit to get the OPRF evaluation at the value $x+1$, because it would have learnt all except the last bits required for the evaluation of $x+1$ already, which goes against the security definition. Even more dangerously, by querying x and then $x+\ell$, the user could learn the OPRF evaluation for all the points between x and $x+\ell$.

Hashed input Instead of x, we pass $H_1(x)$ and use that to calculate the output. We then will show that for appropriately chosen prime value p and evaluation length ℓ, the probability that the OPRF evaluation at some point reveals *any* information about the OPRF evaluation at some other point is small.

Another modification we make is using a list of ℓ random offsets $L \leftarrow\!\!\$\ \mathbb{Z}_p^\ell$ instead of $L = [1, \ldots, \ell]$. We will use this fact, together with Lemma 2.3, to show that the existence of two keys that evaluate to the same PRF value for some x is bounded. We will use $\mathcal{F}_{CRS}$ to generate and retrieve this list. Therefore, we define the output of the Legendre OPRF as

$$H_2\left(x, \left(\frac{H_1(x)+k+L[0]}{p}\right), \ldots, \left(\frac{H_1(x)+k+L[\ell-1]}{p}\right)\right)$$

We present the final version of the Legendre OPRF, with the above-discussed modifications, in Fig. 3.3.

In effect, we evaluate ℓ many bits. The exact bits we determine are dependent on some list L. We also have the user input ℓ random values to mask the output.

Legendre OPRF

Public Parameters: Prime number p, output length ℓ. Distribution D for $\mathcal{F}_{CRS}$ that returns ℓ many values from $\mathbb{Z}_p$ chosen uniformly at random.
Function to compute inside $\mathcal{F}_{ABB}$ for some list L known to both parties:

$$F\left((x, s_0^u, \ldots, s_{\ell-1}^u, m_0, \ldots, m_{\ell-1}), (k, s_0^s, \ldots, s_{\ell-1}^s)\right) \to$$
$$(s_0^s + s_0^u)^2 (k + H_1(x) + L[0]) + m_0, \ldots, (s_{\ell-1}^s + s_{\ell-1}^u)^2 (k + H_1(x) + L[\ell-1]) + m_{\ell-1}$$

Initialisation:
On input (INIT, sid): S picks $k \leftarrow\$ \mathbb{Z}_p$ and stores $\langle sid, k\rangle$

Server Compromise:
On (COMPROMISE, sid), if there is a record $\langle sid, k\rangle$ reveal k to $\mathcal{A}^*$

Offline Evaluation:
On (OFFLINEEVAL, sid, x), send (VALUE, sid) to $\mathcal{F}_{CRS}$, and receive the list L. Then the server retrieves $\langle sid, k\rangle$ and outputs $\left(\text{OFFLINEEVAL}, sid, H_2\left(x, \left(\frac{H_1(x)+k+L[0]}{p}\right), \left(\frac{H_1(x)+k+L[1]}{p}\right), \ldots, \left(\frac{H_1(x)+k+L[\ell-1]}{p}\right)\right)\right)$

Online Evaluation:

- On (EVAL, $sid, ssid, S', x$), U sends (VALUE, sid) to $\mathcal{F}_{CRS}$ and receives the list L. U then sends (INIT, $ssid, \mathbb{Z}_p$) to $\mathcal{F}_{ABB}$. U then samples $2 \cdot \ell$ many elements from $\mathbb{Z}_p$ ($s_0^u, \ldots, s_{\ell-1}^u$ and $m_0, \ldots, m_{\ell-1}$). It then sends a sequence of INPUT commands followed by a sequence of ADD and MULTIPLY commands required to evaluate the function F with list L to $\mathcal{F}_{ABB}$, all including $ssid$. U then stores $\langle x, ssid, m_0 \ldots m_{\ell-1}\rangle$. Finally U sends ℓ many OUTPUT messages to $\mathcal{F}_{ABB}$.
- On (SNDRCOMPLETE, $sid, ssid'$), S sends (VALUE, sid) to $\mathcal{F}_{CRS}$, and receives the list L. It then sends (INIT, $ssid', \mathbb{Z}_p$) to $\mathcal{F}_{ABB}$. S then chooses $s_0^s, \ldots, s_{\ell-1}^s \leftarrow\$ \mathbb{Z}_p$, retrieves the record $\langle sid, k\rangle$ and then sends a sequence of INPUT commands followed by a sequence of ADD and MULTIPLY commands required to evaluate the function F with list L to $\mathcal{F}_{ABB}$, all including $ssid'$. Finally S sends ℓ many OUTPUT messages to $\mathcal{F}_{ABB}$.
- Upon having received ℓ many outputs $c_0, \ldots, c_{\ell-1}$ from $\mathcal{F}_{ABB}$ for some subsession identifier $ssid$, U retrieves the stored $\langle x, ssid, m_0 \ldots m_{\ell-1}\rangle$, and computes $r_i = c_i - m_i$. Then U outputs $\left(\text{EVAL}, sid, ssid, H_2\left(x, \left(\frac{r_0}{p}\right), \ldots, \left(\frac{r_{\ell-1}}{p}\right)\right)\right)$.

Figure 3.3: The Legendre OPRF scheme

We do not describe the exact command sequence the parties send to $\mathcal{F}_{ABB}$ to compute this function but note that if either party does not properly execute the sequence, then $\mathcal{F}_{ABB}$ will ignore the Output messages. In essence, the sequence of commands works as follows:

1. Send sequence of Input messages to load the user and server input into MPC.[1]
2. For each $i \in \{0, \dots, \ell - 1\}$: Recreate the s_i values from the two shares using Add
3. For each $i \in \{0, \dots, \ell - 1\}$: Compute the s_i^2 values using Multiply
4. Compute $H_1(x) + k$ using Add
5. For each $i \in \{0, \dots, \ell - 1\}$: Compute $s_i^2 \cdot (H_1(x) + k)$ using Multiply
6. For each $i \in \{0, \dots, \ell - 1\}$: For $L[i]$ iterations, add s_i^2 to the result of the above step using Add.
7. For each $i \in \{0, \dots, \ell - 1\}$: Add m_i to the result of the above step to mask the output value using Add
8. Send ℓ many Output messages to return the values $m_i + s_i^2(H_1(x) + k + L[i])$.

At the end of this sequence of operations, we make the user abort if it does not get all the ℓ output values. Otherwise, if all the commands run successfully, and the adversary does not prevent the output of some values, we can verify that both parties at the end of the execution will learn:

$$(s_0^s + s_0^u)^2\,(k + H_1(x) + L[0]) + m_0, \dots, (s_{\ell-1}^s + s_{\ell-1}^u)^2\,(k + H_1(x) + L[\ell - 1]) + m_{\ell-1}$$

where only the user knows the values $m_0, \dots, m_{\ell-1}$.

3.2 Insecurity of the Legendre OPRF

In this section, we will analyse the security of the Legendre OPRF, as introduced in Fig. 3.3, using the ideal functionality $\mathcal{F}_{OPRF}$ introduced in Section 3.1.1.

It can be shown that the Legendre OPRF, as described above, is not secure according to the ideal functionality $\mathcal{F}_{OPRF}$ in the universal composability framework. This is because for a scheme to be secure according to $\mathcal{F}_{OPRF}$, the notion requires that the simulator can tell when two inputs to the random oracle H_2 were generated using the same key, which is not the case for the Legendre OPRF.

[1]Note, that the user inputs $H_1(x)$ and not x into the MPC functionality

We sketch a proof to show the insecurity in two parts. First, we show that with the Legendre OPRF, the simulator cannot tell when the same key was used for two evaluations. Then, we will show why this property is necessary for the security notion.

For some list of random indexes L, let

$$\mathrm{PRF}_k(x) = \left(\frac{H_1(x) + k + L[0]}{p}\right), \dots, \left(\frac{H_1(x) + k + L[\ell - 1]}{p}\right)$$

We then have that the input to the random oracle H_2 for the Legendre OPRF is some pair $(x, \mathrm{PRF}_k(x))$.

Indistinguishability of Hash Inputs We show that under the DSLS assumption, for randomly chosen x, x', k, k', any polynomial time simulator, which controls the H_1 random oracle, cannot distinguish between the pairs $((x, \mathrm{PRF}_k(x)), (x', \mathrm{PRF}_k(x')))$ and $((x, \mathrm{PRF}_k(x)), (x', \mathrm{PRF}_{k'}(x')))$ with non-negligible advantage. Concretely, this means that it cannot tell when the same key was used for two evaluations.

Proof sketch: Assume such a simulator $\mathcal{B}$ exists, we construct a polynomial-time adversary against the DSLS problem.
The adversary begins by choosing random elements x and x'. It then queries the random oracle, as set by $\mathcal{B}$, for the values $H_1(x)$ and $H_1(x')$. Then, the adversary uses its DSLS oracle $\mathcal{O}$ to query the values $H_1(x) + L[0], \dots, H_1(x) + L[\ell - 1]$ and $H_1(x') + L[0], \dots, H_1(x') + L[\ell - 1]$. Given the oracle responses, it passes the two elements

$$(x, \mathcal{O}(H_1(x) + L[0]), \dots, \mathcal{O}(H_1(x) + L[\ell - 1]))$$
$$(x', \mathcal{O}(H_1(x') + L[0]), \dots, \mathcal{O}(H_1(x') + L[\ell - 1]))$$

to $\mathcal{B}$ and outputs the value that $\mathcal{B}$ outputs.

In case the adversary is interacting with a Legendre Oracle, we note that this is exactly the first case that $\mathcal{B}$ distinguishes against since the two items will be $((x, \mathrm{PRF}_k(x)), (x', \mathrm{PRF}_k(x')))$ for some key k.

On the other hand, if the oracle $\mathcal{B}$ is interacting with is a random oracle, then the two sequences will, with high probability, appear to be outputs of the PRF for different keys, which simulates the second case for $\mathcal{B}$. To formally show this, the probability of any overlap in the sequences needs to be shown to be sufficiently small and then, using the DSLS assumption, it can be shown that the two random sequences are indistinguishable from PRF outputs for two different keys.

Thus, the advantage of the adversary in the DSLS game is non-negligible if $\mathcal{B}$ has non-negligible advantage. Therefore, we conclude that the advantage of

$\mathcal{B}$ in distinguishing between the two sequences under the DSLS assumption is negligible. □

Next, we sketch a proof as to why, without the simulator being able to distinguish between two such pairs with non-negligible probability, the OPRF cannot be UC secure according to $\mathcal{F}_{OPRF}$.

Proof sketch: We construct an environment as follows: we have two users U_0 and U_1 and a server S. We consider the case where the server is corrupted. The environment chooses two inputs x_0, x_1 and two keys k_0, k_1.
First, the environment queries the hash function four times in a random order to obtain the values $x_{i,j} \leftarrow H_2\left(x_j, PRF_{k_i}(x_j)\right)$ for $i, j \in \{0,1\}$.
In the real world, if the environment then gives a user $U \in \{U_0, U_1\}$ the input x_j and tells the server to use the key k_i in the evaluation, the evaluation will then output $H_2\left(x_j, PRF_{k_i}(x_j)\right)$. Therefore, this must also hold in the simulated world, or the environment can trivially distinguish between the two cases.
Let us consider how the simulator can respond to the initial hash queries. The simulator needs to answer the hash queries with table values F_k from the ideal functionality, as otherwise, with high probability, the users will output different values.

The first option then is for the simulator to set

$$H_2\left(x_j, PRF_{k_0}(x_j)\right) = H_2\left(x_j, PRF_{k_1}(x_j)\right)$$

in which case the environment can recognise with high probability that this is the simulated world as this occurs in the real world with probability $1/2^{\ell}$.
The second option is for the simulator to set the hash outputs for the same input x to be different. The environment now performs two OPRF evaluations using the two users. It randomly assigns x_0 and x_1 as the user inputs and tells the server to use the key k_0 in both evaluations. Since the simulator does not know which input the users receive, for the users to receive the correct value with high probability, the simulator must have set $H_2(x_0, PRF_{k_0}(x_0))$ to be the output of the ideal OPRF functionality $F_k(x_0)$ for some value k and $H_2(x_1, PRF_{k_0}(x_1)) = F_k(x_1)$ for the same k. This is required because the simulator does not learn anything about the user's inputs, so it cannot know which user has received which input. Thus, it cannot consistently use the correct table if they are different.
Since we know, though, that the hash outputs for the same input x differ and the fact that the simulator cannot distinguish between $((x, \mathrm{PRF}_k(x)), (x', \mathrm{PRF}_k(x')))$ and $((x, \mathrm{PRF}_k(x)), (x', \mathrm{PRF}_{k'}(x')))$ with non-negligible probability, the simulator can only blindly assign the values. Thus, the probability that the polynomial-time simulator answered both queries for the same key k with the same table can be bounded as:

$$\Pr\left[H_2\left(x_0, PRF_{k_0}(x_0)\right) = F_k(x_0) \text{ and } H_2\left(x_1, PRF_{k_0}(x_1)\right) = F_k(x_1)\right] \leq \frac{1}{2} + \epsilon$$

where ϵ is negligible. Since if this is not the case the environment can distinguish between the two worlds, the advantage of this environment is non-negligible for any efficient simulator. Therefore showing that the scheme is not UC secure according to $\mathcal{F}_{OPRF}$. □

This problem is avoided by two-hash Diffie-Hellman, one of the OPRFs shown to be secure in this security notion, by embedding a trapdoor into the scheme. In particular, by programming the $H_1(x)$ values in a particular way, the simulator can recover g^k from an input to the second hash function. This allows the simulator to detect in which cases which key was used. As mentioned, though, this is not possible for the Legendre OPRF due to the DSLS assumption. So, the OPRF will not be secure unless one evaluates additional costly checks in the MPC functionality that may reveal information or limit which keys the server can use. We conclude that the scheme outlined above is not secure according to $\mathcal{F}_{OPRF}$ under the decisional SLS assumption when we have a malicious server.

3.3 Correlated OPRF

An alternative UC security notion for OPRFs called *Correlated OPRF* was introduced by Jarecki et al. as a weaker notion, used to show the security of the Diffie-Hellman OPRF with multiplicative blinding [40]. While this is a weaker notion, it still suffices for various applications, such as OPAQUE [39]. In the following, we will first present the Correlated OPRF functionality. Then, we will present a complete proof of security for the Legendre OPRF in the $\mathcal{F}_{\text{ROM,CRS,ABB}}$-Hybrid model.

3.3.1 Ideal functionality

Again, we slightly modify the original correlated OPRF definition by removing the prefixes. That means, upon the user or server sending their input messages for online evaluation, we no longer allow the adversary to make the parties output some value and link a client and server session together. In Section 3.4 we will introduce and prove security with the variant where these prefixes are not removed.

The new definition can be found in Fig. 3.4. The main difference to $\mathcal{F}_{OPRF}$ is that for any two PRF functions F_1, F_2 we now allow them to be correlated on a single value of the adversaries choosing. That means when a new function F' is referenced for the first time, the adversary can provide a list of pairs (F_i, x_i) and the functionality will ensure that $F'(x_i) = F_i(x_i)$, assuming that each function F_i only occurs once in this list. The ideal functionality models this by storing this information as a graph. The instantiated functions are saved in a set of nodes $\mathcal{N}$, and we have a set of edges $\mathcal{E}$ that model the correlations. Each edge between two nodes has a label x, indicating the value on

which the two nodes are correlated. When adding a new node to the graph (i.e., the first time a function is referenced), the CORRELATE function models adding the specified edges to the graph. The practical difference compared to the previous functionality is that a new attack is permitted. In certain cases, a corrupted server can test if a client has previously interacted with the server with a value x. In particular, if the higher-level application allows the server to detect if a client outputs the same output in two interactions, then by answering the two interactions with different tables F_i, F_j where they are correlated on a value x, with $\mathcal{F}_{corOPRF}$ the adversary can with high probability detect if the user's input was the value x in both interactions or not by comparing whether the outputs were the same or not.

3.3.2 Security analysis of the Legendre OPRF

We show that under the DSLS assumption, the Legendre OPRF securely realises $\mathcal{F}_{corOPRF}$ in the $\mathcal{F}_{\text{ROM,CRS,ABB}}$-Hybrid model.

Before we get into the security proof of the Legendre OPRF, we will first prove two supplementary theorems used in the proof.

The first theorem states that the probability of a hash collision in the evaluated sequences is very small. Furthermore, it will also show that the existence of a point P, such that the evaluation of the OPRF at P will reveal information about the evaluation at more than one point, is unlikely.

Theorem 3.1 *Hash Collisions. Assume H_1 behaves like a truly random function mapping bitstrings to $\mathbb{Z}_p$ and let n_{H_1} be the number of values queried of H_1. Let $L = [o_0, o_1, \ldots, o_{\ell-1}]$ be a list of ℓ many random offsets. Consider the probability p_{bad} that there exists x and $x' \neq x$ for which H_1 was queried such that the intersection between the two sequences $[H_1(x) + o_0, \ldots, H_1(x) + o_{\ell-1}]$ and $[H_1(x') + o_0, \ldots, H_1(x') + o_{\ell-1}]$ is not empty.*

Then we have

$$p_{bad} \leq \frac{n_{H_1}^2 \cdot \ell^2}{2 \cdot p}$$

Furthermore, consider the probability $p_{NotGreat}$ that there exists P, x, and $x' \neq x$ such that the intersection between the sequence $[P + o_0, \ldots, P + o_{\ell-1}]$ and $[H_1(x) + o_0, \ldots, H_1(x) + o_{\ell-1}]$ is non empty, and the intersection between the sequences $[P + o_0, \ldots, P + o_{\ell-1}]$ and $[H_1(x') + o_0, \ldots, H_1(x') + o_{\ell-1}]$ is non empty. Then we have,

$$p_{NotGreat} \leq \frac{n_{H_1}^2 \cdot \ell^3}{2 \cdot p}$$

Proof Let us first prove the first statement. Assume that so far there have been $i-1$ hash queries, and consider the probability c_i that in one of the previous $i-1$ distinct queries, one of the evaluated points is equal to a point in

Ideal Correlated OPRF functionality $\mathcal{F}_{corOPRF}$

The OPRF function is parameterised by a public PRF output length ℓ. For every i and x the value $F_i(x)$ is initially undefined. The first time an undefined value $F_i(x)$ is referenced $\mathcal{F}_{corOPRF}$ sets $F_i(x) \leftarrow\$ \{0,1\}^\ell$.

Initialisation:
On message (INIT, *sid*) from party S, if this is the first INIT message for *sid* set $tx = 0$ and send (INIT, *sid*, S) to $\mathcal{A}^*$. From now on use the tag S to denote the unique entity which sent the INIT message for the session identifier *sid*. (Ignore all subsequent INIT messages for *sid*.) Finally, set $\mathcal{N} \leftarrow [S], \mathcal{E} \leftarrow \{\}, \mathcal{G} \leftarrow (\mathcal{N}, \mathcal{E})$

Server Compromise:
On message (COMPROMISE, *sid*) from $\mathcal{A}^*$, declare S as COMPROMISED.
Note: Message (COMPROMISE, *sid*) requires permission from the environment.
//If S is corrupted, then it is declared COMPROMISED as well.

Offline Evaluation:
On (OFFLINEEVAL, *sid* , S^*, x, L) from $P \in \{S, \mathcal{A}^*\}$ do:

- If $P = \mathcal{A}^*$ and $S^* \notin \mathcal{N}$: append S^* to $\mathcal{N}$ and run CORRELATE(S^*, L)
- Ignore message if $P = \mathcal{A}^*$, S not COMPROMISED, and $(S^*, S, x) \in \mathcal{E}$
- Send (OFFLINEEVAL, *sid*, $F_{S^*}(x)$) to P if (i) $P = S$ and $S^* = S$ or (ii) $P = \mathcal{A}^*$ and either $S^* \neq S$ or S COMPROMISED

Online Evaluation:

- On (EVAL, *sid*, *ssid*, S', x) from $P \in \{U, \mathcal{A}^*\}$, send (EVAL, *sid*, *ssid*, P, S') to $\mathcal{A}^*$. Record $\langle ssid, P, x\rangle$
- On (SNDRCOMPLETE, *sid*, *ssid'*) from S, send (SNDRCOMPLETE, *sid*, *ssid'*, S) to $\mathcal{A}^*$, set tx++
- On (RCVCOMPLETE, *sid*, *ssid*, P, S^*, L) from $\mathcal{A}^*$, ignore this message if there is no record $\langle ssid, P, x\rangle$ stored. Else:
  - If $S^* \notin \mathcal{N}$: Append S^* to $\mathcal{N}$, run CORRELATE(S^*, L)
  - If S is not COMPROMISED and ($S^* = S$ $[(S^*, S, x) \in \mathcal{E}$ and $P = \mathcal{A}^*]$):
    If $tx = 0$ ignore this message. Else decrement tx
  - Send $($EVAL$, sid, ssid, F_{S^*}(x))$ to P

CORRELATE (S^*, L):
Reject if list L contains elements $(j, x), (j', x')$ s.t. $j = j'$ and $x \neq x'$.
Else, for all $(j, x) \in L$ s.t. $j \in \mathcal{N}$, add (S^*, j, x) to $\mathcal{E}$ and set $F_{S^*}(x) \leftarrow F_j(x)$

Figure 3.4: The Correlated OPRF functionality $\mathcal{F}_{corOPRF}$. The changes to $\mathcal{F}_{OPRF}$ are highlighted using grey boxes.

the series $H_1(x_i) + o_0, \ldots, H_1(x_i) + o_{\ell-1}$.
Consider the number of points at which the sequence $H_1(x_i)$ can start without causing such a collision. Initially, this is p points, and then in each of the previous $i-1$ iterations, ℓ many points are evaluated. For each of those $(i-1) \cdot \ell$ many points, we effectively remove a set of at most ℓ many possible start-points from not having a collision. Therefore, since we assume $H_1(x_i)$ is chosen uniformly at random it follows that

$$c_i \leq \frac{(i-1) \cdot \ell^2}{p}$$

Let us visualise this for the case where we have $p = 23$, $\ell = 2$ and $L = [0, 4]$. Initially, any point can be the output of our hash function without causing a collision.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

We then hash an item x and find that $H_1(x) = 6$. Thus, we now mark the values 6 and 10 in red, as those are the values we have evaluated.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

$H_1(x) + 0$ (↑ 6) $H_1(x) + 4$ (↑ 10)

Consider some element x' and the possible values for $H_1(x')$. In green, we now mark all the values that would result in a sequence collision. We see that there are ℓ such points for each red element, resulting in a total of less than ℓ^2 many such points, with each further evaluation adding at most another ℓ^2 more.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Therefore, consider the probability p_{bad} that after n_{H_1} many queries there is a collision. Using an union bound we have:

$$p_{bad} \leq \sum_{i=1}^{n_{H_1}} c_i \leq \sum_{i=1}^{n_{H_1}} \frac{(i-1) \cdot \ell^2}{p} \leq \frac{n_{H_1}^2 \ell^2}{2 \cdot p}$$

from which the first statement follows.

