A Fully-Adaptive Threshold Partially-Oblivious PRF

Ruben Baecker[1], Paul Gerhart[2], Daniel Rausch[3], and Dominique Schröder[2,1]

[1] Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany
`ruben.baecker@fau.de`
[2] TU Wien, Vienna, Austria
`{paul.gerhart, dominique.schroeder}@tuwien.ac.at`
[3] University of Stuttgart, Stuttgart, Germany
`daniel.rausch@sec.uni-stuttgart.de`

Abstract. Oblivious Pseudorandom Functions (OPRFs) are fundamental cryptographic primitives essential for privacy-enhancing technologies such as private set intersection, oblivious keyword search, and password-based authentication protocols. We present the first fully adaptive, partially oblivious threshold pseudorandom function that supports proactive key refresh and provides composable security under the One-More Gap Diffie-Hellman assumption in the random oracle model.
Our construction is secure with respect to a new ideal functionality for OPRFs that addresses three critical shortcomings of previous models– specifically, key refresh and non-verifiability issues that rendered them unrealizable. In addition, we identify a gap in a prior work's proof of partial obliviousness and develop a novel proof technique to salvage their scheme.

Keywords: Partially-Oblivious PRF · Threshold Cryptography · Adaptive Security · Universal Composability.

1 Introduction

An Oblivious Pseudorandom Function (OPRF) can be thought of as a secure two-party computation of a pseudorandom function. In this setup, one party, P_1, holds the private key k of the pseudorandom function, while the second party, P_2, holds the input x. At the end of the protocol, P_2 learns the output of the function $f(k, x)$, but nothing else about k. P_1 remains oblivious to x and $f(k, x)$.

OPRFs are well-studied and continue to attract attention due to their critical role in privacy-enhancing technologies. They are integral to primitives such as private set intersection, oblivious keyword search, private information retrieval, and password-based authentication protocols, including Password-Authenticated Secret Sharing, Password-Authenticated Key Exchange, and Single Sign-On systems. In addition to its theoretical significance, OPRFs play a crucial role in practical protocols. A notable example is the Password-Authenticated Key Exchange (PAKE) protocol, OPAQUE [20], which is currently in the process of being standardized [3]. It has the potential to become the standard for password-based authentication on the internet, replacing the current standard, password-over-TLS. Currently, it is already deployed to billions of people on WhatsApp in their end-to-end encrypted backup protocol [31, 12].

1.1 Primitives–Generalizing Oblivious PRFs

Extensive research in the area of OPRFs has led to the development of new variants of OPRFs and protocols [19, 21, 22, 20, 17, 2, 10], enabling diverse applications and achieving novel security properties:

Partially OPRFs. A *partially* oblivious PRF (pOPRF) [14] allows for the disclosure of a portion of the input while still maintaining the confidentiality of the remaining input. pOPRFs provide more granular control over how information is shared, supporting new applications while maintaining strong security guarantees. A key example is password-based authentication protocols [14, 2], which utilize partial input leakage to track login attempts per user by disclosing the username while still safeguarding the user's password [2].

Threshold OPRFs. A t-out-of-n Threshold Oblivious Pseudorandom Function (tOPRF) is a multi-server variant where the key is distributed among n servers. At least t servers must cooperate to compute the PRF. The scheme remains secure as long as no more than $t-1$ servers are compromised. Consequently, no single server can compute the PRF independently. A Distributed Oblivious Pseudorandom Function (dOPRF) is a special case of tOPRF where $t = n$, meaning that all n servers are required to compute the PRF together.

OPRFs with Proactive Key Refresh. Some of the latest protocols incorporate more robust security measures that are designed to maintain security even in the event of a server corruption. The basic concept behind this approach is "proactive key refresh", which involves replacing old keys with new ones to prevent them from being exploited by malicious actors. After a key refresh, a corrupted server is considered honest again. Supporting key refresh is best practice in industries [16, 26, 30] and also required by certain standards, such as PCI DSS [28], and a NIST special publication [1].

1.2 Constructions–The 2HashDH OPRF family

Jarecki *et al.* introduced the 2HashDH family of OPRFs in a series of influential papers (AC'14, EuroS&P'16, ACNS'17, EC'18, AC'24). The constructions are secure within the Universal Composability (UC) framework in the Random Oracle Model (ROM), based on the one-more gap Diffie-Hellman (OM-gapDH) assumption [19, 21, 22, 20, 17]. 2HashDH and its variants serve as essential components in various cryptographic protocols, e.g., [21, 22, 20, 11, 2, 10, 17]. Baum *et al.* (EuroS&P'20) extended the research on OPRFs by proposing a *distributed* pOPRF based on 2HashDH, which is secure against fully-adaptive corruptions [2].

While *distributed* and *threshold* PRFs share some similarities, they highlighted the difficulty of developing a threshold version of their design, identifying it as a significant open problem. This challenge was further emphasized by Casacuberta *et al.* (EuroS&P'22), who recognized threshold pOPRFs secure against fully-adaptive corruptions as a crucial missing component in the landscape of 2HashDH-based constructions [9]. This remains one of the key unresolved issues in the field.

Beyond its theoretical importance, a fully adaptive threshold pOPRF would provide an important basis for adapting existing protocols to a fully-adaptive threshold setting while supporting proactive key refresh. Two prime examples are OPAQUE and PESTO. OPAQUE is a Password-Authenticated Key Exchange protocol [20] that is currently getting standardized [3] and that is deployed to billions of people in the WhatsApp end-to-end encrypted backup protocol [31, 12]. Gu *et al.* already extended OPAQUE to the threshold setting [17] but left fully-adaptive security in combination with proactive key refresh as an open problem. PESTO is a distributed single sign-on scheme [2] that requires the participation of all servers for successful execution, meaning that the unavailability of a single server brings the entire system to a halt. Adapting PESTO to a thresholding variant would greatly improve its robustness and usability, allowing the system to continue functioning even if some servers are offline.

1.3 Our Contributions

Our contribution is threefold (Table 1 provides an overview of which papers are affected by our findings):

Construction: Our primary contribution is the development of the first fully adaptive, partially-obvious threshold pseudorandom function. This construction supports proactive key refresh and is provably secure in a Universal Composability (UC) model [8, 7, 25, 24, 6, 18] under the One-More Gap Diffie-Hellman (OM-B-gapDH) assumption in the ROM.

Foundations: We introduce a new formalization of a UC ideal functionality for oblivious pseudorandom functions (OPRFs), identifying three shortcomings in previous UC ideal functionalities [22, 2]. While one of these problems has been recently and independently discovered [17], the other two have yet to be addressed. These two gaps concern the modeling of key refresh and the fact that the OPRF is not verifiable. The first gap, concerning key refresh, is particularly significant since the current formalization is not realizable. In response, we present the first realizable formalization of an OPRF that supports proactive key refresh.

Technical insights: From a technical point of view, we identify a gap in the proof of partial obliviousness in the constructions of Baum *et al.* [2] and Gu *et al.* [17]. The authors claim that a certain event occurs with negligible probability since its occurrence would otherwise violate the OM-gapDH assumption. However, we show how this event can be triggered without giving the simulator any advantage in breaking the assumption. We explore several approaches to salvaging their construction. Adding a general-purpose non-interactive zero-knowledge proof would solve the problem but at the cost of reduced efficiency. Alternatively, a proof in the Algebraic Group Model (AGM) [15] could solve the problem, but this introduces a stronger trust assumption, along with recent concerns about the soundness of the model [32]. Instead, we introduce a novel proof technique–Domain-Isolating Oracle Programming (DIOP)–that preserves the original construction while establishing its security. DIOP may be of independent interest.

1.4 Related Work

A graphical comparison to related work is shown in Table 1. Oblivious Pseudorandom Functions (OPRFs) have been studied extensively, also in the context of universal composability. Jarecki et al. [19] introduced a verifiable OPRF, but did not achieve partial obliviousness or threshold functionality. Subsequent work by Jarecki et al. [21, 22] continued this line of research; however, [22] contains a gap in the model regarding threshold functionality. Baum et al. [2] claimed a partially oblivious OPRF, but their proof contains a gap regarding partial obliviousness, and their model suffers from labeling and key refresh problems, rendering their ideal functionality unrealizable. Das et al. [10] presented a distributed ($t = n$), partially oblivious OPRF, but lacked full adaptivity and threshold capabilities. Gu et al. [17] provided a thresholding OPRF with partial obliviousness and full adaptivity, but suffers from the same proof gap as [2] and has no support for proactive key refresh.

Comparison to other Blinding Techniques Looking forward, we use a blinding technique to achieve fully-adaptiveness while supporting proactive key refreshes. Camenisch *et al.* also used pairwise blinding values to make an evaluation look independent from key shares [5]. The crucial distinction between our technique and their approach is that in their scheme, the response of every server is necessary for the blinding values to cancel each other out. Therefore, it is only applicable to the distributed ($t = n$) but not to the threshold setting. Baum *et al.* reused the approach of Camenisch *et al.* in [2].

Gu *et al.* use blinding values that cancel each other out to avoid mixing of server responses of different protocol sessions [17]. They use Shamir's secret sharing of zero in the exponent, resulting in the neutral element when combined. Even though their solution works in the threshold setting, it does not suffice for realizing security under fully-adaptive corruption with proactive key refresh. This is because the uniformly random responses cannot be made consistent with a leaked secret key in hindsight.

In independent and concurrent work, Katsumata *et al.* [23] discovered that the blinding technique introduced in [27]–which is essentially the same as ours–can be used to realize fully-adaptive threshold signatures.

Table 1: Comparison of 2HashDH constructions that are proven secure within the UC framework. ○ denotes that a paper claims this property but has a gap regarding the property, and ● denotes that a paper claims and achieves the property.

		JKK14 [19]	JKKX16 [21]	JKKX17 [22]	DGSTV18 [11]	BFHLY20 [2]	DHL22 [10]	GJKNX24 [17]	our work
Properties	Partially oblivious					○	●	○	●
	Verifiable	●			●				
	Threshold			○				●	●
	Distributed ($t = n$)					●	●		
	Proactive key refresh					●			●
	Fully-adaptive					○		●	●
Gaps	Proof gap: partial obliviousness					×		×	
	Model gap: labels					×			
	Model gap: threshold			×					
	Model gap: key refresh					×			

1.5 Road Map and Notation

In Section 2, we provide a technical overview, guiding through the main concepts and contributions of the paper. In Section 3, we provide the necessary background for understanding our work. Section 4 describes the ideal functionality $\mathcal{F}_{\mathrm{tpOPRF}}$. In Section 5, we describe our construction in detail. The main concepts of our proof are explained in Section 6, while the full definition of the simulator and the complete proof are deferred to the full version of the paper.

