How to Obfuscate MPC Inputs

Ian McQuoid [*†] Mike Rosulek[*†] Jiayu Xu[*]

October 27, 2022

Abstract

We introduce the idea of input obfuscation for secure two-party computation (io2PC). Suppose Alice holds a private value x and wants to allow clients to learn $f(x, y_i)$, for their choice of y_i, via a secure computation protocol. The goal of io2PC is for Alice to encode x so that an adversary who compromises her storage gets only oracle access to the function $f(x, \cdot)$. At the same time, there must be a 2PC protocol for computing $f(x, y)$ that takes only this encoding (and not the plaintext x) as input.

We show how to achieve io2PC for functions that have virtual black-box (VBB) obfuscation in either the random oracle model or generic group model. For functions that can be VBB-obfuscated in the random oracle model, we provide an io2PC protocol by replacing the random oracle with an oblivious PRF. For functions that can be VBB-obfuscated in the generic group model, we show how Alice can instantiate a "personalized" generic group. A personalized generic group is one where only Alice can perform the algebraic operations of the group, but where she can let others perform operations in that group via an oblivious interactive protocol.

1 Introduction

Alice has invested significant resources into training a machine-learning classifier. She decides to capitalize on her investment by creating a service where customers can pay her to classify inputs of their choice. The classifier itself is sensitive, and so are the inputs of Alice's clients, so her service uses secure two-party computation (2PC) to perform these classifications. She deploys a server that repeatedly runs the 2PC protocol with customers. This server is a high-value target for attackers, since it must store the details of Alice's proprietary classifier. If a hacker compromises Alice's server it is unavoidable that he learns her classifier… or is it?

Input obfuscation for 2PC. Abstractly, Alice has an input x, and she wants to use a 2PC protocol to allow customers to repeatedly learn $f(x, y_i)$ for any y_i of their choice. An attacker who compromises her computer can gain oracle access to the function $f(x, \cdot)$ by running the 2PC protocol in its head, playing the role of Alice using her private state information which was compromised. In this work, we investigate whether compromising Alice's computer can leak *no more than* oracle access to $f(x, \cdot)$. **Input obfuscation for 2PC (io2PC)** refers to (1) a way for Alice to encode her input x, along with (2) a 2PC protocol for computing functions of x that takes this encoding — not x — as input. The encoding itself should leak only oracle access to the function $f(x, \cdot)$.

[*]Oregon State University, {mcquoidi,rosulekm,xujiay}@oregonstate.edu
[†]First two authors partially supported by NSF award S2356A

Why isn't this trivial? If knowledge of Alice's encoded input is equivalent to having oracle access to $f(x, \cdot)$, then her encoded input is actually a **virtual-black-box (VBB)** obfuscation. So a natural approach is to use a 2PC protocol that takes the obfuscation from Alice, and the input y from Bob evaluates the obfuscation on y and gives the result to Bob.

Unfortunately, this natural approach does not work. The reason is that we require a strong definition of VBB described in Section 2.2 which precludes known constructions of non-trivial functions in the standard model as these obfuscations rely on weakened definitions of VBB [Wee05, Can97, CD08]. It *is* possible to construct VBB for trivial functions such as the constant function, but all (non-trivial) instances of VBB to our knowledge are in an *idealized* model such as the random oracle model [LPS04], the generic group model [BLMZ19], or the generic graded encodings model [BR13]. As the algorithm that evaluates a VBB obfuscation on an input will call the ideal model's oracle, this algorithm cannot be implemented inside a 2PC protocol.

One way to think about io2PC is designing a 2PC protocol for obliviously evaluating an obfuscated program, even if the obfuscation scheme requires an idealized model.

1.1 Overview of Our Results

We first formally define io2PC, and then show how to achieve it for certain classes of functions.

Inspiration from saPAKE. In io2PC we are interested in allowing a server to encode a function in such a way that even on compromise, the adversary only obtains oracle access to the underlying function. This kind of security property is similar to one found in the definition of **strong asymmetric password-authenticated key exchange (saPAKE)** [JKX18]. In saPAKE, a server wants to authenticate clients using passwords and stores only "digests" of the passwords so that when an adversary steals the server's storage, the adversary gains only oracle access to a password-checking functionality (*i.e.* it can submit a password guess and learn whether that guess is correct). In other words, the adversary gains oracle access to a *point function* for each user, with the distinguished point being the user's password. It is therefore natural to think of saPAKE as a special case of io2PC, considering only point functions.

Although the oracle saPAKE protocols provide on compromise is a point function, saPAKE protocols are much stronger than pure point functions as they allow for joint key establishment. To simplify, we can consider lighter VBB obfuscations of point functions in the random oracle model [LPS04]. An obfuscation of the point function $f(x, \cdot)$ simply consists of the value $\mathcal{O}_x = H(x)$, for random oracle H, where the obfuscation can be "evaluated" on y by computing $H(y)$ and comparing it to $\mathcal{O}_x$. As we will see in Section 3, this simple construction doesn't meet the security requirements for io2PC as it allows the oracle interaction in the obfuscation to take place before server compromise. This is exactly the issue that Jareki, Krawczyk, and Xu [JKX18] set out to solve for asymmetric PAKE (aPAKE) protocols. The authors present a compiler which augments an aPAKE protocol by replacing the client's input with the output of an **oblivious pseudo-random function (OPRF)** on the client's input. Roughly, an OPRF is a two-party protocol for evaluating a PRF F on a client's input x and a server's key k where the client learns the PRF output $F(k, x)$ and the server learns nothing. We discuss modeling this primitive in further detail in Section 4.1. This technique allows the server to limit access to the oracle behind an interactive protocol which prevents the adversary from evaluating the oracle calls locally until the server is compromised. It is tempting to apply this compiler directly to our VBB point function, and generally this idea serves as valid intuition for the techniques used in our compilers.

Our result for random-oracle VBB obfuscations. Our main constructions develop and extend the analogy of applying Jareki, Krawczyk, and Xu's compiler directly to VBB obfuscations. We construct $\mathsf{io2PC}$ for a function f, if the related class of functions $\mathcal{C}_f = \{f(x, \cdot) \mid x \in \{0,1\}^n\}$ has a VBB obfuscation in the random oracle model. The obfuscation scheme consists of algorithms Obf and $\mathsf{ObfEval}$ satisfying the following:

- **Correctness**: $\mathsf{ObfEval}^H(\mathsf{Obf}^H(x), y) = f(x, y)$
- **Virtual black box**: For any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that $\mathcal{A}^H(\mathsf{Obf}^H(x))$'s view can be simulated by $\mathcal{S}$ given only black-box access to $f(x, \cdot)$.

In our $\mathsf{io2PC}$ protocol Alice chooses and stores an OPRF key k and uses the keyed OPRF in place of a random oracle to compute $\mathcal{O}_x = \mathsf{Obf}^{\mathsf{OPRF}(k,\cdot)}(x)$. She then stores $\mathcal{O}_x$ instead of x for future interactions. As in the OPAQUE saPAKE protocol [JKX18], we require an OPRF protocol where knowledge of the key k only gives oracle access to $F(k, \cdot)$. Thus, even when an adversary steals the encoding $\mathcal{O}_x$, the OPRF still acts as a random oracle, in terms of observability and programmability, to the simulator. This is what allows us to reduce to VBB security and argue that $\mathcal{O}_x$ leaks no more than oracle access to $f(x, \cdot)$. It is indeed possible to realize such an OPRF protocol in the random oracle model; in this case, the OPRF algorithm itself makes calls to the random oracle. Since our simulator must be efficient, but reduces to the simulator for the VBB obfuscation, our results do not immediately generalize to virtual *grey*-box (VGB) obfuscations. This is because VGB simulators can be inefficient and would not be simulatable under our restrictions.

When a client wants to interactively evaluate $f(x, y)$, the goal is to instead run $\mathsf{ObfEval}^{\mathsf{OPRF}(k,\cdot)}(\mathcal{O}_x, y)$, since Alice holds only $\mathcal{O}_x$ instead of x. However, the two cannot simply run this computation as a 2PC protocol, since OPRF involves calls to the random oracle. Instead, Alice can send $\mathcal{O}_x$ to the client, who runs $\mathsf{ObfEval}^?(\mathcal{O}_x, y)$. The parties can then run an OPRF protocol each time $\mathsf{ObfEval}$ makes an oracle query.

Our result for generic-group obfuscation. In our random-oracle result, we can think of the OPRF as a "personalized" random oracle. It is a random function that only Alice, holding the OPRF key, can evaluate and her evaluations of this function are visible and programmable to the simulator. She can also allow a client to evaluate this function (without leaking the input to Alice) using the OPRF protocol.

Suppose we have a VBB obfuscation now in the generic group model. What is the analogy of a "personalized" generic group? How can Alice instantiate a group, for which only she has the key, which acts as a generic group with respect to the simulator, and yet she can grant access to the group operations via an oblivious protocol? We formalize a personalized generic group as an ideal functionality, and then show how to realize such a functionality. Of course, our protocol is in the generic group model, just as the OPRF ("personalized random oracle") protocol is in the random oracle model.

We show that our main $\mathsf{io2PC}$ technique also applies to VBB obfuscations in the generic group model. In other words, Alice can obfuscate her input, replacing the generic group with her personalized group during the obfuscation process. The client can evaluate the obfuscated program, deferring group operations to the oblivious personalized generic group protocol.

Additionally, we provide example applications of our personalized protocols and show that Canetti, Rothblum, and Varia's hyperplane-membership obfuscation [CRV10] is indeed a VBB obfuscation in the generic group model. Previously, the obfuscation was proven VBB with an inefficient simulator, under the Strong DDH assumption. Using this hyperplane obfuscation in our main protocol, we achieve an io2PC for hyperplane membership.

We conjecture that io2PC is possible for functions that are VBB-obfuscatable in the generic graded encoding model (*i.e.* all circuits [BR14]); however, we leave this result for future work.

1.2 Related Work

Upon server compromise, an adversary learns no more than oracle access to some residual function. Specific instances of this kind of property have been considered previously: in the context of [strong] asymmetric password-authenticated key exchange (aPAKE) [BM93, JKX18], where server compromise should reveal no more than an equality-test oracle; and by Thomas *et al.* [TPY+19], where server compromise should reveal no more than a set-membership oracle. Our study of io2PC systematizes security properties and constructions of this kind, which have previously been studied in an *ad hoc* way.

Beyond the context of server compromise, the more general idea of leaking oracle access appears in some MPC models: In both *non-interactive multiparty computation (NIMPC)* [BGI+14] and the one-pass computation model [HLP11], each party speaks only once in the protocol, with the difference in models being the communication pattern (star topology vs path topology). In these models, it is inevitable that certain types of corruption allow the adversary to re-execute the protocol on different inputs an unlimited number of times. Such an adversary can thereby learn the output of the function on many inputs of its choice, with the honest parties' inputs being fixed. Therefore, the *best possible* security in these models is if the protocol leaks no more than oracle access to this residual function.

Beyond this similarity of defining best-possible security with respect to a residual function oracle, there are important differences between these prior works and ours. In the NIMPC protocols of [BGI+14] and one-pass protocols of [HLP11, GMRW13], the residual functions are completely learnable from oracle queries, either by virtue of being over a small domain, or by being algebraically simple. Our work is meant to be used with unlearnable residual functions — for example, we instantiate our framework with point functions and hyperplane membership queries.

More fundamentally, prior works like [HIJ+17] in the NIMPC model define security in the style of *indistinguishability obfuscation (iO)* — if two vectors of inputs for honest parties result in *functionally identical* residual functions, then the protocol must hide which input vector the honest parties use. This kind of definition for MPC is not conducive to composable security. By contrast, we explicitly require a virtual black-box (VBB) style of security, and define security in the UC framework. Our VBB-style definition also models the fact that, after compromising the server, the adversary must expend some effort each time it wants to evaluate the residual function.

2 Preliminaries

Let κ be the security parameter. We assume that all algorithms have 1^κ as input and do not explicitly write it.

2.1 Idealized Models

In an idealized model, all parties have oracle access to some exponentially large random object. In the *random oracle model*, the random object is a function $H : \{0,1\}^* \to \{0,1\}^n$. In the *ideal permutation model*, the random object is a pair of functions $\Pi, \Pi^{-1} : X \to X$ where Π and Π^{-1} are inverses.

We also consider the generic group model, which we discuss in more detail in Section 5.1.

Immediately below, we define VBB obfuscation in an idealized model, making the definition agnostic with respect to the actual choice of idealized model. We simply let all algorithms have oracle access to some idealized oracle Ora, which may be a random oracle or a generic group.

2.2 Obfuscation

Definition 1. *Let $\mathcal{C}_f = \{f(x, \cdot) \mid x \in \{0,1\}^*\}$ be a class of functions. An **obfuscation** for $\mathcal{C}_f$ (in the Ora-idealized model) is a tuple of polynomial-time algorithms $(\mathsf{Obf}, \mathsf{ObfEval})$, where*

- $\mathsf{Obf}^{\mathsf{Ora}}(x)$ *outputs an **obfuscated program** $\mathcal{O}_x$;*
- $\mathsf{ObfEval}^{\mathsf{Ora}}(\mathcal{O}_x, y)$ *outputs a value z in the range of f.*

The obfuscation satisfies correctness if for all x, y, we have $\mathsf{ObfEval}^{\mathsf{Ora}}(\mathsf{Obf}^{\mathsf{Ora}}(x), y) = f(x, y)$ with overwhelming probability.

We often omit explicitly writing Ora if it is clear from the context.

Looking ahead, we replace the idealized oracle in $\mathsf{ObfEval}$ with an interactive protocol. Hence, we must require that the number of oracle queries does not depend on the input.

Definition 2. *An obfuscation $(\mathsf{Obf}, \mathsf{ObfEval})$ for $\mathcal{C}_f$ has **input-independent** query complexity if there is a polynomial function c such that for all x, y, $\mathsf{ObfEval}^{\mathsf{Ora}}(\mathcal{O}_x, y)$ makes $c(\kappa)$ queries to its Ora oracle. Throughout the paper, we then refer to an obfuscation with this property as a triple $(\mathsf{Obf}, \mathsf{ObfEval}, c)$.*

Virtual black-box (VBB) security means that holding an obfuscated program is equivalent to having oracle access to the function being obfuscated. In our $\mathsf{io2PC}$ protocol, we need to explicitly relate the number of queries an adversary makes to its idealized oracle, and the number of queries the simulator makes to its function oracle.

Definition 3. *An obfuscation $(\mathsf{Obf}, \mathsf{ObfEval}, c)$ has **virtual black-box (VBB) security with simulation rate** r if there exists a polynomial-time simulator $\mathsf{Sim} = (\mathsf{Sim}_0, \mathsf{Sim}_1)$ such that for any polynomial-time adversary $\mathcal{A}$ and any x, the distributions*

$$\{\mathcal{O}_x \leftarrow \mathsf{Obf}^{\mathsf{Ora}}(x); \mathcal{A}^{\mathsf{Ora}}(\mathcal{O}_x)\} \qquad (\textit{real interaction})$$

$$\{(\mathcal{O}_x, \mathit{state}) \leftarrow \mathsf{Sim}_1(); \mathcal{A}^{\mathsf{Sim}_2^{f(x,\cdot)}(\mathit{state})}(\mathcal{O}_x)\} \qquad (\textit{ideal interaction})$$

are indistinguishable, and furthermore in the ideal interaction $Q_S \leq r \cdot \frac{Q_A}{c}$, where Q_S is the number of queries Sim_2 makes to its function oracle, and Q_A is the number of queries $\mathcal{A}$ makes to its oracle interface.

We also need the following extractability property of obfuscation to handle the case where a corrupt server generates an obfuscated program in our $\mathsf{io2PC}$ protocol.

Definition 4. *A VBB obfuscation $(\mathsf{Obf}, \mathsf{ObfEval}, c)$ for $\mathcal{C}_f$ is **extractable** if for any polynomial-time adversary $\mathcal{A}$, there is a polynomial-time algorithm $\mathsf{Extract}$ such that*

$$\Pr\left[\mathsf{ObfEval}^{\mathsf{Ora}}(\mathcal{O}, y) \neq f(x, y) : \begin{array}{c} (y, \mathcal{O}) \leftarrow \mathcal{A}^{\mathsf{Ora}} \\ x := \mathsf{Extract}(\mathcal{O}, \mathcal{H}) \end{array}\right]$$

is negligible, where $\mathcal{H}$ is the list of $\mathcal{A}$'s queries.

In Section 6 we describe examples of obfuscation schemes that satisfy these definitions.

Standard Model VBB Recall that at least trivial VBB obfuscations are possible in the standard model. Even in an idealized model, if ObfEval never queries its oracle, then we have a VBB obfuscation in the standard model. However, our constructions need something slightly stronger than VBB. In particular, Definition 3 and Definition 4 require an idealized model for non-trivial functions. A standard-model VBB allows the evaluator to learn $f(x, \cdot)$ on an unbounded number of inputs, for the cost of 0 oracle queries, making the simulation rate for Definition 3 infinite. A similar observation has been made in the context of asymmetric PAKE[Hes20]: aPAKE seems impossible to achieve in the standard model, as measuring the time of an offline dictionary attack requires counting the adversary's oracle queries. In Definition 4, the simulator's only advantage over a regular adversary is that it can observe the obfuscator's idealized oracle queries.

So our protocol paradigm is incompatible with (at least non-trivial) standard-model VBB. But the spirit of io2PC is possible for standard-model VBB. The server stores an obfuscation $\mathcal{O}_x$ of $f(x, \cdot)$. The parties can do a standard 2PC protocol computing $(\mathcal{O}_x, y) \to \mathsf{ObfEval}(\mathcal{O}_x, y)$, which is possible because ObfEval is a standard-model program. Upon compromising the server, an adversary learns only $\mathcal{O}_x$ which is equivalent to oracle access to $f(x, \cdot)$ by the VBB property. This protocol does not achieve our specific io2PC functionality, though, because the simulator cannot perform the necessary extractions of x from $\mathcal{O}_x$, and of an adversary's oracle queries to $f(x, \cdot)$ after compromising the server to learn $\mathcal{O}_x$.