For the second statement, all the points we previously marked as green would be potential values of P, such that the intersection between the sequence starting at $H_1(x)$ and P have some overlap. Let us now consider all the values $H_1(x')$ in blue, such that the evaluation at one of these green points P would reveal something about the evaluation at x'. Notice that for

each point P, there are ℓ many such points $H_1(x')$, in our example case P and $P-4$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Therefore, if we now were to hash a second element x', the probability that x' and x are such that in no evaluation we can learn information about both x and x' is bounded by

$$\frac{\ell^3}{p}$$

Furthermore, after seeing $n_{H_1} - 1$ hash queries, the probability that the next hash query does not create such a point is

$$\frac{(n_{H_1} - 1) \cdot \ell^3}{p}$$

The result then again follows directly from a union bound over the n_{H_1} hash queries. □

Next, we will show that the probability that there exist two keys $k, k' \neq k$ such that the sequences starting at them are equal is small. From this, we can then conclude that the probability that there exists x, k, k' such that the sequences starting from $H_1(x) + k$ and $H_1(x) + k'$ are equal is also small because $H_1(x) + k$ and $H_1(x) + k'$ are valid keys that could be used directly to find a collision. This implies that it is hard for a malicious server to find two such keys.

Theorem 3.2 *Sequence Collisions. Let p be a prime and $I = [o_0, o_1, \ldots, o_{\ell-1}]$ be a list with ℓ many random elements. Then for ℓ chosen sufficiently large ($\ell \geq c \cdot log_2(p)$ for some $c \approx 2.5$ that depends on p) the probability that two keys k, k' exist such that*

$$\left(\frac{k + o_0}{p}\right), \left(\frac{k + o_1}{p}\right), \ldots, \left(\frac{k + o_{\ell-1}}{p}\right) = \left(\frac{k' + o_0}{p}\right), \left(\frac{k' + o_1}{p}\right), \ldots, \left(\frac{k' + o_{\ell-1}}{p}\right)$$

is bounded by $\frac{1}{2\sqrt{p}}$

Proof We use Lemma 2.3 to prove this theorem. For the Legendre OPRF we instantiate the lemma with $k = 2$ (k in the lemma represents what power residue we are using, whereas K is used for the key) and $a = 0$ (a in the lemma represents the difference between the two values since we want them to be equal we set $a = 0$). Then the lemma states that for a prime p and any $K \neq K'$, that if $I_{K,K'}$ is the set of indices i such that $\left(\frac{K+i}{p}\right) = \left(\frac{K'+i}{p}\right)$, the cardinality of this set $I_{K,K'}$ is bounded as follows:

$$\frac{p}{2} - \sqrt{p} - 1 \leq |I_{K,K'}| \leq \frac{p}{2} + \sqrt{p} + 2$$

Thus, let $K, K' \neq K$ be arbitrary and I be a uniformly random set of ℓ offsets in $\mathbb{Z}_p$. Then, the probability that the PRF sequences for K and K' are equal is the probability that on a random draw of ℓ elements from $\mathbb{Z}_p$ all are in $I_{K,K'}$. If we denote the binomial distribution as $bin(n, p)$, then the probability that the sequences are the same is bounded by

$$\Pr\left[bin\left(\ell, \frac{\frac{p}{2} + \sqrt{p} + 2}{p}\right) = \ell\right]$$

Therefore, we have

$$\Pr[\text{sequences are equal}] \leq \left(\frac{1}{2} + \frac{1}{\sqrt{p}} + \frac{2}{p}\right)^{\ell}$$

We now want to analyse what requirements exist for c such that when $\ell = c \cdot log_2(p)$ the probability the sequences are equal is bounded by $\frac{1}{p^2 \cdot \sqrt{p}}$. Once we have such a c the desired statement follows directly by applying a union bound over the $p^2/2$ many key pairs.

What we require is that

$$\frac{1}{p^2 \cdot \sqrt{p}} \geq \left(\frac{1}{2} + \frac{1}{\sqrt{p}} + \frac{2}{p}\right)^{c \cdot \log_2(p)} = p^{c \cdot \log_2\left(\frac{1}{2} + \frac{1}{\sqrt{p}} + \frac{2}{p}\right)}$$

Which is equivalent to

$$p^{-2.5} \geq p^{c \cdot \log_2\left(\frac{1}{2} + \frac{1}{\sqrt{p}} + \frac{2}{p}\right)}$$

we can see that this holds if $-2.5 \geq c \cdot \log_2\left(\frac{1}{2} + \frac{1}{\sqrt{p}} + \frac{2}{p}\right)$, or equivalently if

$$\begin{aligned} c &\geq \frac{-2.5}{\log_2\left(\frac{1}{2} + \frac{1}{\sqrt{p}} + \frac{2}{p}\right)} \\ &\geq \frac{2.5}{\log_2\left(\frac{2p}{p+2\sqrt{p}+4}\right)} \end{aligned}$$

Given that the value c fulfils this requirement, a union bound over the $\frac{p^2}{2}$ possible key pairs yields the desired statement, since

$$\frac{p^2}{2} \cdot p^{-2.5} = \frac{\sqrt{p}}{2}$$

with which we conclude the proof. □

We also provide sample numbers for the required evaluation length for different prime sizes. For 64-bit primes we can choose $\ell = 161$, for 128-bit primes we can choose $\ell = 321$, and for 256-bit primes we can choose $\ell = 641$.

We now prove the security of the Legendre OPRF. We assume malicious security with adaptive compromise. We do not make any assumptions on our communication channels (such as them being authenticated). For a prime p, where we have ℓ such that it fulfils the bound given by Theorem 3.2, we get approximately $\log_2(\sqrt{p})$ bits of security. In particular, the advantage of a polynomial-time environment in distinguishing between the simulated world and the ideal world is bounded by

$$\frac{n_{H_1}^2 \cdot \ell^3}{p} + \frac{2 \cdot n \cdot \ell}{p} + \frac{1}{2 \cdot \sqrt{p}} + \epsilon_{DSLS} + \frac{n_{H_2}}{2^\ell}$$

where n_{H_1} and n_{H_2} are the number of hash queries made to H_1 and H_2 respectively, n is the number of protocol evaluations that are run, ℓ is our output length, and ϵ_{DSLS} is the advantage of a polynomial-time adversary in the decisional SLS game.

Theorem 3.3 *Security of Legendre OPRF. The Legendre OPRF protocol of Fig. 3.3 securely realises the correlated OPRF functionality $\mathcal{F}_{corOPRF}$ in the $\mathcal{F}_{ROM,CRS,ABB}$-Hybrid model under the decision Shifted Legendre Symbol assumption.*

Proof We present a simulator Sim that for any efficient environment $\mathcal{Z}$ generates a view that is indistinguishable from $\mathcal{Z}$'s interaction with the real world where the parties run the Legendre OPRF protocol of Fig. 3.3. Without loss of generality, we assume that the adversary $\mathcal{A}^*$ is the dummy adversary [15], who does nothing other than pass along all its messages to and from $\mathcal{Z}$. The simulator is shown in Fig. 3.4. We assume that the server identifier is some unique value such that if we choose some random k' it will be unequal to the server identifier.

Proof Sketch: We begin by outlining the main ideas of the proof and will then present the exact details by describing a sequence of game hops.

The simulation strategy can be summarised as follows:

- Both parties honest: In this case Sim just needs to simulate the messages sent from $\mathcal{F}_{ABB}$ in the real world. Since the arithmetic black box masks the output values, the simulator can just choose random values and use those as the Output messages. Like this, by properly timing the RcvComplete message, the view of the environment is unchanged between the real and the simulated world for this case.
- Honest server: In case the server is honest but interacting with a malicious user, Sim chooses a key k on behalf of the server. Whenever

Simulator Sim (sid, p, H_1, H_2)

The simulator obtains as input a session identifier sid indicating which $\mathcal{F}_{corOPRF}$ instance it communicates with, the public parameters p, as well as the description of the hash functions $H_1 : \{0,1\}^* \to \mathbb{Z}_p$, $H_2 : \{0,1\}^* \times \{0,1\}^\ell \to \{0,1\}^\ell$.
By $\mathrm{PRF}_k^{H_1}(x)$ we mean the sequence of ℓ Legendre symbol evaluations with key k, value $H_1(x)$, and the randomly chosen list $L \in \mathbb{Z}_p^\ell$. So $\left(\frac{H_1(x)+k+L[0]}{p}\right), \ldots, \left(\frac{H_1(x)+k+L[\ell-1]}{p}\right)$

The simulator initially sets $T_k \leftarrow []$ and $T_{seen} \leftarrow []$. It chooses ℓ many random values in $\mathbb{Z}_p$ and stores them in L. //T_k stores the keys we know of, T_{seen} stores pairs $(x, \mathrm{PRF}_k^{H_1}(x), k')$ for some unknown k. The simulator uses L to simulate $\mathcal{F}_{CRS}$.

1. On $(\textsc{Init}, sid, S)$ from $\mathcal{F}_{corOPRF}$, pick $k \leftarrow\$ \mathbb{Z}_p$. Set $\mathcal{N}_{\mathrm{SIM}} \leftarrow [S]$ and add k to T_k. From now on when we reference k we mean this key. //Honest server, so we imitate it.
2. On $(\textsc{Compromise}, sid)$ from $\mathcal{A}^*$, send $(\textsc{Compromise}, sid)$ to $\mathcal{F}_{corOPRF}$ and send k to $\mathcal{A}^*$. Record that S is compromised.
3. On $(\textsc{Eval}, sid, ssid, U, S')$ from $\mathcal{F}_{corOPRF}$, record $\langle U, ssid\rangle$.
4. On $(\textsc{SndrComplete}, sid, ssid', S)$ from $\mathcal{F}_{corOPRF}$, record $\langle S, ssid'\rangle$.
5. On messages from user U sending input to $\mathcal{F}_{ABB}$, $(\textsc{Init}, ssid, \mathbb{F})$ followed by a series of $\textsc{Input}$ commands, store all the provided input as z and store $\langle sid, ssid, z, U\rangle$. //Malicious user's input to MPC.
6. On messages from $P \in \{S, \mathcal{A}^*\}$ sending input to $\mathcal{F}_{ABB}$, $(\textsc{Init}, ssid, \mathbb{F})$ followed by a series of $\textsc{Input}$ commands: If the input matches the form of first containing a key k' and then ℓ many values s_i^S, then add k' to T_k. Store all the provided inputs as y and store $\langle sid, ssid, y, P\rangle$. //Malicious server's input to MPC.
7. [Both Honest] Upon having stored $\langle U, ssid\rangle$ and $\langle S, ssid\rangle$ for some $ssid$: Choose ℓ many random elements $r_0, \ldots, r_{\ell-1} \leftarrow\$ \mathbb{Z}_p$ and send a series of ℓ many $(\textsc{Output}, ssid, r_i)$ messages to $\mathcal{A}^*$. If the adversary responds with $\textsc{Deliver}$ for all of them, send $(\textsc{RcvComplete}, sid, ssid, U, S)$ to $\mathcal{F}_{corOPRF}$.
8. [Honest Server]: Upon having stored $\langle S, ssid\rangle$ and $\langle sid, ssid, z, U\rangle$, pick ℓ many values $s_0^S, \ldots, s_{\ell-1}^S \leftarrow\$ \mathbb{Z}_p$ and execute the code of $\mathcal{F}_{ABB}$ just like $\mathcal{F}_{ABB}$ would for the user and a server participating honestly with key k and random masks $s_0^S, \ldots, s_{\ell-1}^S$. //That means if the user sends the correct sequence of commands, compute the ABB function F and then send the ℓ many output messages to $\mathcal{A}^*$. If the user sends some other sequence of commands, then just run as far as the execution with the server would.

9. [Honest User] Upon having stored records $\langle U, ssid \rangle$ and $\langle sid, ssid, y, P \rangle$: Choose a random $x \in \mathbb{Z}_p$, ℓ masking values $m_0, \ldots, m_{\ell-1} \leftarrow\$ \mathbb{Z}_p$ and ℓ s values $s_0^U, \ldots, s_{\ell-1}^U \leftarrow\$ \mathbb{Z}_p$. Then execute the code of $\mathcal{F}_{ABB}$ just like $\mathcal{F}_{ABB}$ would for P with an user playing honestly with input $H_1(x)$ and values $m_0, \ldots, m_{\ell-1}$ and $s_0^U, \ldots, s_{\ell-1}^U$. If the execution runs successfully, and $\mathcal{A}^*$ says DELIVER for all the output messages, then recover the key k^* that P used as input in this session and do:
   - For each item (x, y, k') stored in T_{seen}, if $y = \mathrm{PRF}_{k^*}^{H_1}(x)$, add (k', x) to a new list L.
   - Send $(\text{RCVCOMPLETE}, sid, ssid, U, k^*, L)$ to $\mathcal{F}_{corOPRF}$
10. [Both Malicious] If have stored $\langle sid, ssid, z, U \rangle$ and $\langle sid, ssid, y, P \rangle$ execute the code of $\mathcal{F}_{ABB}$ just like $\mathcal{F}_{ABB}$ would for a user participating with inputs z and a server participating with inputs y.
11. On fresh hash query $H_1(x)$: Set $H_1(x) \leftarrow\$ \mathbb{Z}_p$.
12. On fresh hash query $H_2(x, y)$:
    - If $y = \mathrm{PRF}_k^{H_1}(x)$: pick some new and unique $ssid$ and send $(\text{EVAL}, sid, ssid, S, x)$ and $(\text{RCVCOMPLETE}, sid, ssid, \mathcal{A}^*, S, [])$ to $\mathcal{F}_{corOPRF}$. If $\mathcal{F}_{corOPRF}$ replies $(\text{EVAL}, sid, ssid, r)$ set $H_2(x, y) \leftarrow r$, otherwise abort. //Evaluation on the key we are simulating the server with.
    - If $y = \mathrm{PRF}_{k^*}^{H_1}(x)$ for some k^* stored in T_k: For each item (x', y', k') stored in T_{seen} if $y' = \mathrm{PRF}_{k^*}^{H_1}(x')$ add (k', x') to a new list L. Return the output of $(\text{OFFLINEEVAL}, sid, k^*, x, L)$. Abort if two such k^* exist. //In this case, an evaluation is happening on a value for a key we already know.
    - Otherwise, select some new k^*, add (x, y, k^*) to T_{seen}, send $(\text{OFFLINEEVAL}, sid, k^*, x, [])$ to $\mathcal{F}_{corOPRF}$ and on response $(\text{OFFLINEEVAL}, sid, r)$, set $H_2(x, y) \leftarrow r$
13. On message (VALUE, sid) intended for $\mathcal{F}_{CRS}$, return L.

Figure 3.4: The simulator that shows that the Legendre OPRF UC-realises $\mathcal{F}_{corOPRF}$.

interacting with a malicious user U controlled by $\mathcal{A}^*$, the simulator executes $\mathcal{F}_{ABB}$ just as $\mathcal{F}_{ABB}$ would while playing the role of the server S in it by using k and some random values s_i to produce the messages. Like this, the view of the user is identical in the simulated and the real world. Furthermore, the simulator uses the outer hash function H_2 to ensure consistency with the first case. It does so by programming the values that arise from the PRF evaluations with the key k to be equal to the output of the ideal functionality table for the honest server. Important here is to verify that the amount of correct PRF output values the user receives is bounded by the number of times the server (and by extension the simulator) participate in the MPC protocol, which is where we utilise Theorem 3.1, and the DSLS assumption.

- Malicious server: The case in which we have a malicious server is the most complex. In particular, to maintain consistency, we need to answer hash function evaluations "correctly" without knowing for what key the PRF was evaluated. This is the case in which we then require the correlations in our ideal functionality. To be exact, the simulator Sim , for each hash query where it does not know what key was used, creates a new "simulated" key, which it uses to read a table in the ideal functionality that will only be read for this value. Then, if an honest user performs an evaluation and the server uses some key k, the simulator checks all the previously used "simulated" keys to see in which ones the input could have been generated by the PRF evaluation with k. The simulator then correlates the functionality table of this key k with all of those values. Like this, the simulator can maintain the consistency of the simulation and ensure the users receive the right output.

  It is also in this case where we use Theorem 3.2 to make sure that when programming the hash outputs, we only need to consider a single possible key.

Game hops We now show a sequence of hybrid experiments $\mathbf{G}_0, \ldots, \mathbf{G}_{14}$ where, starting from the real-world execution, we make small incremental changes until we reach the ideal-world execution with the above simulator. We write $\Pr[\mathbf{G}_i]$ as the probability that the environment outputs 1 in game $\mathbf{G}_i$.

Game $\mathbf{G}_0$: This is the real world execution of $\textsc{Exec}_{LegendreOPRF,\mathcal{A}^*,\mathcal{Z}}$

Game $\mathbf{G}_1$: In this game we move everything the protocol parties do to the simulator, which executes all parties. Additionally, we add an ideal functionality that does nothing but forward every input it gets to the simulator. We also add dummy parties that forward input they get from $\mathcal{Z}$ to the functionality to prevent the environment from detecting

the difference. Furthermore, we modify the functionality with dummy interfaces that allow the simulator to let any party produce any output chosen by the simulator. Since these are only syntactical changes, we have

$$\Pr[\mathbf{G}_1] = \Pr[\mathbf{G}_0]$$

Game $\mathbf{G}_2$: We now modify the simulator to abort on any hash collisions or near-collisions of H_1. That means, we abort in case values x and $x' \neq x$ have been queried such that a value S exists where $H_1(x) + L[i] = S + L[j]$ and $H_1(x') + L[i'] = S + L[j']$ for any $i, i', j, j' \in [0, \ldots, \ell - 1]$. If we let n_{H_1} be the number of times hash function H_1 is queried, then we know from the second statement of Theorem 3.1 that the probability of this occurring is bounded by $\frac{n_{H_1}^2 \cdot \ell^3}{2 \cdot p}$. If this event does not occur, then the view of the environment is identical to the previous game. Thus, applying the difference lemma, we get

$$|\Pr[\mathbf{G}_2] - \Pr[\mathbf{G}_1]| \leq \frac{n_{H_1}^2 \cdot \ell^3}{2 \cdot p}$$

Game $\mathbf{G}_3$: We now augment the functionality with tables $F_i(x)$, which are initially uninitialised and on the first reference, are set to a random element in $\{0,1\}^\ell$. Furthermore, we add the OfflineEval interface. The OfflineEval interface, given a message $(\text{OfflineEval}, sid, S^*, x)$ from an entity $P \in \{\mathcal{A}^*, S\}$, responds with $(\text{OfflineEval}, sid, F_{S^*}(x))$ if i) $P = S$ and $S^* = S$ or ii) $P = \mathcal{A}^*$ and $S^* \neq S$ or S is Compromised (although there is no way to mark S as compromised yet). Note that for now, we assume that in case a server is given the OfflineEval command as an input, this is still forwarded to the simulator and then answered by the simulator. For now, the functionality still only forwards the input instead of answering directly to the server.

Since currently no one uses these functionality tables or the new interfaces, we have that

$$\Pr[\mathbf{G}_3] = \Pr[\mathbf{G}_2]$$

Game $\mathbf{G}_4$: The simulator now uses the OfflineEval interface to answer any H_2 hash queries. On any hash query $H_2(x, y)$ instead of answering with a random value, the simulator now instead answers the request by choosing a brand new key k' and responding with $r = F_{k'}(x)$. The simulator stores all such pairs (x, y, k') in a table T_{seen}. Importantly, we assume that the server identifier S is such that it is not an element of $\mathbb{Z}_p$, and we pick our random k' to be distinct from S as well.
Since the hash output is still chosen uniformly at random (the tables

are instantiated with random values), the view of the environment is the same as it was in the previous game, thus we get

$$\Pr[\mathbf{G}_4] = \Pr[\mathbf{G}_3]$$

Game $\mathbf{G}_5$: For each server evaluation with some key k', the simulator now stores k' in a table T_k. In particular, it does this in the case of an honest server where it stores the key k it uses to simulate the server, in case of a malicious server being told to play a round with some key, or when $\mathcal{A}^*$ passes some pair (k', s) as its own input to $\mathcal{F}_{ABB}$. Furthermore, when simulating the honest server with some key k, on a hash query $H_2(x, y)$ where $y = \mathrm{PRF}_k^{H_1}(x)$ instead of answering it with $F_{k'}(x)$ for some new k' the simulator now answers it with $F_k(x)$, it continues to use the OfflineEval interface to learn the table value.
The view of the environment remains unchanged by this, as the output values are still set randomly. Therefore,

$$\Pr[\mathbf{G}_5] = \Pr[\mathbf{G}_4]$$

Game $\mathbf{G}_6$: We now add correlations to our ideal functionality. On initialisation, a node-set $\mathcal{N}$ is set to the set containing only the server S, and an empty edge set $\mathcal{E}$ is initialised. Another argument, L, is added to the OfflineEval interface. If we have that $P = \mathcal{A}^*$ and $S^* \notin \mathcal{N}$ on a OfflineEval query S^* is added to the node set and the Correlate function as described in $\mathcal{F}_{corOPRF}$ is run. That means the functionality rejects the message if the list L contains (i, x) and (i', x') such that $i = i'$ and $x \neq x'$. Otherwise, for all (i, x) in the list L, it adds (S^*, i, x) to the edge set, and sets the functionality table for S^* on the value x to be equal to $F_i(x)$, so it sets $F_{S^*}(x) \leftarrow F_i(x)$. Additionally, there now is an additional check that if S^* and S are correlated on x, then $\mathcal{A}^*$ can no longer use the OfflineEval interface to evaluate that point unless S is compromised. The environment's view remains the same since the simulator does not use these correlations to answer any queries. Thus,

$$\Pr[\mathbf{G}_6] = \Pr[\mathbf{G}_5]$$

Game $\mathbf{G}_7$: We now add the the Compromise interface to the ideal functionality. This operates as described in $\mathcal{F}_{corOPRF}$. When receiving a server compromise message, the simulator now forwards the compromise message to the ideal functionality in addition to revealing k to $\mathcal{A}^*$.
Since whether a server is compromised or not has no effect on the ideal functionality because we never call OfflineEval with the server identity, we get that

$$\Pr[\mathbf{G}_7] = \Pr[\mathbf{G}_6]$$

Game $\mathbf{G}_8$: We now modify our simulator such that whenever it simulates $\mathcal{F}_{ABB}$ for two parties where at least one of them is honest, it now aborts if it occurs that one of the s_i values is zero. Note that for each of those values this will occur with probability $1/p$ since one of the parties is honest. If we let n be the number of evaluations of the Legendre OPRF, then applying the union bound we have

$$|\Pr[\mathbf{G}_8] - \Pr[\mathbf{G}_7]| \leq \frac{n \cdot \ell}{p}$$

Game $\mathbf{G}_9$: In case of a hash function evaluation (x, y) where $y = \mathrm{PRF}_k^{H_1}(x)$ for the key k which we are using to simulate the server, we now respond with $F_S(x)$ to the hash query instead of $F_k(x)$. To do this, we add the OnlineEvaluation interface to the ideal functionality. We add this exactly as it was in $\mathcal{F}_{corOPRF}$, with the only change being that we allow the simulator to also send the SndrComplete message on behalf of the server. We assume that on a SndrComplete message from the server the ideal functionality increments the counter while also continuing to forward the message to the simulator. Similarly, for an Eval message, we assume that the ideal functionality forwards the full message to the simulator. To answer a hash query $H_2(x, y)$ where $y = \mathrm{PRF}_k^{H_1}(x)$ the simulator sends a (Eval, $sid, ssid, S, x$) and a (RcvComplete, $sid, ssid,$ Sim$, S, []$) message to the ideal functionality in rapid succession for some uniquely chosen $ssid$. It aborts if it does not receive a response to the RcvComplete message. If it receives a response (Eval$, sid, ssid, r$) it sets $H_2(x, y) \leftarrow r$

The provision that the simulator is allowed to send the SndrComplete message is required in case of an OfflineEval input to the server. This is still answered by the simulator, who will, before sending the above two messages to the ideal functionality, first send a SndrComplete message. We will remove this in a later game hop.