Notations. Let $\lambda \in \mathbb{N}$ be the security parameter and let ϵ be any negligible function of λ. We denote the set $\{1, \dots, n\}$ by $[n]$. Let $r \leftarrow\$ \mathcal{S}$ denote a uniformly random sample of an element r from a set $\mathcal{S}$. Let $\mathbb{G}$ be an additive cyclic group of prime order q with generator 1. We use the implicit representation of group elements as introduced in [13]: For $a \in \mathbb{Z}_q^*$, we define $[a] = a \cdot [1] \in \mathbb{G}$ as the implicit representation of a in $\mathbb{G}$. Let $\mathbb{BG} = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, q, [1]_1, [1]_2, e)$ be an asymmetric bilinear group, where $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are cyclic groups of order q. The generators of $\mathbb{G}_1$ and $\mathbb{G}_2$ are $[1]_1$ and $[1]_2$ respectively. For $s \in \{1, 2, T\}$ and $a \in \mathbb{Z}_q^*$, we define $[a]_s = a \cdot [1]_s \in \mathbb{G}_s$ as the implicit representation of a in $\mathbb{G}_s$. The function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is an efficiently computable (non-degenerate) bilinear map and because of the bilinearity, we get $e([a]_1, [b]_2) = [a \cdot b]_t$. Note that from $[a] \in \mathbb{G}$ it is generally hard to compute the value a (discrete logarithm problem in $\mathbb{G}$).

2 Technical Overview

This section provides an overview of our major technical contributions, resulting in the first fully adaptive, UC-secure, threshold partially oblivious PRF. In Section 2.1, we gradually extend the standard $\mathsf{2HashDH}$ OPRF to support the threshold setting, proactive key refresh, partial obliviousness, and full adaptive security. In Section 2.2, we formalize an ideal functionality that incorporates these features, identifying three gaps in previous OPRF models. One gap has recently been independently identified [17], but the other two remain unaddressed. Finally, in Section 2.3, we describe our security proof and expose a gap in previous pOPRF proofs where a critical event was incorrectly considered negligible. We show how this event can be triggered without violating the $\mathsf{OM\text{-}gapDH}$ assumption and solve the problem using a novel proof technique that may be of independent interest.

2.1 A Fully-Adaptive Threshold Partially-Oblivious PRF

The primary goal of our work is to construct the first fully adaptive, UC-secure, threshold partially oblivious PRF. In this section, we outline our approach. Starting with the $\mathsf{2HashDH}$ OPRF protocol, we introduce several modifications to incorporate the necessary properties, step by step, leading to the final tpOPRF scheme.

Starting Point: The $\mathsf{2HashDH}$ OPRF [19]. Let $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{G}$ and $\mathsf{H}_f : \{0,1\}^* \to \{0,1\}^\lambda$ be two hash functions. The $\mathsf{2HashDH}$ OPRF computes the following function:

$$F_k(x) = \mathsf{H}_f(x, k \cdot \mathsf{H}_1(x)),$$

where the value x is the client input, and the key k is the server input. The server and client execute an interactive protocol to evaluate the function in a fully oblivious manner. In the first step, the client computes the hash $\mathsf{H}_1(x)$ of its input x, blinds it by multiplying it with a random scalar r, and sends the blinded value $[p] = r \cdot \mathsf{H}_1(x)$ to the server. The server multiplies its key k by the blinded value to get $[o] = k \cdot [p]$ and returns it to the client. The client then unblinds it by multiplying it by the inverse of r, resulting in $k \cdot \mathsf{H}_1(x)$. Finally, the client calculates the output as $F_k(x) = \mathsf{H}_f(x, k \cdot \mathsf{H}_1(x))$. This protocol is fully oblivious since the server only learns a blinded value. Pseudorandomness follows by modeling the hash functions as random oracles.

Extending $\mathsf{2HashDH}$ into the Threshold Setting. We follow the approach of Jarecki *et al.* [22] using Shamir's secret sharing to support threshold security. The underlying PRF remains $F_k(x) = \mathsf{H}_f(x, k \cdot \mathsf{H}_1(x))$, but the key k is shared among n servers, each holding a key share k_i. Any subset of t servers can jointly reconstruct k. To evaluate $F_k(x)$, the client follows the $\mathsf{2HashDH}$ protocol, but interacts with t servers in parallel. Each server provides a partial evaluation,

$\delta_{Sset,i} \cdot k_i \cdot \mathsf{H}_1(x)$, which includes the server-set-specific Lagrange coefficient. The client sums these partial evaluations to obtain $k \cdot \mathsf{H}_1(x)$, then computes the final result as $F_k(x) = \mathsf{H}_f(x, k \cdot \mathsf{H}_1(x))$.

The threshold extension improves both the security and the robustness of the OPRF. Security is improved because an adversary must still interact with at least one honest server to compute the OPRF, even if up to $t-1$ servers are corrupt. Robustness is increased by distributing the server's private key across multiple servers, allowing the OPRF to be computed even if some servers are offline.

Adding Proactive Key Refresh. Threshold security improves both security and robustness, but leaves the problem of compromised servers: as more servers are corrupted, security degrades to the point where it becomes equivalent to single-server security when $t-1$ servers are compromised.

Key refresh mitigates this problem by allowing servers to update their shared key material, resetting corruptions, and maintaining OPRF functionality. During key refresh, *the same key* k is reshared so that PRF evaluations remain consistent, but the refreshed key shares are incompatible with any previously corrupted shares.

In the key refresh protocol, each server shares the integer zero using Shamir secret sharing and sends each resulting share to one of the other servers. Each server then updates its key by adding the shares received from the other servers to its old key. Since only zeros are added, the overall key remains unchanged, but previous key shares become invalid, effectively resetting any corruption and separating the pre- and post-refresh states.

The rationale for not rotating the PRF key k is as follows. With n parties and a threshold of t, consider any period between key refreshes. If the adversary corrupts fewer than t servers (as required by the security definition), he cannot gain sufficient information about k. This is even true in an information-theoretic sense, due to Shamir's secret sharing [29]. Key refresh renders previous shares irrelevant, since they are independent of new shares. Thus, rotation of k is unnecessary. If the adversary corrupts more than t servers, security is compromised, and no cryptographic approach can protect the key. The approach of keeping the underlying key constant has been used in many previous works, such as [5, 22, 2, 17].

Handling Fully-Adaptive Corruptions. We focus on *fully-adaptive adversaries*, which can corrupt servers at any time. This contrasts with the weaker semi-adaptive model, where the adversary must declare the corrupted servers before each key refresh [4]. The challenge in our setting is that standard guessing strategies, such as those used by Gu *et al.* [17] to prove security against fully-adaptive corruption without key refresh, do not easily extend to the key refresh scenario.

In a single period, the simulator has a non-negligible chance of correctly guessing which servers the adversary will corrupt. However, since the number of key refreshes is polynomial in the security parameter, this leads to an exponential increase in loss. As a result, the probability of correctly guessing the corruptions for each key refresh becomes negligible, making guessing strategies ineffective.

Our Approach. Our construction borrows ideas from Camenisch *et al.* [5], but extends them to the threshold setting. To illustrate the main ideas, we start with the observation that fully adaptive security can be achieved if the response of each honest server appears uniformly random and independent of k_i. In addition, the threshold t ensures that the result will be consistent with the PRF key only if t of the responses are combined. This observation forms the basis of our security proof. In Section 2.3, we describe the proof strategy used to establish security under fully-adaptive corruptions.

We introduce an additive blinding value $[\beta_i]$ to make the distribution of the server's response uniformly random and therefore independent of k_i. Now, the server's response is formed as

$$[o_i] \leftarrow \delta_{Sset,i} \cdot k_i \cdot [p] + [\beta_i].$$

To ensure that the sum of all $[o_i]$ still results in $k \cdot [p]$, we carefully craft the blinding values, such that they cancel each other out when summing up the partial evaluations. In more detail, we use a hash function H_b, to compute the blinding value $[\beta_i]$ as the sum of pairwise blinding values $[\beta_{i,j}] \leftarrow \Delta_{i,j} \cdot \mathsf{H}_b(\mathsf{s}_{i,j}, Sset, [p]_1)$ with Δ being the Kronecker delta ($\Delta_{i,j} = 1$ if $i > j$, and $\Delta_{i,j} = -1$ else). The seeds $\mathsf{s}_{i,j}$ are shared values per server pair ($\mathsf{s}_{i,j} = \mathsf{s}_{j,i}$) which leads to

$$\begin{aligned}\sum_{i \in Sset} [\beta_i] &= \sum_{i \in Sset} \sum_{j \in Sset \setminus \{i\}} [\beta_{i,j}] = \sum_{i \in Sset} \sum_{j \in Sset \setminus \{i\}} \Delta_{i,j} \cdot \mathsf{H}_b(\mathsf{s}_{i,j}, \dots) \\ &= \sum_{i \in Sset} \sum_{j \in Sset; j<i} \Delta_{i,j} \cdot \mathsf{H}_b(\mathsf{s}_{i,j}, \dots) + \Delta_{j,i} \cdot \mathsf{H}_b(\mathsf{s}_{j,i}, \dots) \\ &= \sum_{i \in Sset} \sum_{j \in Sset; j<i} \mathsf{H}_b(\mathsf{s}_{i,j}, \dots) - \mathsf{H}_b(\mathsf{s}_{i,j}, \dots) = 0,\end{aligned}$$

such that the blinding values do not influence the final computation when they are all summed up.

Realizing Partial Obliviousness. The final step is to convert the threshold oblivious PRF into a partially oblivious one. The basic idea here is to split the client's input x into two parts: (x_{priv}, x_{pub}). The server remains oblivious to x_{priv}, while x_{pub} is revealed to the server. To bind both inputs to the PRF evaluation, we follow the approach of Everspaugh *et al.* [14] and use bilinear pairings in our pOPRF construction. To illustrate the idea, we start with the 2HashDH construction:

$$F_k(x) = \mathsf{H}_f(x, k \cdot \mathsf{H}_1(x)),$$

but replace the evaluation of the hash function $\mathsf{H}_1(x)$ with $e(H_1(x_{priv}), H_2(x_{pub}))$ and include both messages x_{priv} and x_{pub} as input to the final hash. The hash function H_1 now maps arbitrary bitstrings to the first base group $\mathbb{G}_1$ instead of $\mathbb{G}$

as we are operating in a pairing-friendly group now. Furthermore, we introduce a second hash function H_2 that maps arbitrary bitstrings to the second base group $\mathbb{G}_2$. This leads to the updated PRF

$$F_k(x_{priv}, x_{pub}) = \mathsf{H}_f(x_{priv}, x_{pub}, k \cdot e(H_1(x_{priv}), H_2(x_{pub}))).$$

We omit the threshold setting for better readability for the sake of this overview. Yet, all the presented techniques for a fully adaptive t-out-of-n solution carry over to this modification.

The linearity of the bilinear pairing allows the client to send the private input blinded as before, $[p]_1 \leftarrow r \cdot H_1(x_{priv})$, while sending the public input in the clear. Upon receiving the blinded private input and the public input, the server can combine them by computing the bilinear pairing of $[p]_1$ and $H_2(x_{pub})$. Multiplying this result by the key k yields

$$\begin{aligned}[o]_t = k \cdot e([p]_1, H_2(x_{pub})) &= k \cdot e(r \cdot H_1(x_{priv}), H_2(x_{pub})) \\ &= k \cdot r \cdot e(H_1(x_{priv}), H_2(x_{pub})).\end{aligned}$$

The server sends $[o]_t$ to the client, which can de-blind $[o]_t$ as before to obtain the evaluation $k \cdot e(H_1(x_{priv}), H_2(x_{pub}))$. Finally, the client computes the PRF value by evaluating $\mathsf{H}_f(x_{priv}, x_{pub}, k \cdot e(H_1(x_{priv}), H_2(x_{pub})))$.