3 Defining io2PC

In this section, we formally define io2PC. The ideal functionality is presented in Figure 1. In $\mathcal{F}_{\text{iO2PC}}$ and future functionalities, we leverage the universal composability framework's ability to analyze a single protocol instance by providing unique session and subsession identifiers ($sid, ssid$).

Intuitively, io2PC can be thought of as an extension of VBB obfuscation to an interactive setting where the server may store its obfuscated input for long periods. This setting has been studied in the context of (strong) asymmetric PAKE [GMR06, JKX18], where the server stores a "password file" (*e.g.* the hash of its password) instead of the plain password. Similar to the asymmetric PAKE functionality, this is modeled as follows: In the initialization phase, the server sends its input to $\mathcal{F}_{\text{iO2PC}}$ who stores it. After that, the functionality provides an interface for the adversary to compromise the server — the Compromise query — which corresponds to stealing the server's long-term storage in the real world. This allows the adversary to perform *offline evaluations*, in which it evaluates the function primed on the server's input, without any online interaction.

In an *online evaluation*, the server can use the stored input, or use a replacement input if the server is corrupt. This is meant to model the real-world scenario where a corrupt server executes the protocol on fresh input instead of using stored input. Finally, the client may query against the functionality and receive the function result on the client and server's inputs.

3.1 Simulation Rate

Our eventual io2PC protocol has the following interesting property. A corrupt client may perform k different IOEval sessions in such a way that it eventually learns (only) k outputs of the function, but the simulator cannot extract *any* of the client's inputs until after the kth session.[1] We handle this issue in $\mathcal{F}_{\text{iO2PC}}$ with a *ticketing* mechanism. During each IOEval the client need not immediately

[1] Essentially, our protocol for IOEval simply allows the client to make some fixed number of OPRF queries. Instead of using those OPRF queries for k *sequential* evaluations of the function, the client can schedule the OPRF queries in parallel — *e.g.*., the first query in all k evaluations, then the second query in all k evaluations, etc.

Parameters:

- client C, server S, and ideal adversary $\mathcal{A}^*$

Storage:

- three maps, status, budget and input

On command (Init, sid, x) from S:

1. If $\mathsf{status}[sid]$ is defined: ignore the message.
2. Set $\mathsf{status}[sid] := \mathsf{active}$.
3. Set $\mathsf{input}[sid] := x$.
4. Set $\mathsf{budget}[sid] := 0$.
5. Send $(\mathsf{Init}, sid, \mathsf{S})$ to $\mathcal{A}^*$.

On command $(\mathsf{Compromise}, sid)$ from $\mathcal{A}^*$:

6. Set $\mathsf{status}[sid] := \mathsf{compromised}$.

On command $(\mathsf{OfflineEval}, sid, y)$ from party $\mathsf{P} \in \{\mathcal{A}^*, \mathsf{S}\}$:

7. If $\mathsf{P} = \mathcal{A}^*$, and either $\mathsf{status}[sid] \neq \mathsf{compromised}$ or S is honest: ignore the message.
8. If $\mathsf{status}[sid]$ is undefined: send $(\mathsf{IOEval}, sid, \perp)$ to P.
9. Otherwise, retrieve $x := \mathsf{input}[sid]$ and send $(\mathsf{OfflineEval}, sid, f(x, y))$ to P.

On command $(\mathsf{IOEval}, sid, ssid, x')$ from S:

10. If S is honest, retrieve $x := \mathsf{input}[sid]$, otherwise, set $x := x'$.
11. Send $(\mathsf{IOEval}, sid, ssid, \mathsf{S})$ to $\mathcal{A}^*$.
12. If C is corrupt: set $\mathsf{budget}[sid] := \mathsf{budget}[sid] + r$.
13. Wait for $(\mathsf{IOEval}, sid, ssid, y)$ from C or $(\mathsf{Abort}, sid, ssid)$ from $\mathcal{A}^*$.
14. If honest C sends $(\mathsf{IOEval}, sid, ssid, y)$: send $(\mathsf{IOEval}, sid, ssid, f(x, y))$ to C.
15. If corrupt C sends $(\mathsf{IOEval}, sid, ssid, y)$:
    - Set $\mathsf{budget}[sid] := \mathsf{budget}[sid] - 1$ and send $(\mathsf{IOEval}, sid, ssid, f(x, y))$ to C.
16. If $\mathcal{A}^*$ sends $(\mathsf{Abort}, sid, ssid)$:
    - If S is corrupt, send $(\mathsf{IOEval}, sid, \perp)$ to C.

On command $(\mathsf{Redeem}, sid, ssid, y)$ from $\mathcal{A}^*$:

17. If $\mathsf{status}[sid]$ is undefined: ignore the message.
18. Retrieve $x := \mathsf{input}[sid]$.
19. If $\mathsf{budget}[sid] = 0$, send $(\mathsf{IOEval}, sid, ssid, \perp)$ to $\mathcal{A}^*$.
20. Otherwise set $\mathsf{budget}[sid] := \mathsf{budget}[sid] - 1$ and send $(\mathsf{IOEval}, sid, ssid, f(x, y))$ to $\mathcal{A}^*$.

Figure 1: The functionality $\mathcal{F}_{\text{iO2PC}}$ computing $\mathcal{C}_f$ with simulation rate r

learn the output of f. Rather, the functionality grants a ticket that entitles the client to one evaluation of f, and this evaluation of f can be redeemed at any later point.

More generally, the functionality can grant r tickets for a single IOEval. Intuitively, think of IOEval as granting some resources to the client, which it can use to evaluate f. But there may be "cheap" inputs to f which require r times fewer resources than the worst case, in which case one session of IOEval may provide enough resources for a corrupt client to learn r outputs of the function.

3.2 Server Compromise and Offline Evaluation

Following the treatment of server compromise in aPAKE [GMR06, JKX18], our functionality separates Byzantine server **corruption** and server **compromise**. Upon being compromised, the server only leaks its long-term storage to the adversary but *remains honest*; in other words, a Compromise query does not allow the server to be controlled by the adversary. On the other hand, server corruption not only leaks the entire state of the server to the adversary, but additionally allows for complete control of the server. We consider the *static corruption* model, but crucially, we allow the adversary to *adaptively compromise* an honest server. This is reflected in our $\mathcal{F}_{\text{iO2PC}}$ functionality: the adversary can compromise the server via a Compromise message at any time, which marks the status of the current session compromised, after which the adversary can perform offline evaluations. However, in subsequent online evaluations, a compromised server is still treated as honest.

Furthermore, similar to the (strong) aPAKE functionality, we require that both Compromise and OfflineEval messages be accounted for by the environment. In particular, this means that the ideal adversary (simulator) cannot take certain actions without some corresponding real-world event caused by the real adversary: the ideal adversary cannot send Compromise unless the real adversary compromises the server, and it cannot send OfflineEval unless the real adversary performs some "work" (in the form of random oracle or generic group queries) that corresponds to evaluating f. The rationale is similar to why Byzantine corruptions are accounted for by the environment in the UC framework: to prevent the simulator from corrupting all parties and making the simulation trivial. Indeed, the Compromise and OfflineEval messages can be formally modeled as a special form of corruption — see [Can01, Hes20] for a detailed description.

3.3 Preventing Precomputation

We require that OfflineEval commands sent by a corrupt client are accounted for by the environment, and can only be issued by the environment if the real-world adversary does some observable "work". Crucially, that "work" must happen *after* server compromise. This requirement means that the client cannot "precompute" work before compromise that permits the simulator to send many OfflineEval commands instantly upon server compromise.

This feature is analogous to the definition of **strong** aPAKE. In non-strong aPAKE, an adversary can learn all parties' stored passwords *instantly* upon compromise of the password file. In strong aPAKE, an adversary can only make password guesses after compromise, and these password guesses must be accounted for by the environment — *i.e.* they must correspond to observable work performed *after compromise* by the adversary. In this sense, our $\mathcal{F}_{\text{iO2PC}}$ functionality is analogous to the *strong* flavor of aPAKE.

4 io2PC for Random-Oracle-Model Obfuscation

In this section, we describe a compiler for realizing io2PC from functions that have VBB obfuscation in the random oracle model. Let us first recall the JKX compiler [JKX18] from aPAKE to saPAKE. The compiler works by replacing the input password pw to the starting aPAKE protocol with the evaluation $\mathsf{OPRF}(pw)$. This compiler serves as a source of intuition for an intermediate compiler for io2PC which takes a VBB obfuscation in the random oracle model and replaces the input x to each random oracle evaluation with $\mathsf{OPRF}(x)$.

Recall the point function obfuscation $\mathcal{O}_x = H(x)$ for random oracle H, with evaluation $H(\cdot) \stackrel{?}{=} \mathcal{O}_x$ [LPS04]. Applying this compiler, we arrive at the io2PC protocol in which the server stores $\mathcal{O}'_x = H(\mathsf{OPRF}(x))$ and interactively evaluates $H(\mathsf{OPRF}(\cdot)) \stackrel{?}{=} \mathcal{O}'_x$ by sending $\mathcal{O}'_x$ to the client then acting as the server in an OPRF protocol. This intuitive compiler is not far from the truth, as for random oracle H, $H(\mathsf{OPRF}(\cdot))$ is itself an OPRF, so we simplify slightly by instead replacing all random oracle invocations directly with OPRF invocations. Indeed, with a small modification replacing the OPRF in this compiler with a verifiable OPRF (VOPRF), we achieve the compiler described in Section 4.2.

4.1 Oblivious PRF

An Oblivious Pseudorandom Function (OPRF) [FIPR05] for a Pseudorandom Function (PRF) family $F_{(\cdot)}$ is, generally, a two-party protocol for realizing the functionality where a server who holds a key k and a client who holds an input x evaluate $F_k(x)$ with output $(\epsilon, F_k(x))$. Namely, the client learns $F_k(x)$ and the server learning nothing about the input x or the output $F_k(x)$. OPRFs have found many applications and have been extended to support verification of client and server inputs [JKK14, JL09].

Our functionality $\mathcal{F}_{\mathrm{VOPRF}}$, in Figure 2, for a verifiable OPRF (VOPRF) with active compromise closely follows the functionality of Jarecki, Krawczyk, and Xu [JKX18].

The main difference between the functionality in Figure 2 and the comparable OPRF functionality [JKX18] is the addition of the Abort query. $\mathcal{F}_{\mathrm{VOPRF}}$ is *verifiable* in the sense that it allows for a client to abort in the face of a corrupt server who may, for example, commit to a PRF key through a public key and use a different PRF key during evaluation. Instead of presenting multiple tables indexed by a function parameter as in previous functionalities [JKK14], $\mathcal{F}_{\mathrm{VOPRF}}$ uses a single key provided during initialization and then exposes Abort. This verifiability also models the client's ability to verify consistent key usage between various OPRF interactions with a given server. The client will be assured that all interactions compute the same underlying PRF or else the client can abort.

Our VOPRF functionality Figure 2 differs from others in the literature (*e.g.*, [ADDS21]). Our definition requires that the outputs of the VOPRF are pseudorandom even to the server. This requirement is related to the fact that io2PC (like asymmetric PAKE) requires programmability of outputs from the simulator, even for the server's *non-interactive* evaluations of the OPRF. [Hes20] In particular, this means that to provide input obfuscation to non-trivial functions we must rely on some assumption with stronger programmability than afforded by a CRS. As such, we cannot achieve this functionality outside a *strongly* programmable model such as the random oracle model or the generic group model.

The requirement that the outputs of the VOPRF are pseudorandom even to the server is necessary to realize the intuition that a corrupt client can only gain *oracle access* to the underlying PRF on server compromise. We like to think of such a (V)OPRF as a "personalized" Random Oracle which the server can let another party evaluate, privately, on some input. When an honest

Parameters:
- client C, server S, and ideal adversary $\mathcal{A}^*$

Storage:
- two maps, status and F

On command $(\mathsf{VOPRFInit}, sid)$ from S:
1. If $\mathsf{status}[sid]$ is defined: ignore the message.
2. Set $\mathsf{status}[sid] := \mathsf{active}$.
3. Send $(\mathsf{VOPRFInit}, sid, \mathsf{S})$ to $\mathcal{A}^*$.

On command $(\mathsf{Compromise}, sid)$ from $\mathcal{A}^*$:

4. Set $\mathsf{status}[sid] := \mathsf{compromised}$.

On command $(\mathsf{OfflineEval}, sid, x)$ from party $\mathsf{P} \in \{\mathcal{A}^*, \mathsf{S}\}$:

5. If $\mathsf{status}[sid] \neq \mathsf{compromised}$ or S is not corrupted, and $\mathsf{P} \neq \mathsf{S}$: ignore the message.
6. If $\mathsf{status}[sid]$ is undefined: send $(\mathsf{VOPRFEval}, sid, \perp)$ to P.
7. Otherwise, if $F[x]$ is undefined, set $F[x] \leftarrow \mathbf{H}$, and send $(\mathsf{OfflineEval}, sid, F[x])$ to P.

On command $(\mathsf{VOPRFEval}, sid, ssid, x)$ from C:

8. If $\mathsf{status}[sid]$ is undefined: send $(\mathsf{VOPRFEval}, sid, \perp)$ to C.
9. Send $(\mathsf{VOPRFEval}, sid, ssid, \mathsf{C})$ to $\mathcal{A}^*$, $(\mathsf{VOPRFEval}, sid, ssid)$ to S, and wait for $(\mathsf{SComplete}, sid, ssid)$ from S.
10. Send $(\mathsf{SComplete}, sid, ssid, \mathsf{S})$ to $\mathcal{A}^*$ and wait for either $(\mathsf{Deliver}, sid, ssid)$ or $(\mathsf{Abort}, sid, ssid)$ from $\mathcal{A}^*$.
11. If $\mathcal{A}^*$ sends $(\mathsf{Deliver}, sid, ssid)$:
    - If $F[x]$ is undefined, set $F[x] \leftarrow \mathbf{H}$ and send $(\mathsf{VOPRFEval}, sid, F[x])$ to C.
12. If $\mathcal{A}^*$ instead sends $(\mathsf{Abort}, sid, ssid)$:
    - If S is corrupt, send $(\mathsf{VOPRFEval}, sid, \perp)$ to C.

Figure 2: The functionality $\mathcal{F}_{\mathrm{VOPRF}}$ for evaluating random function F with range $\mathbf{H}$

server is compromised by a corrupt client, the client gains the ability to evaluate this personal random function at will; however, since the outputs of the function are pseudorandom to the server they are also pseudorandom to an adversary who compromises the server's storage. To meet the idea of oracle access, these evaluations must also be observable. With these two properties, we can see that the exposed oracle is analogous to a personalized random oracle.

Jarecki, Kiayias, and Krawczyk [JKK14] provide an efficient UC instantiation of a VOPRF in the random oracle model under a one-more Gap DH assumption. We recall that protocol — called 2HashDH-NIZK therein for its eponymous entry and exit hashes — in Figure 3.

Similar to previous results for the 2HashDH-NIZK protocol [JKK14] in Figure 3 and its non-verifiable derivative [JKX18], we know that 2HashDH-NIZK satisfies our requirements for adaptive compromise, and relative to S's public key, 2HashDH-NIZK satisfies our verifiability requirements. The inclusion of a NIZK does not significantly modify the proof for adaptive compromise, and we may consider the existence of an authenticated channel, mediated through the authenticated channel functionality $\mathcal{F}_{AUTH}$ to provide the server's public key to the client. In situations where the public key of the server is known a priori to the client, we may drop the need for an authenticated channel; however, in the cases we consider for io2PC, the existence of an authenticated channel is already assumed.

<u>Parameters:</u>
Generator g of cyclic group of order q
Random Oracles $H_1(\cdot), H_2(\cdot), H_3(\cdot)$
Client C and Server S

<u>KeyGen:</u>
S samples $k \leftarrow \mathbb{Z}_q$.
S stores k and returns public key g^k.

<u>Compromise:</u>
S returns stored key k.

<u>Offline Evaluation:</u>
On input x, S returns $H_2(g^k, x, H_1(x)^k)$

<u>Online Evaluation:</u>
C, on input x, samples $r \leftarrow \mathbb{Z}_p$ and sends $(H_1(x))^r$ to S.
S, on message b from C sends $h = g^k$, b^k, and $\mathsf{NIZK}^{H_3}(b, b^k, g, g^k)$ to C.
C, on message h, c, π from S, verifies π is a valid proof then returns $H_2(h, x, c^{1/r})$.

Figure 3: VOPRF Protocol 2HashDH-NIZK

4.2 io2PC Protocol

We present our OPRF-based io2PC protocol in Figure 4. In the initialization phase, the server computes an obfuscation of its input x, with the random oracle queries made via evaluating the random function in $\mathcal{F}_{\text{VOPRF}}$ offline. Crucially, after computing the obfuscated input $\mathcal{O}_x$, the server only stores $\mathcal{O}_x$ and *erases the original input* x. In online evaluation, the server sends its storage $\mathcal{O}_x$ to the client, who then runs the obfuscation evaluation procedure to compute the function result with the random oracle queries made via evaluating the random function in $\mathcal{F}_{\text{VOPRF}}$ online (so the client runs evaluation with $\mathcal{F}_{\text{VOPRF}}$ c times).

Our protocol bears a resemblance to the OPAQUE strong aPAKE protocol [JKX18], where the client evaluates an OPRF on its password and obtains a point obfuscation of the password (called the "randomized password" in [JKX18]), receives, from the server, an encryption of the client's authenticated key exchange (AKE) credentials under the randomized password, decrypts and learns its credentials, and then runs an AKE protocol with the server. However, since our goal here is not to establish a key, our protocol is significantly simpler than OPAQUE: the server only needs to send the obfuscation (the randomized password) to the client, and no AKE protocol is run between the client and the server.

4.2.1 Using Verifiable OPRF

Our $\mathsf{io2PC}$ protocol requires a *verifiable* OPRF, meaning that the client should be convinced that the server uses a consistent OPRF key. The alert reader may notice that OPAQUE does not require a verifiable OPRF. However, OPAQUE corresponds to a variant of $\mathsf{io2PC}$ for the special case of point functions, and some situations arise in the special case of $\mathsf{io2PC}$ which are not present in that special case.