Let ϵ be the advantage of the environment in distinguishing between $\mathbf{G}_9$ and $\mathbf{G}_8$. We will perform a reduction showing that if ϵ is not negligible, then we can construct an efficient adversary against the DSLS problem.

Let us first analyse how the view of the environment might have changed due to these modifications. First, for OfflineEval server inputs, the outputs are always answered, so all that has changed is that a different random table is used. This difference is not detectable by the environment. Secondly, if the server is marked compromised, the simulator will always respond to hash queries and the output values will again be set consistently. Therefore, the only case in which

a difference is detectable to the environment is when the ideal functionality ignores the RcvComplete message sent from the simulator. This can occur if the server is not compromised and the number of hash queries for values corresponding to the PRF evaluated at the key k the simulator uses to simulate the honest server is smaller than the number of SndrComplete messages sent. Concretely, this can occur in case the environment (or a user) makes a hash query $H_2(x, \mathrm{PRF}_k^{H_1}(x))$ where the simulator did not evaluate $\mathrm{PRF}_k^{H_1}(x)$ or any of the Legendre symbols that occur in that PRF during the executions of $\mathcal{F}_{ABB}$.

We will now perform our reduction to show that this probability is small. Consider an environment $\mathcal{C}$ that can distinguish between $\mathbf{G}_9$ and $\mathbf{G}_8$. As described above, that means that in $\mathbf{G}_9$ $\mathcal{C}$ made a hash query $H_2(x, \mathrm{PRF}_k^{H_1}(x))$, where the simulator did not evaluate any of the bits of $\mathrm{PRF}_k^{H_1}(x)$, while the server was not compromised. We will construct an adversary $\mathcal{B}$ against the DSLS problem.

We consider a separate game-hop from the above sequence. In this game, the simulation will be aborted once the key k is revealed (so the server is Compromised). But we know that if $\mathcal{C}$ can distinguish between $\mathbf{G}_9$ and $\mathbf{G}_8$ that $\mathcal{C}$ then makes such a hash query before the server is Compromised. Thus, we will use the fact that $\mathcal{C}$ can make such a hash query to build a distinguisher for the DSLS game.

Let us consider the game $\mathbf{G}_9{}'$, which we modify as follows: The simulator no longer picks a key k for the server. Instead, it has access to a truly random oracle $\mathcal{O}_R$.
Whenever the simulator needs to evaluate $\left(\frac{k+H_1(x)+L[i]}{p}\right)$ it instead just queries its own oracle with the value $\mathcal{O}_R(H_1(x)+L[i])$. The simulator aborts on a message Compromise that would require revealing the key k. Furthermore, whenever the simulator needs to provide some value $s^2(k+H_1(x)+L[i])$, as it might in case of a malicious user, it proceeds as follows: It gets a value $b \leftarrow \mathcal{O}_R(H_1(x)+L[i])$. Then, depending on the bit b, it randomly samples elements e from the set of quadratic residues modulo $\mathbb{Z}_p$ or from the set of non-quadratic residues modulo $\mathbb{Z}_p$. Thus, it finds an element e such that $b = \left(\frac{e}{p}\right)$.

Now let us denote with ζ the probability that the environment makes a hash query that causes the simulation to abort in $\mathbf{G}_9$ and with ζ' the probability that the environment makes such a hash query in $\mathbf{G}_9{}'$. Clearly, we have $|\Pr[\mathbf{G}_9] - \Pr[\mathbf{G}_8]| \leq \zeta$ because if no such query occurs, then the environment's view remains the same.

Let us consider ζ', the probability that the environment makes a hash query, resulting in an abort in $\mathbf{G}_9{}'$. Concretely, this occurring means

that for some x' that was never used as a user input, the adversary can learn $y = \mathcal{O}_R(H_1(x) + L[0]), \ldots, \mathcal{O}_R(H_1(x) + L[\ell - 1])$, where none of those bits was previously evaluated. Recall that due to $\mathbf{G}_2$ we know each evaluation will only give information about the OPRF evaluation at a single point. Therefore, this requires the environment to correctly guess ℓ truly random bits. Let n_{H_2} be the number of queries made to the hash function H_2, from the union bound it then follows that this occurs with probability bounded by $\frac{n_{H_2}}{2^\ell}$. Therefore, we know that

$$\zeta' \leq \frac{n_{H_2}}{2^\ell}$$

We now aim to show that the difference between ζ and ζ' can be bounded by the advantage of an adversary against the DSLS problem.

To do this, we assume the existence of an efficient environment $\mathcal{C}$ such that $\mathcal{C}$ can distinguish between $\mathbf{G}_9$ and $\mathbf{G}_9'$ without compromising the server. Using $\mathcal{C}$ as a subroutine, we construct an efficient adversary $\mathcal{B}$ against the DSLS problem.

$\mathcal{B}$ operates as follows: It runs $\mathcal{C}$ with the simulator in the game $\mathbf{G}_9'$ but relays the simulator queries to the truly random oracle to its own oracle provided by the DSLS game. When adversary $\mathcal{C}$ outputs a bit b, $\mathcal{B}$ also outputs the same b. Clearly, $\mathcal{B}$ has an expected polynomial runtime if $\mathcal{C}$ has.[2] Thus, what remains is to show how the advantage of $\mathcal{C}$ relates to the advantage of $\mathcal{B}$. The essential observation here is that since the value s^2 is chosen uniformly at random and we assume that as long as the parties are honest, they will not be zero ($\mathbf{G}_8$), it encodes a random permutation from all quadratic residues to all quadratic residues in $\mathbb{Z}_p$ and the same for all quadratic non-residues. Thus, given $s^2(H_1(x) + k + L[i])$, the only information one can learn about $(H_1(x) + k + L[i])$ is if it is a quadratic residue or not. Therefore, in the case that $\mathcal{B}$'s oracle is computing the Legendre symbol, $\mathcal{B}$ perfectly mimics how the simulator in $\mathbf{G}_9$ operates, as any information given to $\mathcal{C}$ is the same as if $\mathcal{B}$ was playing the simulator in $\mathbf{G}_9$. If $\mathcal{B}$'s oracle is a truly random function, then it simulates the view of $\mathcal{C}$ in $\mathbf{G}_9'$.

Therefore, we conclude that the advantage of $\mathcal{B}$ in the DSLS problem is equal to that of $\mathcal{C}$ in differentiating between the games. Thus, for any environment that does not compromise the server key, we have the following:

$$|\Pr[\mathbf{G}_9] - \Pr[\mathbf{G}_9']| \leq \epsilon_{DSLS}$$

[2]Finding a random element in $\mathbb{Z}_p$ with a specific quadratic residue value requires on average under two attempts.

which also yields that $|\zeta' - \zeta| \leq \epsilon_{DSLS}$.

Combining our insights, we have $\zeta' \leq \frac{n_{H_2}}{2^\ell}$, $|\zeta' - \zeta| \leq \epsilon_{DSLS}$ and $|\Pr[\mathbf{G}_9] - \Pr[\mathbf{G}_8]| \leq \zeta$. Therefore, we conclude that

$$|\Pr[\mathbf{G}_9] - \Pr[\mathbf{G}_8]| \leq \epsilon_{DSLS} + \frac{n_{H_2}}{2^\ell}$$

Game $\mathbf{G}_{10}$: Next, we modify how a hash function request $H_2(x, y)$ is answered for some $y = \mathrm{PRF}_{k'}^{H_1}(x)$ where $k' \neq k$ and $k' \in T_k$. Instead of choosing some brand new k^* and answering the hash query with the ideal functionality evaluation of that value, the simulator instead proceeds as follows:

- If there exist two such keys k' and k'' such that $y = \mathrm{PRF}_{k'}^{H_1}(x) = \mathrm{PRF}_{k''}^{H_1}(x)$ then abort.
- For all values (x', y', k^*) in T_{seen}, if $y' = \mathrm{PRF}_{k'}^{H_1}(x)$, add $([k^*, x'])$ to a list L.
- Set the hash value to be equal to the OfflineEval evaluation of x for the key k', where L is passed as the correlation list.

The correlate aspect does not modify our output for the evaluated x. (Since if (x, y, k^*) would have already been added to T_{seen} for some k^*, then we would have responded to the hash query already). We can also verify that each k^* will occur at most once in the list L because any k^* will only be used once and thus only occur in T_{seen} once. Additionally, each x' will also occur at most once, as the PRF is deterministic.
So all this change does is that instead of answering the query with the ideal functionality on some new key k^*, we now instead answer it on the evaluation of some known key k'. The only case in which the view of the environment changes is if there exist k', k'' in T_k such that $y = \mathrm{PRF}_{k'}^{H_1}(x) = \mathrm{PRF}_{k''}^{H_1}(x)$. From Theorem 3.2 we can conclude that the probability of this occurring is small. This follows from the fact that if two such keys k and k' and value x would exist, then we could find a collision as defined in Theorem 3.2 by setting $k_1 = H_1(x) + k, k_2 = H_1(x) + k'$. Thus, we have

$$|\Pr[\mathbf{G}_{10}] - \Pr[\mathbf{G}_9]| \leq \frac{1}{2 \cdot \sqrt{p}}$$

Game $\mathbf{G}_{11}$: For an honest user, instead of the simulator giving it its output via the dummy interface, the simulator now uses the OnlineEval interface. To do this, we first modify the ideal functionality to no longer pass the user input x to the simulator. Upon receiving an Eval message, the simulator now instead chooses some brand new x^* and uses

that to simulate the user's role in the MPC protocol. The simulator now stores records $\langle U, ssid\rangle$ and $\langle S, ssid\rangle$ when receiving those from the ideal functionality. Furthermore, if both the user and the server for a *ssid* are honest, the simulator instead directly just chooses $r_0, ..., r_\ell$ and sends those as ABB output messages for *ssid* to $\mathcal{A}^*$, instead of simulating the entire $\mathcal{F}_{ABB}$ computation.

Once the adversary has said DELIVER for the ℓ output messages:

- In case the simulator used the key k for the server in the corresponding *ssid*, then the simulator sends (RCVCOMPLETE, $sid, ssid, U, S, []$) to the ideal functionality.
- Otherwise, if the server of *ssid* participated in the protocol with some key k' and value s then for all values (x', y', k^*) in T_{seen}, if $y' = \mathrm{PRF}^{H_1}_{k'}(x)$ add $([k^*, x'])$ to a list L. Send (RCVCOMPLETE, $sid, ssid, U, k', L$) to the functionality.

The view of the environment is unchanged by this since the randomly chosen masking values perfectly mask the MPC output values from it. Therefore, the environment cannot distinguish between whether the simulated OUTPUT messages from $\mathcal{F}_{ABB}$ result from an execution where the input used is the actual user input or some randomly chosen value. Additionally, the environment cannot tell if the simulator uses randomly chosen values in the OUTPUT messages instead.

Furthermore, we observe that the user's output messages are still set identically. Due to how we set the hash values, they corresponded to exactly the values that the ideal functionality will now directly output to the user. Important to note here is that with how we set the list L we pass to the ideal functionality to add our correlations in both $\mathbf{G}_{11}$ and $\mathbf{G}_{10}$ if the environment evaluates a hash query $H_2(x, \mathrm{PRF}^{H_1}_{k'}(x))$ in advance, the simulator will answer that with $F_{k^*}(x)$, and then include $[k^*, x]$ in L when a user runs the evaluation with a corrupted server playing with the key k'. Therefore,

$$\Pr[\mathbf{G}_{11}] = \Pr[\mathbf{G}_{10}]$$

Game $\mathbf{G}_{12}$: We now modify how an OFFLINEEVAL request is handled in case the server is honest. The ideal functionality no longer forwards the request to the simulator but instead responds directly to the server as it should according to the OFFLINEEVAL interface of $\mathcal{F}_{corOPRF}$. This also means the simulator no longer sends SNDRCOMPLETE messages to the ideal functionality on behalf of the server, so we remove the modification that enabled the simulator to do so. Currently, the way such

a request is responded to is by evaluation $F_S(x)$ by sending an Eval and SndrComplete message, followed by a RcvComplete message. If we instead directly allow the server to use the OfflineEval interface, it will similarly be answered by the same table. Therefore, since the output value is still set to the same table, we have

$$\Pr[\mathbf{G}_{12}] = \Pr[\mathbf{G}_{11}]$$

Game $\mathbf{G}_{13}$: We now remove the abort upon a collision in H_1, first introduced in $\mathbf{G}_2$. Furthermore, we remove the aborts in case one of the s values happens to be 0, introduced in $\mathbf{G}_8$. By the same argument as there we have

$$|\Pr[\mathbf{G}_{13}] - \Pr[\mathbf{G}_{12}]| \leq \frac{n_{H_1}^2 \cdot \ell^3}{2 \cdot p} + \frac{n \cdot \ell}{p}$$

We now describe the world we find ourselves in. The ideal functionality Sim interacts with is exactly $\mathcal{F}_{corOPRF}$, except for the fact that it still possesses the dummy interfaces that let the simulator control the output of the parties. H_1 operates exactly as it would in the ideal world, as it just assigns random values to the requested input. H_2 was modified in multiple steps. First, in game $\mathbf{G}_9$, we set it in case y is the evaluation of $\mathrm{PRF}_k^{H_1}(x)$. This corresponds to the first case of how the simulator sets H_2. The second case in the simulation of H_2 corresponds exactly to what we now have, after the modifications in $\mathbf{G}_{10}$. Finally, by how we set the hash output in all other cases, as defined in $\mathbf{G}_3$, our simulator now simulates H_2 as it would in the ideal world execution.
Furthermore, the responses of the simulator to the Init and Compromise are now also the same. We can also see that the evaluation proceeds exactly as it would in the simulated world. Therefore, we can see that this is exactly equivalent to the ideal world when running with simulator Sim, just that the simulator still receives additional input from the ideal functionality, which it does not use, and the presence of the dummy output interfaces of the ideal functionality, which also are unused.

Game $\mathbf{G}_{14}$: In this game we take away the additional information about the inputs the simulator still gets from the functionality. We also remove the dummy interfaces that allowed the simulator to make any party output whatever the simulator wanted. As the simulator did not use either anymore, the distribution of the experiment does not change when the simulator does not get this information. Thus, we have

$$\Pr[\mathbf{G}_{14}] = \Pr[\mathbf{G}_{13}]$$

With that, our proof is complete and we have shown the security of the Legendre OPRF. □

3.3.3 Strengthening of the notion

There are two ways in which the correlated OPRF security notion could be modified to provide slightly stronger security guarantees without requiring modifications to the scheme and proof strategy.

Firstly, the current version of $\mathcal{F}_{corOPRF}$ allows $\mathcal{A}^*$ to let any honest party P compute $F_i(x)$ where F_i may be correlated with F_S on the point x without decrementing the ticket counter. This was used in the security proof for the multiplicative-blinded two-hash Diffie-Hellman and this weaker notion suffices for OPAQUE. We observe, though, that our simulator will never correlate any function with F_S; thus, using the same scheme, simulator, and proof as above, the security in this slightly modified security notion could be shown.

Secondly, the current functionality is for a single server with a single key. Concretely, this implies that we require domain separation between all our hash functions in case multiple servers run the protocol. As it turns out, this is actually not required in our proof. The fact that for H_1 this does not matter follows from the fact that we do not program that random oracle. For H_2, the key insight is that we assume that no two keys exist such that the PRF evaluation at a point x is the same with both of them. Let us now consider the three cases in which we program this random oracle. The first two will not cause any collisions due to the above-stated property that given some value (x, y), there is a unique key for which those would be equal, so unless two honest servers end up using the same key (which occurs with negligible probability), there will be no collisions in this programming. We then observe that the third case is also independent of what servers or keys we are considering. Due to this, we believe our scheme is also secure in a multi-server and multi-key variant of $\mathcal{F}_{corOPRF}$, where domain separation for hash functions is no longer required.

3.4 Correlated OPRF with Prefixes

The security proof uses a slightly modified version of the correlated OPRF functionality in which we removed the prefixes. We did this because, for the majority of OPRF applications the prefixes are not used. In this section, we will argue how our scheme can be modified to work with the original unmodified definition required for OPAQUE. The role the prefixes play is that if for some *sid* both the server and the user output the same prefix *prfx* after sending their Eval or SndrComplete message then the SndrComplete message can only be used to evaluate the OPRF at the input provided in the Eval message with the corresponding prefix. So, while the adversary can still make the user evaluate a different table, that specific SndrComplete can no longer be used for a different evaluation. In Fig. 3.4 we present the

correlated OPRF security notion with prefixes. One easy way of bridging a protocol from the previous definition to this one is by just making the parties output a random prefix. By doing so, honest users and servers in the real world will, with high probability, never use the same prefixes and the simulator can just use random values in the simulation. The applicability of this is limited, though, because for OPAQUE, one of the higher-level applications that uses these prefixes, the protocol aborts if the user and server receive different prefixes.

Therefore, we will instead use the sub-session identifiers (*ssid*s) passed as inputs to $\mathcal{F}_{ABB}$ as our prefixes. In Fig. 3.5 we show the modified Legendre OPRF scheme that is secure according to the correlated OPRF with prefixes security notion. We note one thing that needs to be monitored carefully is that when a user and the server run the session together with some *ssid*, we want the server to output the *ssid* as a prefix only after the user has done so because in the ideal functionality it is not possible for the user to output a prefix after the server has done so. We can ensure this is the case by having the user output the *ssid* upon receiving it while having the server wait until it has received some output from $\mathcal{F}_{ABB}$. That way, we can enforce a strict ordering on the fact that the user will always output the *ssid* before the server does so.

Finally, we show the scheme's security, as stated by Theorem 3.4. Again, we assume malicious security with adaptive compromise.

Theorem 3.4 *Security of Legendre OPRF. The Legendre OPRF protocol with prefixes of Fig. 3.5 securely realises the correlated OPRF functionality $\mathcal{F}_{corOPRF}$ with prefixes in the $\mathcal{F}_{ROM,CRS,ABB}$-Hybrid model under the DSLS assumption.*

Proof In Fig. 3.4 we present the updated simulator. Instead of restating all the individual game hops, we describe the modifications that need to be made to the sequence of game hops used in the proof of Theorem 3.3.

We add one additional game and slightly modify two of the games. With those modifications, we observe that the rest of the proof works completely analogously.

1. **Game $\mathbf{G}_{9_b}$** The game we additionally add occurs after $\mathbf{G}_9$. In this game, we further augment the OnlineEval interface of the functionality by adding the prefixes as described in Fig. 3.4. Furthermore, when the simulator now sends an Eval request to set the hash output, it chooses some new and unique *ssid* and uses that as the prefix (first case of 12. in the simulator). When simulating the OfflineEval interface on behalf of the server by sending the sequence of three messages, we assume that it again uses some new and unique *ssid* in the Eval message and uses that as the prefix there, and does the same for the SndrComplete where it uses a different fresh *ssid* as the prefix.

Ideal Correlated OPRF functionality $\mathcal{F}_{corOPRF}$ with prefixes

The OPRF function is parameterised by a public PRF output length ℓ. For every i and x the value $F_i(x)$ is initially undefined. The first time an undefined value $F_i(x)$ is referenced $\mathcal{F}_{corOPRF}$ sets $F_i(x) \leftarrow\!\$ \{0,1\}^\ell$.

Initialisation:
On message (INIT, *sid*) from party S, if this is the first INIT message for *sid* set $tx = 0$ and send (INIT, *sid*, S) to $\mathcal{A}^*$. From now on use the tag S to denote the unique entity which sent the INIT message for the session identifier *sid*. (Ignore all subsequent INIT messages for *sid*.) Finally, set $\mathcal{N} \leftarrow [S], \mathcal{E} \leftarrow \{\}, \mathcal{G} \leftarrow (\mathcal{N}, \mathcal{E})$

Server Compromise:
On message (COMPROMISE, *sid*) from $\mathcal{A}^*$, declare S as COMPROMISED.
Note: Message (COMPROMISE, *sid*) requires permission from the environment. //If S is corrupted, then it is declared COMPROMISED as well.