2.2 Universal Composability: An Ideal Functionality $\mathcal{F}_{\mathbf{OPRF}}$

In this section, we define a UC functionality $\mathcal{F}_{\mathrm{OPRF}}$, starting with the single-server setting. We then extend the model to handle the threshold setting and key refresh. Lastly, we address the challenge of modeling the non-verifiability of the OPRF. Throughout, we identify three gaps in existing ideal functionalities [19, 9] and propose solutions to resolve them. The first issue regarding the threshold setting was independently identified and resolved by Gu *et al.* [17], the other two remain open.

A Starting Point: $\mathcal{F}_{\mathbf{OPRF}}$. To simplify the discussion in this section, we focus on a streamlined version of the $\mathcal{F}_{\mathrm{OPRF}}$ functionality from [19, 9], illustrated in Figure 1. We shall concentrate on three core properties:

1. the output of $F_k(x)$ must be indistinguishable from that of a random function,
2. the server learns nothing about the input x or the output $F_k(x)$,
3. each execution reveals only a single function evaluation.

The functionality has three interfaces. `Eval` is called by a client to initiate an evaluation request. It takes an input x, stores the request in a queue, and informs the adversary $\mathcal{A}$ of the request. The input x is not leaked to $\mathcal{A}$, which models the obliviousness. `ServerComplete` is called by $\mathcal{A}$ to indicate that the server is willing to participate in an evaluation. The server is informed, and the "ticket counter" is incremented. `ClientComplete` is called by $\mathcal{A}$ to indicate that the evaluation request was successful and that the function value should be returned to the client. The ideal functionality checks if the ticket counter is

The functionality assigns random values $F_S(x) \leftarrow\!\!\$ \{0,1\}^\lambda$ for yet undefined $F_S(x)$.

On $(\texttt{Eval}, x)$ from any client C, do:
- record (C, S, x).
- send $(\texttt{Eval}, C, S)$ to $\mathcal{A}$.

On $(\texttt{ServerComplete}, S)$ from $\mathcal{A}$, do:
- send `ServerComplete` to S;
- increment $tickets(S) + +$.

On `ClientComplete` from $\mathcal{A}$, do:
- if $tickets(S) \leq 0$ ignore the request, otherwise proceed.
- Retrieve record (C, S, x);
- decrement $tickets(S) - -$;
- output $F_S(x)$ to C.

Fig. 1: Simplified UC functionality $\mathcal{F}_{\text{OPRF}}$ [19, 9].

positive to ensure that a server interaction has taken place. It takes the request from the queue and decrements the ticket counter. $\mathcal{F}_{\text{OPRF}}$ returns the function value to the client that it obtains from a table. If the table has no entry for the entered input value, a fresh value is sampled uniformly at random, modeling the indistinguishability from a truly random function. As we have seen, the ticket counter is increased per response from the server and decremented per function value that the client learns. This models the requirement that a client only learns a single function value per server interaction.

Extending $\mathcal{F}_{\text{OPRF}}$ to the Threshold Setting. Compared to the single server setting, an ideal functionality now has to take care of multiple servers. Thus, the first modification is the introduction of individual ticket counters per server, which is incremented when that specific server is involved in an evaluation protocol. While increasing the ticket counter seems intuitive, the question arises of which set of t ticket counters should be decremented once a client learns a function value, i.e., `ClientComplete` is called. To answer this question, we first explain the approach of [22] and explain why it fails. Afterward, we present a solution to the problem.

The Approach of [22]. To decrease the ticket counter when `ClientComplete` is called, the ideal functionality of [22] chooses any t servers with $tickets(S_i) > 0$. Unfortunately, it is not specified exactly which t servers are chosen. We take "choosing the servers at random" as an example to show why this is problematic. We demonstrate the problem with a simple example in which we consider a 2-out-of-3 tOPRF. A sequence of two `Eval` queries (to servers (S_1, S_2) and (S_2, S_3)) and the corresponding four `ServerComplete` queries leads to the following ticket counters:

$$tickets(S_1) = 1, \quad tickets(S_2) = 2, \quad tickets(S_3) = 1$$

A `ClientComplete` query results in one of the following configurations, each with probability $\frac{1}{3}$:

$$tickets(S_1) = 0, \quad tickets(S_2) = 1, \quad tickets(S_3) = 1 \tag{1}$$
$$tickets(S_1) = 0, \quad tickets(S_2) = 2, \quad tickets(S_3) = 0 \tag{2}$$
$$tickets(S_1) = 1, \quad tickets(S_2) = 1, \quad tickets(S_3) = 0 \tag{3}$$

Note that in configuration (2), only server S_2 has tickets left. Therefore, another `ClientComplete` query leads to a different outcome for configuration (2) in comparison to the other configurations. $\mathcal{F}_{\text{tOPRF}}$ ignores the query for configuration (2) because not enough servers have positive ticket counters. In other words, the ideal functionality might, by chance, not answer legitimate requests.

The Solution. The natural solution to the problem is to let the adversary decide which servers were involved in the computation of a certain value. The `ClientComplete` interface is extended to take a server set as input, and the ideal functionality decrements the ticket counters of those servers. As a consequence, the simulator in a proof now has the responsibility of monitoring server responses and associating them with an evaluation [17].

Extending $\mathcal{F}_{\text{tOPRF}}$ to Support Key Refresh We extend $\mathcal{F}_{\text{tOPRF}}$ to support server de-corruption through key refresh. This introduces challenges in managing ticket counters, as handling residual tickets from prior `ServerComplete` operations (before key refresh and de-corruption) is non-trivial.

The Approach of [2]. The strategy of the ideal functionality of [2] is to set all remaining ticket counters to zero once a key refresh happens. At first glance, this strategy seems reasonable, as results obtained from the computation with old keys should be of no relevance in the current epoch with updated keys. Indeed, this intuition holds for each protocol run, in which only some `ServerComplete` were sent. Since there are missing `ServerComplete` calls, after key refresh, the client has no way to obtain the missing responses, as the uncorrupted keys consistent with the responses from before the key refresh are erased. Thus, it seems that the ticket counters can be safely set to zero. However, the intuition does not hold in the situation where all `ServerComplete` were already sent before the key refresh, but the `ClientComplete` was not called yet. In this case, the client obtained all the information that requires the private state of the servers. Moreover, the combination of this obtained information is independent of any epoch, as the combined key stays constant throughout the key refresh. Consequently, a client should be able to use this epoch-independent information in any epoch. Still, the ideal functionality will deny `ClientComplete` as all ticket counters are set to zero. Modeling key refresh as in [2] lets the environment distinguish a protocol in the real world from the ideal functionality.

Our Solution. To address this issue, we decouple the `ClientComplete` check—whether a query is prohibited—from the actual transmission of the function value to the client. Therefore, we introduce an additional interface `CheckServers` that already marks a recorded query as accepted if enough honest servers were involved in answering the query. This allows the simulator to "register" `ClientComplete` queries (using the new `CheckServers` interface) before a key refresh (and, consequently, before erasure of the tickets) while still completing the evaluation once the adversary permits the client to learn the function value.

Handling Function Labels. Counterintuitively, modeling an "unverifiable" OPRF is more complex than modeling a verifiable one. The ideal functionality of a verifiable OPRF requires only a single function table that always returns the correct value. In contrast, an unverifiable OPRF must maintain multiple function tables because corrupted servers may consistently use incorrect keys. Baum *et al.* [2] introduce labels to identify these tables, representing different PRF keys; the adversary specifies a label with each `ClientComplete` query to indicate which key was used in the evaluation. The label `hon` denotes the key used by honest servers.

Rate-limiting guarantees apply only to the `hon` key. A dishonest client could repeatedly combine an honest server's response with different responses of dishonest servers to obtain multiple PRF values from a single interaction with an honest server. Still, these values must match the function tables to ensure consistency for future evaluations between an honest client, a subset of honest servers, and potentially dishonest servers. Therefore, ticket counters are decremented only when the `hon` label is used.

However, the ideal functionality in [2] models an unverifiable OPRF with multiple function tables but treats all labels the same, inadvertently providing rate-limiting guarantees for dishonest executions. This incorrectly implies that any OPRF evaluation–even with an incorrect key–requires interaction with an honest server. Our fix is straightforward: we check and decrement ticket counters only when values associated with the `hon` label are requested, and we ignore the counters for all other labels. Consequently, our approach enforces rate-limiting guarantees for evaluations under the `hon` key, while allowing unrestricted evaluations under incorrect keys.

2.3 Proving Security

We present our security proof for the tpOPRF construction with respect to the ideal functionality. We defer the formal proofs to the full version of the paper. We start with the proof strategy for the standard 2HashDH scheme, extend it to the fully-adaptive threshold setting, and then to the partially oblivious setting, thereby introducing our novel proof technique.

In the context of partial obliviousness, we identify a gap in the proof of previous works [2, 17]. Intuitively, we can create a setting in which the adversary lacks sufficient solutions to break the one-more assumption; while it can compute solutions for one public value, it cannot do so for another public value. (see "Proof Gap: The Gap between One-More and One-More" for details). A straightforward solution to fill this gap would be to use general-purpose non-interactive zero-knowledge proofs (NIZKs) or to exploit the algebraic group model (AGM). However, we deviate from these approaches to preserve weak assumptions and to reduce computational complexity. Instead, we fill the gap in the proof by introducing a novel proof technique that we believe is of independent interest.

The $\mathsf{OM\text{-}gapDH}$ *assumption.* We recall the One-More Gap Diffie-Hellman assumption ($\mathsf{OM\text{-}gapDH}$), which is the standard assumption for proving security of OPRFs [19, 21, 22, 20, 2, 17]. Intuitively, it states that no efficient adversary can solve more Diffie-Hellman instances than allowed. Given a public key $[k]$ and access to three oracles, the adversary's goal is to compute additional solutions. The first oracle, Targ, outputs random group elements $[x]$ that form Diffie-Hellman instances with $[k]$. The second oracle, Help, returns $k \cdot [x']$ for any input $[x']$, effectively solving the Diffie-Hellman problem for $[k], [x']$. The third oracle, DDH, checks whether $[x_1 \cdot x_2] = [x_3 \cdot x_4]$, which tests whether the discrete logarithms between $[x_1], [x_2]$ and $[x_3], [x_4]$ are equal. An adversary wins if it outputs one more solution than the number of queries it made to Help; Formally, after c calls to Help, it must produce $c + 1$ pairs (i, σ) such that $\sigma = k \cdot [x_i]$, where $[x_i]$ is the i-th output of Targ.

Proving $\mathsf{2HashDH}$ *secure.* To prove the UC security of $\mathsf{2HashDH}$, we construct a simulator such that the ideal functionality, in combination with the simulator, is indistinguishable from the real protocol. The main task is to show that the evaluation of the PRF on a given input x matches the value returned by the ideal functionality for x. Therefore, we answer queries to H_f of the form $(x, k \cdot \mathsf{H}_1(x))$ as follows: query the ideal functionality on input x and program the hash function to return the result of that query.

This strategy leads to a problem if the ticket counter of the ideal functionality is zero, and it refuses to answer the query. In that case, the simulator cannot program H_f to return the correct value because the correct value is unknown to the simulator. We show that this event only happens with negligible probability, as we can break the $\mathsf{OM\text{-}gapDH}$ assumption if it does.