First, point-function obfuscation (and hence OPAQUE) requires only a *single* call to the OPRF / random oracle. In the general case, if multiple random oracle queries are required to evaluate an obfuscation, and these oracle queries are replaced by OPRF calls, what should happen if a corrupt server changes its OPRF key between those calls? This is not just a hypothetical question — in the obfuscation presented in Section 6.2, the evaluation algorithm should make some "dummy queries" to its oracle so that the total number of queries does not depend on the input. But the choice of which queries are "dummies" depends on the input. A corrupt server could therefore observe whether changing its OPRF key in an instance leads to any change in the client's output, thereby deducing whether a query is a dummy or not.

Second, point function obfuscation is special because the effect of substituting the "wrong" OPRF key can be easily simulated. Using the wrong OPRF key for a point function makes the point function output false with overwhelming probability, and this can be simulated by the corrupt server simply choosing a random target point for the point function. But in general, it is not immediate that selectively changing the OPRF key is equivalent to choosing a different obfuscated input.

Theorem 5. *Suppose* $(\mathsf{Obf}, \mathsf{ObfEval}, c)$ *is a VBB obfuscation for* $\mathcal{C}_f$ *with simulation rate* r*, in the random oracle model. Then the* io2PC *protocol (Figure 4) realizes the* $\mathcal{F}_{\mathrm{iO2PC}}$ *functionality computing* $\mathcal{C}_f$ *with simulation rate* r *(Figure 1) in the* $\mathcal{F}_{\mathrm{VOPRF}}$*-hybrid world.*

Proof. Consider any polynomial-time environment $\mathcal{Z}$. As standard, assume that the adversary $\mathcal{A}$ is dummy and merely passes messages between $\mathcal{Z}$ and protocol parties. Throughout the proof, we simplify $\mathcal{F}_{\mathrm{iO2PC}}$ to $\mathcal{F}$. We analyze the cases that S is corrupt and C is corrupt separately.

S is corrupt. In this case, we write S^* for the server to stress the fact that it is corrupt. The simulator Sim behaves as follows:

1. On $(\mathsf{VOPRFInit}, sid)$ from S^* aimed at $\mathcal{F}_{\mathrm{VOPRF}}$, Sim initializes a list $\mathcal{H} := []$ and sends $(\mathsf{VOPRFInit}, sid, \mathsf{S}^*)$ to $\mathcal{A}$. Below we assume that there has been an $\mathsf{VOPRFInit}$ message for sid (otherwise Sim simply ignores all messages as there is nothing to simulate).

2. On $(\mathsf{OfflineEval}, sid, x)$ from S^* aimed at $\mathcal{F}_{\mathrm{VOPRF}}$, Sim appends x to the end of $\mathcal{H}$ and sends $(\mathsf{OfflineEval}, sid, F(x))$ (where $F(x) \leftarrow \mathbf{H}$ if undefined) to S^*.

3. On $(\mathsf{IOEval}, sid, ssid, \mathsf{C})$ from $\mathcal{F}$, and $(sid, ssid, \mathcal{O}_x)$ from S^* aimed at C, Sim sends $(\mathsf{VOPRFEval}, sid, ssid, \mathsf{C})$ to $\mathcal{A}$ and waits for $(\mathsf{SComplete}, sid, ssid)$ from S^*.

4. On $(\mathsf{SComplete}, sid, ssid)$ from S^*, Sim sends $(\mathsf{SComplete}, sid, ssid, \mathsf{S}^*)$ to $\mathcal{A}^*$ and waits for $(\mathsf{Deliver}, sid, ssid)$ from $\mathcal{A}^*$.

Parameters:

- Obfuscation $(\mathsf{Obf}, \mathsf{ObfEval}, c)$ for the class of functions $\mathcal{C}_f = \{f(x, \cdot) \mid x \in \{0,1\}^*\}$, in the random oracle model.
- Client C and server S.

On command (Init, sid, x) for S:

1. S: Send $(\mathsf{VOPRFInit}, sid)$ to $\mathcal{F}_{\text{VOPRF}}$
2. S: Run $\mathcal{O}_x \leftarrow \mathsf{Obf}^{?}(x)$, where each time Obf queries its oracle at q:
   - Send $(\mathsf{OfflineEval}, sid, ssid, q)$ to $\mathcal{F}_{\text{VOPRF}}$
   - Receive response $(\mathsf{OfflineEval}, sid, ssid, r)$
   - Give r to Obf as the response to its oracle query
3. S: Store $\mathcal{O}_x$.

On command $(\mathsf{Compromise}, sid)$ from $\mathcal{A}^*$:

4. $\mathcal{A}^*$ must also send $(\mathsf{Compromise}, sid)$ to $\mathcal{F}_{\text{VOPRF}}$
5. $\mathcal{A}^*$ learns $\mathcal{O}_x$.

On command $(\mathsf{IOEval}, sid, ssid)$ for S:

6. S: Send $(sid, ssid, \mathcal{O}_x)$ to C.
7. Both parties set $i := 0$.
8. C: Await command $(\mathsf{IOEval}, sid, ssid, y)$.
9. C: Run $z := \mathsf{ObfEval}^{?}(\mathcal{O}_x, y)$, where each time $\mathsf{ObfEval}$ queries its oracle at q:
   - C: Send $(\mathsf{VOPRFEval}, sid, ssid\|i, q)$ to $\mathcal{F}_{\text{VOPRF}}$
   - S: Await $(\mathsf{VOPRFEval}, sid, ssid\|i)$ from $\mathcal{F}_{\text{VOPRF}}$
   - Both: set $i := i + 1$
   - S: If $i > c$: abort. Otherwise, send $(\mathsf{SComplete}, sid, ssid\|i)$ to $\mathcal{F}_{\text{VOPRF}}$
   - C: Await response $(\mathsf{VOPRFEval}, sid, ssid\|i, r)$ from $\mathcal{F}_{\text{VOPRF}}$
   - C: Give r to Obf as the response to its oracle query
10. C: Output $(\mathsf{IOEval}, sid, ssid, z)$

Figure 4: The $\mathsf{io2PC}$ protocol for computing function f, based on a VBB obfuscation in the random oracle model.

5. On $(\mathsf{Deliver}, sid, ssid)$ from $\mathcal{A}$, Sim sends another $(\mathsf{VOPRFEval}, sid, ssid, \mathsf{C})$ message to $\mathcal{A}$ and another $(\mathsf{VOPRFEval}, sid, ssid)$ message to S^*. Sim repeats this loop of sending $\mathsf{VOPRFEval}$ and receiving $\mathsf{Deliver}$, until it receives a total number of c $\mathsf{Deliver}$ messages from $\mathcal{A}$. If Sim receives an $(\mathsf{Abort}, sid, ssid)$ from $\mathcal{A}$ during this cycle, it sends $(\mathsf{Abort}, sid, ssid)$ to $\mathcal{F}$.

6. Sim computes $x := \mathsf{Extract}(\mathcal{O}_x, \mathcal{H})$ and sends $(\mathsf{IOEval}, sid, ssid, x)$ to $\mathcal{F}$ (as from a corrupt server).

We now argue that $\mathcal{Z}$'s distinguishing advantage between the ideal world and the real world is negligible. Let y be C's input. In the real world, on $(\mathsf{IOEval}, sid, ssid, y)$ from $\mathcal{Z}$ and $(sid, ssid, \mathcal{O}_x)$ from S^*, C runs $\mathsf{ObfEval}^{F(\cdot)}(\mathcal{O}_x, y)$. This consists of evaluating $F(\cdot)$ with $\mathcal{F}_{\mathrm{VOPRF}}$ c times; each time $\mathcal{A}$ receives an $(\mathsf{VOPRFEval}, sid, ssid, \mathsf{C})$ message and S^* receives an $(\mathsf{VOPRFEval}, sid, ssid)$ message, and the evaluation proceeds only after $\mathcal{A}$ sends a $(\mathsf{Deliver}, sid, ssid)$ message. (If $\mathcal{A}$ sends an $(\mathsf{Abort}, sid, ssid)$ message instead, C aborts immediately.) By the description of Sim, this is exactly $\mathcal{A}$'s view in the ideal world. [2]

It remains to analyze C's output. Here we assume that $\mathcal{A}$ never sends $(\mathsf{Abort}, sid, ssid)$ aimed at $\mathcal{F}_{\mathrm{VOPRF}}$ (otherwise C outputs $(\mathsf{IOEval}, sid, ssid, \perp)$ in both worlds, as shown above). Observe that Sim answers $\mathsf{OfflineEval}$ queries via lazy sampling, so $\mathcal{Z}$'s view in the ideal world when sending $(\mathsf{OfflineEval}, sid, x)$ messages aimed $\mathcal{F}_{\mathrm{VOPRF}}$, is identical to its view when querying (sid, x) to a random oracle functionality with range $\mathbf{H}$. Therefore, in the ideal world, C outputs $(\mathsf{IOEval}, sid, ssid, f(x, y))$. On the other hand, in the real world, C outputs $(\mathsf{IOEval}, sid, ssid, \mathsf{ObfEval}^{F(\cdot)}(\mathcal{O}_x, y))$.

We now construct a reduction $\mathcal{R}_{\mathsf{Extract}}$ that, given $\mathcal{Z}$, attacks the extractability property of $(\mathsf{Obf}, \mathsf{ObfEval})$. $\mathcal{R}_{\mathsf{Extract}}$ acts as the combination of Sim, $\mathcal{F}_{\mathrm{iO2PC}}$, and C by running their codes, and interacts with the adversarial parties $\mathcal{Z}$, $\mathcal{A}$, and S^* (where $\mathcal{A}$ and S^* are both controlled by $\mathcal{Z}$), but with the following exceptions: in step 2 of Sim, $\mathcal{R}_{\mathsf{Extract}}$ queries $F(x) := H(x)$ if undefined, rather than sampling $F(x) \leftarrow \mathbf{H}$; in step 3 of Sim, $\mathcal{R}_{\mathsf{Extract}}$ outputs $(y, \mathcal{O}_x)$. Clearly, $\mathcal{R}_{\mathsf{Extract}}$'s advantage in breaking the extractability property of $(\mathsf{Obf}, \mathsf{ObfEval})$, is equal to $\mathcal{Z}$'s distinguishing advantage between the two worlds. Therefore, $\mathcal{Z}$'s distinguishing advantage is negligible, completing the proof.

C is corrupt. In this case, we write C^* for the client to stress the fact that it is corrupt. The simulator Sim behaves as follows:

1. On $(\mathsf{Init}, sid, \mathsf{S})$ from $\mathcal{F}$, Sim runs $\mathsf{SimObf}_1()$, the first phase of the VBB simulator for $(\mathsf{Obf}, \mathsf{ObfEval})$. Let $\mathcal{O}_x$ be SimObf_1's output. Below we assume that there has been such an Init message (otherwise Sim simply ignores all messages as there is nothing to simulate).

2. On $(\mathsf{Compromise}, sid)$ from $\mathcal{A}$ aimed at S and $\mathcal{F}_{\mathrm{VOPRF}}$, Sim marks sid $\mathsf{compromised}$, sends $(\mathsf{Compromise}, sid)$ to $\mathcal{F}$, and sends $(sid, \mathcal{O}_x)$ to $\mathcal{A}$.[3]

3. On $(\mathsf{OfflineEval}, sid, p)$ from $\mathcal{A}$ aimed at $\mathcal{F}_{\mathrm{VOPRF}}$, if sid is marked $\mathsf{compromised}$, Sim runs $\mathsf{SimObf}_2()$ with the adversary querying $H(p)$. (Note that while interacting with Sim_2, Sim needs to play the role of both the adversary in Theorem 1 and the oracle $f(x, \cdot)$.) When SimObf_2 makes a query y to its oracle $f(x, \cdot)$, Sim sends $(\mathsf{OfflineEval}, sid, y)$ to $\mathcal{F}$, and on

[2] This is assuming w.l.o.g. that C must evaluate F sequentially — *e.g.*, it never starts the second evaluation before receiving the result of the first evaluation.

[3] Following *e.g.*, [JKX18], we assume that $\mathcal{A}$ always sends a $\mathsf{Compromise}$ message to S and $\mathcal{F}_{\mathrm{VOPRF}}$ simultaneously. These two actions correspond to a single action in the real protocol, *i.e.* compromising the server.

$\mathcal{F}$'s response $(\mathsf{OfflineEval}, sid, z)$, Sim sends z to SimObf_2 as the response to its query. Finally, when SimObf_2 outputs q as the response to the adversary's $H(p)$ query, Sim sends $(\mathsf{OfflineEval}, sid, q)$ to $\mathcal{A}$.

4. On $(\mathsf{IOEval}, sid, ssid, \mathsf{S})$ from $\mathcal{F}$, Sim sends $(sid, ssid, \mathcal{O}_x)$ to C^*.

5. On $(\mathsf{VOPRFEval}, sid, ssid, p)$ from C^* aimed at $\mathcal{F}_{\mathrm{VOPRF}}$ and $(\mathsf{VOPRFEval}, sid, ssid)$ from C^*, if there have been more than c such messages, Sim does not do anything.
   Otherwise, Sim runs $\mathsf{SimObf}_2()$ with the adversary querying $H(p)$. When SimObf_2 makes a query y to its function oracle $f(x, \cdot)$:

   - If this is the first such query since the last $(\mathsf{IOEval}, sid, ssid, \mathsf{S})$ from $\mathcal{F}$, Sim sends $(\mathsf{IOEval}, sid, ssid, y)$ to $\mathcal{F}$.
   - Otherwise, Sim sends $(\mathsf{Redeem}, sid, ssid, y)$ to $\mathcal{F}$.

   On $\mathcal{F}$'s response $(\mathsf{IOEval}, sid, ssid, z)$, Sim sends z to SimObf_2 as the response to its query. Finally, when both of the followings happen: (1) SimObf_2 outputs q as the response to the adversary's $H(p)$ query, and (2) $\mathcal{A}$ sends $(\mathsf{Deliver}, sid, ssid)$ aimed at $\mathcal{F}_{\mathrm{VOPRF}}$, Sim sends $(\mathsf{VOPRFEval}, sid, ssid, q)$ to C^*.

We now argue that $\mathcal{Z}$'s distinguishing advantage between the ideal world and the real world is negligible. In the real world, if the number of $(\mathsf{VOPRFEval}, sid, ssid)$ messages from C^* exceeds c, they do not have any effect; $(\mathsf{Abort}, sid, ssid)$ messages from $\mathcal{A}^*$ aimed at $\mathcal{F}_{\mathrm{VOPRF}}$ also do not have any effect (because S is honest). By the description of Sim, this is also the case in the ideal world. Below we assume that C^* sends at most c $(\mathsf{VOPRFEval}, sid, ssid)$ messages, and $\mathcal{A}^*$ never sends $(\mathsf{Abort}, sid, ssid)$.

We construct a reduction $\mathcal{R}_{\mathsf{VBB}}$ that, given $\mathcal{Z}$, distinguishes between the two distributions in the definition of the virtual black box property of $(\mathsf{Obf}, \mathsf{ObfEval})$ (Theorem 1). $\mathcal{R}_{\mathsf{VBB}}$ first receives $\mathcal{O}_x$. Then it acts as the combination of Sim (only steps 2-5), $\mathcal{F}_{\mathrm{iO2PC}}$, and S by running their codes, and interacts with $\mathcal{Z}$, $\mathcal{A}$, and C^*, but with the following exceptions: in steps 3 and 5, $\mathcal{R}_{\mathsf{VBB}}$ queries $H(p)$. Finally, $\mathcal{R}_{\mathsf{VBB}}$ copies $\mathcal{Z}$'s output. Note that if $\mathcal{R}_{\mathsf{VBB}}$ is in the ideal interaction, the adversary's H queries are simulated by SimObf_2, so $\mathcal{Z}$'s view, in this case, is identical to its view in the ideal world; whereas if $\mathcal{R}_{\mathsf{VBB}}$ is in the real interaction, $\mathcal{Z}$'s view is identical to its view in the real world. Therefore, $\mathcal{Z}$'s distinguishing advantage is equal to $\mathcal{R}_{\mathsf{VBB}}$'s advantage, which is negligible. $\square$

5 io2PC for Generic-Group Obfuscations

5.1 Generic Groups

Generic groups were introduced by Shoup [Sho97] as a way to model an idealized cyclic group where the only allowable operations are the standard group operations. Consider an encoding $\sigma : \mathbb{Z}_p \to \{0,1\}^*$ of group elements (without loss of generality, the cyclic group of order p) into strings. The group operation (on encoded elements) is defined by the function $\mathsf{mult}_\sigma(\sigma(x), \sigma(y)) = \sigma(x + y \bmod p)$. In Shoup's generic group model, parties have access to an oracle for mult_σ for a *uniformly chosen* encoding σ, along with an encoding of the group generator. Under such a random encoding, the encoding of a group element leaks nothing about that item's "identity" (*i.e.* its discrete log).

Maurer [Mau05] proposed a slightly different model of generic groups, where the encoding of elements is not bijective; *i.e.* each group element may have many valid encodings. In this model, a

$\mathsf{dlog} :=$ empty map
$g^* \leftarrow \{0,1\}^{2\kappa}$
$\mathsf{dlog}[g^*] := 1$

$\underline{\mathsf{ZeroTest}(g_1):}$
  if $\mathsf{dlog}[g_1]$ undefined: $\mathsf{dlog}[g_1] \leftarrow \mathbb{Z}_p$
  return $\mathsf{dlog}[g_1] \stackrel{?}{=} 0$

$\underline{\mathsf{Mult}(g_1, g_2):}$
  if $\mathsf{dlog}[g_1]$ undefined: $\mathsf{dlog}[g_1] \leftarrow \mathbb{Z}_p$
  if $\mathsf{dlog}[g_2]$ undefined: $\mathsf{dlog}[g_2] \leftarrow \mathbb{Z}_p$
  $g_3 \leftarrow \{0,1\}^{2\kappa}$
  $\mathsf{dlog}[g_3] := \mathsf{dlog}[g_1] + \mathsf{dlog}[g_2] \bmod p$
  return g_3

$\underline{\mathsf{gen}():}$
  return g^*

Figure 5: A generic group oracles for group of order p, with handles of length 2κ

stateful oracle maintains a mapping $D : \{0,1\}^* \to \mathbb{Z}_p$, where an abstract *handle* h represents the group element $D[h] \in \mathbb{Z}_p$. A party can query its oracle to multiply handles h_1, h_2 — to do this, the oracle chooses a new handle h_3 and records $D[h_3] = D[h_1] + D[h_2] \bmod p$.