Offline Evaluation:
On (OFFLINEEVAL, *sid* , S^*, x, L) from $P \in \{S, \mathcal{A}^*\}$ do:

- If $P = \mathcal{A}^*$ and $S^* \notin \mathcal{N}$: append S^* to $\mathcal{N}$ and run CORRELATE(S^*, L)
- Ignore message if $P = \mathcal{A}^*$, S not COMPROMISED, and $(S^*, S, x) \in \mathcal{E}$
- Send (OFFLINEEVAL, *sid*, $F_{S^*}(x)$) to P if (i) $P = S$ and $S^* = S$ or (ii) $P = \mathcal{A}^*$ and either $S^* \neq S$ or S COMPROMISED

Online Evaluation:

- On (EVAL, *sid*, *ssid*, S', x) from $P \in \{U, \mathcal{A}^*\}$, send (EVAL, *sid*, *ssid*, P, S') to $\mathcal{A}^*$. On *prfx* from $\mathcal{A}^*$, reject it if *prfx* was used before. Else record $\langle ssid, P, x, prfx, 0 \rangle$ and send (Prefix, *ssid*, *prfx*) to P.
- On (SNDRCOMPLETE, *sid*, *ssid'*) from S, send (SNDRCOMPLETE, *sid*, *ssid'*, S) to $\mathcal{A}^*$. On *prfx'* from $\mathcal{A}^*$ send (Prefix, *ssid*, *prfx'*) to S. If there is a record $\langle ssid, P, x, prfx', 0 \rangle$ s.t. $prfx = prfx'$ change it to $\langle ssid, P, x, prfx', 1 \rangle$, else set tx++
- On (RCVCOMPLETE, *sid*, *ssid*, P, S^*, L) from $\mathcal{A}^*$, ignore this message if there is no record $\langle ssid, P, x, prfx, ok? \rangle$ stored. Else:
  - If $S^* \notin \mathcal{N}$: Append S^* to $\mathcal{N}$, run CORRELATE(S^*, L)
  - If S is not COMPROMISED and ok? = 0 do:
    If ($S^* = S$ or $[(S^*, S, x) \in \mathcal{E}$ and $P = \mathcal{A}^*]$):
    If $tx = 0$ ignore this message. Else decrement tx
  - Send (EVAL, *sid*, *ssid*, $F_{S^*}(x)$) to P

Correlate (S^*, L):
Reject if list L contains elements $(j, x), (j', x')$ s.t. $j = j'$ and $x \neq x'$.
Else, for all $(j, x) \in L$ s.t. $j \in \mathcal{N}$, add (S^*, j, x) to $\mathcal{E}$ and set $F_{S^*}(x) \leftarrow F_j(x)$

Figure 3.4: The Correlated OPRF functionality $\mathcal{F}_{corOPRF}$ with prefixes from [40]. The changes to $\mathcal{F}_{corOPRF}$ without prefixes are highlighted using grey boxes.

Note that in both these cases, only the simulator receives the prefix from the ideal functionality. Therefore, the view of the environment is unchanged by whether that prefix is output. Furthermore, because no prefix is ever repeated, each interaction continues to first increase and decrease the counter.

For the honest server, instead of having the simulator use the dummy interface to make the server output the prefix *ssid*, it now instead sends the prefix as a reply to the SndrComplete message from the ideal functionality. It still does this at the same point in time (once the first output message is delivered). This corresponds to the modifications to cases 7. and 8. in the simulator. We observe that the server's output is the same as before because, in both cases, the ideal functionality will give the server the output *ssid* once the first output value is available. Finally, we note that the *ok?* value of a record will never change in this game because, for the Eval messages, we choose some fresh *ssid*, so we will never have the case that a SndrComplete message will receive a prefix from the simulator that had been used already in an Eval message. Therefore, we have

$$\Pr[\mathbf{G}_{9b}] = \Pr[\mathbf{G}_9]$$

2. **Game $\mathbf{G}_{11}$'** We replace Game $\mathbf{G}_{11}$ with the following game instead: For an honest user, instead of the simulator giving it its output via the dummy interface, the simulator now uses the OnlineEval interface. To do this, we modify the ideal functionality to no longer pass the user input x to the simulator. Upon receiving an Eval message, the simulator now instead chooses some brand new x^* and uses that to simulate the user's role in the ABB protocol. For the prefix, upon receiving the $(\text{Eval}, sid, ssid, U, S')$ message, the simulator responds by sending *ssid* to the ideal functionality.

   When receiving those from the ideal functionality, the simulator now stores records $\langle U, ssid\rangle$ and $\langle S, ssid\rangle$. Furthermore, if both the user and the server for some *ssid* are honest, the simulator instead directly just chooses $r_0, ..., r_\ell$ and sends those as ABB output messages for *ssid* to $\mathcal{A}^*$ instead of simulating the entire $\mathcal{F}_{ABB}$ computation.

   Once the adversary has said Deliver for the ℓ output messages:

Legendre OPRF with prefixes

Public Parameters: Prime number p, output length ℓ. Distribution D for $\mathcal{F}_{CRS}$ that returns ℓ many values from $\mathbb{Z}_p$ chosen uniformly at random.
Function to compute inside $\mathcal{F}_{ABB}$ for some list L known to both parties:

$$F\left(\left(x, s_0^u, \ldots, s_{\ell-1}^u, m_0, \ldots, m_{\ell-1}\right), \left(k, s_0^s, \ldots, s_{\ell-1}^s\right)\right) \to$$
$$\left(s_0^s + s_0^u\right)^2 \left(k + H_1(x) + L[0]\right) + m_0, \ldots, \left(s_{\ell-1}^s + s_{\ell-1}^u\right)^2 \left(k + H_1(x) + L[\ell-1]\right) + m_{\ell-1}$$

Initialisation:
On input (Init, sid): S picks $k \leftarrow\$ \mathbb{Z}_p$ and stores $\langle sid, k\rangle$

Server Compromise:
On $(\text{Compromise}, sid)$, if there is a record $\langle sid, k\rangle$ reveal k to $\mathcal{A}^*$

Offline Evaluation:
On $(\text{OfflineEval}, sid, x)$, send (Value, sid) to $\mathcal{F}_{CRS}$, and receive the list L. Then the server retrieves $\langle sid, k\rangle$ and outputs $\left(\text{OfflineEval}, sid, H_2\left(x, \left(\frac{H_1(x)+k+L[0]}{p}\right), \left(\frac{H_1(x)+k+L[1]}{p}\right), \ldots, \left(\frac{H_1(x)+k+L[\ell-1]}{p}\right)\right)\right)$

Online Evaluation:

- On $(\text{Eval}, sid, ssid, S', x)$, U Outputs $(\text{Prefix}, ssid, ssid)$, U then sends (Value, sid) to $\mathcal{F}_{CRS}$ and receives the list L. U then sends $(\text{Init}, ssid, \mathbb{Z}_p)$ to $\mathcal{F}_{ABB}$. U then samples $2 \cdot \ell$ many elements from $\mathbb{Z}_p$ ($s_0^u, \ldots, s_{\ell-1}^u$ and $m_0, \ldots, m_{\ell-1}$). It then sends a sequence of Input commands followed by a sequence of Add and Multiply commands required to evaluate the function F with list L to $\mathcal{F}_{ABB}$, all including $ssid$. U then stores $\langle x, ssid, m_0 \ldots m_{\ell-1}\rangle$. Finally U sends ℓ many Output messages to $\mathcal{F}_{ABB}$.
- On $(\text{SndrComplete}, sid, ssid')$, S sends (Value, sid) to $\mathcal{F}_{CRS}$, and receives the list L. It then sends $(\text{Init}, ssid', \mathbb{Z}_p)$ to $\mathcal{F}_{ABB}$. S then chooses $s_0^s, \ldots, s_{\ell-1}^s \leftarrow\$ \mathbb{Z}_p$, retrieves the record $\langle sid, k\rangle$ and then sends a sequence of Input commands followed by a sequence of Add and Multiply commands required to evaluate the function F with list L to $\mathcal{F}_{ABB}$, all including $ssid'$. Finally S sends ℓ many Output messages to $\mathcal{F}_{ABB}$.
- Upon receiving the first output from $\mathcal{F}_{ABB}$, S outputs $(\text{Prefix}, ssid, ssid)$.
- Upon having received ℓ many outputs $c_0, \ldots, c_{\ell-1}$ from $\mathcal{F}_{ABB}$ for some subsession identifier $ssid$, U retrieves the stored $\langle x, ssid, m_0 \ldots m_{\ell-1}\rangle$, and computes $r_i = c_i - m_i$. Then U outputs $\left(\text{Eval}, sid, ssid, H_2\left(x, \left(\frac{r_0}{p}\right), \ldots, \left(\frac{r_{\ell-1}}{p}\right)\right)\right)$.

Figure 3.5: The Legendre OPRF scheme with prefixes. Changes to the prefix-less version are marked in grey boxes .

Simulator Sim (sid, p, H_1, H_2)

The simulator obtains as input a session identifier sid indicating which $\mathcal{F}_{corOPRF}$ instance it communicates with, the public parameters p, as well as the description of the hash functions $H_1 : \{0,1\}^* \to \mathbb{Z}_p$, $H_2 : \{0,1\}^* \times \{0,1\}^\ell \to \{0,1\}^\ell$. By $\mathrm{PRF}_k^{H_1}(x)$ we mean the sequence of ℓ Legendre symbol evaluations with key k, value $H_1(x)$, and the randomly chosen list $L \in \mathbb{Z}_p^\ell$. So $\left(\frac{H_1(x)+k+L[0]}{p}\right), \ldots, \left(\frac{H_1(x)+k+L[\ell-1]}{p}\right)$

The simulator initially sets $T_k \leftarrow []$ and $T_{seen} \leftarrow []$. It chooses ℓ many random values in $\mathbb{Z}_p$ and stores them in L. //T_k stores the keys we know of, T_{seen} stores pairs $(x, \mathrm{PRF}_k^{H_1}(x), k')$ for some unknown k. The simulator uses L to simulate $\mathcal{F}_{CRS}$.

1. On $(\textsc{Init}, sid, S)$ from $\mathcal{F}_{corOPRF}$, pick $k \leftarrow\$ \mathbb{Z}_p$. Set $\mathcal{N}_{\text{SIM}} \leftarrow [S]$ and add k to T_k. From now on when we reference k we mean this key. //Honest server, so we imitate it.
2. On $(\textsc{Compromise}, sid)$ from $\mathcal{A}^*$, send $(\textsc{Compromise}, sid)$ to $\mathcal{F}_{corOPRF}$ and send k to $\mathcal{A}^*$. Record that S is compromised.
3. On $(\textsc{Eval}, sid, ssid, U, S')$ from $\mathcal{F}_{corOPRF}$, record $\langle U, ssid\rangle$. Send $ssid$ to $\mathcal{F}_{corOPRF}$ as U's prefix.
4. On $(\textsc{SndrComplete}, sid, ssid', S)$ from $\mathcal{F}_{corOPRF}$, record $\langle S, ssid'\rangle$.
5. On messages from user U sending input to $\mathcal{F}_{ABB}$, $(\textsc{Init}, ssid, \mathbb{F})$ followed by a series of $\textsc{Input}$ commands, store all the provided input as z and store $\langle sid, ssid, z, U\rangle$. //Malicious user's input to MPC.
6. On messages from $P \in \{S, \mathcal{A}^*\}$ sending input to $\mathcal{F}_{ABB}$, $(\textsc{Init}, ssid, \mathbb{F})$ followed by a series of $\textsc{Input}$ commands: If the input matches the form of first containing a key k' and then ℓ many values s_i^S, then add k' to T_k. Store all the provided inputs as y and store $\langle sid, ssid, y, P\rangle$. //Malicious server's input to MPC.
7. [Both Honest] Upon having stored $\langle U, ssid\rangle$ and $\langle S, ssid\rangle$ for some $ssid$: Choose ℓ many random elements $r_0, \ldots, r_{\ell-1} \leftarrow\$ \mathbb{Z}_p$ and send a series of ℓ many $(\textsc{Output}, ssid, r_i)$ messages to $\mathcal{A}^*$. Once the adversary responds with $\textsc{Deliver}$ to one of them, send $ssid$ to $\mathcal{F}_{corOPRF}$ as the server's prefix. If the adversary responds with $\textsc{Deliver}$ for all of them, send $(\textsc{RcvComplete}, sid, ssid, U, S)$ to $\mathcal{F}_{corOPRF}$.

8. [Honest Server]: Upon having stored $\langle S, ssid \rangle$ and $\langle sid, ssid, z, U \rangle$, pick ℓ many values $s_0^S, \ldots, s_{\ell-1}^S \leftarrow\$ \mathbb{Z}_p$ and execute the code of $\mathcal{F}_{ABB}$ just like $\mathcal{F}_{ABB}$ would for the user and a server participating honestly with key k and random masks $s_0^S, \ldots, s_{\ell-1}^S$. Upon receiving DELIVER for the first time for an output message, send $ssid$ to $\mathcal{F}_{corOPRF}$ as the server's prefix. //That means if the user sends the correct sequence of commands, compute the MPC function F and then send the ℓ many output messages to $\mathcal{A}^*$. If the user sends some other sequence of commands, then just run as far as the execution with the server would.

9. [Honest User] Upon having stored records $\langle U, ssid \rangle$ and $\langle sid, ssid, y, P \rangle$: Choose a random $x \in \mathbb{Z}_p$, ℓ masking values $m_0, \ldots, m_{\ell-1} \leftarrow\$ \mathbb{Z}_p$ and ℓ s values $s_0^U, \ldots, s_{\ell-1}^U \leftarrow\$ \mathbb{Z}_p$. Then execute the code of $\mathcal{F}_{ABB}$ just like $\mathcal{F}_{ABB}$ would for P with an user playing honestly with input $H_1(x)$ and values $m_0, \ldots, m_{\ell-1}$ and $s_0^U, \ldots, s_{\ell-1}^U$. If the execution runs successfully, and $\mathcal{A}^*$ says DELIVER for all the output messages, then recover the key k^* that P used as input in this session and do:

   - For each item (x, y, k') stored in T_{seen}, if $y = \text{PRF}_{k^*}^{H_1}(x)$, add (k', x) to a new list L.
   - Send (RCVCOMPLETE, $sid, ssid, U, k^*, L$) to $\mathcal{F}_{corOPRF}$

10. [Both Malicious] If have stored $\langle sid, ssid, z, U \rangle$ and $\langle sid, ssid, y, P \rangle$ execute the code of $\mathcal{F}_{ABB}$ just like $\mathcal{F}_{ABB}$ would for a user participating with inputs z and a server participating with inputs y.

11. On fresh hash query $H_1(x)$: Set $H_1(x) \leftarrow\$ \mathbb{Z}_p$.

12. On fresh hash query $H_2(x, y)$:

   - If $y = \text{PRF}_k^{H_1}(x)$ pick some new and unique $ssid$ and send (EVAL, $sid, ssid, S, x$) to $\mathcal{F}_{corOPRF}$, upon receiving the corresponding EVAL message from $\mathcal{F}_{corOPRF}$ send $ssid$ as prefix. Then send (RCVCOMPLETE, $sid, ssid, \mathcal{A}^*, S, []$) to $\mathcal{F}_{corOPRF}$. If $\mathcal{F}_{corOPRF}$ replies (EVAL, $sid, ssid, r$) set $H_2(x, y) \leftarrow r$, otherwise abort. //Evaluation on the key we are simulating the server with.

- If $y = \text{PRF}^{H_1}_{k^*}(x)$ for some k^* stored in T_k: For each item (x', y', k') stored in T_{seen} if $y' = \text{PRF}^{H_1}_{k^*}(x')$ add (k', x') to a new list L. Return the output of $(\textsc{OfflineEval}, sid, k^*, x, L)$. Abort if two such k^* exist. //In this case, an evaluation is happening on a value for a key we already know.
- Otherwise, select some new k^*, add (x, y, k^*) to T_{seen}, send $(\textsc{OfflineEval}, sid, k^*, x, [])$ to $\mathcal{F}_{corOPRF}$ and on response $(\textsc{OfflineEval}, sid, r)$, set $H_2(x, y) \leftarrow r$

13. On message $(\textsc{Value}, sid)$ intended for $\mathcal{F}_{CRS}$, return L.

Figure 3.4: The simulator that demonstrated that the Legendre OPRF with prefixes UC-realises $\mathcal{F}_{corOPRF}$ with prefixes. Modifications to the variant without prefixes are given in grey boxes.

- In case the simulator used the key k for the server in the corresponding *ssid*, then the simulator sends $(\textsc{RcvComplete}, sid, ssid, U, S, [])$ to the ideal functionality.
- Otherwise, if the server of *ssid* participated in the protocol with some key k' and value s then for all values (x', y', k^*) in T_{seen}, if $y' = \text{PRF}^{H_1}_{k'}(x)$, add $([k^*, x'])$ to a list L. Send $(\textsc{RcvComplete}, sid, ssid, U, k', L)$ to the functionality.

To summarise the three changes

a) The user now receives output from the ideal functionality, no longer via the dummy interface

b) Simulator now simulates $\mathcal{F}_{ABB}$ with random input for honest users. If both parties are honest, then it just uses some random values as the $\mathcal{F}_{ABB}$ output

c) Eval messages are now answered with *ssid* as the prefix.

Let us now discuss what advantage the environment has in distinguishing between this and the previous game based on these three modifications.

a) The user's output messages are still set in the same way. The prefix is now output via the Eval-prefix interface instead of the dummy interface, but it is still set to the same value. For the output, due to how we set the hash values, they corresponded to exactly the values that the ideal functionality will now directly output to the user. Important to note here is that with how we set the list L we pass to the ideal functionality to add our correlations in both $\mathbf{G}_{11}'$ and $\mathbf{G}_{10}$, if the environment evaluates a hash query $H_2(x, \text{PRF}^{H_1}_{k'}(x))$ in advance, the simulator will answer that with $F_{k^*}(x)$, and then include $[k^*, x]$ in L

when a user runs the evaluation with a corrupted server playing with the key k'.

b) The view of the environment is unchanged by this as well. We note that the randomly chosen masking values perfectly mask the MPC output values from it. Therefore, if the user is honest, the environment cannot distinguish between whether the simulated OUTPUT messages from $\mathcal{F}_{ABB}$ result from an execution where the input used is the actual user input or some randomly chosen value. Additionally, the environment cannot tell if the simulator uses randomly chosen values in the OUTPUT messages instead.

c) This is the only modification that may be noticeable for the environment. In particular, if later in the execution a server uses the same *ssid* as a prefix on the SNDRCOMPLETE message and we then end up in a situation where the *tx* value causes a RCVCOMPLETE to fail. This can occur if no RCVCOMPLETE is sent for *ssid* since then the counter would have been incremented by one in the previous game but not in this game.

Assume an environment $\mathcal{Z}$ exists, which causes this case to occur. We construct an alternative environment $\mathcal{Z}'$ that can distinguish between $\mathbf{G}_{10}$ and $\mathbf{G}_0$ and show that the probability of $\mathcal{Z}$ causing such a change to occur is bounded by the probability of $\mathcal{Z}'$ distinguishing between the two games.

$\mathcal{Z}'$ operates as follows: It internally runs $\mathcal{Z}$ and forwards all messages from $\mathcal{Z}$ to the corresponding parties and all inputs it receives to $\mathcal{Z}$ while maintaining a counter as is done by the functionality in $\mathbf{G}_{11}'$. Once $\mathcal{Z}$ manages to cause the sending of a RCVCOMPLETE that would be ignored due to the *tx* value by the ideal functionality of $\mathbf{G}_{11}'$, $\mathcal{Z}'$ first completes any remaining open sessions by sending OUTPUT commands to $\mathcal{F}_{ABB}$. It then sends the message that $\mathcal{Z}$ sent that caused an ignored RCVCOMPLETE message.

By sending all the OUTPUT messages, we can see that any time the counter was incremented in $\mathbf{G}_{10}$ but not in $\mathbf{G}_{11}'$ (due to the user and server using the same *ssid*), we would now have caused all those sessions to complete. Therefore, in $\mathbf{G}_{10}$ the counter would have been decremented for each of those sessions again, and the counter would now be equivalent in both cases. If the subsequent request failed in $\mathbf{G}_{11}'$ then it would also fail in $\mathbf{G}_{10}$. The environment $\mathcal{Z}'$ can detect if this occurs as either finishing an evaluation does not output anything or the simulator aborts, since the two cases in which the simulator of $\mathbf{G}_{10}$ sends a RCVCOMPLETE message to the ideal functionality, are either when someone performs a hash query of H_2 or when finalising

the output for an honest user. Because in $\mathbf{G}_0$ no aborts will occur, $\mathcal{Z}'$ can distinguish between the two games.

Therefore, we conclude that

$$|\Pr[\mathbf{G}_{11}{}'] - \Pr[\mathbf{G}_{10}]| \leq |\Pr[\mathbf{G}_{10}] - \Pr[\mathbf{G}_0]|$$

3. The final modification required for the proof is in Game $\mathbf{G}_{13}$. There, we need to replace any references to $\mathcal{F}_{corOPRF}$ with the variant with prefixes. So, where before it would have been said that the ideal functionality now operates exactly like $\mathcal{F}_{corOPRF}$, the proof would now need to say that it operates exactly like $\mathcal{F}_{corOPRF}$ with prefixes. The rest remains the same.

We observe that with the above modifications, the remaining game-hops remain valid. Therefore, we conclude the proof. □

With that, we have shown how the Legendre OPRF can be utilised as part of OPAQUE in place of two-hash Diffie Hellman.

3.5 Higher-Power Residue OPRF

One method of increasing the efficiency of schemes that rely on the Legendre symbol is to use higher-power residues instead, as introduced in Section 2.2.3. Using the a-th power residue, each element now provides us with $\log_2(a)$ many output bits instead of 1 when using the Legendre symbol. This reduces the amount of exchanged data at the cost of increased local computation.

We now discuss the modifications that need to be made to our scheme if the generalised power residue is used in place of the Legendre symbol.

1. Wherever we evaluated the Legendre symbol, we now use the a-th power power residue instead.
2. Instead of the decisional SLS assumption, we now need a variant for the higher-power residue. I.e. that for some unknown k, an oracle that returns $\mathcal{L}_p^a(x+k)$ is indistinguishable from an oracle that returns random values in $\{0, \ldots, a-1\}$.
3. Instead of using s^2 as a masking value, we now instead utilise s^a
4. We can adjust Theorem 3.2, to consider the a-th power residue. Since Lemma 2.3 also applies to higher-power residues, we then get the following requirement for the value c

$$c \geq \frac{2.5}{\log_2(\frac{a \cdot p}{p + a\sqrt{p} + 2 \cdot a})}$$

This means that if we, for example, have a 128-bit prime and $a = 128$ then we can use $\ell = 46$ instead of 321, or if we have a 256-bit prime and $a = 256$ then we can use $\ell = 81$ instead of 641.

We present the Power Residue OPRF in Fig. 3.5. We recall that for $a = 2$ we recover the Legendre OPRF, so when we mention the *Higher*-Power Residue OPRF, we are considering the cases where a is greater than two. The Power Residue OPRF arises from applying the above-discussed modifications to the Legendre OPRF. For the sake of brevity, we will not include the full security proof of the scheme, but we observe that with the above steps, it works completely analogously to the security proof for the Legendre OPRF. We will consider using both the Legendre symbol and higher-power residues in our benchmarks. The main performance trade-off will be the fact that we require fewer elements when using a larger a, resulting in reduced bandwidth requirements, but needing more communication rounds to compute the masking values s^a, which requires $\log_2(a)$ communication rounds. Finally, we also note that the extension to an OPRF with prefixes works completely analogously to the Legendre OPRF.