The simulator has to solve three main tasks: (i) injecting challenge elements from Targ, (ii) scalar multiplications with the key k for every `ServerComplete` query, and (iii) solution extractions. Task (ii) can be forwarded to the Help oracle of the $\mathsf{OM\text{-}gapDH}$ assumption. Tasks (i) and (iii) are solved by exploiting the random oracles. Random oracle H_1 answers every query with a challenge element obtained from Targ. The simulator observes random oracle H_f and checks every query with the DDH oracle if it is of the form $(x, k \cdot \mathsf{H}_1(x))$. If so, the input to H_f is a valid solution to the $\mathsf{OM\text{-}gapDH}$ assumption.

Note that we tie the ticket counter of the ideal functionality to the number of Help queries that the simulator does not have a solution for. Every time `ServerComplete` is called, the ticket counter is incremented and the simulator queries Help exactly once. Every time H_f is queried on a valid input, the simulator obtains a solution and calls `ClientComplete`, which decrements the ticket counter. This allows the simulator to query the ideal functionality for every valid query $(x, k \cdot \mathsf{H}_1(x))$ to H_f that the environment obtained through interacting with the simulator. If the environment submits an additional valid input to H_f for which the simulator cannot query the ideal functionality, it breaks the $\mathsf{OM\text{-}gapDH}$ assumption by providing that "one-more" solution.

Fully-Adaptive Threshold. The main observation that enables fully-adaptive security is that we can make a server's response look uniformly random and, thereby, independent from k_i while the corresponding server is honest. Only when t responses are combined, the computation as a whole is consistent with the PRF key. Now, the simulator can sample the first $t-1$ responses uniformly at random and only compute the last response in a way that the combination of all t responses is consistent. In more detail, the simulator computes the last response $[o_j]$ via

$$[o_j] = k \cdot [p] - \sum_{i \in Sset \setminus \{j\}} [o_i],$$

where $k \cdot [p]$ is obtained from the Help oracle that the assumption provides. This allows the simulator to leave honest key shares undefined and only sample key shares of corrupted servers just in time once a corruption is requested.

To make things consistent, once the server is corrupted and the secret key is defined, we program the random oracles. Therefore, the simulator goes through all previously computed answers and samples undefined blinding values $\beta_{i,j} \leftarrow \mathsf{H}_b(\mathsf{s}_{i,j}, Sset, [p])$ uniformly at random until there is only one undefined blinding value left. It programs H_b for that value as

$$\mathsf{H}_b(\mathsf{s}_{i,j}, Sset, [p]) \leftarrow$$

$$[o_i] - \delta_{Sset,i} \cdot k_i \cdot [p] - \sum_{l \in Sset \setminus \{i,j\}} \Delta_{i,l} \cdot \mathsf{H}_b(\mathsf{s}_{i,l}, Sset, [p])$$

It is important to see that there must be at least one undefined blinding value for this to work. This holds true because the adversary can corrupt at most $t-1$ servers such that at the point of corruption, at least one server remains honest. The pairwise blinding value between the now corrupted server and the server that remains honest is still undefined and can now be used to make things consistent.

Partial Obliviousness. Before we introduce the challenges associated with proving security for a partially oblivious PRF, we quickly recap how we changed our construction to transition from a fully to a partially oblivious setting. Most importantly, we switch to a pairing-friendly group consisting of two groups $\mathbb{G}_1, \mathbb{G}_2$, a target group $\mathbb{G}_T$, and a bilinear pairing e that maps two group elements of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively to $\mathbb{G}_T$. We use the notation $e([x]_1, [y]_2)$ to denote the bilinear pairing. Furthermore, we use two hash functions H_1 and H_2 instead of a single one. H_1 is used to hash the private input to $\mathbb{G}_1$ and the public input is hashed to $\mathbb{G}_2$ using H_2. The hash values are combined using the bilinear pairing. The combined element is multiplied by the key k, resulting in

$$k \cdot e(H_1(x_{priv}), H_2(x_{pub})).$$

Given that we are no longer working in a single group, the $\mathsf{OM\text{-}gapDH}$ assumption does not fit the setting anymore, and we switch to the bilinear version of the assumption called the One-More Bilinear Gap Diffie-Hellman ($\mathsf{OM\text{-}B\text{-}gapDH}$) assumption. Instead of one, there are now two Targ oracles that return random group elements from $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively. The Help and DDH oracles operate in the target group. A valid solution is of the form (i, j, σ) such that $\sigma = k \cdot e([x_i]_1, [y_j]_2)$ where $[x_i]$ is the i-th response and $[y_j]_2$ is the j-th response from the Targ_1 and Targ_2 oracle, respectively. An adversary breaks the $\mathsf{OM\text{-}B\text{-}gapDH}$ assumption if it provides $c + 1$ distinct solutions while querying the Help oracle at most c times.

The proof strategy for adapting $\mathsf{2HashDH}$ to the partially oblivious setting, as proposed by Baum *et al.* [2] and later employed by Gu *et al.* [17], involves utilizing the random oracles H_1 and H_2 to embed challenges from Targ_1 and Targ_2, respectively, rather than relying solely on H_1. However, this approach does not work.

Proof Gap: The Gap Between One-More and One-More. The pOPRF ideal functionality has separate ticket counters for each public input x_{pub}. This captures the requirement that a client interacting with a server for x_{pub} only learns a single PRF value $f_k(\cdot, x_{pub})$. The use case of password-based authentication makes this requirement clear. Here, pOPRFs prevent brute-force attacks against the password by restricting authentication attempts per user using the password as the private input and the username as the public input. If an adversary could submit a password guess associated with a public value x_{pub}' but obtain a PRF value for a different x_{pub}, then the adversary could bypass the limit on authentication attempts by using different usernames for each password guess. In a way, each x_{pub} value opens up a new "domain" and with it a new "one-more condition". The environment can distinguish the real from the ideal world if it breaks any one of the "one-more conditions".

In contrast, the $\mathsf{OM\text{-}B\text{-}gapDH}$ assumption does not distinguish between different inputs when counting Help queries. Rather, it counts all Help queries regardless of the input. As a result, the assumption only has a single "one-more condition" that must be broken to break the assumption. This gap between the multiple domains (or "one-more conditions") in the ideal functionality and the single domain (or "one-more condition") can lead to a situation where the environment distinguishes the two worlds, but the simulator is unable to break the $\mathsf{OM\text{-}B\text{-}gapDH}$ assumption, effectively breaking the security reduction.

We demonstrate this discrepancy by sketching an environment that uses the simulator to break the $\mathsf{OM\text{-}B\text{-}gapDH}$ assumption. Intuitively, the environment sits between the simulator and the $\mathsf{OM\text{-}B\text{-}gapDH}$ assumption and simulates an instance of the assumption that we call $\mathsf{OM\text{-}B\text{-}gapDH}^{\mathcal{E}}$ (see Figure 2).

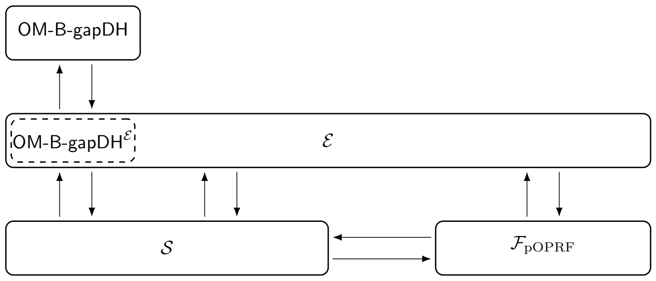

Fig. 2: The environment $\mathcal{E}$ that uses the simulator $\mathcal{S}$ of [2] to break the $\mathsf{OM\text{-}B\text{-}gapDH}$ assumption.

$\mathsf{OM\text{-}B\text{-}gapDH}^{\mathcal{E}}$ is simulated such that all queries to $\mathsf{Targ}_1^{\mathcal{E}}$ and $\mathsf{Targ}_2^{\mathcal{E}}$ are forwarded to the corresponding oracles of $\mathsf{OM\text{-}B\text{-}gapDH}$. The counterexample consists of five simple steps (the explicit message flow can be seen in the full version of the paper):

1. The environment sends a regular evaluation request $([p]_1, x_{pub_1})$ to the server. Note that any party (including $\mathcal{A}$) can send those requests to the server. The simulator computes $\mathsf{H}_2(x_{pub_1})$ using $\mathsf{Targ}_2^{\mathcal{E}}$, which the environment forwards to Targ_2. The simulator answers the query using the $\mathsf{Help}^{\mathcal{E}}$ oracle $[o]_t \leftarrow \mathsf{Help}^{\mathcal{E}}(e([p]_1, \mathsf{H}_2(x_{pub_1})))$, which the environment forwards to Help. This leads to the ticket counters
$$tickets(x_{pub_1}) = 1, \quad tickets(x_{pub_2}) = 0$$
and the internal counters of the assumptions
$$c^{\mathcal{E}} = 1, \quad c = 1.$$
2. The environment sends a re-randomized version of the first evaluation request $(a \cdot [p]_1, x_{pub_1})$. Therefore, the environment can answer the following $\mathsf{Help}^{\mathcal{E}}$ query sent by the simulator to $\mathsf{OM\text{-}B\text{-}gapDH}^{\mathcal{E}}$ without a Help query to $\mathsf{OM\text{-}B\text{-}gapDH}$ as it knows the re-randomization factor a ($[o']_t \leftarrow a \cdot [o]_t$). This leads to the ticket counters
$$tickets(x_{pub_1}) = 2, \quad tickets(x_{pub_2}) = 0$$
and the internal counters of the assumptions
$$c^{\mathcal{E}} = 2, \quad c = 1.$$
3. The environment prepares the evaluation for (x_{pub_2}, x_{priv}) by sending the inputs to the corresponding ROs H_1 and H_2. The simulator answers those queries using $\mathsf{Targ}_1^{\mathcal{E}}$ and $\mathsf{Targ}_2^{\mathcal{E}}$, which the environment forwards to Targ_1 and Targ_2, respectively.

4. The environment computes the solution $[o'']_t \leftarrow k \cdot e(\mathsf{H}_1(x_{priv}), \mathsf{H}_2(x_{pub_2}))$ for that evaluation using a Help query to $\mathsf{OM\text{-}B\text{-}gapDH}$. At this point, the number of Help queries to $\mathsf{OM\text{-}B\text{-}gapDH}$ and $\mathsf{OM\text{-}B\text{-}gapDH}^{\mathcal{E}}$ are
$$c^{\mathcal{E}} = 2, \quad c = 2.$$
5. The environment sends the valid query $(x_{pub_2}, x_{priv}, [o'']_t)$ to the random oracle H_f. To program the answer correctly, the simulator has to send a query to the ideal functionality. The ideal functionality refuses to answer because the ticket counter for x_{pub_2} is $tickets(x_{pub_2}) = 0$ (the fact that $tickets(x_{pub_1}) = 2$ does not make a difference here). According to the proofs in [2] and [17], the simulator now breaks the $\mathsf{OM\text{-}B\text{-}gapDH}^{\mathcal{E}}$ assumption. Because the number of $\mathsf{Help}^{\mathcal{E}}$ queries is $c^{\mathcal{E}} = 2$, it outputs three valid solutions, which the environment forwards to $\mathsf{OM\text{-}B\text{-}gapDH}$.