In Shoup's generic group model, every group element x has a unique encoding $\sigma(x)$, which means that it is trivial to test equality of group elements. In Maurer's model, the oracle must provide an equality-test function — *i.e.* given handles h_1, h_2 the oracle returns $D[h_1] \stackrel{?}{=} D[h_2]$.

The two generic group models are equivalent in terms of algorithmic power (*e.g.*, the discrete log problem is equally difficult in both models) [JS08]. However, the distinction is important when incorporating generic groups into a larger cryptographic system. For example, in the Shoup model one may compute a hash of a group element's encoding, so that anyone who can compute the same group element can also compute the same hash. In the Maurer model, two parties may compute two different handles for the "same" group element, so one must be more careful about the distinction between handles and the group elements they represent.

In this work we use a generic group model more similar to Maurer's model. The details are given in Figure 5. Group elements may be represented by many handles, and the oracle must therefore provide an explicit equality test feature. Without loss of generality, we provide a zero-test feature as it is simpler.

In Maurer's model, the handles can be sequential numbers — *i.e.* the ith oracle query is given handle "i". This suffices to reason about non-interactive algorithms. In our setting, the generic group oracle is a common resource shared among many parties (similar to a random oracle), and in that case sequence numbers would reveal how many group operations other parties have performed. Our generic group oracle therefore chooses new handles uniformly at random.

Conventions. Although technically a generic group is modeled as an oracle, it becomes too cumbersome to notate all group operations as oracle calls. Instead, we use standard (multiplicative) group notation to denote operations in the group, as is standard.

A group requires both a group operation and inverses. Since we always consider groups of known order p, inverses can be computed by raising to the $p-1$ power, which can be done with the group-multiplication oracle, so we do not provide a separate explicit group-inverse oracle.

Our generic group formulation assumes that every handle represents *some* group element. Any handle not specifically generated by the oracle corresponds to a uniformly chosen group element. Hence, parties can generate [handles of] random group elements at any time.

Standard Concepts. The generic group oracle uses a map dlog to keep track of the discrete log of every handle. The discrete log of a group element is of course an element of $\mathbb{Z}_p$. A common proof technique in the generic group model is to keep track of discrete logs *symbolically*.

We use the `mathbb` font to denote formal variables like $\mathbb{K}, \mathbb{R}$. Then we extend the contents of dlog to contain not only scalars from $\mathbb{Z}_p$ but rational functions in these formal variables — *i.e.* $\mathsf{dlog}[\cdot] \in \mathbb{Z}_p(\mathbb{K}, \mathbb{R}, \ldots)$. When multiplying group elements, the new handle's dlog value is still recorded as $\mathsf{dlog}[g_3] = \mathsf{dlog}[g_1] + \mathsf{dlog}[g_2]$, but now addition denotes (symbolic) addition of functions over the formal variables.

In our security proofs, we write an expression like "$g^{a\mathbb{K}+b}$" to indicate that the simulator generates a new group handle whose dlog value is the symbolic expression $a\mathbb{K} + b$. Our convention is that lowercase letters like a, b will denote scalars from $\mathbb{Z}_p$.

In a standard generic-group security proof, a random group element like g^r will be replaced by a symbolic one $g^{\mathbb{R}}$. After an adversary performs group operations, other group elements may have dlog-values that are expressions including $\mathbb{R}$. A zero-test on such a group element is performed by checking whether the dlog of that group element is *identically (symbolically) zero*. A standard argument shows that symbolic zero-tests are indistinguishable from real/concrete zero-test, provided that all symbolic dlog expressions have bounded degree, and the dlog formal variables take the place of uniformly chosen (concrete) discrete log values.

5.2 Personalized Generic Group

In the previous sections, we saw that we can convert a VBB obfuscation into an $\mathsf{io2PC}$ protocol by replacing a random oracle with an oblivious PRF. We like to think of an OPRF as a kind of "personalized" random oracle. It is a random function, to which only the server has the key; yet the server can allow the client to evaluate the function on a private input. Even if the server's key to the OPRF is stolen, the adversary's access to the random function is observable/programmable to the simulator.

In this section we extend this analogy from random oracles to generic groups. A **personalized generic group (PGG)** is a group to which only the server has the key; yet the server can allow the client to perform the group operation on private inputs. If the server's key to the group is stolen, the adversary's access to the group is analogous to a true generic group.

We formally define a personalized generic group as an ideal functionality $\mathcal{F}_{\mathsf{pgg}}$ in Figure 6. The functionality maintains a map DLog associating discrete logs with handles, similar to a standard generic group. The server can perform group operations at any time by sending appropriate commands ($\mathsf{OfflineMult}, \mathsf{OfflineZeroTest}$) to the functionality. The client can perform group operations, but only interactively ($\mathsf{OnlineMult}, \mathsf{OnlineZeroTest}$) and only with approval from the server. These group operations are oblivious — the server does not learn which group elements the client is operating on. Only after designating the session as $\mathsf{compromised}$ can a corrupt client also gain the ability to perform the group operations unilaterally.

We point out some other notable aspects of the definition: There are a few things that a client can do non-interactively, *i.e.* without the server's assistance and approval. A client can freely "clone" a handle, resulting in another handle with the same DLog value. In our eventual PGG protocol, this is indeed possible, but does not seem to give obvious advantage to the client. The client can also generate a handle representing a random group element since by default, all handles correspond to uniform group elements.

A corrupt server can learn all discrete logs of all handles. This makes the simulation considerably easier, but does not seem to represent any issues with our usage of the functionality. Note that if the server is honest but a corrupt client compromises the session, the client cannot learn discrete logs. This helps reflect the fact that the corrupt client can obtain at most oracle access to a generic group upon compromising the session. It is likely that our PGG protocol could be proven secure without letting the simulator for a corrupt server learn all discrete logs, but at the cost of increased proof complexity.

The OnlineMult and OfflineMult commands are not analogous. OfflineMult is more powerful than OnlineMult, since it allows the caller (either the server or a client after the session is compromised) to perform arbitrary linear combinations of group elements, not just a single group operation between two elements. We could define the PGG ideal functionality so that OnlineMult is more powerful than a single group operation, and our protocol could achieve this feature in a natural way. We have chosen to model only the minimal functionality of OnlineMult.

The simulator for a $\mathcal{F}_{\mathsf{pgg}}$ protocol should only call OfflineMult at most once for each multiplication made (after compromise) in the common group by the corrupt client; and it should call OfflineZeroTest at most once for each zero-test made in the common group by the corrupt client. *i.e.* an adversary must expend new effort for each OfflineMult and OfflineZeroTest, and furthermore that effort must be expended *after compromise.* However, recall that an OfflineMult is more powerful than a single multiplication. A corrupt client could perform a single multiplication in the common group that is as powerful as a single OfflineMult (which performs a more powerful linear combination of group elements) in the personalized group.[4] For this reason, it is much **more important to measure an adversary's effort in terms of zero-tests and not group multiplications,** since the simulator does not precisely preserve the number of group multiplications between the common group and personalized group. Only the zero-tests are preserved exactly.

5.3 Protocol for Personalized Generic Groups

In this section we describe our protocol for a personalized generic group. The protocol is in the ideal permutation model and uses a generic group itself. This leads to a high potential for confusion. We differentiate between the **common group** and the **personalized group**:

The common group is the generic group that is used by the protocol. Both parties have unrestricted oracle access to the operations of this group. Group element handles are associated with their discrete log via a map that we call dlog. In the security proof, the simulator finds it useful to play the role of this generic group and maintain the dlog map differently (*e.g.*, with symbolic expressions rather than scalars from $\mathbb{Z}_p$).

The personalized group is one that is realized by the protocol. In the ideal functionality for this personalized group, handles are associated with their discrete log via a map that we call DLog. The goal of this personalized group is to carefully restrict the client's access to the operations that involve DLog, via an interactive protocol.

Main Idea. In our protocol, a "key" for a personalized generic group consists of a key k to a strong PRP F, with forward and inverse evaluation denoted F^+ and F^- respectively, and a random generator $\widehat{g}$ (of the common group). An element in the personalized group with discrete log x is represented by a handle of the form $(F_k(m), \widehat{g}^x g^m)$ for $m \in \mathbb{Z}$ providing a multiplicative blind g^m. This kind of encoding can be motivated as follows:

A client who doesn't know the "key" to the personalized group can only create handles of random elements, because the action of $F_k^{\pm}$ is unpredictable.

After compromising the server and learning the "key" $(k, \widehat{g})$, an adversary can invert the PRP to obtain m, then remove the g^m blinding term to obtain simply $\widehat{g}^x$. In other words, after compro-

[4]This is indeed possible in our protocol but would be mitigated if the common-group oracle had a group-multiplication feature exactly as powerful as OfflineMult of $\mathcal{F}_{\mathsf{pgg}}$.

mising the server, the handle becomes equivalent to knowing $\widehat{g}^x$. The adversary can now perform group operations on these unblinded values of the form $\widehat{g}^x$, without the server's help. But since we are in a generic group, the simulator can continue to observe the adversary's group operations on these values.

With the help of the server, it is possible for the client to perform group operations on two of these handles:

1. Consider two handles of the form (c_1, g_1) and (c_2, g_2), where $c_1 = F_k(m_1), c_2 = F_k(m_2)$ and $g_1 = \widehat{g}^{x_1} g^{m_1}, g_2 = \widehat{g}^{x_2} g^{m_2}$. The client can perform $g_3 = g_1 g_2 = \widehat{g}^{x_1+x_2} g^{m_1+m_2}$. This is half of a valid handle for the element with discrete log $x_1 + x_2$. If the client can obtain an encryption of $F_k(m_1 + m_2)$, they will be able to construct a complete and correct handle.

2. The client and server run a 2PC protocol, where the client provides c_1, c_2, and the server provides k, and the client learns $F_k(m_1 + m_2)$. The server learns nothing. Note that this 2PC protocol involves no group operations in the common generic group – it merely involves arithmetic on exponents and PRP evaluation.

Similarly, the client can perform a zero-test with the server's help:

1. Given a handle (c_1, g_1) the client wants to know whether these have the form $c_1 = F_k(m_1)$ and $g_1 = \widehat{g}^0 g^{m_1} = g^{m_1}$. In other words, the client should learn whether c_1 encrypts the discrete log of g_1.

2. Our approach again involves enlisting the help of a 2PC protocol. The client provides c_1 and the server provides k, so m_1 can be obtained inside the 2PC functionality. The functionality provides two basic functions: First, it chooses a random s and lets the client learn $g^{s \cdot m_1}$ using the value of m_1 that it computed. Next, it allows the client to raise any group element of its choice to the s power. Assuming the client chooses to compute g_1^s, the result equals $g^{s \cdot m_1}$ if and only if $g_1 = g^{m_1}$. We discuss exactly how this is done below.

Details and fine print. The full details of our protocol are given in Figure 8, where the separate 2PC functionality invoked by the parties is described in Figure 7. This "helper functionality" is a typical reactive functionality that can be securely realized by any standard 2PC protocol. The preceding outline captures the main intuition of our protocol, but there are several necessary modifications required for technical reasons.

First, the handles are "wrapped" in an ideal permutation $\Pi^{\pm}$ — *i.e.* a valid handle is h of the form $\Pi(c_1, g_1)$ where c_1, g_1 are as described above, and Π is an ideal permutation. By enlisting the ideal permutation, the simulator can observe every time a new handle is generated or an existing handle is "unpacked" into its two components.

In our outline, the parties run an oblivious protocol that allows a client, who holds handles $\Pi(F_k(m_1), \widehat{g}^{x_1} g^{m_1})$ and $\Pi(F_k(m_2), \widehat{g}^{x_2} g^{m_2})$, to obtain a new handle $\Pi(F_k(m_1+m_2), \widehat{g}^{x_1+x_2} g^{m_1+m_2})$. However, this new handle would leak its "history" to anyone who holds k, which would be undesirable. The new handle should instead have a fresh mask m_3, rather than a mask $m_1 + m_2$ derived from its parent handles. When the parties run a 2PC to let the client learn its new ciphertext, the client should instead learn $F_k(m_3)$ for a fresh m_3. Then the client needs to learn a correction term $\Delta = g^{m_3 - m_1 - m_2}$, so it can complete the handle as $\Pi\big(F_k(m_3), (g_1 \cdot g_2 \cdot \Delta) = \widehat{g}^{x_1+x_2} g^{m_3}\big)$. Since the 2PC functionality itself cannot generate group elements (this would require contacting the generic group oracle for the common group), it delegates this task to the server. *i.e.* it gives $m_3 - m_1 - m_2$ to the server, who generates and sends $\Delta = g^{m_3 - m_1 - m_2}$ to the client.

Parameters:

- group order p
- handle length ℓ
- client C and server S

Storage:

- map DLog; our convention is uninitialized entries of DLog are sampled uniformly from $\mathbb{Z}_p$ before being used.
- map status

On input $(\mathsf{Init}, sid, h \in \{0,1\}^\ell)$ from server S:

1. If $\mathsf{status}[sid]$ already defined: abort.
2. Set $\mathsf{status}[sid] := \mathsf{active}$.
3. Send $(\mathsf{Init}, sid, \mathsf{S}, h)$ to C.
4. Set $\mathsf{DLog}[h] := 1$.

On $(\mathsf{Compromise}, sid)$ from $\mathcal{A}^*$:

5. Set $\mathsf{status}[sid] := \mathsf{compromised}$.

On input $(\mathsf{OnlineMult}, sid, ssid, h_1, h_2)$ from C:

6. Give $(\mathsf{OnlineMult}, sid, ssid)$ to S and await response $(\mathsf{Deliver}, sid, ssid)$
7. Sample $h_3 \leftarrow \{0,1\}^\ell$.
8. Set $\mathsf{DLog}[h_3] := \mathsf{DLog}[h_1] + \mathsf{DLog}[h_2] \bmod p$.
9. Give $(\mathsf{OnlineMult}, sid, ssid, h_3)$ to P.

On input $(\mathsf{OfflineMult}, sid, ssid, u_0, (u_1, h_1), \ldots, (u_n, h_n))$ from party $P \in \{\mathsf{S}, \mathcal{A}^*\}$:

10. If $\mathsf{status}[sid] \neq \mathsf{compromised}$ and $P = \mathcal{A}^*$: do nothing.
11. Sample $h' \leftarrow \{0,1\}^\ell$.
12. Set $\mathsf{DLog}[h'] := u_0 + u_1\mathsf{DLog}[h_1] + \cdots + u_n\mathsf{DLog}[h_n] \bmod p$.
13. Give $(\mathsf{OfflineMult}, sid, ssid, h')$ to P.

On input $(cmd \in \{\mathsf{OnlineZeroTest}, \mathsf{OfflineZeroTest}\}, sid, ssid, h)$ from party P:

14. If $cmd = \mathsf{OfflineZeroTest}$:
    If $P \notin \{\mathcal{A}^*, \mathsf{S}\}$, or $P = \mathcal{A}^*$ and $\mathsf{status}[sid] \neq \mathsf{compromised}$: do nothing.
15. If $cmd = \mathsf{OnlineZeroTest}$:
    If $P = \mathsf{C}$: Give $(\mathsf{OnlineZeroTest}, sid, ssid)$ to S and await response $(\mathsf{Deliver}, sid, ssid)$.
16. Give $(cmd, sid, ssid, [\mathsf{DLog}[h] \stackrel{?}{=} 0])$ to P.

On input $(\mathsf{Identify}, h)$ from corrupt S:

17. Give $\mathsf{DLog}[h]$ to S

On input $(\mathsf{Register}, v)$ from corrupt S:

18. $h \leftarrow \{0,1\}^\ell$
19. $\mathsf{DLog}[h] := v$
20. Give h to C

On input $(\mathsf{CloneHandle}, h)$ from corrupt C:

21. $h' \leftarrow \{0,1\}^\ell$
22. $\mathsf{DLog}[h'] := \mathsf{DLog}[h]$
23. Give h' to C

Figure 6: The personalized generic group functionality $\mathcal{F}_{\mathsf{pgg}}$.

However, the server may cheat and send a different group element than the functionality intended. To prevent this, the functionality authenticates the group element with a one-time MAC. The functionality gives random MAC key α, β to the client, and gives s and its one-time MAC $\mu = \alpha s + \beta$ to the server. Now the server can send both g^s and g^μ to the client, who can check the MAC in the exponent via $(g^s)^\alpha \cdot g^\beta \stackrel{?}{=} (g^\mu)$.

There are two conceptual steps in the zero-test protocol which are more complicated than our high-level outline. First, the 2PC helper functionality wants the client to learn the group element $g^{m_1 s}$. It delegates this to the server, using the same method above with one-time MACs, so that the client can be sure that it receives the intended group element. Actually, we must blind the exponent $m_1 s$ from the server (since it also learns s below, and it should not learn m_1 which is tied to this particular handle) — so the functionality gives a random z to the client, and asks the server to deliver $g^{m_1 s + z}$. The client can unblind by multiplying with g^{-z}.

Second, the 2PC functionality gives s to the server so that the server can take part in a blind exponentiation protocol (raising a group element of the client's choice to the s power). It is important to ensure that the server raises the client's element to the correct power, since otherwise the server could easily cause a zero-test to fail even when it should correctly succeed. For this, we (1) have the functionality deliver the value g^s to the client (via delegating to the server), and (2) have the server run a simple *verifiable* exponentiation protocol, where the client can be convinced that its group element was indeed raised to the s power.