Power Residue OPRF

Public Parameters: Prime number p, output length ℓ. Power residue being used a. Distribution D for $\mathcal{F}_{CRS}$ that returns ℓ many values from $\mathbb{Z}_p$ chosen uniformly at random.
Function to compute inside $\mathcal{F}_{ABB}$ for some list L known to both parties:

$$F\left((x, s_0^u, \ldots, s_{\ell-1}^u, m_0, \ldots, m_{\ell-1}), (k, s_0^s, \ldots, s_{\ell-1}^s)\right) \to$$
$$(s_0^s + s_0^u)^a (k + H_1(x) + L[0]) + m_0, \ldots, (s_{\ell-1}^s + s_{\ell-1}^u)^a (k + H_1(x) + L[\ell-1]) + m_{\ell-1}$$

Initialisation:
On input (Init, sid): S picks $k \leftarrow\$ \mathbb{Z}_p$ and stores $\langle sid, k\rangle$

Server Compromise:
On $(\text{Compromise}, sid)$, if there is a record $\langle sid, k\rangle$ reveal k to $\mathcal{A}^*$

Offline Evaluation:
On $(\text{OfflineEval}, sid, x)$, send (Value, sid) to $\mathcal{F}_{CRS}$, and receive the list L. Then the server retrieves $\langle sid, k\rangle$ and outputs $\left(\text{OfflineEval}, sid, H_2\left(x, \mathcal{L}_p^a(H_1(x) + k + L[0]), \ldots, \mathcal{L}_p^a(H_1(x) + k + L[\ell-1])\right)\right)$

Online Evaluation:

- On $(\text{Eval}, sid, ssid, S', x)$, U sends (Value, sid) to $\mathcal{F}_{CRS}$ and receives the list L. U then sends $(\text{Init}, ssid, \mathbb{Z}_p)$ to $\mathcal{F}_{ABB}$. U then samples $2 \cdot \ell$ many elements from $\mathbb{Z}_p$ ($s_0^u, \ldots, s_{\ell-1}^u$ and $m_0, \ldots, m_{\ell-1}$). It then sends a sequence of Input commands followed by a sequence of Add and Multiply commands required to evaluate F with list L to $\mathcal{F}_{ABB}$, all including $ssid$. U then stores $\langle x, ssid, m_0 \ldots m_{\ell-1}\rangle$. Finally U sends ℓ many Output messages to $\mathcal{F}_{ABB}$.
- On $(\text{SndrComplete}, sid, ssid')$, S sends (Value, sid) to $\mathcal{F}_{CRS}$, and receives the list L. It then sends $(\text{Init}, ssid', \mathbb{Z}_p)$ to $\mathcal{F}_{ABB}$. S then chooses $s_0^s, \ldots, s_{\ell-1}^s \leftarrow\$ \mathbb{Z}_p$, retrieves the record $\langle sid, k\rangle$ and then sends a sequence of Input commands followed by a sequence of Add and Multiply commands required to evaluate F with list L to $\mathcal{F}_{ABB}$, all including $ssid'$. Finally S sends ℓ many Output messages to $\mathcal{F}_{ABB}$.
- Upon having received ℓ many outputs $c_0, \ldots, c_{\ell-1}$ from $\mathcal{F}_{ABB}$ for some subsession identifier $ssid$, U retrieves the stored $\langle x, ssid, m_0 \ldots m_{\ell-1}\rangle$, and computes $r_i = c_i - m_i$. Then U outputs $\left(\text{Eval}, sid, ssid, H_2\left(x, \mathcal{L}_p^a(r_0), \ldots, \mathcal{L}_p^a(r_{\ell-1})\right)\right)$.

Figure 3.5: The Power Residue OPRF, which arises from generalising the Legendre symbols to higher powers.

Chapter 4

Alternative Uses of Correlated OPRFs

While we have shown that the Legendre OPRF fulfils the security notion $\mathcal{F}_{corOPRF}$, it remains an open question what further applications this security notion may have. Jarecki, Krawczyk, and Xu [40] provided brief arguments for why certain previous constructions for Device-enhanced PAKE, a variant of OPAQUE with outsourced envelope, and threshold OPRFs are not UC secure when a correlated OPRF is used in place of an OPRF. Here, we formally investigate the security of an OPRF-based construction for a Password-Protected Secret Sharing (PPSS) protocol when it is initialised with a correlated OPRF.

4.1 Password-Protected Secret Sharing based on Correlated OPRF

We consider the scheme introduced by Jarecki et al. [36], which enables a user, using some password, to share a secret among n servers such that the compromise of any t of them leaks no information about the secret or the password. Knowing the password, it is then possible to recover the same secret by communicating with $t + 1$ such servers. This scheme uses a UC secure OPRF and we will show that if this is replaced by a correlated OPRF, the PPSS scheme is no longer UC secure.

First, in Fig. 4.1 we present the ideal functionality for password-protected secret sharing taken from [36]. We slightly modify it to better fit how $\mathcal{F}_{corOPRF}$ works, as the OPRF functionality that $\mathcal{F}_{corOPRF}$ is based on is a modification of the one used originally for the PPSS scheme. In particular, we now assume that the server inputs (SInit and SRec messages) are given as inputs to the servers, which then pass them along to the ideal functionality. In the original functionality, the adversary sent these to the ideal functionality and they were then sent as messages to the servers. The proof below works anal-

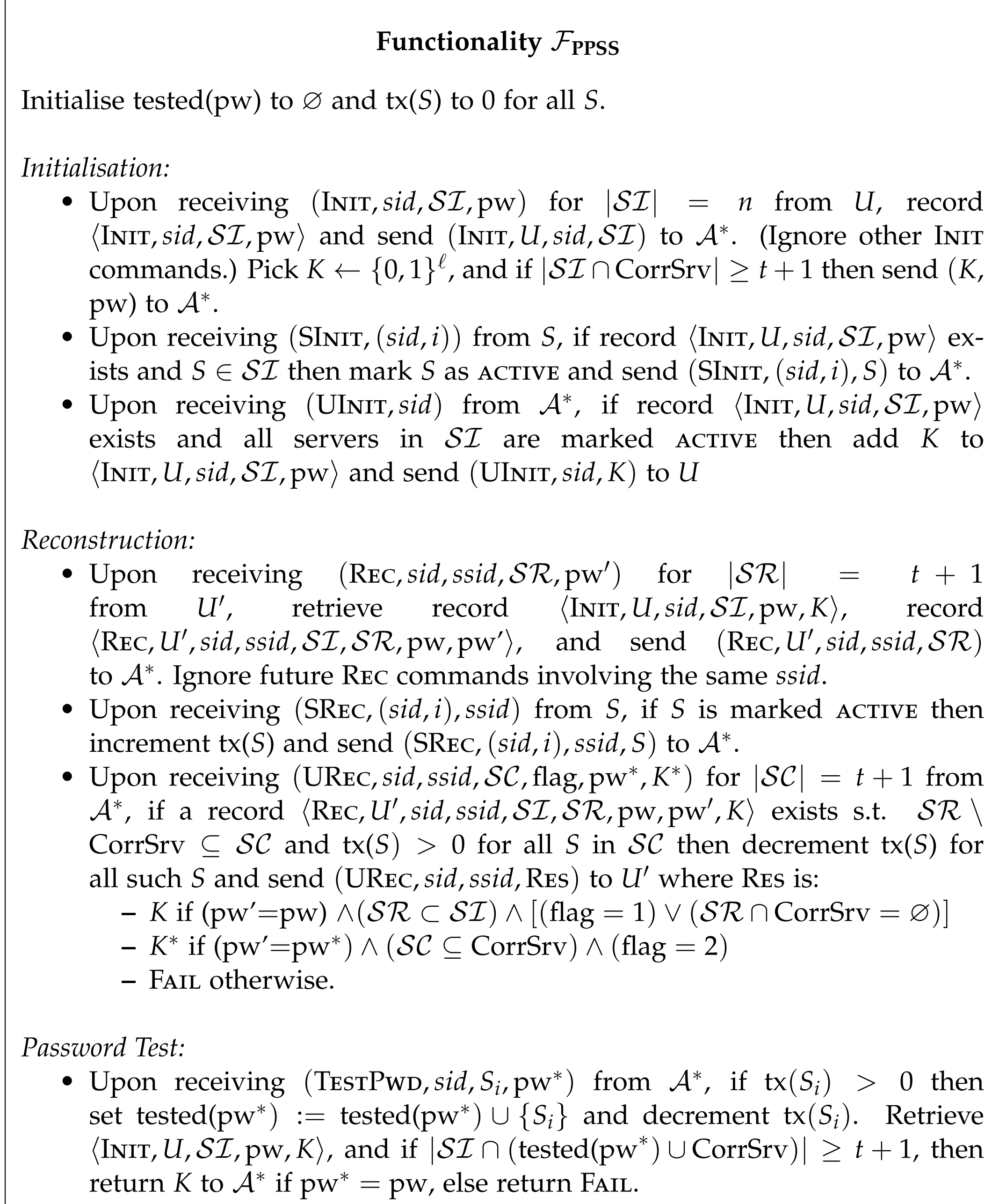
ogously for the original definition, as either the environment determines the number of such messages sent or learns how many such messages are sent. Additionally, the *sid* the servers now receive includes some index i, which denotes their identity.

Effectively, there are three interfaces to the ideal functionality. Using some password, the *Initialisation* interface allows a user to obtain a random secret value by secret sharing it between a set $\mathcal{SI}$ of n servers. This requires all the servers to participate in the session as well (by sending their SInit messages). Upon completion, the user receives the secret value K.
If, at a later point in time, someone with the same password wants to recover K, they can use the *Reconstruction* interface. For this, a subset of t+1 servers $\mathcal{SR}$ is specified. All those servers again need to participate in the protocol, this time by sending a corresponding SRec message, which causes the counter for that server to be incremented. The adversary can then complete the evaluation. In case all the $t+1$ servers are corrupted (i.e., members of CorrSrv) then the adversary can choose a K^* and pw so that if the user uses the password pw it receives the output K^*. Otherwise, if the user uses the correct password, the adversary can either let the session work or, if at least one of the $t+1$ servers is corrupted, can make the recovery fail.

The final interface allows the adversary to perform a *Password Test*, which lets the adversary test if a specific password was used in a session. This requires the adversary to decrement at least $t+1$ counters of servers in the session though, so if less than $t+1$ servers are corrupted the adversary only has a limited amount of times it can test passwords.

The counters in this scheme serve a similar purpose to the counters in the OPRF functionalities. They ensure that the number of completed sessions a server is part of is bounded by the number of times the server participated in the protocol.

Finally, in Fig. 4.2 we present the scheme shown to be secure when initialised with an OPRF [36], initialised with a correlated OPRF. Again, we make the required modifications for the differences between the two OPRF functionalities, in particular with $\mathcal{F}_{corOPRF}$ for any *sid* we now have only a single server, we have subsession identifiers, and it is now the server that sends the SndrComplete message. The idea of the scheme is that to hide a secret, the user evaluates the OPRF with all the servers using the input password and then uses the outputs to mask the secret shares. Those secret shares are then stored on all the servers.
To recover the secret, the user just needs to recover $t+1$ secret shares, which can be done by communicating with $t+1$ servers, unmasking those shares, and then recovering the entire secret. An additional commitment value is used and stored on all the servers to make sure that a single malicious server cannot just change one of the secret shares and then make the recovery out-



Functionality $\mathcal{F}_{\text{PPSS}}$

Initialise tested(pw) to $\varnothing$ and tx(S) to 0 for all S.

Initialisation:

- Upon receiving $(\textsc{Init}, sid, \mathcal{SI}, \text{pw})$ for $|\mathcal{SI}| = n$ from U, record $\langle \textsc{Init}, sid, \mathcal{SI}, \text{pw} \rangle$ and send $(\textsc{Init}, U, sid, \mathcal{SI})$ to $\mathcal{A}^*$. (Ignore other $\textsc{Init}$ commands.) Pick $K \leftarrow \{0,1\}^\ell$, and if $|\mathcal{SI} \cap \text{CorrSrv}| \geq t+1$ then send (K, pw) to $\mathcal{A}^*$.
- Upon receiving $(\text{S}\textsc{Init}, (sid, i))$ from S, if record $\langle \textsc{Init}, U, sid, \mathcal{SI}, \text{pw} \rangle$ exists and $S \in \mathcal{SI}$ then mark S as $\textsc{active}$ and send $(\text{S}\textsc{Init}, (sid, i), S)$ to $\mathcal{A}^*$.
- Upon receiving $(\text{U}\textsc{Init}, sid)$ from $\mathcal{A}^*$, if record $\langle \textsc{Init}, U, sid, \mathcal{SI}, \text{pw} \rangle$ exists and all servers in $\mathcal{SI}$ are marked $\textsc{active}$ then add K to $\langle \textsc{Init}, U, sid, \mathcal{SI}, \text{pw} \rangle$ and send $(\text{U}\textsc{Init}, sid, K)$ to U

Reconstruction:

- Upon receiving $(\textsc{Rec}, sid, ssid, \mathcal{SR}, \text{pw}')$ for $|\mathcal{SR}| = t+1$ from U', retrieve record $\langle \textsc{Init}, U, sid, \mathcal{SI}, \text{pw}, K \rangle$, record $\langle \textsc{Rec}, U', sid, ssid, \mathcal{SI}, \mathcal{SR}, \text{pw}, \text{pw}' \rangle$, and send $(\textsc{Rec}, U', sid, ssid, \mathcal{SR})$ to $\mathcal{A}^*$. Ignore future $\textsc{Rec}$ commands involving the same *ssid*.
- Upon receiving $(\text{S}\textsc{Rec}, (sid, i), ssid)$ from S, if S is marked $\textsc{active}$ then increment tx(S) and send $(\text{S}\textsc{Rec}, (sid, i), ssid, S)$ to $\mathcal{A}^*$.
- Upon receiving $(\text{U}\textsc{Rec}, sid, ssid, \mathcal{SC}, \text{flag}, \text{pw}^*, K^*)$ for $|\mathcal{SC}| = t+1$ from $\mathcal{A}^*$, if a record $\langle \textsc{Rec}, U', sid, ssid, \mathcal{SI}, \mathcal{SR}, \text{pw}, \text{pw}', K \rangle$ exists s.t. $\mathcal{SR} \setminus \text{CorrSrv} \subseteq \mathcal{SC}$ and $\text{tx}(S) > 0$ for all S in $\mathcal{SC}$ then decrement tx(S) for all such S and send $(\text{U}\textsc{Rec}, sid, ssid, \textsc{Res})$ to U' where $\textsc{Res}$ is:
  - K if $(\text{pw}'{=}\text{pw}) \wedge (\mathcal{SR} \subset \mathcal{SI}) \wedge [(\text{flag} = 1) \vee (\mathcal{SR} \cap \text{CorrSrv} = \varnothing)]$
  - K^* if $(\text{pw}'{=}\text{pw}^*) \wedge (\mathcal{SC} \subseteq \text{CorrSrv}) \wedge (\text{flag} = 2)$
  - $\textsc{Fail}$ otherwise.

Password Test:

- Upon receiving $(\textsc{TestPwd}, sid, S_i, \text{pw}^*)$ from $\mathcal{A}^*$, if $\text{tx}(S_i) > 0$ then set $\text{tested}(\text{pw}^*) := \text{tested}(\text{pw}^*) \cup \{S_i\}$ and decrement $\text{tx}(S_i)$. Retrieve $\langle \textsc{Init}, U, \mathcal{SI}, \text{pw}, K \rangle$, and if $|\mathcal{SI} \cap (\text{tested}(\text{pw}^*) \cup \text{CorrSrv})| \geq t+1$, then return K to $\mathcal{A}^*$ if $\text{pw}^* = \text{pw}$, else return $\textsc{Fail}$.

Figure 4.1: Ideal (t,n)-Threshold Password-Protected Secret Sharing Functionality [36].

put some different key.

There is one further difference in the OPRF functionalities. The correlated OPRF functionality $\mathcal{F}_{corOPRF}$ does not utilise authenticated channels, which results in the fact that in the RcvComplete message, the adversary can influence the return value even if the server is honest. In the following, though, we will not use that and only specify some $S^* \neq S$ in our RcvComplete messages for corrupted servers. Therefore, the following proof would work completely analogously for a variant of the correlated OPRF functionality where the adversary can only influence the output of sessions occurring with a malicious server.

Theorem 4.1 *Security of PPSS scheme with correlated OPRF. The OPRF-based Password-Protected Secret Sharing scheme introduced in [36] and presented in Fig. 4.2 is no longer secure in the UC framework when the OPRF is replaced by a correlated OPRF.*

Proof The key idea is that if one of the servers correlates F_i and F_j on some value x and then uses F_i when the user initialises/stores their secret on the servers and F_j in a later evaluation with the same user, the simulator would need to be able to know if the user uses password x as input or not to simulate this case correctly. If the user input is x, then the later evaluation/retrieval by the user will work as intended, whereas if the user input is not x then this will fail with high probability. What then remains to be shown is that the simulator does not have a way to test whether the user input is x in the constructed case.

We present an environment $\mathcal{Z}$ that will, with high probability, be able to distinguish between the real and simulated world for all efficient simulators when the threshold t is greater than one. $\mathcal{Z}$ works as follows for some set of n servers $\mathcal{SI}$. Assume one of these servers S^* is compromised and all others are honest, and there is some honest user U. With ℓ we denote the length of the secret or key, which is protected by the PPSS scheme.

- Send $(\text{SInit}, (sid, i))$ messages as input to all the servers, where the value of i is set to be the position of the server in $\mathcal{SI}$. Make the malicious server S^* participate in the protocol normally.
- Send $(\text{Init}, sid, \mathcal{SI}, x_0)$ for some randomly chosen x_0 as input to U.
- Send $(\text{RcvComplete}, (sid, i), ssid_i, U, S_i, [])$ to all the OPRF sessions, where $ssid_i$ comes from the Eval message outputs. The only exception is for S^* where we replace S_i by some randomly chosen $\mathcal{F}_{corOPRF}$ table label F.
- Wait for output (UInit, sid, K) from U. If no such output occurs, return 1 and abort (simulated world).
- Choose some $x_1 \neq x_0$ uniformly at random and some unique $ssid$.

Password-Protected Secret Sharing scheme

Public Parameters: Security parameter ℓ, threshold parameters $t, n \in \mathbb{N}$, $t \leq n$, field $\mathbb{F} := GF(2^\ell)$, instance of commitment scheme COM, hash function H with range $\{0,1\}^{2\ell}$.

Init *for user U:*

1. On input $(\textsc{Init}, sid, \{S_1, \ldots, S_n\}, \text{pw})$, pick $s \leftarrow\$ \mathbb{F}$ and parse $H(s)$ as $[r||K]$.
2. Generate $(s_1, \ldots, s_n)$ as a (t, n) Shamir's secret-sharing of s over $\mathbb{F}$ and set $s := (s_1, \ldots, s_n)$.
3. Send $(\textsc{Eval}, (sid, i), ssid, S_i, \text{pw})$ to $\mathcal{F}_{corOPRF}$ where $ssid$ is some uniquely chosen subsession identifier, and wait for response $(\textsc{Eval}, (sid, i), ssid, p_i)$ for all $i \in [n]$.
4. Compute $e_i := s_i \oplus p_i$ for $i \in [n]$, set $e := (e_1, \ldots, e_n)$ and compute $C := \text{COM}((\text{pw}, e, s); r)$.
5. Set $\omega := (e, C)$ and send $(\textsc{Send}, (sid, i, 0), S_i, \omega)$ to $\mathcal{F}_{\text{AUTH}}$ for $i \in [n]$.
6. If $\mathcal{F}_{\text{AUTH}}$ returns $(\textsc{Sent}, (sid, i, 1), S_i, U, \textsc{ack})$ for all $i \in [n]$, output $(\text{U}\textsc{Init}, sid, K)$.

Init *for server S:*

1. On input $(\text{S}\textsc{Init}, (sid, i))$, create a record $\langle sid, i \rangle$, send $(\textsc{Init}, (sid, i))$ to $\mathcal{F}_{corOPRF}$, followed by $(\textsc{SndrComplete}, (sid, i), ssid)$ for some uniquely chosen subsession identifier $ssid$.
2. On message $(\textsc{Sent}, (sid, i, 0), U, S, \omega)$ from $\mathcal{F}_{\text{AUTH}}$ for some existing record $\langle sid, i \rangle$, append ω to the record and send $(\textsc{Send}, (sid, i, 1), U, \textsc{ack})$ to $\mathcal{F}_{\text{AUTH}}$.

Rec *for user U:*

1. On input $(\textsc{Rec}, sid, ssid, \mathcal{SR}, \text{pw}')$, send $(\textsc{Eval}, (sid, j), ssid, S'_j, \text{pw}')$ to $\mathcal{F}_{corOPRF}$ for all $S'_j \in \mathcal{SR}$
2. If $\mathcal{F}_{corOPRF}$ returns $\big(\textsc{Eval}, (sid, j), ssid, \sigma_j\big)$ and $\mathcal{F}_{\text{AUTH}}$ returns $\Big(\textsc{Sent}, (sid, ssid, j, 1), S'_j, U, (i_j, \omega_j)\Big)$ for all $S'_j \in \mathcal{SR}$, then if $i_{j_1} = i_{j_2}$ or if $\omega_{j_1} \neq \omega_{j_2}$ for any $j_1 \neq j_2$ then output $(\text{U}\textsc{Rec}, sid, ssid, \textsc{Fail})$ and stop. Otherwise set $p'_{i_j} := \sigma_j$ for all j and set $I := \{i_j\}$.
3. Parse any ω_j as (e', C'), parse e' as $(e'_1, \ldots, e'_n)$, and set $s'_i := e'_i \oplus p'_i$ for all $i \in I$.
4. Recover s' and the shares s'_i for $i \notin I$ by interpolating points (i, s'_i) for $i \in I$.
5. Set $s' = (s'_1, \ldots, s'_n)$, parse $H(s')$ as $[r'||K']$, and output $(\text{U}\textsc{Rec}, sid, ssid, \textsc{Res})$ for $\textsc{Res} = K'$ if $C' = \text{COM}((\text{pw}', e', s'); r')$ and $\textsc{Res} = \textsc{Fail}$ otherwise.

Rec *for server S:*

1. On input $(\text{S}\textsc{Rec}, (sid, i), ssid)$, send $(\textsc{SndrComplete}, (sid, i), ssid)$ to $\mathcal{F}_{corOPRF}$. If hold a record $\langle sid, i, \omega \rangle$ then send $(\textsc{Send}, (sid, ssid, i, 1), U, (i, \omega))$ to $\mathcal{F}_{\text{AUTH}}$.