Note that all queries to $\mathsf{Targ}_1^{\mathcal{E}}$ and $\mathsf{Targ}_2^{\mathcal{E}}$ were directly forwarded to the corresponding Targ oracle of the $\mathsf{OM\text{-}B\text{-}gapDH}$ assumption, such that every valid solution to $\mathsf{OM\text{-}B\text{-}gapDH}^{\mathcal{E}}$ is also a valid solution to $\mathsf{OM\text{-}B\text{-}gapDH}$. Therefore, the environment breaks the $\mathsf{OM\text{-}B\text{-}gapDH}$ assumption with the three solutions provided by the simulator, which leads to a contradiction because we assume $\mathsf{OM\text{-}B\text{-}gapDH}$ to be hard. Consequently, a simulator that returns those three solutions cannot exist.

Fixing the Gap with Heavy Solutions. The above approach fails because a missing solution for one public value cancels out the additional solution for another public value. If the simulator were able to get a solution directly from the Help query without having to wait for the solution from a query to H_f, there would be no Help query without a solution that could cancel out the additional solution.

A straightforward modification that allows the simulator to compute a valid solution for each Help query is to require the client to send a witness extractable zero-knowledge proof to the server with every `Eval` query. This proof shows that the client knows a private value x_{priv} and a blinding factor r such that $[p]_1 = r \cdot \mathsf{H}_1(x_{priv})$. When the simulator receives such a query, it can extract x_{priv} and r from the proof and submit $e(\mathsf{H}_1(x_{priv}), \mathsf{H}_2(x_{pub}))$ to the Help oracle. The response from Help is a valid solution to the $\mathsf{OM\text{-}B\text{-}gapDH}$ assumption, and the simulator can compute the server's response by re-blinding the result with r. This approach has a significant drawback since the statement of the NIZK involves a hash function, such that the zero-knowledge proof must be general-purpose. This results in significant computational overhead, making this method impractical for many real-world applications.

A similar approach without the computational overhead is to use the Algebraic Group Model (AGM) [15]. On an intuitive level, the AGM requires the client to send an algebraic representation alongside each group element it sends. An honest client would send $[p]_1 = r \cdot \mathsf{H}_1(x_{priv})$ to the simulator with its `Eval` query. Similar to the approach using NIZKs, the simulator would learn the blinding factor r, solving the issues. Handling queries from dishonest clients is slightly more complicated but still possible. Yet, we also deviate from this approach since the AGM is considered a strong assumption, and we try to use as weak assumptions as possible.

Our Novel Proof Technique. As we have seen, we can avoid the issue if we immediately get a solution from every Help query. A different approach is to avoid "unnecessary" Help queries, namely those from all domains that are not the ones where the break happens, such that they cannot cancel out additional solutions.

To see how we can simulate the output of the Help oracle in certain domains, we take a step back and observe that for computing the value

$$k \cdot e([p]_1, \mathsf{H}_2(x_{pub})),$$

we have to "combine" three components: (i) the uniformly random value $[p]_1$ with $[p]_1 = r \cdot \mathsf{H}_1(x_{priv})$, (ii) the hash of the public value $\mathsf{H}_2(x_{pub})$, and (iii) the key k. To "combine" these three elements, we can "combine" two group elements using the bilinear pairing while the third element can only be included with scalar multiplication. In the real protocol, the two group elements have to be $[p]_1$ and $\mathsf{H}_2(x_{pub})$ because for scalar multiplication, the discrete logarithm must be known, which cannot be computed efficiently. Likewise, the client cannot extract the key k from the public key $[k]$ such that the key has to be included by the server via scalar multiplication.

To avoid the "unnecessary" Help queries in our proof, we refrain from the combination of the real protocol and, instead, include the public key as an element of the bilinear pairing. Since we can only combine two elements using the pairing and have to incorporate the third element by scalar multiplication, we need to know the discrete logarithm of either $[p]_1$ or $\mathsf{H}_2(x_{pub})$. Since the value $[p]_1$ is uniformly random and chosen by the client, $[p]_1$ is not in question as a candidate for incorporation via scalar multiplication. Therefore, we input both $[k]_2$ and $[p]_1$ in the pairing leading to the element

$$e([p]_1, [k]_2) = k \cdot e([p]_1, [1]_2)$$

In order to also include the value $\mathsf{H}_2(x_{pub})$ into the computation, we use the programmability of the random oracle and program $\mathsf{H}_2(x_{pub}) = [a]_2$ for a known but uniformly random scalar a. Knowing the trapdoor a, we can compute

$$a \cdot e([p]_1, [k]_2) = a \cdot k \cdot e([p]_1, [1]_2) = k \cdot e([p]_1, [a]_2) = k \cdot e([p]_1, \mathsf{H}_2(x_{pub})),$$

which preserves correctness. This strategy brings us one step closer to achieving our goal of closing the gap between the "multi-domain" and "single-domain" worlds of the ideal functionality and the assumption.

While this solution helps us with the domain issue, it also imposes a challenge for the reduction: If the environment manages to provide one more valid query to H_f, the simulator has to use this valid query to output an additional valid solution to the $\mathsf{OM\text{-}B\text{-}gapDH}$ assumption. Otherwise, the simulator cannot break the $\mathsf{OM\text{-}B\text{-}gapDH}$ assumption in case the environment distinguishes the real from the ideal world. By the $\mathsf{OM\text{-}B\text{-}gapDH}$ assumption, a solution is only valid if both $\mathsf{H}_1(x_{priv})$ and $\mathsf{H}_2(x_{pub})$ are challenge elements returned by Targ_1 and Targ_2. This is not the case if we follow our new strategy always, and H_2 always returns values $[a]_2$ that it knows a to. Therefore, we guess the public value $x_{pub}{}^*$ that the break happens for and deviate from our trapdoor-injection technique for $\mathsf{H}_2(x_{pub}{}^*)$ by returning a challenge element from Targ_2. In a way, this isolates the crucial one of the domains (or "one-more conditions") from the others, such that the others cannot cancel out solutions for the crucial one. This closes the aforementioned gap between multiple "one-more conditions" in the ideal functionality and the single "one-more condition" in the assumption because only a single public value is connected with the assumption. We summarize the differences in how the random oracles are connected to the Targ_1, Targ_2, and Help oracles and compare our Domain-Isolating Oracle Programming technique to the work of [2, 17], in Table 2.

		simulator in [2, 17]	DIOP
$\mathsf{H}_1(x_{priv})$		Targ_1	Targ_1
$\mathsf{H}_2(x_{pub})$	$x_{pub} = x_{pub}{}^*$	Targ_2	Targ_2
	else		$[a]_2 = a \cdot [1]_2$
$k \cdot e([p]_1, \mathsf{H}_2(x_{pub}))$	$x_{pub} = x_{pub}{}^*$	$\mathsf{Help}(e([p]_1, \mathsf{H}_2(x_{pub})))$	$\mathsf{Help}(e([p]_1, \mathsf{H}_2(x_{pub}{}^*)))$
	else		$a \cdot e([p]_1, [k]_2)$

Table 2: Comparison between our novel Domain-Isolating Oracle Programming (DIOP) proof technique and the strategy used by Baum *et al.* [2] and Gu *et al.* [17].

3 Preliminaries

OPRF. An OPRF is a two-party protocol between a client and a server realizing functionality $(f_k(x), \perp) \leftarrow \langle \mathcal{C}(x), \mathcal{S}(k) \rangle_{\mathsf{Eval}}$ for a PRF family f_k.

The $\mathsf{OM\text{-}B\text{-}gapDH}$ *assumption.* We already introduced the related $\mathsf{OM\text{-}gapDH}$ assumption on an intuitive level in Section 2.3 and formally define $\mathsf{OM\text{-}B\text{-}gapDH}$ here. The "One-More Bilinear Gap Diffie-Hellman" (or $\mathsf{OM\text{-}B\text{-}gapDH}$) assumption is a variant of the $\mathsf{OM\text{-}gapDH}$ assumption that works on an asymmetric bilinear group. Consequently, it has two oracles $\mathsf{Targ}_1, \mathsf{Targ}_2$ that return random elements from groups $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively. Solutions now have the form (i, j, σ) such that $\sigma = k \cdot e([x_i]_1, [y_i]_2)$, with x_i returned by the i-th call to Targ_1 and y_i returned by the j-th call to Targ_2. The Help and DDH oracles take group elements from $\mathbb{G}_T$. Figure 3 shows the $\mathsf{OM\text{-}B\text{-}gapDH}$ game.

Definition 1 ($\mathsf{OM\text{-}B\text{-}gapDH}$). *The* One-More Bilinear Gap Diffie-Hellman *assumption holds in bilinear group* $\mathbb{BG}$ *of prime order* q *if for any* PPT *adversary*

$\mathcal{A}$ there exists a negligible function $\mathsf{negl}(\cdot)$ *such that*

$$\Pr[\mathsf{OM\text{-}B\text{-}gapDH}_{\mathcal{A}}(\mathbb{BG})] \leq \mathsf{negl}(\lambda),$$

where the randomness is taken over the random coins of the experiment and the adversary.

```
$\mathsf{OM\text{-}B\text{-}gapDH}(\mathbb{BG})$
    $k \leftarrow\!\$\; \mathbb{Z}_q^*$
    $c, c_{1,t}, c_{2,t} \leftarrow 0$
    $\mathbb{O} := \{\mathsf{Targ}_1, \mathsf{Targ}_2, \mathsf{Help}, \mathsf{DDH}\}$
    $(i_1, j_1, \sigma_1), \ldots, (i_\ell, j_\ell, \sigma_\ell) \leftarrow \mathcal{A}^{\mathbb{O}}([k]_2)$
    assert $c < \ell$
    assert $\forall \alpha : (i_\alpha \leq c_{1,t}) \wedge (j_\alpha \leq c_{2,t})$
    assert $\forall \alpha \neq \beta : (i_\alpha, j_\alpha) \neq (i_\beta, j_\beta)$
    return $\forall \alpha : k \cdot e([x_{i_\alpha}]_1, [y_{j_\alpha}]_2) = \sigma_\alpha$

$\mathsf{Targ}_1()$
    $c_{1,t} \leftarrow c_{1,t} + 1$
    $[x_{c_{1,t}}]_1 \leftarrow\!\$\; \mathbb{G}_1$
    return $[x_{c_{1,t}}]_1$

$\mathsf{Targ}_2()$
    $c_{2,t} \leftarrow c_{2,t} + 1$
    $[y_{c_{2,t}}]_2 \leftarrow\!\$\; \mathbb{G}_2$
    return $[y_{c_{2,t}}]_2$

$\mathsf{DDH}([x_1]_t, [y_1]_t, [x_2]_t, [y_2]_t)$
    $d_1 \leftarrow \mathsf{dLog}_{[x_1]_t}([y_1]_t)$
    $d_2 \leftarrow \mathsf{dLog}_{[x_2]_t}([y_2]_t)$
    return $d_1 = d_2$

$\mathsf{Help}([z]_t)$
    $c \leftarrow c + 1$
    return $k \cdot [z]_t$
```

Fig. 3: Security game for $\mathsf{OM\text{-}B\text{-}gapDH}$.