Security.

Theorem 6. *The protocol in Figure 8 UC-securely realizes $\mathcal{F}_{\mathsf{pgg}}$ (Figure 6) in the generic group and ideal-permutation model, when F is a strong PRP.*

We provide a sketch of the main ideas here. The case of a corrupt server is considerably easier. It is easy to see that the server's view during $\mathsf{OnlineMult}$, $\mathsf{OnlineZeroTest}$ gives no information about the client's choice of handles, since all the communication is mediated through the helper functionality $\mathcal{F}_{\mathsf{helper}}$. The $\mathcal{F}_{\mathsf{pgg}}$ functionality allows a corrupt server to both learn the discrete log for any handle, and also directly register a handle with a chosen discrete log. The simulator can use these features to intercept all the adversary's $\Pi^{\pm}$ oracle queries and relay discrete log information between the functionality and the actual group elements used for $h = \Pi(\cdot, g_1)$.

The case of a corrupt client is considerably more complex, but the main idea is as follows. In the real world, handles have the form $h = \Pi(c, \widehat{g}^{\mathsf{DLog}[h]} g^{F_k^{-1}(c)})$. We use the technique of symbolic discrete logs (described in Section 5.1) to model the adversary's ignorance of certain values. The adversary does not know the discrete log of $\widehat{g}$, so we represent it by a formal variable $\mathbb{K}$. Before the session is compromised, the adversary does not know $F_k^{-1}(c)$ for any c, so we represent this value by formal variable $\mathbb{M}_c$. The adversary initially does not know anything about the $\mathsf{DLog}[h]$ values, so we represent them by formal variables $\mathbb{D}_h$.

The adversary can only gain information about group elements (in the common group) through a zero-test. When the adversary makes such a zero-test, the simulator observes it and checks the dlog value of that group element. This dlog is a symbolic expression over the formal variables. Formal variables $\mathbb{M}_c$ and $\mathbb{K}$ represent values that are random from the adversary's point a view, so if the dlog expression is not symbolically equal to zero as a function of those variables, then the zero test in the real world would succeed only with negligible probability. Hence, the simulator can simply claim that the zero-test fails. However, the dlog expression may contain $\mathbb{D}_h$ terms which represent concrete $\mathsf{DLog}[h]$ values, and depending on the actual values in $\mathsf{DLog}[h]$ the dlog expression may or may not be identically zero as a function of the other formal variables. In that case, the

Parameters:

- modulus p
- strong PRP $F^{\pm} : \{0,1\}^{\kappa} \times \mathbb{Z}_p \to \mathbb{Z}_p$
- client C and server S

Storage: value k, initially sampled as $k \leftarrow \{0,1\}^{\kappa}$

On command $(\mathsf{HelpInit}, sid)$ from S:

1. Sample $\alpha, \beta, c, s \leftarrow \mathbb{Z}_p$
2. $\mu = \alpha s + \beta$ *// one-time MAC of s under key (α, β)*
3. Send $(\mathsf{HelpInit}, sid, \alpha, \beta, c)$ to C and send $(\mathsf{HelpInit}, sid, s, \mu, c, k)$ to S.

On command $(\mathsf{HelpMult}, sid, ssid, c_1, c_2)$ from C:

4. Send $(\mathsf{HelpMult}, sid, ssid)$ to S await response $(\mathsf{Deliver}, sid, ssid)$.
5. $m_1 := F_k^{-1}(c_1)$; $m_2 := F_k^{-1}(c_2)$.
6. $\alpha, \beta, m_3 \leftarrow \mathbb{Z}_p$.
7. $c_3 = F_k(m_3)$.
8. $s = m_3 - m_1 - m_2 \bmod p$.
9. $\mu = \alpha s + \beta \bmod p$. *// one-time MAC of s under key (α, β)*
10. Give $(\mathsf{HelpMult}, sid, ssid, \alpha, \beta, c_3)$ to C and give $(\mathsf{HelpMult}, sid, ssid, s, \mu)$ to S

On command $(\mathsf{HelpZeroTest}, sid, ssid, c)$ from C:

11. Send $(\mathsf{HelpZeroTest}, sid, ssid)$ to S and await response $(\mathsf{Deliver}, sid, ssid)$
12. $m := F_k^{-1}(c)$.
13. $\alpha, \beta, \gamma, s, z \leftarrow \mathbb{Z}_p$.
14. $t := sm + z$
15. $\mu = \alpha s + \beta t + \gamma \bmod p$. *// one-time MAC of (s, t) under key (α, β, γ)*
16. Give $(\mathsf{HelpZeroTest}, sid, ssid, \alpha, \beta, \gamma, z)$ to C and
    give $(\mathsf{HelpZeroTest}, sid, ssid, s, t, \mu)$ to S

Figure 7: Helper functionality $\mathcal{F}_{\mathsf{helper}}$ for our personalized generic group protocol.

Parameters:

- generic group $\langle g \rangle$ of prime order p, with handles of length 2κ
- strong PRP $F^{\pm}$
- ideal permutation $\Pi^{\pm} : (\mathbb{Z}_p \times \{0,1\}^{2\kappa}) \to (\mathbb{Z}_p \times \{0,1\}^{2\kappa})$
- client C and server S

On input (Init, sid) for server S:

1. S: Send $(\mathsf{HelpInit}, sid)$ to $\mathcal{F}_{\mathsf{helper}}$
2. C: Receive $(\mathsf{HelpInit}, sid, \alpha, \beta, c)$ from $\mathcal{F}_{\mathsf{helper}}$
3. S: Receive $(\mathsf{HelpInit}, sid, s, \mu, c, k)$ from $\mathcal{F}_{\mathsf{helper}}$
4. S: Store $(\widehat{g} := g^{s-m}, k)$, where $m := F_k^{-1}(c)$.
5. S: Send $(S = g^s, M = g^{\mu})$ to C.
6. C: If $S^{\alpha} \cdot g^{\beta} \neq M$: abort.
7. C: Output $(\mathsf{Init}, sid, h = \Pi(c, S))$

On input $(\mathsf{Compromise}, sid)$ from $\mathcal{A}^*$:

8. $\mathcal{A}^*$ should learn $(k, \widehat{g})$

On input $(\mathsf{OnlineMult}, sid, ssid, h_1, h_2)$ for C:

9. C: $(c_1, g_1) = \Pi^{-1}(h_1)$; $(c_2, g_2) = \Pi^{-1}(h_2)$.
10. C: Send $(\mathsf{HelpMult}, sid, ssid, c_1, c_2)$ to $\mathcal{F}_{\mathsf{helper}}$
11. S: Await $(\mathsf{Deliver}, sid, ssid)$ from environment and forward it to $\mathcal{F}_{\mathsf{helper}}$
12. C: Receive $(\mathsf{HelpMult}, sid, ssid, \alpha, \beta, c_3)$ from $\mathcal{F}_{\mathsf{helper}}$
13. S: Receive $(\mathsf{HelpMult}, sid, ssid, s, \mu)$ from $\mathcal{F}_{\mathsf{helper}}$
14. S: Send $(S = g^s, M = g^{\mu})$ to C.
15. C: If $S^{\alpha} \cdot g^{\beta} \neq M$: abort.
16. C: Output $(\mathsf{OnlineMult}, sid, ssid, h_3 = \Pi(c_3, g_1 \cdot g_2 \cdot S))$

On input $(\mathsf{OnlineZeroTest}, sid, ssid, h_1)$ for C:

17. C: $(c_1, g_1) = \Pi^{-1}(h_1)$.
18. C: Send $(\mathsf{HelpZeroTest}, sid, ssid, c_1)$ to $\mathcal{F}_{\mathsf{helper}}$
19. S: Await $(\mathsf{Deliver}, sid, ssid)$ from environment and forward it to $\mathcal{F}_{\mathsf{helper}}$
20. C: Receive $(\mathsf{HelpZeroTest}, sid, ssid, \alpha, \beta, \gamma, z)$ from $\mathcal{F}_{\mathsf{helper}}$
21. S: Receive $(\mathsf{HelpZeroTest}, sid, ssid, s, t, \mu)$ from $\mathcal{F}_{\mathsf{helper}}$
22. S: Send $(S = g^s, T = g^t, M = g^{\mu})$ to C.
23. C: If $S^{\alpha} \cdot T^{\beta} \cdot g^{\gamma} \neq M$: abort.
24. C: $a, b, c \leftarrow \mathbb{Z}_p$; $A := g_1^a \cdot g^b$; $C := g_1^c$; send (A, C) to S.
25. S: Send $A' = A^s$ and $C' = C^s$ to C
26. C: If $(A')^c \neq (C')^a \cdot S^{bc}$: abort
27. C: Output $(\mathsf{OnlineZeroTest}, sid, ssid, [(C')^{1/c} \stackrel{?}{=} T \cdot g^{-z}])$

On input $(\mathsf{OfflineMult}, sid, ssid, u_0, (u_1, h_1), \ldots, (u_n, h_n))$ for S:

28. S: For each $i \in [n]$ do: $(c_i, g_i) = \Pi^{-1}(h_i)$; $m_i := F_k^{-1}(c_i)$
29. S: $m^* \leftarrow \mathbb{Z}_p$; $c^* := F_k(m^*)$; $h^* = \Pi(c^*, \widehat{g}^{u_0} \prod_i g_i^{u_i} \cdot g^{m^* - \sum_i m_i})$
30. S: Output $(\mathsf{OfflineMult}, sid, ssid, h^*)$

On input $(\mathsf{OfflineZeroTest}, sid, ssid, h_1)$ for S:

31. S: $(c_1, g_1) = \Pi^{-1}(h_1)$; $m_1 := F_k^{-1}(c_1)$
32. S: Output $(\mathsf{OfflineZeroTest}, sid, ssid, [g_1 \stackrel{?}{=} g^{m_1}])$

Figure 8: Our personalized generic group protocol.

simulator must know whether the concrete DLog values make the dlog expression identically zero. We carefully analyze what kinds of expressions are possible in dlog, and show that this situation only happens when the simulator needs to know whether a single DLog[h] value is zero, and then only after the adversary has done an OnlineZeroTest on h. In all other cases, the concrete values in DLog have no bearing on whether a dlog expression is symbolically zero (at least before session compromise).

When the adversary compromises the session, it learns the PRP key k. This makes the $F_k^{-1}(c)$ values no longer uncertain from the adversary's point of view. We model this by having the simulator replace every formal variable $\mathbb{M}_c$ with a concrete value $F_k^{-1}(c)$, after compromise. This changes what kinds of expressions the adversary is able to make appear in dlog. After the compromise, there are more situations where the concrete values in DLog[h] have a bearing on whether a dlog expression is symbolically zero. In those cases, the simulator can use OfflineMult, OfflineZeroTest to learn the relevant information about those DLog values.

Throughout the simulation, act as an honest generic group oracle, maintaining the mapping dlog as in Figure 5

At initialization time:

1. Await command (HelpInit, sid) from S to $\mathcal{F}_{\mathsf{helper}}$
2. Sample $s, m, \mu \leftarrow \mathbb{Z}_p$; $k \leftarrow \{0,1\}^\kappa$; and $c = F_k(m)$
3. Give (HelpInit, sid, s, μ, c, k) to S on behalf of $\mathcal{F}_{\mathsf{helper}}$
4. If S sends (g^s, g^μ) to C, then send (Init, $sid, h = \Pi(c, g^s)$) to $\mathcal{F}_{\mathsf{pgg}}$; otherwise abort.
5. Remember $\widehat{g} = g^{s-m}$ and k

When $\mathcal{A}^*$ makes a fresh query $\Pi^{-1}(h)$:

6. Send (Identify, h) to $\mathcal{F}_{\mathsf{pgg}}$ and receive response (Identify, v)
7. $m \leftarrow \mathbb{Z}_p$; $c := F_k(m)$
8. Return $(c, \widehat{g}^v \cdot g^m)$ as the result from Π^{-1}

When $\mathcal{A}^*$ makes a fresh query $\Pi(c_1, g_1)$:

9. $m_1 = F_k^{-1}(c_1)$
10. $v = (\mathsf{dlog}[g_1] - m_1)/\mathsf{dlog}[\widehat{g}]$
11. Send (Register, v) to $\mathcal{F}_{\mathsf{pgg}}$ and receive response (Register, h).
12. Return h as the response from Π

Upon receiving (OnlineMult, $sid, ssid$) message from $\mathcal{F}_{\mathsf{pgg}}$:

13. Choose random $s, \mu \leftarrow \mathbb{Z}_p$ and give them to S as outputs from $\mathcal{F}_{\mathsf{helper}}$.
14. If S sends (g^s, g^μ) to C, then send (Deliver, $sid, ssid$) to $\mathcal{F}_{\mathsf{pgg}}$; otherwise abort.

Upon receiving (OnlineZeroTest, $sid, ssid$) message from $\mathcal{F}_{\mathsf{pgg}}$:

15. Choose random $s, t, \mu \leftarrow \mathbb{Z}_p$ and give them to S as outputs from $\mathcal{F}_{\mathsf{helper}}$.
16. If S sends (g^s, g^t, g^μ) to C, then continue; otherwise abort.
17. Choose $A, C \leftarrow \langle g \rangle$ and give them to S on behalf of C.
18. If S sends (A^s, C^s) to C, then send (Deliver, $sid, ssid$) to $\mathcal{F}_{\mathsf{pgg}}$; otherwise abort.

Figure 9: Simulator for a corrupt server

Throughout the simulation, act as an honest generic group oracle, maintaining the mapping dlog as in Figure 5, except where indicated below.

Storage:

- k, $\widehat{g}$;
- map dlog, whose values are rational functions in formal variables $\mathbb{K}$, $\{\mathbb{D}_h\}_h$, $\{\mathbb{M}_c\}_c$, $\{\mathbb{S}_i\}_i$;
- map $\mathsf{canonical}$, where all values initialized to uniform values

Upon receiving $(\mathsf{HelpInit}, sid, h)$ from $\mathcal{F}_{\mathsf{pgg}}$:

1. Choose $k \leftarrow \{0,1\}^\kappa$; $\alpha, \beta, s, m \leftarrow \mathbb{Z}_p$; $c := F_k(m)$
2. Give $(\mathsf{HelpInit}, sid, \alpha, \beta, c)$ to C on behalf of $\mathcal{F}_{\mathsf{helper}}$
3. Send (g^s, g^{as+b}) to C on behalf of S.
4. $\widehat{g} := g^{\mathbb{K}}$.
5. $\mathsf{canonical}[c] := h$

When $\mathcal{A}^*$ sends $(\mathsf{Compromise}, sid)$:

6. For every formula stored in dlog: replace every formal variable $\mathbb{M}_c$ with concrete value $F_k^{-1}(c)$
7. Give $(k, \widehat{g})$ to $\mathcal{A}^*$

Upon message $(\mathsf{HelpMult}, sid, c_1, c_2)$ from C to $\mathcal{F}_{\mathsf{helper}}$:

8. $m_1 := F_k^{-1}(c_1)$; $m_2 := F_k^{-1}(c_2)$
9. $h_1 := \mathsf{canonical}[c_1]$; $h_2 := \mathsf{canonical}[c_2]$
10. Send $(\mathsf{OnlineMult}, sid, ssid, h_1, h_2)$ to $\mathcal{F}_{\mathsf{pgg}}$; receive h_3 in response
11. $\alpha, \beta, m_3 \leftarrow \mathbb{Z}_p$; $c_3 := F_k(m_3)$
12. $\mathsf{canonical}[c_3] = h_3$
13. If $\mathsf{compromised}$: $S := g^{(\mathbb{D}_{h_3}+m_3)-(\mathbb{D}_{h_1}+m_1)-(\mathbb{D}_{h_2}+m_2)}$;
    else: $S := g^{(\mathbb{D}_{h_3}+\mathbb{M}_{c_3})-(\mathbb{D}_{h_1}+\mathbb{M}_{c_1})-(\mathbb{D}_{h_2}+\mathbb{M}_{c_2})}$
14. Send $(S, S^\alpha \cdot g^\beta)$ to C on behalf of S

Upon message $(\mathsf{HelpZeroTest}, sid, c_1)$ from C to $\mathcal{F}_{\mathsf{helper}}$:

15. $m_1 := F_k^{-1}(c_1)$
16. $h_1 := \mathsf{canonical}[m_1]$
17. $\alpha, \beta, \gamma, z \leftarrow \mathbb{Z}_p$;
18. $S := g^{\mathbb{S}_i}$ where $\mathbb{S}_i$ is an unused formal variable
19. if $\mathsf{compromised}$: $T := g^{\mathbb{S}_i \cdot m_1}$; else $T := g^{\mathbb{S}_i \cdot \mathbb{M}_{c_1}}$
20. Send $(S, T, S^\alpha \cdot T^\beta \cdot g^\gamma)$ to C on behalf of S
21. Await message (A, C) from C to S
22. Send $(\mathsf{OnlineZeroTest}, sid, ssid, h_1)$ to $\mathcal{F}_{\mathsf{pgg}}$; receive t in response
23. Give $(A^{\mathbb{S}_i}, C^{\mathbb{S}_i})$ to C on behalf of S
24. If t: replace every instance of $\mathbb{D}_{h_1}$ in a formula in dlog with 0

Figure 10: Simulator for a corrupt client, part 1

When $\mathcal{A}^*$ makes a fresh query $\Pi^{-1}(h)$:

25. $m \leftarrow \mathbb{Z}_p$; $c := F_k(m)$
26. $\mathsf{canonical}[c] := h$
27. If $\mathsf{compromised}$: return $(c, g^{\mathbb{K}\mathbb{D}_h + m})$; else return $(c, g^{\mathbb{K}\mathbb{D}_h + \mathbb{M}_c})$

When $\mathcal{A}^*$ makes a fresh query $\Pi(c_1, g_1)$:

28. if $\mathsf{canonical}[c_1]$ undefined:
    - Send $(\mathsf{RandHandle}, sid)$ to $\mathcal{F}_{\mathsf{pgg}}$ and receive h_1 in response
    - Set $\mathsf{canonical}[c_1] := h_1$;
    - Return h_1
29. If $\mathsf{dlog}[g_1]$ has the form $\mathbb{K}\mathbb{D}_{h_1} + \mathbb{M}_{c_1}$ for some h_1:

    // only possible pre-compromise
    - Send $(\mathsf{CloneHandle}, sid, h_1)$ to $\mathcal{F}_{\mathsf{pgg}}$ and receive h^* in response
    - Return h^*
30. If $\mathsf{dlog}[g_1]$ has the form $\mathbb{K}(u_0 + u_1\mathbb{D}_{h_1} + u_n\mathbb{D}_{h_n}) + m_1$, where $m_1 := F_k^{-1}(c_1)$:

    // only possible post-compromise
    - Send $(\mathsf{OfflineMult}, sid, ssid, u_0, (u_1, h_1), \ldots, (u_n, h_n))$ to $\mathcal{F}_{\mathsf{pgg}}$;
      Receive response $(\mathsf{OfflineMult}, sid, ssid, h^*)$
    - Set $\mathsf{canonical}[c_1] := h^*$ unless it is already defined;
    - Update $\mathsf{dlog}[g_1] := \mathbb{K}\mathbb{D}_{h^*} + m_1$
    - Return h^*

When $\mathcal{A}^*$ multiplies two generic group elements g_1, g_2:

31. $g_3 \leftarrow \{0,1\}^{2\kappa}$
32. $\mathsf{dlog}[g_3] := \mathsf{dlog}[g_1] + \mathsf{dlog}[g_2]$ *// as symbolic formulas*
33. If $\mathsf{dlog}[g_3]$ has the form $\mathbb{K}(u_0 + u_1\mathbb{D}_{h_1} + u_n\mathbb{D}_{h_n}) + m$:

    // only possible post-compromise
    - Send $(\mathsf{OfflineMult}, sid, ssid, u_0, (u_1, h_1), \ldots, (u_n, h_n))$ to $\mathcal{F}_{\mathsf{pgg}}$;
      Receive response $(\mathsf{OfflineMult}, sid, ssid, h^*)$
    - Update $\mathsf{dlog}[g_3] := \mathbb{K}\mathbb{D}_{h^*} + m_1$
34. Return g_3

When $\mathcal{A}^*$ tests $g_1 \stackrel{?}{=} 1$ for a group element:

35. If $\mathsf{dlog}[g_1]$ is of the form $\mathbb{K} \cdot \mathbb{D}_h$ for some h:
    - Send $(\mathsf{OfflineZeroTest}, sid, ssid, h)$ to $\mathcal{F}_{\mathsf{pgg}}$; receive response t
    - If t: replace every occurrence of $\mathbb{D}_h$ in a formula in dlog with 0
    - Return t
36. Else: return $\mathsf{dlog}[g_1] \stackrel{?}{\equiv} 0$ *// is the formula identically zero?*

Figure 11: Simulator for a corrupt client, part 2

5.3.1 Proof

Corrupt Server

Lemma 7. *The protocol in Figure 8 UC-securely realizes $\mathcal{F}_{\mathsf{pgg}}$ (Figure 6) against a corrupt server.*

Proof. The simulator for this case is given in Figure 9. Throughout the proof, we assume without loss of generality that the adversary does not make a query to Π or Π^{-1} whose answer is already known. I.e., it does not repeat queries to $\Pi^{\pm}$; it does not query $\Pi(x)$ if a previous query $\Pi^{-1}(y) = x$ was made; it does not query $\Pi^{-1}(y)$ if a previous query $\Pi(x)$ was made. All the adversary's queries to $\Pi^{\pm}$ are therefore *fresh*, and the simulator's behavior is described in terms of its responses to fresh queries.

We prove that the simulation is indistinguishable from the real protocol interaction, using the following sequence of hybrids:

Real interaction: The adversary interacts with an honest client running the protocol, honest generic group oracle, honest ideal permutation $\Pi^{\pm}$, and honest $\mathcal{F}_{\mathsf{helper}}$ functionality. The generic group oracle maintains dlog which maps group elements (handles) to their discrete logs.

Hybrid 1: Same as above, but let every *fresh* query to Π or Π^{-1} output a uniform result, rather than ensuring that Π remains a permutation. Then abort if any of the following bad events happen:

- There is a repeated output of Π or of Π^{-1}.
- Maintain a set $\mathcal{C}$. Each time a *fresh* call to Π^{-1} returns $(c_1, \cdot)$, add c_1 to $\mathcal{C}$. Each time $\mathcal{F}_{\mathsf{helper}}$ outputs a value c or c_3, add it to $\mathcal{C}$. Abort if a value already exists in $\mathcal{C}$ when it is being added.
- Abort if a *fresh* query to Π^{-1} results in output $(\cdot, g_1)$ where $\mathsf{dlog}[g_1]$ already defined.
- Abort if during an $\mathsf{OnlineMult}$ session or Init, the server receives S, M from $\mathcal{F}_{\mathsf{helper}}$, sends something other than (S, M) to the client, and yet the client does not abort.
- Abort if during an $\mathsf{OnlineZeroTest}$ session, the server receives S, T, M from $\mathcal{F}_{\mathsf{helper}}$, sends something other than (S, T, M) to the client, and yet the client does not abort.
- Abort if during an $\mathsf{OnlineZeroTest}$ session, the server receives s from $\mathcal{F}_{\mathsf{helper}}$ and (A, C) from the client, and responds with something other than (A^s, C^s), and yet the client does not abort.

The hybrids are indistinguishable if the probability of a bad event is negligible. The first three bad events have negligible probability, by the birthday bound (and the fact that F is a permutation – since often m is chosen uniformly and c is computed as $c = F_k(m)$). The next two bad events have negligible probability because the M is a one-time MAC verified by the client. In Lemma 8 we show that the probability of the final event is negligible.

Hybrid 2: Same as the previous hybrid, except for the following changes:

- The random values chosen by $\mathcal{F}_{\mathsf{helper}}$ during Init, which determine $\widehat{g}$, can be made at the beginning of time.
- Maintain a map DLog. Every time there is a fresh query $\Pi(c_1, g_1) = h_1$ or $\Pi^{-1}(h_1) = (c_1, g_1)$ by any party, set $\mathsf{DLog}[h_1] = (\mathsf{dlog}[g_1] - F_k^{-1}(c_1))/\mathsf{dlog}[\widehat{g}]$. Here dlog is the table from the honest generic group oracle.

These changes have no effect on the adversary's view.

Writing the formula for DLog differently, we get the invariant $h_1 = \Pi(F_k(m_1), \widehat{g}^{\mathsf{DLog}[h_1]} \cdot g^{m_1})$.

In the interactive protocols in this hybrid, the honest client either aborts or reliably receives the S, T values intended by $\mathcal{F}_{\mathsf{helper}}$. If an honest client runs $\mathsf{OnlineMult}$ with h_1 and h_2 and finally outputs h_3, then (using the notation in the protocol and in $\mathcal{F}_{\mathsf{helper}}$) we get:

$$\begin{aligned} h_3 = \Pi(c_3, g_1 g_2 S) &= \Pi\Big(F_k(m_3), g_1 \cdot g_2 \cdot g^s\Big) \\ &= \Pi\Big(F_k(m_3), (\widehat{g}^{\mathsf{DLog}[h_1]} g^{m_1})(\widehat{g}^{\mathsf{DLog}[h_2]} g^{m_2}) g^{m_3 - m_1 - m_2}\Big) \\ &= \Pi\Big(F_k(m_3), \widehat{g}^{\mathsf{DLog}[h_1] + \mathsf{DLog}[h_2]} g^{m_3}\Big) \end{aligned}$$

Hence $\mathsf{DLog}[h_3] = \mathsf{DLog}[h_1] + \mathsf{DLog}[h_2]$ always holds. Similarly, if the honest client does not abort during Init, then it outputs:

$$h = \Pi(c, g^s) = \Pi(F_k(m), g^{s-m} \cdot g^m) = \Pi(F_k(m), \widehat{g} \cdot g^m)$$

Hence $\mathsf{DLog}[h] = 1$.

If an honest client runs $\mathsf{OnlineZeroTest}$ with h_1 and does not abort, then the server has sent the correct $S = g^s, T = g^{sm_1 + z}$ value, and the correct $(A' = A^s, C' = C^s)$ values. In that case the client correctly outputs the result of:

$$\begin{aligned} (C')^{1/c} \stackrel{?}{=} T \cdot g^{-z} \iff (C^s)^{1/c} \stackrel{?}{=} g^{sm_1 + z} g^{-z} \iff (g_1^{cs})^{1/c} \stackrel{?}{=} g^{sm_1} \\ \iff g_1^s \stackrel{?}{=} g^{sm_1} \iff g_1 \stackrel{?}{=} \widehat{g}^0 g^{m_1} \iff \mathsf{DLog}[h_1] \stackrel{?}{=} 0 \end{aligned}$$

Hybrid 3: Based on the invariants defined above, we make the following modifications to the previous hybrid. Recall that every time a fresh query to $\Pi^{\pm}$ is made, we update DLog according to the rule above. In this hybrid, we modify what happens when it is the honest client who makes such a query.

The honest client only queries Π at the end of $\mathsf{OnlineZeroTest}$ and Init. In Init, the client queries $\Pi(c, S)$, where c and S are known to the adversary. We can therefore assume without loss of generality that the adversary makes this initial query to Π instead of the client. In $\mathsf{OnlineZeroTest}$ the client queries $h_3 = \Pi(c_3, g_3)$ where c_3 and g_3 are never used again in the interaction until someone queries $\Pi^{-1}(h_3)$. Therefore, it would not change the adversary's view if we instead:

- Choose h_3 uniformly (this is equivalent because we are considering h_3 resulting from a *fresh* query to Π)
- Record $\mathsf{DLog}[h_3] = \mathsf{DLog}[h_1] + \mathsf{DLog}[h_2]$
- Later, only if/when $\Pi^{-1}(h_3)$ is needed, sample a uniform m_3 (this is equivalent to sampling $c_3 = F_k(m_3)$ uniformly) and respond with $(F_k(m_3), \widehat{g}^{\mathsf{DLog}[h_3]} \cdot g^{m_3})$

The honest client only queries Π^{-1} at the beginning of $\mathsf{OnlineMult}$ or of $\mathsf{OnlineZeroTest}$. If this is a fresh query, then the output will be of Π^{-1} will be uniform. The output of this query is used only as input (c_1, c_2) to $\mathcal{F}_{\mathsf{helper}}$ (which gives no information about it to the server) or as part of the input to a final call to Π. But after making the previous change described above, this call to Π is no longer even made. So it would not change the adversary's view if the honest client did not make this query to Π^{-1}, and its uniform output was chosen later only if/when the adversary makes the query.

Hybrid 4: Same as the previous hybrid, except that we change how fresh Π^{-1} queries by the adversary are handled. In the previous hybrid, a random (c_1, g_1) was chosen as output from $\Pi^{-1}(h_1)$, and based on that value we computed $\mathsf{DLog}[h_1]$. Clearly the value of $\mathsf{DLog}[h_1]$ is uniform in this case. It would be equivalent then to first choose $\mathsf{DLog}[h_1]$, then choose random c_1 and then set $g_1 = \widehat{g}^{\mathsf{DLog}[h_1]} \cdot g^{F_k^{-1}(c_1)}$.

With this change, the adversary's view in the interaction is now identical to the simulation.

- During Init, $\mathcal{F}_{\mathsf{helper}}$ gives uniform output to the adversary; the honest party aborts if the adversary does not send the expected group elements to the client; Π is called resulting in output h; and a value $\mathsf{DLog}[h] = 1$ is recorded. In the simulation, $\mathsf{DLog}[h]$ is recorded in $\mathcal{F}_{\mathsf{pgg}}$ by the simulator sending the appropriate Init command.

- When the adversary makes a fresh $\Pi^{-1}(h)$ query, the corresponding $\mathsf{DLog}[h]$ value is fetched (if that value doesn't exist, a random one is chosen); the output of Π^{-1} becomes the correct values representing $\mathsf{DLog}[h]$. In the simulation, $\mathsf{DLog}[h]$ is consulted by the simulator sending an $\mathsf{Identify}$ command to $\mathcal{F}_{\mathsf{pgg}}$.

- When the adversary makes a fresh Π query, its DLog value is computed and stored for a random handle h which is chosen as the output of Π. In the simulation, the simulator computes the DLog value, and $\mathcal{F}_{\mathsf{pgg}}$ samples h and records $\mathsf{DLog}[h]$ when the simulator sends a $\mathsf{Register}$ command.

- During $\mathsf{OnlineMult}$ and $\mathsf{OnlineZeroTest}$, the adversary sees random outputs from $\mathcal{F}_{\mathsf{helper}}$. As a side effect of $\mathsf{OnlineMult}$, a new random handle is registered in DLog. As a side effect of $\mathsf{OnlineZeroTest}$, the honest client outputs the result of $\mathsf{DLog}[h_1] \stackrel{?}{=} 0$.

□

Lemma 8. *If during an $\mathsf{OnlineZeroTest}$ session, a corrupt server receives s from $\mathcal{F}_{\mathsf{helper}}$ and (A, C) from the client, and responds with something other than (A^s, C^s), then the client will abort with overwhelming probability.*

Proof. In $\mathsf{OnlineZeroTest}$, it is easy to see that the client's group elements A and C are uniformly distributed from the adversary's view. From the client's view they are computed as $A = g_1^a \cdot g^b$ and $C = g_1^c$ for uniform a, b, c. The client also holds the correct value g^s, as it was authenticated by $\mathcal{F}_{\mathsf{helper}}$. The server responds with (A', C') and the client aborts if $(A')^c \neq (C')^a (g^s)^{bc}$.

Consider one particular execution of $\mathsf{OnlineZeroTest}$ and suppose we represent the discrete logs of A and C symbolically. Specifically, $\mathsf{dlog}[A] = \delta\mathbb{A} + \mathbb{B}$ and $\mathsf{dlog}[C] = \delta\mathbb{C}$, where $\delta = \mathsf{dlog}[\widehat{g}]$. Here $\mathbb{A}, \mathbb{B}, \mathbb{C}$ are formal variables. Note that $\delta = 0$ with only negligible probability — we assume $\delta \neq 0$ below.

The client will verify whether $\mathbb{C} \cdot \mathsf{dlog}[A'] = \mathbb{A} \cdot \mathsf{dlog}[C'] + \mathbb{B}\mathbb{C}s$, after replacing every occurrence of $\mathbb{A}$ with the concrete value a, and likewise for $\mathbb{B}, \mathbb{C}$. Note that $\mathsf{dlog}[A']$ and $\mathsf{dlog}[C']$ are symbolic expressions that may include $\mathbb{A}, \mathbb{B}, \mathbb{C}$ as well. Indeed, each of these two dlog values may be any linear combination of $\mathsf{dlog}[A], \mathsf{dlog}[C]$ and 1, since $\mathsf{dlog}[A]$ and $\mathsf{dlog}[C]$ are the only dlog values that include formal variables. Write:

$$\mathsf{dlog}[A'] = u(\underbrace{\delta\mathbb{A} + \mathbb{B}}_{\mathsf{dlog}[A]}) + v(\underbrace{\delta\mathbb{C}}_{\mathsf{dlog}[C]}) + w$$

$$\mathsf{dlog}[C'] = x(\underbrace{\delta\mathbb{A} + \mathbb{B}}_{\mathsf{dlog}[A]}) + y(\underbrace{\delta\mathbb{C}}_{\mathsf{dlog}[C]}) + z$$

for scalars $u, v, w, x, y, z \in \mathbb{Z}_p$. Then the client will check the condition

$$\mathbb{C} \cdot \mathsf{dlog}[A'] = \mathbb{A} \cdot \mathsf{dlog}[C'] + \mathbb{B}\mathbb{C} \cdot s$$
$$\iff \mathbb{C}\Big[u(\delta\mathbb{A} + \mathbb{B}) + v(\delta\mathbb{C}) + w\Big] = \mathbb{A}\Big[x(\delta\mathbb{A} + \mathbb{B}) + y(\delta\mathbb{C}) + z\Big] + \mathbb{B}\mathbb{C} \cdot s$$

We will now prove the contrapositive of the statement in the lemma. Namely, suppose the client aborts with less than overwhelming probability — *i.e.*, the equation holds with non-negligible probability. We will show that the server must have sent the "correct" A', C' values. If the two sides of this equation are not identically equal as polynomials over the formal variables, then they can only be equal with negligible probability over a random assignment to the formal variables. So under our hypothesis, the two sides of the equation are identically equal as polynomials, and we may equate corresponding coefficients.

The coefficient of $\mathbb{B}\mathbb{C}$ on the right is s, and on the left is u; hence $u = s$. With $u = s$ the coefficient of $\mathbb{A}\mathbb{C}$ on the left is $u\delta = s\delta$. The coefficient of $\mathbb{A}\mathbb{C}$ on the right is $y\delta$. Since $\delta \neq 0$ we must have $y = s$. The monomial $\mathbb{C}^2$ is possible on the left with coefficient $v\delta$ but impossible on the right; hence $v = 0$. Similarly, monomial $\mathbb{C}$ is possible on the left but not the right; monomials $\mathbb{A}^2$ and $\mathbb{A}$ are possible on the right but not the left; hence $w = x = z = 0$.

In summary, if the equation holds with more than negligible probability, $\mathsf{dlog}[A'] = s \cdot \mathsf{dlog}[A]$ and $\mathsf{dlog}[C'] = s \cdot \mathsf{dlog}[C]$, as desired. □

Corrupt Client

Lemma 9. *The protocol in Figure 8 UC-securely realizes $\mathcal{F}_{\mathsf{pgg}}$ (Figure 6) against a corrupt client.*

Proof. The simulator for this case is given in Figures 10 and 11. Throughout the proof, we assume without loss of generality that the adversary does not make a query to Π or Π^{-1} whose answer is already known. I.e., it does not repeat queries to $\Pi^{\pm}$; it does not query $\Pi(x)$ if a previous query $\Pi^{-1}(y) = x$ was made; it does not query $\Pi^{-1}(y)$ if a previous query $\Pi(x)$ was made. All the adversary's queries to $\Pi^{\pm}$ are therefore *fresh*, and the simulator's behavior is described in terms of its responses to fresh queries.