Figure 4.2: The Password-Protected Secret Sharing scheme from [36], initialised with a correlated OPRF.

- Send $(\text{Rec}, sid, ssid, \mathcal{SR}, x_0)$ to U, causing U to start the reconstruction phase. $\mathcal{SR}$ consists of t honest servers and the corrupted server S_i.
- For all servers in $\mathcal{SR}$ other than S^*, send them a $(\text{SRec}, (sid, i), ssid)$ message to make them send SndrComplete to the correlated OPRF. Then send RcvComplete messages to $\mathcal{F}_{corOPRF}$. For S^* instead send SndrComplete as for the others, and then for the RcvComplete message send $(\text{RcvComplete}, sid, ssid, U, F^*, [F, x_b])$ for b a bit chosen uniformly at random and F^* chosen arbitrarily such that $F^* \neq F$.
- If the user does not output anything, then output 1 (simulated world).
- If $b = 0$ and the user outputs some value K, then output 0 (real world).
- If $b = 1$ and the user outputs Fail, then output 0 (real world).
- Else: Output 1 (simulated world).

First, let us analyse what occurs in the execution with the real world using this environment.
In the real world, the protocol will run to completion. The user will output (UInit, sid, K) in the first phase. The second phase will work successfully in case $b = 0$ because all the OPRF output values sent to the user will be the same as in the initialisation. Therefore, the user will output some value K. On the other hand, if $b = 1$ then, with probability $1 - 1/2^\ell$, the user will return Fail as the restored s will be different, causing the commitment to change.

Therefore, the simulator must simulate things such that the user's output is set analogously in the simulated world. Let us analyse what information the simulator can learn about the value of x_0 in the execution. From the initialisation phase, since only one server is compromised and we assume t is greater than one, all the simulator learns is that the Init takes place, but nothing further about the value x_0. Similarly, in the reconstruction phase, the simulator does not learn any further information about the value x_0.

The only opportunity the simulator has for performing a check on the content of the value of x_0 is by using the TestPwd interface. We observe that the only way for the simulator to do this is by causing the counter in t honest servers to be decremented. However, the environment can ensure that these counters are only incremented t times in total (once for each of the t honest servers) by only sending the above sequence of messages and nothing else. We know these same counters need to be decremented t times for the user to receive its output. Therefore, the simulator cannot make use of this interface in the above case, as otherwise, the counters would have been decremented beyond 0 leading to no output.

The idea of the proof now lies in the fact that since the simulator cannot learn any information about x_0, it can only guess what the value of b is because without knowing what x_0 is, it cannot distinguish between x_0 and x_1. Therefore, the simulator needs to guess with probability 1/2 if it wants

the user to output K or FAIL.

Let us now analyse the advantage of the environment. In the real world, it will always output 0 except if the two OPRF tables are equal on the value x_0 by pure chance, which occurs with probability $1/2^\ell$. In the ideal world, the environment will correctly identify that it is in the ideal world with probability at least 1/2. Therefore, the advantage of the environment is $1 - \frac{1}{2^\ell} - \frac{1}{2}$, which is non-negligible for $\ell > 1$. □

Chapter 5

Performance

5.1 Overview

To benchmark our OPRFs' efficiency, we implement them using the MP-SPDZ framework [43]. This allows us to run the OPRFs with the UC secure *Mascot* MPC protocol [44] and measure their concrete performance. The Mascot MPC protocol provides security in the presence of malicious adversaries, although only with static corruptions. We also note that this library does not offer any security guarantees in regard to quantum adversaries, so the results should be taken with a grain of salt. However, since the security of the Mascot protocol is based on oblivious transfer and symmetric primitives, it is possible to instantiate it based on post-quantum hardness assumptions. While certain post-quantum MPC libraries exist, to our knowledge, none offer security in the malicious model or include security proofs in the UC framework [13], which is why we use the MP-SPDZ framework for our implementation. Another advantage of this framework is that it allows us to benchmark our OPRFs with different underlying MPC protocols. Thus, we will also consider the performance of our schemes in a *semi-honest* environment. This means both parties are expected to execute the protocol as specified, and the MPC protocol ensures that they do not learn any information they should not have during the evaluation [31]. We will use both the *Semi* and *Hemi* MPC protocols for this setting. Semi is a variant of Mascot where the parts required for malicious security are removed [43]. Similarly, Hemi denotes a modified version of LowGear [45], where again, the parts required for malicious security are removed [43].

Our implementations are available on GitHub.[1].

Phases Many MPC protocols, including Mascot, can be split into two phases, an *offline* (also commonly called *preprocessing*) phase and an *online* phase.

[1] https://github.com/lucasdodgson/MP-SPDZ

The offline phase can occur between the parties before either of the parties' input is known and generates values, which are then used, in combination with the parties' input, in the online phase. In general, the offline phase is much more expensive, in computation and required communication, than the online phase. Thus, for our comparison, we will consider the cost of only the online phase and the combined cost of both the offline and online phases. Furthermore, we note that for the offline phase, there are hyperparameters which have a noticeable effect on the performance. We will focus mainly on the *batch size*, which specifies how much preprocessing data is generated in a round of communication. Using a large batch size reduces the number of communication rounds but may require more data to be exchanged between the two parties. Therefore, we consider a variety of potential batch sizes for the evaluation, except for the Hemi MPC protocol where the batch size does not have any influence.

Security Parameters We perform our evaluations using a 128-bit prime, which offers 64 bits of security, and a 256-bit prime, which offers 128 bits of computational security. In both cases, the MPC library provides 64 bits of statistical security. Note that if in a table we have a column called *prime* the entries of this column will be the bit-length of the prime, not the value of the prime itself.

Performance Metrics To evaluate the performance of our OPRFs, we measure and report the amount of bandwidth that needs to be sent between the two parties, the number of communication rounds, as well as the amount of time an execution takes in a simulated WAN environment. The timing measurements are taken with both parties running on the same host, with the *tc* command being used to add a latency of 100ms and a bandwidth limitation of 50 megabits per second. The machine used to run both parties has an Intel i7-1185G7 @ 3.00 GHz $\times$ 8 CPU and 32 GB of RAM. We also verify that certain configurations have a significantly smaller RAM requirement. Finally, we also include the performance of our schemes when ten evaluations are performed in parallel to analyse what performance benefits can be gained in those cases. We round all our values to at least two decimal digits.

One final comment is regarding what exactly we denote by "rounds". In this case, we mean the number of concrete rounds the protocol implementation requires for the execution. This depends strongly on the underlying MPC protocol and the concrete implementation. An alternative approach would be to analyse only the number of actual MPC protocol calls that require communication, each of which the implementation may use multiple rounds for. In our case, that would be four rounds for the online phases of our OPRFs. Another reason to be cautious with the reported rounds is that, in some cases, the number of reported rounds does not correspond to the

Protocol	Performance – Rounds, Data (MB)		
	1 parallel	10 parallel	100 parallel
Legendre OPRF, batch size 65	290, 28.33	2352, 273.34	22982, 2726.18
Legendre OPRF, batch size 185	158, 36.63	872, 276.36	8134, 2721.86
Legendre OPRF, batch size 325	114, 38.82	518, 274.26	4650, 2716.51
Legendre OPRF, batch size 645	92, 51.71	290, 278.56	2374, 2718.83
Legendre OPRF, batch size 1285	92, 102.85	180, 303.71	1226, 2746.79
Legendre OPRF, batch size 3250	92, 259.86	114, 387.05	518, 2738.21
Legendre OPRF, batch size 6500	92, 519.56	92, 520.15	290, 2804.57
Legendre OPRF, online phase	14, 0.067	14, 0.668	14, 6.68
Legendre OPRF improved, online phase	14, 0.031	14, 0.309	14, 3.085

Table 5.1: Bandwidth and communication rounds requirements of the Legendre OPRF using the Mascot MPC protocol and a 128-bit prime.

number of sequential communication rounds. For example, a protocol with 44 rounds of communication, which for a network delay of 100ms means the evaluation should take at least 4.4 seconds to run, may run in under 4.4 seconds. This comes from the fact that certain rounds can take place in parallel. Nevertheless, the reported rounds still provide a decent approximation of the number of sequential rounds.

5.2 Legendre OPRF

In the following, the scheme denoted as "Legendre OPRF" functions exactly like the scheme we presented and proved secure in Chapter 3. We note that if we only measure the online phase, we can improve the performance by performing certain computations in the preprocessing phase. In particular, the multiplicative masking values, s^2, and the user's output masking values are independent of the parties' actual inputs. They thus can be generated and squared as part of the preprocessing phase.[2] Therefore, we also implement a variation of the Legendre OPRF, denoted as "Legendre OPRF improved" where this is done. For the online phase results, we will include both variants' performance.

5.2.1 128-bit prime

First, we consider the case where we have a 128-bit prime and use an evaluation length of 321. In Table 5.1 we present the number of rounds required for the evaluation and the required bandwidth in megabytes, considering the cases where 1, 10, and 100 parallel evaluations are taking place.

[2] $\mathcal{F}_{ABB}$ does not differentiate between the online and offline phases and therefore does not support any interfaces for modelling such behaviour as part of the UC-proof.

Protocol	Time (s)	
	1 parallel	10 parallel
Legendre OPRF, batch size 65	33.68 ± 0.01	268.90 ± 0.86
Legendre OPRF, batch size 185	18.88 ± 0.04	119.70 ± 0.03
Legendre OPRF, batch size 325	15.49 ± 0.00	88.21 ± 0.02
Legendre OPRF, batch size 645	16.37 ± 0.01	70.95 ± 0.03
Legendre OPRF, batch size 1285	25.50 ± 0.01	68.02 ± 0.03
Legendre OPRF, batch size 3250	53.02 ± 0.03	76.97 ± 0.04
Legendre OPRF, batch size 6500	98.67 ± 0.21	98.15 ± 0.05
Legendre OPRF, online phase only	1.61 ± 0.01	2.42 ± 0.01
Legendre OPRF improved, online phase only	1.41 ± 0.00	1.85 ± 0.00

Table 5.2: Runtime of the Legendre OPRF in a simulated WAN setting, evaluated using the Mascot MPC protocol and a 128-bit prime.

Furthermore, in Table 5.2 we include the running times for the protocol in our simulated WAN environment. The reported value is the mean measurement over 100 executions. For this, we consider only the case where we run 1 or 10 sessions in parallel.

When we measure the cost of both the online and offline phases, we observe that at least 92 rounds of communication and over 28MB of bandwidth are required. From the timing information, we can see that for a single evaluation, the batch size of 325 is optimal, with the protocol requiring under 16 seconds for the evaluation. When we consider the case where we do ten evaluations in parallel, the best performing of the evaluated batch sizes is 1285, with the required time being just under 68 seconds.

When evaluating multiple values in parallel, the evaluation time, number of rounds, and bandwidth grow at a lower rate than the number of evaluations being performed. For example, the required time to evaluate ten elements only takes around 4.4 times as long as to evaluate a single element. We attribute this to the fact that some constant overheads remain the same if 1, 10 or 100 functions are being evaluated and because the rounds can take place simultaneously.

If we consider only the cost of the online phase, we observe that our bandwidth requirement drops to 67KB, and the required number of communication rounds is now only 14 for all considered evaluations. Furthermore, this yields a drastically reduced execution time and the cost increase from parallel evaluations is even smaller. The improved variant further reduces the bandwidth requirements by a factor of over two.

Protocol	Performance – Rounds, Data (MB)		
	1 parallel	10 parallel	100 parallel
Legendre OPRF, batch size 65	518, 203.20	4650, 2012.58	45836, 20043.6
Legendre OPRF, batch size 185	224, 216.55	1682, 2028.29	16140, 20020.5
Legendre OPRF, batch size 325	158, 240.15	982, 2027.86	9924, 20044.5
Legendre OPRF, batch size 645	114, 291.10	518, 2011.86	4672, 20025.3
Legendre OPRF, batch size 1285	92, 395.22	290, 2056.50	2374, 20037.9
Legendre OPRF, batch size 3250	92, 999.00	158, 2399.84	982, 20270.2
Legendre OPRF, batch size 6500	92, 1997.61	114, 2932.28	518, 20269.7
Legendre OPRF, online phase	14, 0.267	14, 2.667	14, 26.67
Legendre OPRF improved, online phase	14, 0.124	14, 1.23	14, 12.31

Table 5.3: Bandwidth and communication rounds requirements of the Legendre OPRF using the Mascot MPC protocol and a 256-bit prime.

5.2.2 256-bit prime

Using a 256-bit prime, our scheme has 128-bit computational security, for which we use an evaluation length of 641. Intuitively, the bandwidth requirements should increase by approximately four when compared to using a 128-bit prime. This is because the evaluation length increases by a factor of two, and all the elements are twice the size. As we can see in Table 5.3 this is approximately the increase we observe. The increase is smaller when we use a large batch size because we already generate enough preprocessing to cover parts of the increased evaluation length. When we consider the online phase only, we again observe an increase in bandwidth by a factor close to four while the number of required rounds remains the same. In Table 5.4 we can see the runtimes in the simulated WAN environment. The change in runtime is much harder to predict since it depends on whether the number of rounds or the bandwidth contributed a more significant part to the runtimes. For example, if we are measuring only the online phase, we can see that for a single evaluation, there is almost no increase. In contrast, there is a significantly more substantial increase for other configurations.

5.2.3 Semi-honest security

As already stated, the weaker security notion of semi-honest security also suffices for some use cases. Hence, it is of interest to also know the performance of the OPRFs in that setting. Here, we limit ourselves to considering the replacement of Mascot by a semi-honest OPRF protocol. We note that a more thorough treatment of semi-honest security would also revisit specific aspects of the security proof to optimise the scheme further. In particular, we recall that the evaluation length is larger than the number of output bits because we needed to show that a malicious server cannot find two keys for which the output is equal. This is no longer required in the semi-honest setting, so a shorter evaluation length would also suffice. For example, for

Protocol	Time (s)	
	1 parallel	10 parallel
Legendre OPRF, batch size 65	76.93 ± 0.02	722.12 ± 0.02
Legendre OPRF, batch size 185	55.17 ± 0.01	485.86 ± 0.03
Legendre OPRF, batch size 325	53.91 ± 0.03	425.47 ± 0.07
Legendre OPRF, batch size 645	59.84 ± 0.03	395.22 ± 0.09
Legendre OPRF, batch size 1285	75.52 ± 0.04	375.99 ± 0.13
Legendre OPRF, batch size 3250	178.68 ± 0.11	418.28 ± 0.17
Legendre OPRF, batch size 6500	349.02 ± 0.22	506.63 ± 0.25
Legendre OPRF, online phase	1.64 ± 0.01	3.39 ± 0.00
Legendre OPRF improved, online phase	1.42 ± 0.00	2.25 ± 0.01

Table 5.4: Runtime of the Legendre OPRF in a simulated WAN setting, evaluated using the Mascot MPC protocol and a 256-bit prime.

Prime	All Phases	Protocol	Batch Size	Rounds	Data (MB)	Time (s)
128	✓	Semi	645	44	4.09	3.14 ± 0.00
128	✓	Hemi	—	15	3.22	3.56 ± 0.02
256	✓	Semi	1285	44	26.60	7.29 ± 0.00
256	✓	Hemi	—	15	10.71	5.79 ± 0.01
128	✗	Semi	—	3	0.03	0.21 ± 0.00
128	✗	Hemi	—	3	0.03	0.21 ± 0.00
256	✗	Semi	—	3	0.12	0.22 ± 0.00
256	✗	Hemi	—	3	0.12	0.22 ± 0.00

Table 5.5: Best performing variants of the Legendre OPRF in the semi-honest model. All phases indicates whether the measurements are for all the phases or only the online phase.

a 256-bit prime, we believe one could also use an evaluation length of 256 instead of 641. Nevertheless, we leave this to potential future works.

For the complete benchmark results in the semi-honest model, we refer to Appendix A.1.1. In Table 5.5, we present the best-performing variants for both the Hemi and Semi MPC protocols. We recall here that the batch size did not affect the performance of the Hemi MPC protocol. We can see that the online performance for both MPC protocols is almost the same, requiring 0.12MB of communication for 256-bit primes with three rounds of communication taking around 0.22 seconds.[3] In case we consider both the online and offline costs, we can see that for a 128-bit prime, the protocol can run in just over 3 seconds and for a 256-bit prime in 5.8 seconds. For a 128-bit prime, we can see that while Hemi has both lower bandwidth and round re-

[3]Note that this is the performance of the improved variant. An additional round of communication can be saved because the s^2 values are available from the beginning.

quirements, its execution takes longer than Semi, while for a 256-bit prime, its execution is faster. This, we attribute to certain constant overheads with Hemi that are larger than they are for Semi. In both cases, the Legendre OPRF can gain significant performance improvements by changing the underlying MPC protocol if the stronger property of malicious security is not required.

5.3 Higher-Power Residue OPRF

As introduced in Section 3.5, the generalisation of the Legendre symbol to higher-power residues can also be used as an OPRF. This results in a reduced evaluation length ℓ, reducing the amount of data that needs to be communicated. The downside is that using the a-th power residue requires us to compute s^a instead of s^2, which increases the number of required rounds of communication. Here, we consider the powers of two between four and 256 as values for a.

For the online phase benchmarks, similarly to s^2, the computation of s^a could be done as part of the offline phase. Unfortunately, the used library does not directly support this, so we perform that computation as part of the online phase.

While the full results can be found in Appendix A.1.2, we present here the results of the optimal configurations in terms of the required time for the various powers and a single parallel session. One thing that is important to note is that for the timing estimate, we now use the mean of 5 runs instead of 100 as was done previously. This is due to the increased number of parameters we test with the Power Residue OPRF.

When considering malicious security, the results for the online and offline phases can be seen in Table 5.6. As expected, the amount of data that needs to be sent is reduced compared to the Legendre OPRF at the cost of increased communication rounds. Nevertheless, there is a noticeable performance benefit, with the optimal time being reduced from around 54s to under 38 for a 256-bit prime.

The performance of just the online phase can be seen in Table 5.7. For the online phase, the increased rounds have a more significant effect than the reduced data requirements, leading to an overall slower evaluation than the Legendre OPRF. We expect that if the computation of the masking values s^a is performed as part of the offline phase, then the Power Residue OPRF would also offer an advantage compared to the Legendre OPRF for this case. However, this advantage would only be minor because the amount of data that needs to be sent for this configuration is already very small, with the communication rounds presenting the most significant performance bottleneck.

Prime	Power Residue	Batch Size	Rounds	Data (MB)	Time (s)
128	4	250	115	29.88	14.72 ± 0.03
128	8	450	94	36.10	14.17 ± 0.01
128	16	450	95	36.10	14.18 ± 0.01
128	32	450	96	36.09	14.32 ± 0.01
128	64	450	97	36.09	14.38 ± 0.01
128	128	185	120	22.14	13.83 ± 0.03
128	256	185	121	22.14	13.86 ± 0.00
256	4	250	159	184.74	44.81 ± 0.04
256	8	250	160	184.71	44.79 ± 0.05
256	16	185	183	163.31	42.87 ± 0.03
256	32	185	184	163.30	43.12 ± 0.04
256	64	250	141	148.80	37.77 ± 0.05
256	128	185	164	136.74	37.63 ± 0.03
256	256	185	165	136.73	37.64 ± 0.02

Table 5.6: Evaluation of the Power Residue OPRF with the Mascot MPC protocol when measuring all phases.

Prime	Power Residue	Rounds	Data (MB)	Time (s)
128	4	15	0.03	1.63 ± 0.02
128	8	16	0.02	1.65 ± 0.09
128	16	17	0.02	1.87 ± 0.14
128	32	18	0.02	1.86 ± 0.05
128	64	19	0.02	2.09 ± 0.01
128	128	20	0.02	2.1 ± 0.07
128	256	21	0.02	2.38 ± 0.07
256	4	15	0.10	1.64 ± 0.00
256	8	16	0.10	1.64 ± 0.00
256	16	17	0.09	1.86 ± 0.00
256	32	18	0.09	1.88 ± 0.00
256	64	19	0.09	2.12 ± 0.01
256	128	20	0.09	2.21 ± 0.01
256	256	21	0.09	2.51 ± 0.01

Table 5.7: Evaluation of the Power Residue OPRF online phase using the Mascot MPC protocol.

Prime	All Phases	Protocol	Power	Batch Size	Rounds	Data (MB)	Time (s)
128	✓	Semi	8	450	46	2.87	3.13 ± 0.00
128	✓	Hemi	4	—	16	3.20	3.55 ± 0.00
256	✓	Semi	8	1000	46	20.68	6.37 ± 0.00
256	✓	Hemi	4	—	16	10.65	5.83 ± 0.05
128	✗	Semi	8	—	5	0.02	0.41 ± 0.00
128	✗	Hemi	4	—	4	0.03	0.41 ± 0.00
256	✗	Semi	4	—	4	0.10	0.45 ± 0.00
256	✗	Hemi	4	—	4	0.10	0.45 ± 0.00

Table 5.8: Best performing variants of the Higher-Power Residue OPRF in the semi-honest model. All phases indicates whether the measurements are for all the phases or only the online phase.

5.3.1 Semi-honest performance

We also run our benchmarks using Hemi and Semi as the underlying MPC protocols. Again, we note that we do not include a thorough treatment of the semi-honest model and use the same evaluation length as we did for malicious security. The best-performing variants can be seen in Table 5.8, whereas the full results are again available in Appendix A.1.2. For the sake of brevity, we only list the best-performing powers. Again, we observe that the additional rounds of communication for the online phase only cause the execution to be slower despite the slightly lower data requirements. In case the performance of both phases is considered for semi-honest security, the Power Residue OPRF performs marginally better for 128-bit primes, whereas the Legendre OPRF outperforms the Power Residue OPRF for 256-bit primes.

5.4 Ram Usage

The amount of RAM the evaluation requires is another critical consideration for the real-world usability of a protocol. Many systems only have limited available RAM, so the number of entities that can run a protocol requiring hundreds of Gigabytes of RAM is very limited.

To get a better estimate on the estimated RAM usage of our OPRFs, we reproduced the benchmarks on a system limited to 4GB and on a system limited to 2GB of RAM.[4] These represent configurations where there are under 2GB and 1GB, respectively, of RAM available to each party during the execution.[5]

For the Legendre OPRF, we can reproduce all our results on the system with 4GB of RAM with batch sizes up to and including 1285 and 10 parallel

[4]Swap was disabled for both cases.

[5]A part of the available RAM is also utilised by the running Operating System.