4 Ideal Functionality $\mathcal{F}_{\mathrm{tpOPRF}}$

We define our ideal threshold partially oblivious PRF functionality $\mathcal{F}_{\mathrm{tpOPRF}}$ in Figure 4. It formalizes secrecy of private inputs x_{priv}, correctness of the output while all servers are honest (but not verifiability of the output, i.e., correctness when some corrupted servers are involved), and rate limiting with respect to evaluations of the "correct" PRF. Since $\mathcal{F}_{\mathrm{tpOPRF}}$ follows the discussion in Section 2, Figure 4 is mostly self-explanatory, except for two details.

Firstly, $\mathcal{F}_{\mathrm{tpOPRF}}$ guarantees correctness of the output only if all servers have been honest *for the entire duration* from start to end of an eval request. Some works, such as [2], define a slightly stronger notion that guarantees correctness if all servers are honest when the output is generated, even if some servers were malicious while the request was ongoing. This is too strong for protocols that do not involve clients in key refresh, such as the one in this paper: a malicious server might send an incorrect response to a client and then perform a key refresh to become honest again before the response is delivered. Since the client is unaware of the key refresh, once he receives the response, he would still finish his previous eval request and generate an incorrect output, even though all servers are honest at the time of output. We note that slightly weakening the correctness property is not an issue. Correctness guarantees of non-verifiable OPRFs as modeled by $\mathcal{F}_{\mathrm{tpOPRF}}$ are already extremely weak to begin with and basically serve more as a

The functionality keeps several independent random functions $F_{label}(x_{pub}, x_{priv})$, where *label* is a string that denotes a specific function. The special label $label = \texttt{hon}$ denotes the "correct" function for which correctness and rate limiting are provided. It assigns random values $F_{label}(x_{pub}, x_{priv}) \leftarrow\$ \{0,1\}^{\lambda}$ whenever not yet defined.

Initialize $tickets(S_i, x_{pub}) \leftarrow 0, \forall i \in \{1, \ldots, \mathsf{n}\}$.

On $(\texttt{Eval}, x_{pub}, x_{priv})$ from any client C, do:
- record (C, x_{pub}, x_{priv}).
- send $(\texttt{Eval}, C, x_{pub})$ to $\mathcal{A}$.

On $(\texttt{ServerComplete}, x_{pub})$ from server S_i, do:
- increment $tickets(S_i, x_{pub}) + +$ and notify $\mathcal{A}$.

On $(\texttt{CheckServers}, C, \mathit{HonServerSet})$ from the adversary $\mathcal{A}$, do:
- retrieve (but keep) record (C, x_{pub}, x_{priv}).
- let n_c be the number of currently corrupted servers. Check that *HonServerSet* has size at least $\mathsf{t} - n_c$ and contains only honest servers S_i with $tickets(S_i, x_{pub}) \geq 1$ that are willing to help with a PRF evaluation; if this fails, abort.
- Decrement $tickets(S_i, x_{pub}) - -$ for all $S_i \in \mathit{HonServerSet}$.
- Mark the record (C, x_{pub}, x_{priv}) as accepted by the servers and return control to $\mathcal{A}$.

On $(\texttt{ClientComplete}, C, label)$ from the adversary $\mathcal{A}$, do:
- retrieve (and delete) record (C, x_{pub}, x_{priv}).
- if all S_i, $i \in \{1, \ldots, \mathsf{n}\}$ have been honest since the record was stored, set $label \leftarrow \texttt{hon}$.
- if $label = \texttt{hon}$ and the record (C, x_{pub}, x_{priv}) is not marked as accepted by the servers (see `CheckServers` command), abort and return an error to $\mathcal{A}$.
- output $F_{label}(x_{pub}, x_{priv})$ to C.

On `HonestKeyRefresh` from the adversary $\mathcal{A}$, do:
- reset all S_i to be honest and reset all *tickets* to 0.

On $(\texttt{Corrupt}, P)$ from the adversary $\mathcal{A}$, do:
- adaptively corrupt party P (for servers S_i only until at most $\mathsf{t} - 1$ are corrupt).

In addition, $\mathcal{A}$ can also start PRF evaluations on arbitrary x_{pub}, x_{priv}, and *label* and obtain the output for himself/a corrupted client. The logic is the same as for `Eval`, `CheckServers`, `ClientComplete` but with one difference: unlike honest clients, $\mathcal{A}$ only has to determine x_{priv} when the output is generated, i.e., as part of the `ClientComplete` step. This slight relaxation does not weaken intended security guarantess but is formally necessary for the simulator in our security proof.

Fig. 4: Summary of our ideal threshold partially oblivious PRF functionality $\mathcal{F}_{\text{tpOPRF}}$ with n servers S_i and threshold $1 \leq \mathsf{t} \leq \mathsf{n}$.

functional sanity check rather than a security property. If an application requires correct OPRF outputs, it would rather use a verifiable OPRF.

Secondly, in $\mathcal{F}_{\text{OPRF}}$ (cf. Figure 1), which we used as our starting point, the adversary $\mathcal{A}$ decides when a server is willing to complete a PRF evaluation by sending the `ServerComplete` message; higher-level protocols are only notified of this decision. In our $\mathcal{F}_{\text{tpOPRF}}$ (cf. Figure 4), this is reversed, and we let higher-level protocols decide by sending the `ServerComplete` message, with the adversary only being notified. Both of these definitions formalize the same security guarantees–there will be no more evaluations than permitted by honest servers–and are therefore interchangeable for analyzing the security of real OPRF protocols. They do, however, differ in the context of hybrid protocols. Since the adversary decides in $\mathcal{F}_{\text{OPRF}}$, a security proof of a hybrid protocol on top of $\mathcal{F}_{\text{OPRF}}$ cannot rely on any specific rate-limiting algorithm being used. In contrast, and if desired, one can build hybrid protocols on top of $\mathcal{F}_{\text{tpOPRF}}$ that fix a specific rate-limiting algorithm–say, servers are only willing to evaluate on public inputs following a certain format–and then use this property in a security analysis. Due to this potential advantage, we chose the latter definitional style for $\mathcal{F}_{\text{tpOPRF}}$.

5 Construction

In this section, we present our construction for the UC-secure fully-adaptive partially oblivious PRF. Because the motivation for the chosen building blocks has been detailed in Section 2.1, we omit a high-level overview here and instead provide a detailed description of the complete protocol flow.

Setup. To set up tpOPRF, one samples a bilinear group $\mathbb{BG}$, samples a total of five hash functions, and stores the group description, the threshold parameters (t, n), and the description of the hash functions in the public parameters pp. A formal description of the setup procedure is given in Figure 5.

```
Setup(λ, n, t)
    $\mathbb{BG} \leftarrow\!\!\$\ \mathsf{GGen}(\lambda)$
    $\mathsf{H}_1 \leftarrow\!\!\$\ \{\mathsf{H} : \{0,1\}^* \to \mathbb{G}_1\}$
    $\mathsf{H}_2 \leftarrow\!\!\$\ \{\mathsf{H} : \{0,1\}^* \to \mathbb{G}_2\}$
    $\mathsf{H}_f \leftarrow\!\!\$\ \{\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda\}$
    $\mathsf{H}_b \leftarrow\!\!\$\ \{\mathsf{H} : \{0,1\}^* \to \mathbb{G}_t\}$
    $\mathsf{H}_s \leftarrow\!\!\$\ \{\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda\}$
    $\mathsf{pp} \leftarrow (\mathbb{BG}, \mathsf{n}, \mathsf{t}, \mathsf{H}_1, \mathsf{H}_2, \mathsf{H}_f, \mathsf{H}_b, \mathsf{H}_s)$
    return pp
```

```
Key-Gen(pp)
    $k \leftarrow\!\!\$\ \mathbb{Z}_q^*$
    $\{k_i\}_{i \in [n]} \leftarrow\!\!\$\ \mathsf{Share}(k, \mathsf{n}, \mathsf{t})$
    for $i \in [\mathsf{n}]$ do
        for $j \in [\mathsf{n}] \setminus \{i\}$ do
            if $i < j$ : $\mathsf{s}_{i,j} \leftarrow\!\!\$\ \{0,1\}^\lambda$
            else : $\mathsf{s}_{i,j} \leftarrow \mathsf{s}_{j,i}$
    $\mathsf{sk}_i \leftarrow (k_i, \{\mathsf{s}_{i,j}\}_{j \in [\mathsf{n}] \setminus \{i\}})$
    return $\{\mathsf{sk}_i\}_{i \in [\mathsf{n}]}$
```

Fig. 5: tpOPRF Setup and Key-Generation

Key Generation. The key generation algorithm samples a random key k, which will serve as the final PRF key. This key will never change during the lifetime of an OPRF (note that the key refresh will change the shares of this key, but not the key itself). The key generation algorithm shares k using Shamir secret sharing with parameters (t, n), such that each set of t servers can reconstruct k. In addition, it samples uniformly random bilateral seeds $\mathsf{s}_{i,j}$ for each pair of servers $(i, j) \in [\mathsf{n}]^2_{i \neq j}$. It holds, that $\mathsf{s}_{i,j} = \mathsf{s}_{j,i}$. Looking ahead to the evaluation protocol, the blinding factors of two corresponding servers will cancel each other out since $\mathsf{s}_{i,j} = \mathsf{s}_{j,i}$. We formally describe the key generation algorithm in Figure 5.

Evaluation. In the evaluation protocol, the client takes as input a public value x_{pub}, a private value x_{priv}, and public parameters pp. Each server takes as input its secret key sk_i and public parameters pp. The client chooses a set of t servers *Sset* from the set of acceptable server sets $\mathsf{T}_{\mathsf{set}}$. First, the client hashes the private input x_{priv} to group $\mathbb{G}_1$ and blinds that hash by multiplying it with a uniformly random value r, resulting in $[p]_1 \leftarrow r \cdot H_1(x_{priv})$. It sends the blinded value $[p]_1$ along with x_{pub} and $Sset$ to each server in $Sset$.

After receiving the triple from the client, each server contained in $Sset$ performs the following steps: it evaluates the pairing on the blinded value $[p]_1$ and the public value x_{pub}, which is hashed to group $\mathbb{G}_2$, resulting in $[o]_t \leftarrow e([p]_1, H_2(x_{pub}))$. The server computes a blinding value $[\beta_i]_t$ as the sum of pairwise blinding values $[\beta_{i,j}]_t$, each multiplied by the Kronecker delta $\varDelta_{i,j}$. The pairwise blinding values are computed as the hash of the mutual seed $\mathsf{s}_{i,j}$ and the message received from the client, resulting in fresh blinding values per request. For one of the servers computing $[\beta_{i,j}]_t$, $\varDelta_{i,j}$ is 1, and for the other, it is -1. Therefore, when added, they cancel each other out. The server multiplies $[o]_t$ with its key share k_i and the Lagrange coefficient $\delta_{Sset,i}$ corresponding to the server set. Finally, it adds the blinding value and sends the resulting $[u_i]_t$ back to the client.

Once the client has received all $[u_i]_t$, it adds them up to obtain $r \cdot k \cdot e(\mathsf{H}_1(x_{priv}), \mathsf{H}_2(x_{pub}))$. Note that the sum of all $[u_i]_t$ does not contain any blinding values because the blinding values consist only of pairs $\varDelta_{i,j} \cdot [\beta_{i,j}]_t$ and $\varDelta_{j,i} \cdot [\beta_{j,i}]_t$ that add up to zero. The client computes the final PRF value by computing the hash of the two input values x_{pub}, x_{priv} and the unblinded response from the servers $[u]_t/r = k \cdot e(\mathsf{H}_1(x_{priv}), \mathsf{H}_2(x_{pub}))$, resulting in $y \leftarrow \mathsf{H}_f\left(x_{pub}, x_{priv}, \frac{[u]_t}{r}\right)$. We formally describe the evaluation protocol in Figure 6.