We prove that the simulation is indistinguishable from the real protocol interaction, using the following sequence of hybrids:

Real interaction: The adversary interacts with an honest server running the protocol, honest generic group oracle, honest ideal permutation $\Pi^{\pm}$, and honest $\mathcal{F}_{\mathsf{helper}}$ functionality. The generic group oracle maintains dlog which maps group elements (handles) to their discrete logs.

Hybrid 1: Same as above, but let every *fresh* query to Π or Π^{-1} output a uniform result, rather than ensuring that Π remains a permutation. Then abort if any of the following bad events happen:

- There is a repeated output of Π or of Π^{-1}.
- Maintain a set $\mathcal{C}$. Each time a *fresh* call to Π^{-1} returns $(c_1, \cdot)$, add c_1 to $\mathcal{C}$. Each time $\mathcal{F}_{\mathsf{helper}}$ outputs a value c or c_3, add it to $\mathcal{C}$. Abort if a value already exists in $\mathcal{C}$ when it is being added.
- Abort if a *fresh* query to Π^{-1} results in output $(\cdot, g_1)$ where $\mathsf{dlog}[g_1]$ already defined.

The hybrids are indistinguishable if the probability of a bad event is negligible, and these three bad events have negligible probability as in the security proof for a corrupt server.

Hybrid 2: Same as the previous hybrid, except for some added bookkeeping. First choose k and (the random values chosen by $\mathcal{F}_{\mathsf{helper}}$ that determine) $\widehat{g}$ at the beginning of time.

Maintain a map DLog. Every time there is a fresh query $\Pi(c_1, g_1) = h_1$ or $\Pi^{-1}(h_1) = (c_1, g_1)$ by any party, set $\mathsf{DLog}[h_1] = (\mathsf{dlog}[g_1] - F_k^{-1}(c_1))/\mathsf{dlog}[\widehat{g}]$. Here dlog is the table from the honest generic group oracle. Writing the formula for DLog differently, we get the invariant $h_1 = \Pi(F_k(m_1), \widehat{g}^{\mathsf{DLog}[h_1]} \cdot g^{m_1})$.

Additionally, maintain a map $\mathsf{canonical}$. Every time there is a fresh query $\Pi(c_1, g_1) = h_1$ or $\Pi^{-1}(h_1) = (c_1, g_1)$, and $\mathsf{canonical}[c_i]$ is not yet defined, set $\mathsf{canonical}[c_1] = h_1$. Assume without loss of generality that every time the adversary provides a c_i value to $\mathcal{F}_{\mathsf{helper}}$, it also samples a random g_i and queries $\Pi(c_i, g_i)$. This has the effect of setting $\mathsf{canonical}[c_i]$ if it is not already defined (it also sets $\mathsf{DLog}[\mathsf{canonical}[c_i]]$).

Also assume without loss of generality that at the end of Init, the adversary queries on Π as it is instructed — *i.e.*, $h = \Pi(c, S)$. Again, this query has the effect of setting dlog and $\mathsf{canonical}$ as above. These changes involve only additional bookkeeping, so they have no effect on the adversary's view. With these changes, we have the invariant that the values $\mathsf{canonical}[c_i]$ and $\mathsf{DLog}[\mathsf{canonical}[c_i]]$ are always well-defined for any c_i value that the simulator observes.

The meaning of $\mathsf{canonical}[c]$ is the following: In $\mathsf{OnlineMult}$ and $\mathsf{OnlineZeroTest}$ the corrupt client sends only c_i values to $\mathcal{F}_{\mathsf{helper}}$. The simulator must relate these c_i values to complete handles (h_i values). The handle $\mathsf{canonical}[c_1]$ is the first handle that the simulator has seen to be associated with c_1.

Hybrid 3: Consider an honest server who runs $\mathsf{OfflineMult}$ on $u_0, (u_1, h_1), \ldots, (u_n, h_n)$ resulting in output h^*. We can easily see that $\mathsf{DLog}[h^*] = u_0 + \sum_i u_i \mathsf{DLog}[h_i]$ by applying the invariant of DLog to all terms. The server finally computes h^* by calling Π on (c^*, g^*) where c^* is uniform. It would change nothing in the adversary's view if the honest server never called $\Pi^{\pm}$, but we simply chose h^* uniformly and recorded $\mathsf{DLog}[h^*]$ as above. Then only later if/when the adversary made the query $\Pi^{-1}(h^*)$ would c^* and g^* be sampled.

Similarly, consider an honest server who runs $\mathsf{OfflineZeroTest}$ on h_1. Again, the invariant on DLog shows that the server's output is logically equivalent to $\mathsf{DLog}[h_1] \stackrel{?}{=} 0$. It would change nothing in the adversary's view if the honest server's output were simply computed by consulting $\mathsf{DLog}[h_1] \stackrel{?}{=} 0$, and it never queried Π^{-1}.

Finally, assume without loss of generality that the adversary queries Π according to the specified protocol, at the end of $\mathsf{OnlineMult}$. In other words, after providing c_1, c_2, the adversary queries $(c_1, g_1) = \Pi^{-1}(\mathsf{canonical}[c_1])$ and $(c_2, g_2) = \Pi^{-1}(\mathsf{canonical}[c_2])$. It then queries $h_3 = \Pi(c_3, g_1 g_2 S)$ as instructed. This has the effect of setting $\mathsf{canonical}[c_3]$. By some algebraic simplifications applying the invariant of DLog, we can further see that $\mathsf{DLog}[\mathsf{canonical}[c_3]] = \mathsf{DLog}[\mathsf{canonical}[c_1]] + \mathsf{DLog}[\mathsf{canonical}[c_2]]$ will always hold.

After these modifications, the honest server no longer queries $\Pi^{\pm}$. If the adversary makes a fresh query $\Pi^{-1}(h_1)$ and $\mathsf{DLog}[h_1]$ is already defined, then the result is computed as $(c_1, \widehat{g}^{\mathsf{DLog}[h_1]} \cdot g^{F_k^{-1}(c_1)})$ for random c_1.

In the case that the adversary makes a fresh query $\Pi^{-1}(h_1)$ where $\mathsf{DLog}[h_1]$ is not already defined, we proceed as before by choosing a random output (c_1, g_1) of Π^{-1} and computing $\mathsf{DLog}[h_1] = (\mathsf{dlog}[g_1] - F_k^{-1}(c_1))/\mathsf{dlog}[\widehat{g}]$. But we are already conditioning on the event that g_1 sampled in this way avoids all known group elements. Hence $\mathsf{dlog}[g_1]$ is uniform, so this is equivalent to first choosing $\mathsf{DLog}[h_1]$ uniformly and then solving $g_1 = \widehat{g}^{\mathsf{DLog}[h_1]} g^{F_k^{-1}(c_1)}$.

So in both cases, the adversary's response to fresh Π^{-1} is computed in the same way: refer to $\mathsf{DLog}[h_1]$ (sampling it uniformly if it doesn't already exist) and giving $(F_k(m), \widehat{g}^{\mathsf{DLog}[h_1]} g^m)$ as the result.

Note that DLog is still updated on every fresh query to Π.

Hybrid 4: Same as the previous hybrid, except we extend dlog and DLog to contain rational functions over formal variables $\mathbb{K}, \{\mathbb{M}_c\}_c, \{\mathbb{D}_h\}_h, \{\mathbb{S}_i\}_i$. In this hybrid we will simply be "renaming" values: $\mathbb{K}$ will represent the discrete log of $\widehat{g}$; $\mathbb{M}_c$ will represent $F_k^{-1}(c)$; $\mathbb{D}_h$ will represent $\mathsf{DLog}[h]$, and $\mathbb{S}_i$ will represent the s-value used in the ith $\mathsf{OnlineZeroTest}$.

We separately maintain a map s mapping formal variables to their concrete values. For group elements that the client receives, we express their discrete logs in terms of symbolic expressions:

- In response to a fresh query $\Pi^{-1}(h)$ made *before storage compromise*, sample uniform c and give response $(c, g^{\mathbb{K}\mathbb{D}_h+\mathbb{M}_c})$.
- In response to a fresh query $\Pi^{-1}(h)$ made *after storage compromise*, sample uniform c and give response $(c, g^{\mathbb{K}\mathbb{D}_h+F_k^{-1}(c)})$.
- During Init, give random c and $S = g^{\mathbb{K}+\mathbb{M}_c}$. (We assume Init happens before storage compromise.)
- During $\mathsf{OnlineMult}$ *before storage compromise*, let $h_1 = \mathsf{canonical}[c_1]$, $h_2 = \mathsf{canonical}[c_2]$, and pre-emptively choose h_3. $S = g^{(\mathbb{K}\mathbb{D}_{h_3}+\mathbb{M}_{c_3})-(\mathbb{K}\mathbb{D}_{h_1}+\mathbb{M}_{c_1})-(\mathbb{K}\mathbb{D}_{h_2}+\mathbb{M}_{c_2})}$
- During $\mathsf{OnlineMult}$ *after storage compromise*, do as above, but compute $m_i = F_k^{-1}(c_i)$ and give $S = g^{(\mathbb{K}\mathbb{D}_{h_3}+m_3)-(\mathbb{K}\mathbb{D}_{h_1}+m_1)-(\mathbb{K}\mathbb{D}_{h_2}+m_2)}$
- During $\mathsf{OnlineZeroTest}$ *before storage compromise*, let $\mathbb{S}_i$ be an unused formal variable and give $S = g^{\mathbb{S}_i}, T = g^{\mathbb{S}_i\mathbb{M}_{c_1}}, A' = A^{\mathbb{S}_i}, C' = C^{\mathbb{S}_i}$.
- During $\mathsf{OnlineZeroTest}$ *after storage compromise*, let $\mathbb{S}_i$ be an unused formal variable and give $S = g^{\mathbb{S}_i}, T = g^{\mathbb{S}_i m_1}, A' = A^{\mathbb{S}_i}, C' = C^{\mathbb{S}_i}$, where $m_i = F_k^{-1}(c_1)$.

For every fresh query $h_1 = \Pi(c_1, g_1)$, we symbolically set $\mathsf{DLog}[h_1] = (\mathsf{dlog}[g_1] - \mathbb{M}_{c_1})/\mathbb{K}$ before compromise, or $\mathsf{DLog}[h_1] = (\mathsf{dlog}[g_1] - F_k^{-1}(c_1))/\mathbb{K}$ after compromise.

When the server storage is compromised, we replace every formal variable $\mathbb{M}_c$ appearing in DLog or dlog with the scalar $F_k^{-1}(c)$.

When any party performs a zero-test with respect to dlog (i.e., the adversary using its generic group oracle) or DLog (*i.e.*, the honest server during $\mathsf{OfflineZeroTest}$), we check whether the formula evaluates to zero when $\mathbb{K}$ is assigned $\mathsf{dlog}[\widehat{g}]$; each $\mathbb{D}_h$ is assigned $\mathsf{DLog}[h]$; each $\mathbb{M}_c$ is assigned $F_k^{-1}(c)$; and each $\mathbb{S}_i$ is assigned $\mathsf{s}[\mathbb{S}_i]$.

There is no change to the adversary's view by making these extensive changes, because we have simply temporarily replaced some concrete scalar values with a formal variable, but substituted the concrete scalars back in during any zero-test. The only subtle issue worth observing is that we have replaced $g^{m_3-m_1-m_2}$ with something of the form $g^{(\mathbb{K}\mathbb{D}_{h_3}+\mathbb{M}_{c_3})-(\mathbb{K}\mathbb{D}_{h_1}+\mathbb{M}_{c_1})-(\mathbb{K}\mathbb{D}_{h_2}+\mathbb{M}_{c_2})}$, but we will replace each $\mathbb{D}_h$ with the corresponding $\mathsf{DLog}[h]$, and we have already established that $\mathsf{DLog}[h_3] - \mathsf{DLog}[h_1] - \mathsf{DLog}[h_2] = 0$ holds for the handles involved in $\mathsf{OnlineMult}$.

Hybrid 5: We modify the zero-tests in the previous hybrid. When testing whether a formula evaluates to zero, we replace each $\mathbb{D}_h$ formal variable with concrete value $\mathsf{DLog}[h]$ as before. But we then test whether the resulting formula — which still may contain $\mathbb{M}_c$, $\mathbb{S}_i$, $\mathbb{K}$ formal variables — is identically zero.

This hybrid differs from the previous one only in the bad event that a symbolic expression, which is not identically zero, happens to evaluate to zero on the concrete values that we assign to its formal variables. To show that this bad event has negligible probability, we observe the following:

- Expressions in dlog and DLog are always rational functions of the formal variables
- Every *single* formal variable has degree at most 1 in an expression in dlog and DLog. We may have monomials containing several formal variables (*e.g.*, $\mathbb{K} \cdot \mathbb{D}_h$), but never any squared or higher variables. The adversary may choose any group element A and have it raised to the $\mathbb{S}_i$ power during $\mathsf{OnlineZeroTest}$, but each time it is with respect to a different $\mathbb{S}_i$ variable, so we can never accumulate squared terms from this process.
- The remaining formal variables are $\mathbb{K}$, $\mathbb{M}_c$, and $\mathbb{S}_i$. During a zero-test: $\mathbb{K}$ would be replaced with $\mathsf{dlog}[\widehat{g}]$ which is uniform; $\mathbb{S}_i$ would be replaced with $\mathsf{s}[\mathbb{S}_i]$ which is uniform. If any $\mathbb{M}_c$ terms are in the expression, then it must be before server storage compromise, and before the PRP key k is given to the adversary. Formal variables $\mathbb{M}_c$ would be replaced with $F_k^{-1}(c)$ concrete values; a simple reduction to the strong PRP security of F shows that these $F_k^{-1}(c)$ values are indistinguishable from random, pre-compromise.

Overall, the bad event happens only when a rational function, which is not identically zero and has degree-1 in every individual variable, evaluates to 0 on a uniform assignment to its formal variables. For any particular rational function, this happens with probability at most $1/p$, which is negligible.

After making the changes in this hybrid, note that the concrete values corresponding to $\mathbb{K}$ and $\mathbb{S}_i$ are no longer used; and concrete values corresponding to $\mathbb{M}_c$ (*i.e.*, $F_k^{-1}(c)$ values) are no longer used pre-compromise.

Hybrid 6: In the previous hybrid, symbolic expressions can be assigned to DLog when the adversary makes the query $h_1 = \Pi(c_1, g_1)$ — *e.g.*., this query might result in the assignment $\mathsf{DLog}[h_1] = (\mathsf{dlog}[g_1] - \mathbb{M}_{c_1})/\mathbb{K}$. We modify the behavior in this case, as follows:

- If $\mathsf{dlog}[g_1]$ has the form $\mathbb{K}(u_0 + \sum_i u_i \mathbb{D}_{h_i}) + \mathbb{M}_{c_1}$, or the form $\mathbb{K}(u_0 + \sum_i u_i \mathbb{D}_{h_i}) + F_k^{-1}(c_1)$, for some set of u_i's and h_i's, then simply set $\mathsf{DLog}[h_1] = u_0 + \sum_i u_i \mathsf{DLog}[h_i]$. Previously, the assignment would have resulted in symbolic expression $\mathsf{DLog}[h_1] = u_0 + \sum_i u_i \mathbb{D}_{h_i}$. Since $\mathbb{D}_h$ values are always replaced by corresponding $\mathsf{DLog}[h]$ values during zero-tests, the change has no effect on the adversary.
- In all other cases, set $\mathsf{DLog}[h_1]$ to be a uniform value.

We must argue that the second change is indistinguishable to the adversary. Note that the DLog values are accessed only during a zero test — at all other times, the actual values of DLog are "hidden" behind a formal variable $\mathbb{D}_h$.

In the previous hybrid, the particular problematic $\mathsf{DLog}[h_1]$ held a symbolic expression that contained (at the very least) a term $\mathbb{K}$ in the denominator. Any zero-tests that involve such a term will return false because the expression will not be identically zero. After the change to this hybrid, $\mathsf{DLog}[h_1]$ contains a random scalar. The change is noticeable from the adversary's view only if they are able to "cancel off" the random scalar in some expression, making some zero-test succeed in this hybrid where it would have failed in the previous hybrid. Since the scalar assigned to $\mathsf{DLog}[h_1]$ is uniform and independent of the adversary's view, the probability of this event is negligible, so the hybrids are indistinguishable.

Now observe that after making these changes, DLog contains only scalar values at all times.

We make some observations about what kinds of group elements the adversary can generate in this hybrid.

- If it is pre-compromise, then the adversary has only seen group elements of the form $g^{\mathbb{K}\mathbb{D}_h + \mathbb{M}_c}$, where $h = \mathsf{canonical}[c]$, or in the case of $\mathsf{OnlineMult}$, it sees the product of 3 such elements.

In other words, $\mathbb{D}_h$ appears only for $h = \mathsf{canonical}[c]$ for some c, and it always appears with $\mathbb{K}$ and $\mathbb{M}_c$ as a "bundle" of the form $\mathbb{K}\mathbb{D}_h + \mathbb{M}_c$. The only other way $\mathbb{D}_h$ can appear in dlog is if the adversary asks for some group element to be raised to the $\mathbb{S}_i$ power in $\mathsf{OnlineZeroTest}$. This will have the effect of multiplying the dlog-expression for each such "bundle" by a $\mathbb{S}_i$ term.

- If it is pre-compromise, then the only group elements involving $\mathbb{M}_c$ that the adversary has seen are those described above (with dlog containing bundles of the form $\mathbb{K}\mathbb{D}_h + \mathbb{M}_c$), and elements with $\mathsf{dlog} = \mathbb{S}_i\mathbb{M}_c$, only when it invokes $\mathsf{OnlineZeroTest}$ with that c.