Protocol	Rounds	Data (MB)	Time (s)
Legendre OPRF online phase	14	0.12	1.42 ± 0.00
Legendre OPRF all phases	158	240.15	53.91 ± 0.03
Higher-Power Residue OPRF online phase	15	0.10	1.64 ± 0.00
Higher-Power Residue OPRF all phases	164	136.74	37.63 ± 0.03
Round-Optimal LWE VOPRF [3]	2	131072.00	> 20971
Round-Optimal SIDH VOPRF [5]	2	8.70	> 1.39

Table 5.9: Comparison of post-quantum OPRF protocols that offer malicious security.

evaluations when using the Mascot MPC protocol. Similarly, all presented evaluations with both Semi and Hemi run successfully. On the other hand, for the system with 2GB of RAM, the protocol no longer successfully runs when using the Mascot protocol with a batch size of 1285 and a 256-bit prime. For smaller batch sizes, as well as when using a 128-bit prime or a semi-honest MPC protocol, the execution is still successful. Notably, based on the timing measurements, the configuration offering the optimal runtime in the simulated WAN environment for Mascot and a 256-bit prime is still successful on the 2GB system. If a system with even more strict memory requirements is used, we also note that a valid option is to utilise smaller batch sizes.

For the higher-power residues, the actual execution has a smaller RAM requirement than for the Legendre OPRF, so once compiled, the batch sizes that worked with the Legendre OPRF will also work with the Power Residue OPRF. However, one consideration for this setting is that the compilation is more RAM intensive when higher powers are used. For example, without further tweaking the compilation parameters on the 2GB system, the compilation will fail once powers of 32 and above are used. Assuming, though, that the compilation is done on a system with sufficient RAM, the execution works as expected.

5.5 Comparison

Finally, we include a comparison to existing post-quantum OPRF protocols, both for malicious and semi-honest security. For malicious security, we use the table from [5] as a baseline, keeping only the maliciously secure schemes. In Table 5.9 we can see the comparison. It is important to note that the other OPRFs we are comparing are verifiable OPRFs, which is a stronger security notion. Even though this property is required for their malicious security, the comparison should be taken with a grain of salt.

We note that, to our knowledge, no complete implementations are available for the other two maliciously secure OPRFs. The bandwidth requirements

Protocol	Rounds	Data (MB)	Time (s)	Malicious Client	Verifiable
Legendre OPRF online phase	3	0.12	0.22 ± 0.00	✗	✗
Legendre OPRF all phases	15	10.71	5.79 ± 0.01	✗	✗
Higher-Power Residue OPRF online phase	4	0.10	0.45 ± 0.00	✗	✗
Higher-Power Residue OPRF all phases	16	10.65	5.83 ± 0.05	✗	✗
Garbled Circuits OPRF [27]	—	6.79	2.07 ± 0.02	✓	✗
Garbled Circuits OPRF – non post-quantum instantiation [27]	—	0.30	0.28 ± 0.02	✓	✗
OPUS [33]	—	0.02	35.29 ± 0.04	✗	✗
CSIDH-based OPRF [12]	3	>0.42	—	✓	✗
Round-Optimal SIDH OPRF [5]	2	3.00	—	✓	✗
Round-Optimal SIDH VOPRF [5]	2	8.70	—	✓	✓

Table 5.10: Comparison of post-quantum OPRF protocols that offer semi-honest security.

are estimated on element sizes and do not account for various overheads. The provided runtime is a lower bound derived from the amount of data that needs to be communicated between the parties. We also recall that the SIDH VOPRF [5] relies on a trusted setup being available.

We can see that our OPRFs have significantly increased round complexity. At the same time, when we allow for a preprocessing phase, they have a significantly lower bandwidth requirement. Still, when compared to a quantum-insecure OPRF, such as two-hash Diffie Hellman that requires two rounds of communication and 70 bytes of communication, there remains a significant performance gap [36, 27].

For semi-honest security, we use the table from [27] as our baseline. The comparison can be seen in Table 5.10. We use a "—" when a value is not available. The Legendre OPRF offers the best performance regarding the required time when we allow for a preprocessing phase. Again, though, it is important to note that the concrete instantiation of the Legendre OPRF does not offer post-quantum security, which the other OPRFs in the list do, except for the second Garbled Circuit variant.

For the semi-honest model, again, the caveat with verifiability applies, and additionally that various models are secure even in the presence of a mali-

cious client, which this specific instantiation of our scheme is not.

For both cases, it is important to note that due to the used MPC protocol not being post-quantum, the performance of our scheme should be taken with a grain of salt. Still, our scheme brings significant performance benefits for malicious security when looking at the required amount of data. In the semi-honest model, if we look at the online phase, the Legendre and Power Residue OPRFs have the best time and close to the lowest required data.

Chapter 6

Conclusion

In this thesis, we introduced and analysed the Legendre OPRF. We studied its security using three variations of OPRF security notions in the Universal Composability framework and proved it secure according to two of those in the malicious setting, thereby showing the security of various protocols, including OPAQUE, when instantiated with the Legendre OPRF. Additionally, we considered the generalisation of the Legendre OPRF to higher-power residues and discussed the performance tradeoff that arises in doing so.

Our performance results show that both OPRFs can offer a performance benefit over alternative post-quantum OPRFs if we allow for a separate preprocessing phase. We also provide implementations of our OPRFs, which we could not find for the alternative maliciously-secure post-quantum OPRFs. The OPRFs also offer the advantage of parallelising well, yielding further performance advantages for some use cases. Depending on the configuration, the cost increase for performing ten evaluations in parallel was between a factor of two and seven. Furthermore, the OPRFs will benefit from any improvements to the underlying MPC protocol, as most of the cost is due to the MPC protocol and its communication requirements. Our results also show the performance advantages in case a weaker security notion, such as semi-honest security, suffices for an application.

We also examined the security properties of correlated OPRFs, proving that one password-protected secret sharing protocol is no longer secure when the used OPRF is a correlated OPRF. This highlights an important limitation of the Legendre and Power Residue OPRFs and their achieved security notion.

While the Legendre OPRF offers competitive performance compared to alternative post-quantum OPRFs, there remains a significant performance gap when compared to maliciously secure OPRF protocols, such as two-hash Diffie Hellman [40], which are insecure against quantum adversaries. We believe further improvements to be required for a broader applicability of

post-quantum OPRFs.

6.1 Future Work

We list here some of the avenues of potential future work we identified during the thesis. Firstly, we would have liked to explore some of the potential extensions of the Legendre OPRF. In particular, modifying the scheme to have the verifiability property would be of interest. We believe that including a hashed commitment of the key, which is verified as part of the MPC protocol, would suffice for this but this requires a separate security proof. Similarly, it would also be of significant interest to show the Legendre OPRF's security in the case of a semi-honest server according to the non-correlated OPRF security notion [39]. We believe this to be possible because the proof only uses correlations for the case where the server uses a different key than it started with. Showing this security would further open the applications for which the Legendre OPRF is a valid candidate.

Additionally, while we analysed one further use case of OPRFs (PPSS) when instantiated with a correlated OPRF, there remain open questions about the applicability of the security notion to other applications, which would be of great interest to analyse further.

Finally, in the implementation, there are various potential extensions we identified. In particular, implementing and evaluating the scheme in a post-quantum MPC library would be of great interest once those become available. The same applies to an implementation where the masking value is completely calculated as part of the preprocessing for the higher-power residues, in which case we expect them to be the better-performing variant in the online phase.

Appendix A

Appendix

A.1 Additional Benchmarks

In this section, we include further benchmark results, which we did not include in Chapter 5. It contains the full benchmark results in the semi-honest model and for the higher-power residues. We also present the results for 10 parallel evaluations here, which were omitted from the main part for the sake of brevity.

A.1.1 Complete results for the semi-honest model

Semi First, we present the full benchmarks for the Legendre OPRF in the Semi MPC protocol. The results for both the offline and online phases can be seen in Table A.1 and Table A.2 for 128 and 256-bit primes respectively. In Table A.3 the results for only the online phase are shown.

Hemi Next, we present the results for the Hemi MPC protocol. Table A.4 contains the results for both the online and offline phases for both 128-bit and 256-bit primes, whereas Table A.5 contains the results for both the offline and online phases.

A.1.2 Complete results for the Power Residue OPRF

Here, we include further results on the performance of the Power Residue OPRF, when the considered power is greater than two. For these, the timing measurements are based on 5 repetitions instead of 100. We evaluate the batch sizes of 65, 150, 185, 250, 300, 350, 450, 600, 645, and 1000. However, in the tables, we will only include the best-performing batch sizes in terms of rounds, bandwidth requirement, or time.

Parallel	Batch Size	Rounds	Bandwidth (MB)	Time (s)
1	65	143	4.30	11.69 ± 0.10
1	185	77	4.74	6.69 ± 0.05
1	325	55	4.14	4.45 ± 0.00
1	645	44	4.09	3.14 ± 0.00
1	1285	44	8.03	4.16 ± 0.00
1	3250	44	20.10	6.42 ± 0.00
1	6500	44	40.07	10.12 ± 0.01
10	65	1122	42.06	105.73 ± 0.28
10	185	418	41.04	44.57 ± 0.40
10	325	253	40.90	27.01 ± 0.01
10	645	143	40.40	19.68 ± 0.01
10	1285	88	40.15	13.38 ± 0.00
10	3250	55	40.55	11.08 ± 0.00
10	6500	44	40.53	10.2 ± 0.01

Table A.1: Benchmarks results for the Legendre OPRF using the Semi MPC protocol and a 128-bit prime. Both online and offline phases.

Parallel	Batch Size	Rounds	Bandwidth (MB)	Time (s)
1	65	253	27.28	24.59 ± 0.01
1	185	110	26.92	13.43 ± 0.03
1	325	77	26.97	9.92 ± 0.03
1	645	55	26.72	8.37 ± 0.01
1	1285	44	26.60	7.29 ± 0.00
1	3250	44	66.84	14.66 ± 0.02
1	6500	44	133.40	26.35 ± 0.03
10	65	2211	269.58	228.3 ± 0.05
10	185	803	268.70	110.79 ± 0.08
10	325	473	269.14	83.4 ± 0.09
10	645	253	266.69	66.43 ± 0.03
10	1285	143	265.48	57.8 ± 0.02
10	3250	77	268.43	53.73 ± 0.03
10	6500	55	268.39	51.02 ± 0.05

Table A.2: Benchmarks results for the Legendre OPRF using the Semi MPC protocol and a 256-bit prime. Both online and offline phases.

Prime	Improved	Parallel	Rounds	Bandwidth (MB)	Time (s)
128	✗	1	3	0.05	0.41 ± 0.00
128	✓	1	3	0.03	0.21 ± 0.00
128	✗	10	3	0.51	0.87 ± 0.00
128	✓	10	3	0.31	0.72 ± 0.01
256	✗	1	3	0.20	0.44 ± 0.00
256	✓	1	3	0.12	0.22 ± 0.00
256	✗	10	3	2.05	1.23 ± 0.01
256	✓	10	3	1.23	0.99 ± 0.01

Table A.3: Benchmark results for the Legendre OPRF using the Semi MPC protocol, when considering only the cost of the online phase.

Prime Size	Parallel	Rounds	Bandwidth (MB)	Time (s)
128	1	15	3.22	3.56 ± 0.02
128	10	15	3.68	3.63 ± 0.02
256	1	15	10.71	5.79 ± 0.01
256	10	15	12.56	6.05 ± 0.03

Table A.4: Benchmark results for the Legendre OPRF using the Hemi MPC protocol, when there is not a separate preprocessing phase.

Prime Size	Improved	Parallel	Rounds	Bandwidth (MB)	Time (s)
128	✗	1	3	0.05	0.40 ± 0.00
128	✓	1	3	0.03	0.21 ± 0.00
128	✗	10	3	0.51	0.86 ± 0.00
128	✓	10	3	0.31	0.68 ± 0.01
256	✗	1	3	0.20	0.43 ± 0.00
256	✓	1	3	0.12	0.22 ± 0.00
256	✗	10	3	2.05	1.20 ± 0.00
256	✓	10	3	1.23	1.01 ± 0.02

Table A.5: Benchmark results for the Legendre OPRF using the Hemi MPC protocol, when considering only the cost of the online phase

All phases

First, we present the results for all phases, so the measurements are for both the online and offline phases.

Mascot The results for the Power Residue OPRF with the Mascot MPC protocol can be seen in Table A.6 for a 128-bit prime and in Table A.7 for a 256-bit prime.

Semi For the Semi MPC protocol, the results can be seen in Table A.6 for a 128-bit prime and Table A.7 for a 256-bit prime.

Hemi In Table A.10 and Table A.11 the results for the Hemi MPC protocol with a 128-bit and 256-bit prime, respectively, are shown.

Online Phase For the online phase results, we consider both 128 and 256-bit primes and both the basic variant and the variant with improved preprocessing. The powers we consider for the OPRF are powers of two between 4 and 256. The cost of the Mascot online phase for the Power Residue OPRF can be seen in Table A.12. The results for Semi in Table A.13 and for Hemi in Table A.14

Parallel	Power Residue	Batch Size	Rounds	Bandwidth (MB)	Time (s)
1	4	65	247	23.20	28.08 ± 0.01
1	4	250	115	29.88	14.72 ± 0.03
1	4	600	93	48.09	16.14 ± 0.01
1	8	65	226	20.65	25.17 ± 0.00
1	8	450	94	36.10	14.17 ± 0.01
1	16	65	227	20.64	25.2 ± 0.01
1	16	450	95	36.10	14.18 ± 0.01
1	32	65	206	18.09	22.32 ± 0.01
1	32	450	96	36.09	14.32 ± 0.01
1	64	65	207	18.09	22.55 ± 0.02
1	64	450	97	36.09	14.38 ± 0.01
1	128	65	208	18.09	22.76 ± 0.02
1	128	185	120	22.14	13.83 ± 0.03
1	128	450	98	36.09	14.61 ± 0.01
1	256	65	209	18.08	22.99 ± 0.04
1	256	185	121	22.14	13.86 ± 0.00
1	256	450	99	36.09	14.64 ± 0.02
10	4	185	651	199.73	88.85 ± 0.16
10	4	1000	181	236.30	56.3 ± 0.07
10	8	65	1556	176.83	194.54 ± 3.30
10	8	645	226	202.84	53.81 ± 0.03
10	8	1000	182	236.23	56.22 ± 0.06
10	16	65	1461	165.25	186.57 ± 0.05
10	16	600	227	188.69	51.49 ± 0.11
10	16	1000	161	197.22	47.96 ± 0.06
10	32	65	1402	157.46	176.21 ± 2.65
10	32	645	206	177.64	47.82 ± 0.03
10	32	1000	162	197.20	48.13 ± 0.03
10	64	65	1381	154.90	176.2 ± 0.02
10	64	645	207	177.63	47.99 ± 0.06
10	64	1000	163	197.19	48.43 ± 0.06
10	128	65	1338	149.78	170.47 ± 0.02
10	128	645	208	177.61	48.33 ± 0.06
10	128	1000	164	197.18	49.24 ± 0.26
10	256	150	605	147.17	80.64 ± 1.73
10	256	600	209	165.25	46.51 ± 0.10
10	256	1000	165	197.17	48.94 ± 0.02

Table A.6: Benchmark results for the Power Residue OPRF using the Mascot MPC protocol and a 128-bit prime, considering the cost of both the online and offline phases.

Parallel	Power Residue	Batch Size	Rounds	Bandwidth (MB)	Time (s)
1	4	65	401	151.06	58.56 ± 0.02
1	4	250	159	184.74	44.81 ± 0.04
1	4	1000	93	307.56	63.46 ± 0.17
1	8	65	380	141.69	55.19 ± 0.01
1	8	250	160	184.71	44.79 ± 0.05
1	8	1000	94	307.53	63.88 ± 0.22
1	16	65	359	132.33	51.8 ± 0.03
1	16	185	183	163.31	42.87 ± 0.03
1	16	1000	95	307.51	63.87 ± 0.28
1	32	65	338	122.98	48.45 ± 0.01
1	32	185	184	163.30	43.12 ± 0.04
1	32	1000	96	307.50	63.76 ± 0.20
1	64	65	339	122.98	48.75 ± 0.15
1	64	250	141	148.80	37.77 ± 0.02
1	64	1000	97	307.50	63.79 ± 0.12
1	128	65	340	122.97	48.7 ± 0.02
1	128	185	164	136.74	37.63 ± 0.03
1	128	1000	98	307.50	64.28 ± 0.24
1	256	65	341	122.97	48.98 ± 0.02
1	256	185	165	136.73	37.64 ± 0.02
1	256	1000	99	307.49	64.35 ± 0.16
10	4	65	3431	1464.53	531.08 ± 0.10
10	4	645	401	1495.45	304.66 ± 0.55
10	4	1000	291	1600.12	311.65 ± 0.31
10	8	65	3048	1293.41	469.54 ± 0.09
10	8	1000	270	1456.38	282.63 ± 0.41
10	16	65	2849	1202.51	437.89 ± 0.22
10	16	1000	249	1312.78	256.11 ± 0.46
10	32	65	2724	1145.04	417.3 ± 0.05
10	32	645	338	1217.31	248.93 ± 0.22
10	32	1000	250	1312.68	256.54 ± 0.51
10	64	150	1183	1114.79	288.38 ± 0.33
10	64	450	449	1172.49	247.45 ± 0.33
10	64	1000	251	1312.63	256.24 ± 0.78
10	128	65	2608	1092.89	400.0 ± 0.04
10	128	450	450	1172.45	248.05 ± 0.37
10	128	1000	252	1312.58	257.75 ± 0.35
10	256	65	2543	1064.83	391.54 ± 0.11
10	256	600	341	1132.35	235.34 ± 0.35
10	256	1000	253	1312.54	258.72 ± 0.44

Table A.7: Benchmark results for the Power Residue OPRF using the Mascot MPC protocol and a 256-bit prime, considering the cost of both the online and offline phases.

Parallel	Power Residue	Batch Size	Rounds	Bandwidth (MB)	Time (s)
1	4	250	56	3.21	4.0 ± 0.04
1	4	600	45	3.80	3.29 ± 0.00
1	8	450	46	2.87	3.13 ± 0.00
1	16	150	69	2.91	5.54 ± 0.00
1	16	450	47	2.87	3.34 ± 0.00
1	32	65	103	2.60	7.09 ± 0.04
1	32	450	48	2.87	3.34 ± 0.00
1	64	65	104	2.60	7.78 ± 0.44
1	64	450	49	2.87	3.55 ± 0.00
1	128	185	61	2.40	4.45 ± 0.10
1	128	450	50	2.87	3.57 ± 0.00
1	256	185	62	2.40	4.75 ± 0.04
1	256	450	51	2.87	3.78 ± 0.00
10	4	600	122	30.07	16.54 ± 0.06
10	4	1000	89	31.24	12.16 ± 0.02
10	8	150	354	27.66	35.48 ± 0.91
10	8	1000	90	31.19	12.19 ± 0.01
10	16	250	212	25.23	21.98 ± 0.93
10	16	1000	80	25.00	10.1 ± 0.02
10	32	350	158	24.20	15.97 ± 0.00
10	32	1000	81	24.98	10.29 ± 0.01
10	64	350	159	24.19	15.99 ± 0.00
10	64	1000	82	24.97	10.53 ± 0.01
10	128	185	259	23.44	25.8 ± 0.59
10	128	1000	83	24.96	10.84 ± 0.05
10	256	600	106	22.54	13.44 ± 0.01
10	256	1000	84	24.96	10.95 ± 0.03

Table A.8: Benchmark results for the Power Residue OPRF using the Semi MPC protocol and a 128-bit prime, considering the cost of both the online and offline phases.

Parallel	Power Residue	Batch Size	Rounds	Bandwidth (MB)	Time (s)
1	4	65	199	20.46	$18.90.0 \pm 0.00$
1	4	1000	45	20.70	6.39 ± 0.02
1	8	450	57	18.65	7.06 ± 0.00
1	8	1000	46	20.68	6.37 ± 0.00
1	16	65	179	17.73	16.64 ± 0.01
1	16	1000	47	20.67	6.57 ± 0.00
1	32	65	169	16.38	15.43 ± 0.01
1	32	1000	48	20.67	6.58 ± 0.00
1	64	150	93	15.62	10.48 ± 0.01
1	64	1000	49	20.66	6.87 ± 0.01
1	128	250	72	15.58	8.07 ± 0.00
1	128	1000	50	20.66	6.97 ± 0.01
1	256	185	84	15.39	9.61 ± 0.00
1	256	1000	51	20.66	7.28 ± 0.00
10	8	65	1487	179.61	152.76 ± 0.00
10	8	1000	134	185.79	42.78 ± 0.00
10	16	1000	124	165.18	38.46 ± 0.00
10	32	350	279	159.25	48.58 ± 0.01
10	32	1000	125	165.12	38.72 ± 0.00
10	64	300	313	155.18	51.03 ± 0.01
10	64	1000	126	165.08	39.23 ± 0.00
10	128	185	479	153.40	65.66 ± 0.14
10	128	1000	127	165.06	39.80 ± 0.00
10	256	600	172	148.72	41.58 ± 0.00
10	256	1000	128	165.02	40.79 ± 0.01

Table A.9: Benchmark results for the Power Residue OPRF using the Semi MPC protocol and a 256-bit prime, considering the cost of both the online and offline phases.

Parallel	Power Residue	Rounds	Bandwidth (MB)	Time (s)
1	4	16	3.20	3.55 ± 0.03
1	8	17	3.20	3.75 ± 0.03
1	16	18	3.19	3.88 ± 0.06
1	32	19	3.19	3.92 ± 0.01
1	64	20	3.19	3.98 ± 0.04
1	128	21	3.19	4.14 ± 0.00
1	256	22	3.19	4.16 ± 0.01
10	4	16	3.52	3.68 ± 0.06
10	8	17	3.47	3.82 ± 0.04
10	16	18	3.45	3.81 ± 0.02
10	32	19	3.43	4.0 ± 0.01
10	64	20	3.43	4.06 ± 0.01
10	128	21	3.42	4.32 ± 0.01
10	256	22	3.41	4.5 ± 0.08

Table A.10: Benchmark results for the Power Residue OPRF using the Hemi MPC protocol and a 128-bit prime, considering the cost of both the online and offline phases.