Evaluation Protocol

Client$(x_{pub}, x_{priv}, Sset, \mathsf{pp})$		**i-th Server**$(\mathsf{sk}_i, \mathsf{pp})$
Choose $Sset \leftarrow\$ \mathsf{T}_{\mathsf{set}}$ servers for current round.		
$r \leftarrow\$ \mathbb{Z}_q^*$		**parse** sk_i **as** $(k_i, \{\mathsf{s}_{i,j}\}_{j \in [\mathsf{n}] \setminus \{i\}})$
$[p]_1 \leftarrow r \cdot H_1(x_{priv})$	$\xrightarrow{([p]_1, x_{pub}, Sset)}$	$[o]_t \leftarrow [p]_1 \cdot H_2(x_{pub})$
		$[\beta_{i,j}]_t \leftarrow \mathsf{H}_b(\mathsf{s}_{i,j}, Sset, x_{pub}, [p]_1)$
		$[\beta_i]_t \leftarrow \sum_{j \in Sset \setminus \{i\}} \varDelta_{i,j} \cdot [\beta_{i,j}]_t$
		// $\varDelta_{i,j} = 1$ if $i > j$, -1 else
$[u]_t \leftarrow \sum_{i \in Sset} [u_i]_t$	$\xleftarrow{[u_i]_t}$	$[u_i]_t \leftarrow \delta_{Sset,i} \cdot k_i \cdot [o]_t + [\beta_i]_t$
$y \leftarrow \mathsf{H}_f\left(x_{pub}, x_{priv}, \frac{[u]_t}{r}\right)$		// $\delta_{Sset,i} = \prod_{j \in Sset \setminus \{i\}} \frac{j}{j-i}$
return y		**return** x_{pub}

Fig. 6: tpOPRF Evaluation

Key Refresh. In the key refresh protocol, each server updates its key share k_i' and computes new bilateral seeds $\mathsf{s}_{i,j}'$ that match between two servers ($\mathsf{s}_{i,j}' = \mathsf{s}_{j,i}'$).

Every server computes a Shamir secret sharing of zero and sends the shares to the other servers. After receiving the shares from all other servers, it adds these zero shares to its old share k_i. The resulting share k'_i is independent of the old one because the added shares are uniformly random. Note that reconstructing the key with the updated shares still yields the initial key k, as only sharings of zero were added to the old shares. Every two servers perform a Diffie-Hellman key exchange and hash the result to obtain a fresh seed $\mathsf{s}'_{i,j}$, that is consistent but random $\mathsf{s}'_{i,j} = \mathsf{s}'_{j,i}$.

We deliberately employ non-verifiable secret sharing in this protocol. While verifiable secret sharing (VSS) could protect against dishonest shares during the refresh phase, the system's robustness is already limited by two unavoidable adversarial capabilities: (i) a network-level adversary can drop messages to prevent a client from receiving server responses, and (ii) a compromised server can erase its internal state, including its share. Given these limitations, our primary focus is on preserving security, rather than ensuring robustness against every possible form of disruption. Should stronger robustness be desired–for instance, to defend against incorrect shares during refresh–substituting the underlying secret-sharing scheme with a verifiable variant would be a natural extension.

We depict the key refresh protocol in Figure 7.

```
i-th Server($\mathsf{sk}_i, \mathsf{pp}$)
1 :  $\{k_{i,j}\} \leftarrow\$ \mathsf{Share}(0, n, t)$
2 :  for $j \in [\mathsf{n}] \setminus \{i\}$ do
3 :      $a_{i,j} \leftarrow\$ \mathbb{Z}_q^*$,   $[A_{i,j}]_t \leftarrow a_{i,j} \cdot [1]_t$
4 :  send $(k_{i,j}, [A_{i,j}]_t)$ to $\mathcal{S}_j$
5 :  receive $(k_{j,i}, [A_{j,i}]_t)$ from $\mathcal{S}_j$
6 :  $k'_i \leftarrow k_i + \sum_{j \in [\mathsf{n}]} k_{j,i}$
7 :  for $j \in [\mathsf{n}] \setminus \{i\}$ do
8 :      $[B_{i,j}]_t \leftarrow a_{i,j} \cdot [A_{j,i}]_t$
9 :      $\mathsf{s}'_{i,j} \leftarrow \mathsf{H}_s([B_{i,j}]_t)$
10 : return $(k'_i, \{\mathsf{s}'_{i,j}\}_{j \in [\mathsf{n}] \setminus \{i\}})$
```

Fig. 7: tpOPRF Key Refresh

6 Proof

In this section, we give a high-level idea of our proof and defer to the full version of the paper for the formal proof of security. We prove our construction to be UC-secure and show that our protocol $\mathcal{P}_{\text{tpOPRF}}$ realizes the ideal tpOPRF functionality $\mathcal{F}_{\text{tpOPRF}}$ (see the full version of the paper for the formal definition of $\mathcal{F}_{\text{tpOPRF}}$). As part of the security proof, we construct a simulator such that–as is common–an environment cannot distinguish whether it interacts with the real protocol $\mathcal{P}_{\text{tpOPRF}}$ or the ideal functionality $\mathcal{F}_{\text{tpOPRF}}$ (connected to the simulator). In other words, we show that an adversary learns at most as much from the real protocol as it learns from the ideal protocol. We prove the indistinguishability of $\mathcal{P}_{\text{tpOPRF}}$ and $\mathcal{F}_{\text{tpOPRF}}$ under the $\mathsf{OM\text{-}B\text{-}gapDH}$ assumption.

We assume server-sided authenticated channels, which can be realized using TLS. We require authenticated channels to ensure correctness in the presence of honest servers: without authentication, the adversary could arbitrarily tamper with messages from honest servers, potentially violating correctness. However, we emphasize that authenticated communication is not necessary for security, and our security guarantees hold even in the absence of authenticated channels. Additionally, we assume secure erasure for the client, which is equivalent to assuming that the client only gets corrupted after it has no pending evaluation queries.

We model both honest and dishonest executions of the key refresh protocol. Honest key refresh is used to restore security guarantees for previously compromised parties, while dishonest key refresh does not result in de-corruption. We formalize the following implication: if key refresh is executed correctly–i.e., it proceeds without adversarial interference, is executed atomically, and takes place over secure channels–then it effectively resets the internal state of the involved parties, thereby restoring their security guarantees. We refer to this scenario as honest key refresh. Our model also allows the adversary to trigger a dishonest key refresh, where corrupted parties execute the protocol under adversarial scheduling and potentially over compromised channels. In these cases, the resulting key material remains under adversarial control–security is neither broken nor restored, and previously corrupted servers remain compromised.

Theorem 1. *Let $\lambda \in \mathbb{N}$ be the security parameter. Let $\mathcal{P}_{tpOPRF}$ be our protocol and $\mathcal{F}_{tpOPRF}$ be the ideal* tpOPRF *functionality as defined in the full version of the paper. Let* $\mathsf{n}, \mathsf{t} \in \mathbb{N}$ *be the number of servers, respectively the number of servers necessary to evaluate in $\mathcal{P}_{tpOPRF}$. Suppose the* $\mathsf{OM\text{-}B\text{-}gapDH}$ *assumption holds true, then*

$$\mathcal{P}_{tpOPRF} \leq \mathcal{F}_{tpOPRF}$$

in the $(\mathcal{F}_{\text{auth}}, \mathcal{F}_{\text{ro}}^{i \in [1,2,f,b,s]})$-hybrid world.

The simulator $\mathcal{S}$ operates as a single machine connected to the ideal functionality $\mathcal{F}_{\text{tpOPRF}}$ and the environment $\mathcal{E}$ through their network interfaces. During execution, $\mathcal{S}$ accepts and processes all incoming messages. Internally, it mostly emulates the real-world protocol $\mathcal{P}_{\text{tpOPRF}}$ and uses the information leaked by $\mathcal{F}_{\text{tpOPRF}}$ to construct messages for the environment indistinguishable from those originating from real parties. In the event of server compromise, $\mathcal{S}$ outputs a secret key, adequately responds to adversarial messages, and consistently answers queries to the random oracles. We only consider the behavior of honest parties, forwarding messages addressed to corrupt parties to the environment. The complete simulator strategy of $\mathcal{S}$ is depicted in the full version of the paper.

The main challenges in constructing the simulator involve simulating the client's behavior without access to the private input x_{priv}, handling evaluation finalization without this private input, and ensuring consistency with the ideal functionality during key refresh and server corruptions. To address these challenges, the simulator employs the following strategies:

1. **Evaluation Initialization:** Since the simulator lacks knowledge of the client's private input x_{priv}, it must simulate the client's initialization phase in a way that is indistinguishable from the real protocol. To achieve this, the simulator selects a random scalar $a \in \mathbb{Z}_q^*$ and computes $[p]_1 = a \cdot [1]_1$, effectively mimicking the blinding operation performed in the real protocol, where $[p]_1 = r \cdot \mathsf{H}_1(x_{priv})$ with a random blinding factor r. This approach ensures that $[p]_1$ remains a uniformly random group element, preserving the distribution expected by the servers. By doing so, the simulator maintains consistency in the server's view, ensuring that the messages it sends are indistinguishable from those in the real protocol, despite not knowing x_{priv}.
2. **Evaluation Finalization:** In the evaluation finalization phase, the simulator must determine the PRF key used by potentially dishonest servers to interact correctly with the ideal functionality. Without knowledge of x_{priv}, the simulator reconstructs the public key pk' from the received messages. It uses the scalar a from the initialization phase and retrieves the trapdoor y injected in the random oracle $\mathsf{H}_2(x_{pub})$ to compute $\mathsf{pk}' = a^{-1} \cdot y^{-1} \cdot [u]_t$, where $[u]_t$ is the aggregate response from the servers. If pk' is not already associated with a label, it assigns a new label and stores the tuple $(label, [1]_1, [1]_2, \mathsf{pk}')$. This allows the simulator to send the `FinishEval` message to the ideal functionality with the correct label, ensuring that the PRF outputs correspond to those in the real protocol. By accurately simulating this phase, the simulator maintains the consistency required for the indistinguishability of the two executions.
3. **Random Oracle Queries:** The simulator must handle queries to the random oracles in a manner that preserves consistency between the simulated protocol and the ideal functionality. It does so by programming the random oracles and embedding trapdoors when necessary. For example, when a query to H_f (the final hash function) is received, especially from a corrupted client, the simulator determines the public key pk' consistent with the computation of $[u]_t$ by using the trapdoors embedded in $\mathsf{H}_1(x_{priv})$ and $\mathsf{H}_2(x_{pub})$. If pk' corresponds to the honest key label, the simulator checks whether sufficient servers were involved in the computation to satisfy the ideal functionality's requirements. It programs the random oracle to return the PRF value obtained from the ideal functionality, ensuring that any evaluations performed by the adversary result in outputs consistent with those expected in the ideal world. This careful management of random oracle responses prevents the adversary from distinguishing between the real and ideal executions based on the outputs of these oracles.