Parallel	Power Residue	Rounds	Bandwidth (MB)	Time (s)
1	4	16	10.65	5.83 ± 0.05
1	8	17	10.63	5.98 ± 0.04
1	16	18	10.62	6.2 ± 0.07
1	32	19	10.61	6.29 ± 0.06
1	64	20	10.61	6.36 ± 0.06
1	128	21	10.61	6.6 ± 0.02
1	256	22	10.60	6.78 ± 0.03
10	4	16	11.94	6.17 ± 0.05
10	8	17	11.74	6.3 ± 0.04
10	16	18	11.63	6.41 ± 0.03
10	32	19	11.57	6.69 ± 0.06
10	64	20	11.53	7.14 ± 0.04
10	128	21	11.51	7.85 ± 0.05
10	256	22	11.48	8.87 ± 0.03

Table A.11: Benchmark results for the Power Residue OPRF using the Hemi MPC protocol and a 256-bit prime, considering the cost of both the online and offline phases.

Prime Size	Improved	Parallel	Power Residue	Rounds	Bandwidth (MB)	Time (s)
128	✗	1	4	15	0.04	1.62 ± 0.00
128	✗	1	8	16	0.04	1.82 ± 0.00
128	✗	1	16	17	0.03	1.84 ± 0.01
128	✗	1	32	18	0.03	2.03 ± 0.00
128	✗	1	64	19	0.03	2.04 ± 0.00
128	✗	1	128	20	0.03	2.26 ± 0.00
128	✗	1	256	21	0.03	2.32 ± 0.01
128	✓	1	4	15	0.03	1.63 ± 0.02
128	✓	1	8	16	0.02	1.65 ± 0.09
128	✓	1	16	17	0.02	1.87 ± 0.14
128	✓	1	32	18	0.02	1.86 ± 0.05
128	✓	1	64	19	0.02	2.09 ± 0.01
128	✓	1	128	20	0.02	2.1 ± 0.07
128	✓	1	256	21	0.02	2.38 ± 0.07
128	✗	10	4	15	0.44	2.01 ± 0.05
128	✗	10	8	16	0.36	1.9 ± 0.00
128	✗	10	16	17	0.32	1.91 ± 0.00
128	✗	10	32	18	0.30	2.13 ± 0.01
128	✗	10	64	19	0.29	2.23 ± 0.01
128	✗	10	128	20	0.27	2.52 ± 0.03
128	✗	10	256	21	0.26	2.7 ± 0.03
128	✓	10	4	15	0.26	1.7 ± 0.07
128	✓	10	8	16	0.24	1.92 ± 0.11
128	✓	10	16	17	0.23	1.96 ± 0.10
128	✓	10	32	18	0.23	2.01 ± 0.12
128	✓	10	64	19	0.23	2.26 ± 0.19
128	✓	10	128	20	0.22	2.45 ± 0.05
128	✓	10	256	21	0.22	2.88 ± 0.07
256	✗	1	4	15	0.17	1.65 ± 0.00
256	✗	1	8	16	0.14	1.84 ± 0.00
256	✗	1	16	17	0.13	1.87 ± 0.01
256	✗	1	32	18	0.12	2.08 ± 0.00
256	✗	1	64	19	0.11	2.14 ± 0.01
256	✗	1	128	20	0.11	2.39 ± 0.01
256	✗	1	256	21	0.11	2.52 ± 0.01
256	✓	1	4	15	0.10	1.64 ± 0.00
256	✓	1	8	16	0.10	1.64 ± 0.00
256	✓	1	16	17	0.09	1.86 ± 0.00
256	✓	1	32	18	0.09	1.88 ± 0.00
256	✓	1	64	19	0.09	2.12 ± 0.01
256	✓	1	128	20	0.09	2.21 ± 0.01
256	✓	1	256	21	0.09	2.51 ± 0.01
256	✗	10	4	15	1.74	3.29 ± 0.00
256	✗	10	8	16	1.44	3.29 ± 0.00
256	✗	10	16	17	1.28	3.22 ± 0.01
256	✗	10	32	18	1.19	3.36 ± 0.01
256	✗	10	64	19	1.13	3.52 ± 0.01
256	✗	10	128	20	1.09	3.84 ± 0.03
256	✗	10	256	21	1.05	4.81 ± 0.04
256	✓	10	4	15	1.03	2.33 ± 0.04
256	✓	10	8	16	0.96	2.4 ± 0.05
256	✓	10	16	17	0.92	2.6 ± 0.01
256	✓	10	32	18	0.90	2.91 ± 0.05
256	✓	10	64	19	0.89	3.1 ± 0.02
256	✓	10	128	20	0.88	3.6 ± 0.03
256	✓	10	256	21	0.97	4.73 ± 0.03

Table A.12: Benchmark results for the Power Residue OPRF using the Mascot MPC protocol, considering only the online phase.

Prime Size	Improved	Parallel	Power Residue	Rounds	Bandwidth (MB)	Time (s)
128	✗	1	4	4	0.04	0.42 ± 0.00
128	✗	1	8	5	0.03	0.62 ± 0.00
128	✗	1	16	6	0.03	0.63 ± 0.00
128	✗	1	32	7	0.03	0.84 ± 0.01
128	✗	1	64	8	0.03	0.86 ± 0.00
128	✗	1	128	9	0.03	1.07 ± 0.00
128	✗	1	256	10	0.02	1.11 ± 0.00
128	✓	1	4	4	0.03	0.42 ± 0.00
128	✓	1	8	5	0.02	0.41 ± 0.00
128	✓	1	16	6	0.02	0.62 ± 0.00
128	✓	1	32	7	0.02	0.62 ± 0.00
128	✓	1	64	8	0.02	0.84 ± 0.00
128	✓	1	128	9	0.02	0.84 ± 0.00
128	✓	1	256	10	0.02	1.08 ± 0.01
128	✗	10	4	4	0.36	0.73 ± 0.04
128	✗	10	8	5	0.31	0.89 ± 0.01
128	✗	10	16	6	0.28	0.72 ± 0.00
128	✗	10	32	7	0.27	0.96 ± 0.00
128	✗	10	64	8	0.26	1.04 ± 0.00
128	✗	10	128	9	0.25	1.39 ± 0.01
128	✗	10	256	10	0.24	1.59 ± 0.01
128	✓	10	4	4	0.26	0.65 ± 0.00
128	✓	10	8	5	0.24	0.66 ± 0.00
128	✓	10	16	6	0.23	0.67 ± 0.00
128	✓	10	32	7	0.23	0.69 ± 0.00
128	✓	10	64	8	0.22	0.95 ± 0.00
128	✓	10	128	9	0.22	1.08 ± 0.01
128	✓	10	256	10	0.22	1.54 ± 0.01
256	✗	1	4	4	0.14	0.46 ± 0.00
256	✗	1	8	5	0.12	0.68 ± 0.00
256	✗	1	16	6	0.11	0.69 ± 0.00
256	✗	1	32	7	0.11	0.94 ± 0.00
256	✗	1	64	8	0.10	1.04 ± 0.02
256	✗	1	128	9	0.10	1.39 ± 0.02
256	✗	1	256	10	0.10	1.59 ± 0.02
256	✓	1	4	4	0.10	0.44 ± 0.00
256	✓	1	8	5	0.10	0.44 ± 0.00
256	✓	1	16	6	0.09	0.66 ± 0.00
256	✓	1	32	7	0.09	0.68 ± 0.00
256	✓	1	64	8	0.09	0.96 ± 0.02
256	✓	1	128	9	0.09	1.07 ± 0.01
256	✓	1	256	10	0.09	1.52 ± 0.00
256	✗	10	4	4	1.43	1.44 ± 0.04
256	✗	10	8	5	1.23	1.54 ± 0.05
256	✗	10	16	6	1.13	1.79 ± 0.04
256	✗	10	32	7	1.06	2.1 ± 0.06
256	✗	10	64	8	1.03	2.82 ± 0.05
256	✗	10	128	9	1.00	3.87 ± 0.02
256	✗	10	256	10	0.97	5.72 ± 0.04
256	✓	10	4	4	1.02	1.13 ± 0.04
256	✓	10	8	5	0.96	1.27 ± 0.06
256	✓	10	16	6	0.92	1.57 ± 0.01
256	✓	10	32	7	0.90	2.09 ± 0.04
256	✓	10	64	8	0.89	2.61 ± 0.07
256	✓	10	128	9	0.88	3.64 ± 0.07
256	✓	10	256	10	0.87	5.62 ± 0.02

Table A.13: Benchmark results for the Power Residue OPRF using the Semi MPC protocol, considering only the online phase.

Prime Size	Improved	Parallel	Power Residue	Rounds	Bandwidth (MB)	Time (s)
128	✗	1	4	4	0.04	0.41 ± 0.00
128	✗	1	8	5	0.03	0.61 ± 0.00
128	✗	1	16	6	0.03	0.62 ± 0.00
128	✗	1	32	7	0.03	0.83 ± 0.00
128	✗	1	64	8	0.03	0.84 ± 0.00
128	✗	1	128	9	0.03	1.07 ± 0.01
128	✗	1	256	10	0.02	1.1 ± 0.01
128	✓	1	4	4	0.03	0.41 ± 0.00
128	✓	1	8	5	0.02	0.41 ± 0.00
128	✓	1	16	6	0.02	0.62 ± 0.00
128	✓	1	32	7	0.02	0.62 ± 0.00
128	✓	1	64	8	0.02	0.84 ± 0.00
128	✓	1	128	9	0.02	0.85 ± 0.00
128	✓	1	256	10	0.02	1.08 ± 0.00
128	✗	10	4	4	0.36	0.67 ± 0.01
128	✗	10	8	5	0.31	0.87 ± 0.00
128	✗	10	16	6	0.28	0.7 ± 0.00
128	✗	10	32	7	0.27	0.94 ± 0.01
128	✗	10	64	8	0.26	1.02 ± 0.01
128	✗	10	128	9	0.25	1.3 ± 0.02
128	✗	10	256	10	0.24	1.53 ± 0.04
128	✓	10	4	4	0.26	0.65 ± 0.00
128	✓	10	8	5	0.24	0.66 ± 0.00
128	✓	10	16	6	0.23	0.68 ± 0.00
128	✓	10	32	7	0.23	0.7 ± 0.00
128	✓	10	64	8	0.22	0.96 ± 0.00
128	✓	10	128	9	0.22	1.05 ± 0.01
128	✓	10	256	10	0.22	1.41 ± 0.01
256	✗	1	4	4	0.14336	0.45 ± 0.00
256	✗	1	8	5	0.123264	0.66 ± 0.00
256	✗	1	16	6	0.11264	0.68 ± 0.01
256	✗	1	32	7	0.106496	0.92 ± 0.01
256	✗	1	64	8	0.10272	0.99 ± 0.01
256	✗	1	128	9	0.100096	1.32 ± 0.01
256	✗	1	256	10	0.09728	1.53 ± 0.01
256	✓	1	4	4	0.10	0.45 ± 0.00
256	✓	1	8	5	0.10	0.45 ± 0.00
256	✓	1	16	6	0.09	0.67 ± 0.00
256	✓	1	32	7	0.09	0.7 ± 0.00
256	✓	1	64	8	0.09	0.97 ± 0.01
256	✓	1	128	9	0.09	1.09 ± 0.01
256	✓	1	256	10	0.09	1.46 ± 0.01
256	✗	10	4	4	1.4336	1.31 ± 0.01
256	✗	10	8	5	1.23264	1.43 ± 0.04
256	✗	10	16	6	1.1264	1.58 ± 0.02
256	✗	10	32	7	1.06496	1.78 ± 0.11
256	✗	10	64	8	1.0272	2.32 ± 0.04
256	✗	10	128	9	1.00096	3.11 ± 0.07
256	✗	10	256	10	0.9728	4.31 ± 0.03
256	✓	10	4	4	1.02	1.19 ± 0.04
256	✓	10	8	5	0.96	1.21 ± 0.05
256	✓	10	16	6	0.92	1.45 ± 0.01
256	✓	10	32	7	0.90	1.68 ± 0.04
256	✓	10	64	8	0.89	2.02 ± 0.01
256	✓	10	128	9	0.88	2.7 ± 0.02
256	✓	10	256	10	0.87	3.93 ± 0.09

Table A.14: Benchmark results for the Power Residue OPRF using the Hemi MPC protocol, considering only the online phase.

Bibliography

[1] Michel Abdalla, Mario Cornejo, Anca Nitulescu, and David Pointcheval. Robust password-protected secret sharing. In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows, editors, *ESORICS 2016: 21st European Symposium on Research in Computer Security, Part II*, volume 9879 of *Lecture Notes in Computer Science*, pages 61–79, Heraklion, Greece, September 26–30, 2016. Springer, Heidelberg, Germany.

[2] Martin R. Albrecht, Alex Davidson, Amit Deo, and Daniel Gardham. Crypto dark matter on the torus: Oblivious PRFs from shallow PRFs and FHE. *Cryptology ePrint Archive*, 2023.

[3] Martin R. Albrecht, Alex Davidson, Amit Deo, and Nigel P. Smart. Round-optimal verifiable oblivious pseudorandom functions from ideal lattices. In Juan Garay, editor, *PKC 2021: 24th International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 12711 of *Lecture Notes in Computer Science*, pages 261–289, Virtual Event, May 10–13, 2021. Springer, Heidelberg, Germany.

[4] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[5] Andrea Basso. A post-quantum round-optimal oblivious PRF from isogenies. 2023. https://eprint.iacr.org/2023/225.

[6] Andrea Basso, Péter Kutas, Simon-Philipp Merz, Christophe Petit, and Antonio Sanso. Cryptanalysis of an oblivious prf from supersingular isogenies. In *Advances in Cryptology–ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Informa-*

tion Security, Singapore, December 6–10, 2021, Proceedings, Part I 27, pages 160–184. Springer, 2021.

[7] Carsten Baum, Tore Frederiksen, Julia Hesse, Anja Lehmann, and Avishay Yanai. Pesto: proactively secure distributed single sign-on, or how to trust a hacked server. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 587–606. IEEE, 2020.

[8] Daniel J Bernstein and Tanja Lange. Post-quantum cryptography. *Nature*, 549(7671):188–194, 2017.

[9] Ward Beullens, Tim Beyne, Aleksei Udovenko, and Giuseppe Vitto. Cryptanalysis of the Legendre PRF and generalizations. *IACR Transactions on Symmetric Cryptology*, 2020(1):313–330, 2020.

[10] Ward Beullens and Cyprien Delpech de Saint Guilhem. LegRoast: Efficient post-quantum signatures from the Legendre PRF. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, pages 130–150, Paris, France, April 15–17, 2020. Springer, Heidelberg, Germany.

[11] Dan Boneh, Yuval Ishai, Alain Passelègue, Amit Sahai, and David J. Wu. Exploring crypto dark matter: New simple PRF candidates and their applications. In *Theory of Cryptography Conference*, pages 699–729. Springer, 2018.

[12] Dan Boneh, Dmitry Kogan, and Katharine Woo. Oblivious pseudorandom functions from isogenies. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020, Part II*, volume 12492 of *Lecture Notes in Computer Science*, pages 520–550, Daejeon, South Korea, December 7–11, 2020. Springer, Heidelberg, Germany.

[13] Niklas Büscher, Daniel Demmler, Nikolaos P. Karvelas, Stefan Katzenbeisser, Juliane Krämer, Deevashwer Rathee, Thomas Schneider, and Patrick Struck. Secure two-party computation in a quantum world. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *ACNS 20: 18th International Conference on Applied Cryptography and Network Security, Part I*, volume 12146 of *Lecture Notes in Computer Science*, pages 461–480, Rome, Italy, October 19–22, 2020. Springer, Heidelberg, Germany.

[14] Jan Camenisch and Anja Lehmann. Privacy-preserving user-auditable pseudonym systems. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 269–284. IEEE, 2017.

[15] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press.

[16] Sílvia Casacuberta, Julia Hesse, and Anja Lehmann. SoK: Oblivious pseudorandom functions. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 625–646. IEEE, 2022.

[17] Dario Catalano, Ronald Cramer, Giovanni Di Crescenzo, Ivan Darmgård, David Pointcheval, Tsuyoshi Takagi, Ronald Cramer, and Ivan Damgård. Multiparty computation, an introduction. *Contemporary cryptology*, pages 41–87, 2005.

[18] Melissa Chase and Peihan Miao. Private set intersection in the internet setting from lightweight oblivious PRF. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part III*, volume 12172 of *Lecture Notes in Computer Science*, pages 34–63, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany.

[19] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPDZ2k: Efficient MPC mod 2^k for dishonest majority. Cryptology ePrint Archive, Report 2018/482, 2018. https://eprint.iacr.org/2018/482.

[20] Ivan Damgård. On the randomness of Legendre and Jacobi sequences. In Shafi Goldwasser, editor, *Advances in Cryptology – CRYPTO'88*, volume 403 of *Lecture Notes in Computer Science*, pages 163–172, Santa Barbara, CA, USA, August 21–25, 1990. Springer, Heidelberg, Germany.

[21] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264, Santa Barbara, CA, USA, August 17–21, 2003. Springer, Heidelberg, Germany.

[22] Ivan Damgård and Claudio Orlandi. Multiparty computation for dishonest majority: From passive to active security at low cost. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 558–576, Santa Barbara, CA, USA, August 15–19, 2010. Springer, Heidelberg, Germany.

[23] Gareth T Davies, Sebastian Faller, Kai Gellert, Tobias Handirk, Julia Hesse, Máté Horváth, and Tibor Jager. Security analysis of the what-

sapp end-to-end encrypted backup protocol. *Cryptology ePrint Archive*, 2023.

[24] Emiliano De Cristofaro and Gene Tsudik. Experimenting with fast private set intersection. In *International Conference on Trust and Trustworthy Computing*, pages 55–73. Springer, 2012.

[25] Itai Dinur, Steven Goldfeder, Tzipora Halevi, Yuval Ishai, Mahimna Kelkar, Vivek Sharma, and Greg Zaverucha. MPC-friendly symmetric cryptography from alternating moduli: Candidates, protocols, and applications. In *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part IV 41*, pages 517–547. Springer, 2021.

[26] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 823–852, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany.

[27] Sebastian Faller, Astrid Ottenhues, and Johannes Ernst. Composable oblivious pseudo-random functions via garbled circuits. *Cryptology ePrint Archive*, 2023.

[28] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *TCC 2005: 2nd Theory of Cryptography Conference*, volume 3378 of *Lecture Notes in Computer Science*, pages 303–324, Cambridge, MA, USA, February 10–12, 2005. Springer, Heidelberg, Germany.

[29] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 1986.

[30] Lorenzo Grassi, Christian Rechberger, Dragos Rotaru, Peter Scholl, and Nigel P. Smart. MPC-friendly symmetric key primitives. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 430–443, Vienna, Austria, October 24–28, 2016. ACM Press.

[31] Carmit Hazay and Yehuda Lindell. A note on the relation between the definitions of security for semi-honest and malicious adversaries. *Cryptology ePrint Archive*, 2010.

[32] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. *Journal of Cryptology*, 33(4):1732–1786, 2020.

[33] Lena Heimberger, Fredrik Meisingseth, and Christian Rechberger. Oprfs from isogenies: Designs and analysis. *Cryptology ePrint Archive*, 2023.

[34] Dennis Hofheinz and Jörn Müller-Quade. Universally composable commitments using random oracles. In Moni Naor, editor, *TCC 2004: 1st Theory of Cryptography Conference*, volume 2951 of *Lecture Notes in Computer Science*, pages 58–76, Cambridge, MA, USA, February 19–21, 2004. Springer, Heidelberg, Germany.

[35] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014, Part II*, volume 8874 of *Lecture Notes in Computer Science*, pages 233–253, Kaoshiung, Taiwan, R.O.C., December 7–11, 2014. Springer, Heidelberg, Germany.

[36] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 276–291. IEEE, 2016.

[37] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17: 15th International Conference on Applied Cryptography and Network Security*, volume 10355 of *Lecture Notes in Computer Science*, pages 39–58, Kanazawa, Japan, July 10–12, 2017. Springer, Heidelberg, Germany.

[38] Stanislaw Jarecki, Hugo Krawczyk, and Jason Resch. Updatable oblivious key management for storage systems. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 379–393, 2019.

[39] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 456–486, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany.

[40] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. On the (in)security of the diffie-hellman oblivious PRF with multiplicative blinding. In Juan Garay, editor, *PKC 2021: 24th International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 12711 of *Lecture Notes in Computer Science*, pages 380–409, Virtual Event, May 10–13, 2021. Springer, Heidelberg, Germany.

[41] Novak Kaluđerović, Thorsten Kleinjung, and Dusan Kostic. Improved key recovery on the legendre PRF. *Cryptology ePrint Archive*, 2020.

[42] Panos Kampanakis and Tancrède Lepoint. Do we need to change some things? open questions posed by the upcoming post-quantum migration to existing standards and deployments. *Cryptology ePrint Archive*, 2023.

[43] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[44] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 830–842, Vienna, Austria, October 24–28, 2016. ACM Press.

[45] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 158–189, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany.

[46] Dmitry Khovratovich. Key recovery attacks on the Legendre PRFs within the birthday bound. *Cryptology ePrint Archive*, 2019.

[47] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 818–829, Vienna, Austria, October 24–28, 2016. ACM Press.

[48] Luciano Maino and Chloe Martindale. An attack on sidh with arbitrary starting curve. *Cryptology ePrint Archive*, 2022.

[49] Vasileios Mavroeidis, Kamer Vishi, Mateusz D Zych, and Audun Jøsang. The impact of quantum computing on present cryptography. *arXiv preprint arXiv:1804.00200*, 2018.

[50] Omid Mir, Michael Roland, and René Mayrhofer. Decentralized, privacy-preserving, single sign-on. *Security and Communication Networks*, 2022:1–18, 2022.

[51] Moni Naor and Omer Reingold. On the construction of pseudo-random permutations: Luby-Rackoff revisited (extended abstract). In *29th Annual ACM Symposium on Theory of Computing*, pages 189–199, El Paso, TX, USA, May 4–6, 1997. ACM Press.

[52] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. *Journal of the ACM*, 51(2):231–262, 2004.

[53] National Institute of Standards and Technology. NIST three draft fips for post-quantum cryptography. 2023. `https://csrc.nist.gov/news/2023/three-draft-fips-for-post-quantum-cryptography`, Accessed: 2023-08-29.

[54] István András Seres, Máté Horváth, and Péter Burcsi. The legendre pseudorandom function as a multivariate quadratic cryptosystem: security and applications. *Applicable Algebra in Engineering, Communication and Computing*, pages 1–31, 2023.

[55] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[56] Wim van Dam and Sean Hallgren. Efficient quantum algorithms for shifted quadratic character problems. *arXiv preprint quant-ph/0011067*, 2000.

[57] Xiaoyi Yang, Yanqi Zhao, Sufang Zhou, and Lianhai Wang. A lightweight delegated private set intersection cardinality protocol. *Computer Standards & Interfaces*, 87:103760, 2024.