We demonstrate that any discrepancies between the real and ideal executions occur with negligible probability, primarily due to rare failure events. These events include situations where the adversary manages to generate valid protocol messages without sufficient interaction with honest servers or exploits collisions in the hash functions. We bound the probability of these failure events using the One-More Gap Diffie-Hellman ($\mathsf{OM\text{-}B\text{-}gapDH}$) assumption or statistical bounds. Specifically, we show that if an adversary could cause the simulator to fail with non-negligible probability, it would imply an ability to solve the $\mathsf{OM\text{-}B\text{-}gapDH}$ problem, which contradicts the assumed hardness of this problem.

We use our Domain-Isolating Oracle Programming (DIOP) proof technique and guess the public value $x_{pub}{}^*$ for which the break happens to inject into that "domain" the $\mathsf{OM\text{-}B\text{-}gapDH}$ assumption by setting $\mathsf{H}_2(x_{pub}{}^*) \leftarrow \mathsf{Targ}_2$. For all other x_{pub}, we inject a trapdoor into H_2 so we can simulate everything without using the Help oracle of the $\mathsf{OM\text{-}B\text{-}gapDH}$ assumption. We also inject challenge elements from Targ_1 into H_1. By carefully designing the simulator to handle all possible interactions and ensuring that any other differences between the simulated and real protocol executions are statistically negligible, we conclude that our tpOPRF protocol achieves composable security within the UC framework.

Acknowledgments

This work was partially supported by Deutsche Forschungsgemeinschaft as part of the Research and Training Group 2475 "Cybercrime and Forensic Computing" (grant number 393541319/GRK2475/1-2019), and by the Smart Networks and Services Joint Undertaking (SNS JU) under the European Union's Horizon Europe research and innovation program in the scope of the CONFIDENTIAL6G project under Grant Agreement 101096435. The contents of this publication are the sole responsibility of the authors and do not in any way reflect the views of the EU. his research was partly funded by SBA Research (SBA-K1 NGC), a COMET Center within the COMET – Competence Centers for Excellent Technologies Programme and funded by BMIMI, BMWET, and the federal state of Vienna.

References

[1] Elaine Barker. *NIST Special Publication 800-57 Part 1, Revision 5.* `https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf`. [Online; accessed 21-January-2025].

[2] C. Baum et al. "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)*. Los Alamitos, CA, USA: IEEE Computer Society, 2020, pp. 587–606. DOI: `10.1109/EuroSP48549.2020.00044`.

[3] Daniel Bourdrez et al. *The OPAQUE Augmented PAKE Protocol.* `https://datatracker.ietf.org/doc/draft-irtf-cfrg-opaque/`. [Online, accessed 10/02/24].

[4] Julian Brost et al. "Threshold Password-Hardened Encryption Services". In: *ACM CCS 2020: 27th Conference on Computer and Communications Security*. Ed. by Jay Ligatti et al. ACM Press, Nov. 2020, pp. 409–424. DOI: 10.1145/3372297.3417266.

[5] Jan Camenisch, Anja Lehmann, and Gregory Neven. "Optimal Distributed Password Verification". In: *ACM CCS 2015: 22nd Conference on Computer and Communications Security*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. ACM Press, Oct. 2015, pp. 182–194. DOI: 10.1145/2810103.2813722.

[6] Jan Camenisch et al. "iUC: Flexible Universal Composability Made Simple". In: *Advances in Cryptology – ASIACRYPT 2019, Part III*. Ed. by Steven D. Galbraith and Shiho Moriai. Vol. 11923. Lecture Notes in Computer Science. Springer, Cham, Dec. 2019, pp. 191–221. DOI: 10.1007/978-3-030-34618-8_7.

[7] Ran Canetti. "Universally Composable Security". In: *J. ACM* 67.5 (2020), 28:1–28:94. DOI: 10.1145/3402457. URL: https://doi.org/10.1145/3402457.

[8] Ran Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *42nd Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Oct. 2001, pp. 136–145. DOI: 10.1109/SFCS.2001.959888.

[9] S. Casacuberta, J. Hesse, and A. Lehmann. "SoK: Oblivious Pseudorandom Functions". In: *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. Los Alamitos, CA, USA: IEEE Computer Society, 2022, pp. 625–646. DOI: 10.1109/EuroSP53844.2022.00045.

[10] Poulami Das, Julia Hesse, and Anja Lehmann. "DPaSE: Distributed Password-Authenticated Symmetric-Key Encryption, or How to Get Many Keys from One Password". In: *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2022, 682–696. DOI: 10.1145/3488932.3517389.

[11] Alex Davidson et al. "Privacy Pass: Bypassing Internet Challenges Anonymously". In: *Proceedings on Privacy Enhancing Technologies* 2018.3 (July 2018), pp. 164–180. DOI: 10.1515/popets-2018-0026.

[12] Gareth T. Davies et al. "Security Analysis of the WhatsApp End-to-End Encrypted Backup Protocol". In: *Advances in Cryptology – CRYPTO 2023, Part IV*. Ed. by Helena Handschuh and Anna Lysyanskaya. Vol. 14084. Lecture Notes in Computer Science. Springer, Cham, Aug. 2023, pp. 330–361. DOI: 10.1007/978-3-031-38551-3_11.

[13] Alex Escala et al. "An Algebraic Framework for Diffie-Hellman Assumptions". In: *Advances in Cryptology – CRYPTO 2013, Part II*. Ed. by Ran Canetti and Juan A. Garay. Vol. 8043. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Aug. 2013, pp. 129–147. DOI: 10.1007/978-3-642-40084-1_8.

[14] Adam Everspaugh et al. "The Pythia PRF Service". In: *USENIX Security 2015: 24th USENIX Security Symposium*. Ed. by Jaeyeon Jung and Thorsten Holz. USENIX Association, Aug. 2015, pp. 547–562.

[15] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. "The Algebraic Group Model and its Applications". In: *Advances in Cryptology – CRYPTO 2018, Part II*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10992. Lecture Notes in Computer Science. Springer, Cham, Aug. 2018, pp. 33–62. DOI: 10.1007/978-3-319-96881-0_2.

[16] Google. *Google Cloud KMS Documentation: Key rotation*. https://cloud.google.com/kms/docs/key-rotation. [Online; accessed 21-January-2025].

[17] Yanqi Gu et al. "Threshold PAKE with Security Against Compromise of All Servers". In: *Advances in Cryptology - ASIACRYPT 2024 - 30th International Conference on the Theory and Application of Cryptology and Information Security, Kolkata, India, December 9-13, 2024, Proceedings, Part V*. Ed. by Kai-Min Chung and Yu Sasaki. Vol. 15488. Lecture Notes in Computer Science. Springer, 2024, pp. 66–100. DOI: 10.1007/978-981-96-0935-2_3. URL: https://doi.org/10.1007/978-981-96-0935-2_3.

[18] Dennis Hofheinz and Victor Shoup. "GNUC: A New Universal Composability Framework". In: *Journal of Cryptology* 28.3 (July 2015), pp. 423–508. DOI: 10.1007/s00145-013-9160-y.

[19] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. "Round-Optimal Password-Protected Secret Sharing and T-PAKE in the Password-Only Model". In: *Advances in Cryptology – ASIACRYPT 2014, Part II*. Ed. by Palash Sarkar and Tetsu Iwata. Vol. 8874. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Dec. 2014, pp. 233–253. DOI: 10.1007/978-3-662-45608-8_13.

[20] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. "OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-computation Attacks". In: *Advances in Cryptology – EUROCRYPT 2018, Part III*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10822. Lecture Notes in Computer Science. Springer, Cham, 2018, pp. 456–486. DOI: 10.1007/978-3-319-78372-7_15.

[21] Stanislaw Jarecki et al. "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)*. 2016, pp. 276–291. DOI: 10.1109/EuroSP.2016.30.

[22] Stanislaw Jarecki et al. "TOPPSS: Cost-Minimal Password-Protected Secret Sharing Based on Threshold OPRF". In: *ACNS 17: 15th International Conference on Applied Cryptography and Network Security*. Ed. by Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi. Vol. 10355. Lecture Notes in Computer Science. Springer, Cham, July 2017, pp. 39–58. DOI: 10.1007/978-3-319-61204-1_3.

[23] Shuichi Katsumata, Michael Reichle, and Kaoru Takemure. "Adaptively Secure 5 Round Threshold Signatures from MLWE/MSIS and DL with

Rewinding". In: *Advances in Cryptology – CRYPTO 2024, Part VII*. Ed. by Leonid Reyzin and Douglas Stebila. Vol. 14926. Lecture Notes in Computer Science. Springer, Cham, Aug. 2024, pp. 459–491. DOI: 10.1007/978-3-031-68394-7_15.

[24] Ralf Küsters, Max Tuengerthal, and Daniel Rausch. "The IITM Model: A Simple and Expressive Model for Universal Composability". In: *Journal of Cryptology* 33.4 (Oct. 2020), pp. 1461–1584. DOI: 10.1007/s00145-020-09352-1.

[25] Ueli Maurer. "Constructive Cryptography - A Primer (Invited Paper)". In: *FC 2010: 14th International Conference on Financial Cryptography and Data Security*. Ed. by Radu Sion. Vol. 6052. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Jan. 2010, p. 1. DOI: 10.1007/978-3-642-14577-3_1.

[26] OWASP Cheat Sheets Series Team. *Cryptographic Storage Cheat Sheet*. https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html. [Online; accessed 12-February-2025].

[27] Rafaël Del Pino et al. "Threshold Raccoon: Practical Threshold Signatures from Standard Lattice Assumptions". In: *Advances in Cryptology – EUROCRYPT 2024, Part II*. Ed. by Marc Joye and Gregor Leander. Vol. 14652. Lecture Notes in Computer Science. Springer, Cham, May 2024, pp. 219–248. DOI: 10.1007/978-3-031-58723-8_8.

[28] Security Standards Council. *PCI DSS v4.0.1 (Section 15, 3.7.4 and 3.7.5)*. https://docs-prv.pcisecuritystandards.org/PCI%20DSS/Standard/PCI-DSS-v4_0_1.pdf. [Online; accessed 21-January-2025].

[29] Adi Shamir. "How to Share a Secret". In: *Communications of the Association for Computing Machinery* 22.11 (Nov. 1979), pp. 612–613. DOI: 10.1145/359168.359176.

[30] Jeremy Stieglitz. *The curious case of faster AWS KMS symmetric key rotation*. https://aws.amazon.com/blogs/security/the-curious-case-of-faster-aws-kms-symmetric-key-rotation/. [Online; accessed 12-February-2025].

[31] WhatsApp. *Security of End-To-End Encrypted Backups*. www.whatsapp.com/security/WhatsApp_Security_Encrypted_Backups_Whitepaper.pdf. [Online, accessed 10/02/24]. 2021.

[32] Cong Zhang, Hong-Sheng Zhou, and Jonathan Katz. "An Analysis of the Algebraic Group Model". In: *Advances in Cryptology – ASIACRYPT 2022, Part IV*. Ed. by Shweta Agrawal and Dongdai Lin. Vol. 13794. Lecture Notes in Computer Science. Springer, Cham, Dec. 2022, pp. 310–322. DOI: 10.1007/978-3-031-22972-5_11.