With these two observations in mind, the only way that the adversary can cancel a $\mathbb{M}_c$ term from a bundle $\mathbb{K}\mathbb{D}_h + \mathbb{M}_c$ is to do a zero-test involving c to obtain $\mathbb{S}_i\mathbb{M}_c$, and also raise the corresponding bundle to the $\mathbb{S}_i$ power to cancel off $\mathbb{S}_i\mathbb{M}_c$, leaving behind a term $\mathbb{S}_i\mathbb{K}\mathbb{D}_h$.

Note that every distinct call to $\mathsf{OnlineZeroTest}$ will involve a distinct $\mathbb{S}_i$. This $\mathbb{S}_i$ is tied to a particular $\mathbb{M}_c$ variable, and $\mathbb{M}_c$ only ever appears in a bundle with $\mathbb{D}_h$ for $h = \mathsf{canonical}[c]$. So, if any two terms of the form $\mathbb{S}_i\mathbb{K}\mathbb{D}_h$ have different h subscripts, then they must also have different i subscripts.

We can now observe the following about this hybrid. When an adversary queries Π using g_1 with $\mathsf{dlog}[g_1]$ of the form $\mathbb{K}(u_0 + \sum_i \mathbb{D}_{h_i}) + \mathbb{M}_c$, the value DLog will be assigned in a special way as described above. But following the reasoning above, before compromise, there is no way to isolate a term $\mathbb{K}\mathbb{D}_h$ without either a multiplicative $\mathbb{S}_i$ or an additive $\mathbb{M}_c$. So the only way this case can happen is if $u = 0$ and the sum is over one item — *i.e.*, $\mathsf{dlog}[g_1]$ has the form $\mathbb{K}\mathbb{D}_h + \mathbb{M}_c$, where we must also have $h = \mathsf{canonical}[c]$. In this case, the hybrid simply sets $\mathsf{DLog}[h'] = \mathsf{DLog}[h]$ for the new handle h'.

We make a further observation about how zero-tests work in this hybrid. When an adversary makes a zero-test for some g_1 (with the common generic group oracle, *i.e.*, with respect to the dlog map), we replace each $\mathbb{D}_h$ term in $\mathsf{dlog}[g_1]$ with $\mathsf{DLog}[h]$ in the expression and see if the result is identically zero. There are two cases to consider:

- No linear combination of $\mathbb{D}_h$ variables and 1 divides $\mathsf{dlog}[g_1]$. Then no matter what the values of $\mathsf{DLog}[h]$ are, the result after substituting them will still be a nonzero symbolic expression, and the zero-test will return false.

- Some linear combination of $\mathbb{D}_h$ variables and 1 divides $\mathsf{dlog}[g_1]$. If this is pre-compromise, then by our previous reasoning, the only way this can happen is if $\mathsf{dlog}[g_1]$ is of the form $\mathbb{S}_i\mathbb{K}\mathbb{D}_h$, where the corresponding $\mathsf{OnlineZeroTest}$ has been performed. A linear combination containing more than one $\mathbb{D}_h$ variable cannot divide $\mathsf{dlog}[g_1]$ because either the corresponding $\mathbb{M}_c$ terms will be present as additive terms, or each $\mathbb{D}_h$ has been isolated from its $\mathbb{M}_c$ partner but paired with *different* $\mathbb{S}_i$ variables.

  In the case $\mathsf{dlog}[g_1] = \mathbb{S}_i\mathbb{K}\mathbb{D}_h$, if $\mathsf{DLog}[h] = 0$ then the expression is identically zero and the zero-test succeeds. Otherwise, the result is a symbolic expression that is not identically zero, and the zero-test fails.

- If this is post-compromise, then the only way a linear combination of $\mathbb{D}_h$ values divides $\mathsf{dlog}[g_1]$, other than the case above, is if $\mathsf{dlog}[g_1] = \mathbb{K}(u_0 + \sum_i u_i\mathbb{D}_{h_i})$. No $\mathbb{D}_h$ term can be separated from $\mathbb{K}$, and if any $\mathbb{D}_h$ is multiplied by $\mathbb{S}_i$, then no other $\mathbb{D}_h$ can be multiplied by the same $\mathbb{S}_i$, and we are in the previous case. Again, if $u_0 + \sum_i u_i\mathsf{DLog}[h_i] = 0$ then the entire expression is identically zero and the zero-test succeeds. Otherwise, the result is a symbolic expression that is not identically zero, and the zero-test fails.

Ideal interaction: The previous hybrid has identical behavior to the ideal interaction, only with some operations delegated to the ideal functionality. One can see that the adversary is given symbolic group elements following the description above. Upon compromise, all $\mathbb{M}_c$ variables are replaced with their corresponding concrete values in dlog.

During $\mathsf{OnlineMult}$, the interaction registers a new handle whose DLog is the sum of those for $\mathsf{canonical}[c_1]$ and $\mathsf{canonical}[c_2]$. In the simulation, the maintenance of DLog is handled by $\mathcal{F}_{\mathsf{pgg}}$.

Before compromise, if the adversary performs a zero-test on an element whose dlog contains a $\mathbb{D}_h$ variable, that zero-test will fail unless a previous $\mathsf{OnlineZeroTest}$ was performed for the corresponding c (such that $\mathsf{canonical}[c] = h$) and succeeded. In the simulation, the simulator can issue a $\mathsf{OnlineZeroTest}$ command to learn whether $\mathsf{DLog}[h] = 0$, and simulate the result of the zero-test accordingly.

After compromise, the adversary can perform its own zero-tests, but only for linear combinations of handles. The simulator can handle this by sending an $\mathsf{OfflineMult}$ command (for the same linear combination) and then sending a $\mathsf{OfflineZeroTest}$ command to learn whether the linear combination is zero. □

5.4 io2PC Protocol for Generic-Group Obfuscation

Finally, with a personalized generic group, we can realize io2PC for any function that has a suitable VBB obfuscation in the generic group model. The protocol is essentially the same as our io2PC for random-oracle obfuscation (Figure 4), but we replace the OPRF with a personalized generic group. We give the details in Figure 12.

Theorem 10. *Suppose* $(\mathsf{Obf}, \mathsf{ObfEval}, c)$ *is a VBB obfuscation for* $\mathcal{C}_f$ *with simulation rate* r*, in the generic group model. Then the io2PC protocol (Figure 12) realizes the* $\mathcal{F}_{\mathrm{iO2PC}}$ *functionality computing* $\mathcal{C}_f$ *with simulation rate* r *(Figure 1) in the* $\mathcal{F}_{\mathsf{pgg}}$*-hybrid world.*

The proof is essentially identical to that of Theorem 5, with the obvious changes replacing the random oracle / OPRF with generic group / personalized group.

6 Compatible Obfuscations

In this section we discuss obfuscations that are compatible with our io2PC approach, namely those that are input-independent, virtual black-box, and extractable.

6.1 Point Functions

For the point function, *i.e.* the function family $\mathcal{C}_f$ where $f(x, y) = (x \stackrel{?}{=} y)$, there is a simple obfuscation in the random oracle model. We only sketch the scheme and its security argument: given a random oracle H with range $\mathbf{H}$, let $\mathsf{Obf}(x)$ output $H(x)$, and $\mathsf{ObfEval}(\mathcal{O}_x, y)$ output $(H(y) \stackrel{?}{=} \mathcal{O}_x)$. Clearly, this scheme is correct and input-independent with query rate $c = 1$. The VBB simulator chooses $\mathcal{O}_x \leftarrow \mathbf{H}$ and answers the adversary's $H(y)$ queries as follows: it learns whether $y = x$ via querying $f(x, y)$, and if so, it returns $\mathcal{O}_x$; otherwise it returns a random element in $\mathbf{H}$. The simulation rate is 1 as $Q_S = Q_A$.

For the extractability property, $\mathsf{Extract}(\mathcal{O}, \mathcal{H})$ checks if there is an $x \in \mathcal{H}$ such that $H(x) = \mathcal{O}$. If there is more than one such x, $\mathsf{Extract}$ aborts; if there is exactly one such x, it outputs x; if there is no such x, it outputs $\perp$. It is not hard to see that the probability of the bad event in the definition of extractability is negligible (it happens only if $\mathcal{A}$ finds a collision in H, or finds $\mathcal{O}, y$ with $\mathcal{O} = H(y)$ without querying $H(y)$).

Parameters:

- Obfuscation $(\mathsf{Obf}, \mathsf{ObfEval}, c)$ for the class of functions $\mathcal{C}_f = \{f(x, \cdot) \mid x \in \{0,1\}^*\}$, in the generic-group model.
- Client C and server S.

On command (Init, sid, x), S for S:

1. S: Send (Init, sid) to $\mathcal{F}_{\mathsf{pgg}}$
2. S: Receive response (Init, sid, h)
3. S: Run $\mathcal{O}_x \leftarrow \mathsf{Obf}^{?}(x)$, where each time Obf queries its oracle:
   - If the query is of the form $\mathsf{Mult}(h_1, h_2)$:
     - Send $(\mathsf{OfflineMult}, sid, ssid, 0, (1, h_1), (1, h_2))$ to $\mathcal{F}_{\mathsf{pgg}}$
     - Receive response $(\mathsf{OfflineMult}, sid, ssid, h_3)$
     - Give h_3 to Obf as the response to its oracle query
   - If the query is of the form $\mathsf{ZeroTest}(h_1)$:
     - Send $(\mathsf{OfflineZeroTest}, sid, ssid, h_1)$ to $\mathcal{F}_{\mathsf{pgg}}$
     - Receive response $(\mathsf{OfflineZeroTest}, sid, ssid, b)$
     - Give b to Obf as the response to its oracle query
4. S: Store $\mathcal{O}_x$.

On command $(\mathsf{Compromise}, sid)$ from $\mathcal{A}^*$:

5. $\mathcal{A}^*$ must also send $(\mathsf{Compromise}, sid)$ to $\mathcal{F}_{\mathsf{pgg}}$
6. $\mathcal{A}^*$ learns $\mathcal{O}_x$.

On command $(\mathsf{IOEval}, sid, ssid)$ for S:

7. S: Send $(sid, ssid, \mathcal{O}_x)$ to C.
8. Both parties set $i := 0$
9. C: Await command $(\mathsf{IOEval}, sid, ssid, y)$.
10. C: Run $z := \mathsf{ObfEval}^{?}(\mathcal{O}_x, y)$, where each time $\mathsf{ObfEval}$ queries its oracle:
    - If the query is of the form $\mathsf{Mult}(h_1, h_2)$:
      - C: Send $(\mathsf{OnlineMult}, sid, ssid\|i, h_1, h_2)$ to $\mathcal{F}_{\mathsf{pgg}}$
      - S: Await $(\mathsf{OnlineMult}, sid, ssid\|i)$ from $\mathcal{F}_{\mathsf{pgg}}$
      - S: Send $(\mathsf{Deliver}, sid, ssid\|i)$ to $\mathcal{F}_{\mathsf{pgg}}$
      - C: Await response $(\mathsf{OnlineMult}, sid, ssid\|i, h_3)$ from $\mathcal{F}_{\mathsf{pgg}}$
      - C: Give h_3 to Obf as the response to its oracle query
    - If the query is of the form $\mathsf{ZeroTest}(h_1)$:
      - C: Send $(\mathsf{OnlineZeroTest}, sid, ssid\|i, q)$ to $\mathcal{F}_{\mathsf{pgg}}$
      - S: Await $(\mathsf{OnlineZeroTest}, sid, ssid\|i)$ from $\mathcal{F}_{\mathsf{pgg}}$
      - Both: set $i := i + 1$
      - S: If $i > c$: abort. Otherwise, send $(\mathsf{Deliver}, sid, ssid\|i)$ to $\mathcal{F}_{\mathsf{pgg}}$
      - C: Await response $(\mathsf{OnlineZeroTest}, sid, ssid\|i, b)$ from $\mathcal{F}_{\mathsf{pgg}}$
      - C: Give b to Obf as the response to its oracle query
11. C: Output $(\mathsf{IOEval}, sid, ssid, z)$

Figure 12: The $\mathsf{io2PC}$ protocol for computing function f, based on a VBB obfuscation in the generic group model.

6.2 Hyperplane Membership

Extending the idea of point-function obfuscation above, we may consider the same function in higher dimensional spaces. In this section, we provide a new proof for a hyperplane membership protocol in the generic group model.

Let p be a prime with $||p|| = \kappa$ and $d = \mathsf{poly}(\kappa)$. For $\boldsymbol{x} \in \mathbb{Z}_p^d$, define function $F_{\boldsymbol{x}} : \mathbb{Z}_p^d \to \{\text{FALSE}, \text{TRUE}\}$ as

$$F_{\boldsymbol{x}}(\boldsymbol{y}) = \begin{cases} \text{TRUE} & \text{if } \langle \boldsymbol{x}, \boldsymbol{y} \rangle = 0 \\ \text{FALSE} & \text{otherwise} \end{cases}$$

i.e. $F_{\boldsymbol{x}}$ computes membership in the subspace of $\mathbb{Z}_p^d$ containing all vectors orthogonal to $\boldsymbol{x}$. We use $\mathcal{F}_p^d$ to denote the function family $\{F_{\boldsymbol{x}}\}$. Obfuscation of $\mathcal{F}_p^d$ has been considered previously [CRV10], and we recall the construction below.

6.2.1 Obfuscation

The obfuscation in Figure 13 is due to Canetti, Rothblum, and Varia [CRV10] whose proof is based on strong DDH assumption proven in the GGM, but the proof constructs an inefficient simulator in the dimension of the ambient space. We reconsider the protocol and prove for an efficient simulator with access to a global GGM.

Parameters:
Generic group G with handle space $\mathbf{H}$.
Public generator g of prime order p.
Ambient space dimension d.

$\mathsf{Obf}(\boldsymbol{x})$:
Sample a generator γ of G.
Return $\mathcal{O}_x = (\gamma^{x_i})_{i \in [d]}$.

$\mathsf{ObfEval}(\mathcal{O}_x, \boldsymbol{y})$:
Interpret $\mathcal{O}_x$ as $(o_i)_{i \in [d]} \in \mathbf{H}^d$.
Return $\prod_i o_i^{x_i} \stackrel{?}{=} g^0$

Figure 13: VBB Hyperplane Membership Obfuscation

On input $\boldsymbol{x} = (x_i)_{i \in [d]} \in \mathbb{Z}_p^d$ and input $\boldsymbol{y} = (y_i)_{i \in [d]} \in \mathbb{Z}_p^d$, correctness is immediately evident as $\mathsf{ObfEval}(\mathsf{Obf}(\boldsymbol{x}), \boldsymbol{y})$ computes $\prod_i (\gamma^{x_i})^{y_i} = \gamma^{\langle \boldsymbol{x}, \boldsymbol{y} \rangle} \stackrel{?}{=} \gamma^0$. The obfuscation algorithm $\mathsf{Obf}(\boldsymbol{x})$ must be careful about optimizing its generic group operations, however. Even if $x_i = x_j$ for distinct i, j, the obfuscation algorithm must ensure that distinct handles are generated for γ^{x_i} and γ^{x_j}; *e.g.*, by separately multiplying by g^0. Finally, note that depending on how Figure 13 is implemented, the number of multiplication queries that $\mathsf{ObfEval}$ makes is data dependent. Specifically, when evaluating exponentiation through squaring $\mathsf{ObfEval}$ will compute g^2 with one query while computing g^{127} will require 12 queries. To make the total number of multiplication queries a constant, we may simply require a constant-time exponentiation algorithm.

In our previous definition for simulation rate, we stated that for an obfuscation to have simulation rate r, it must hold that $Q_S \leq r \cdot \frac{Q_A}{c}$. However, the GGM oracle has two interfaces for queries: the multiplication query Mult and the zero test query $\mathsf{ZeroTest}$. As we stated earlier (see Section 5.2), it is much more important to measure an adversary's effort in terms of zero-tests and not group multiplications. If we only count $\mathsf{ZeroTest}$ queries, the obfuscation scheme is indeed limited by a single query with $Q_S = Q_A$. In the theorem below, the statements about query rate and simulation rate refer only to $\mathsf{ZeroTest}$ queries.

6.2.2 Virtual Black-Box Property

Theorem 11. *The scheme in Figure 13 is a VBB obfuscation Definition 3 for F in the Generic Group Model, with query rate $c = 1$ and simulation rate $r = 1$.*

Proof Sketch:
The simulator Sim replaces the obfuscation $\mathcal{O}_x$ with uniformly sampled handles $\mathcal{O} \leftarrow \mathbf{H}^d$ and then plays the role of the two GG oracles Mult and $\mathsf{ZeroTest}$. In the real world, the obfuscation uses a sampled generator γ with uniform discrete logarithm and since this value is outside the adversary's view, we represent it with the formal variable $\mathbb{K}$. Sim then catalogs the symbolic discrete logarithms of all multiplications the adversary makes relative to handles $\{o_i\}_{i\in[d]}$, comprising $\mathcal{O}$, and the public generator g. As the adversary can only gain information about relations between group elements through a zero-test, it can't tell if $\mathcal{O}$ was replaced until it interacts with the $\mathsf{ZeroTest}$ oracle. When the adversary makes such a zero-test, Sim checks the discrete logarithm of that group element. By construction, the discrete logarithm of these queries will take on the form of a polynomial $\mathbb{K}\left(\sum_i a_i x_i\right) + z$, for coefficients $a_i, z \in \mathbb{Z}_p$, relative to base g. Noting that $\sum_i a_i x_i$ is exactly $\langle \boldsymbol{x}, \boldsymbol{a} \rangle$, Sim may then check if this combination is zero by querying the function oracle $f(\boldsymbol{x}, \boldsymbol{a})$. But since the simulator does not need to know the x_i to make the query, the simulator may run agnostic of the input $\boldsymbol{x}$.

Proof. Let $\mathcal{A}$ be any polynomial-time adversary. The simulator $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$ is constructed as follows:

- Sim_1 samples $\mathcal{O} \leftarrow \mathbf{H}^d$ and outputs $(\mathcal{O}, \mathcal{O})$.
- Sim_2, on input $\mathcal{O}$, creates formal variables $\mathbb{K}$ representing the discrete logarithm of γ relative to base g, and $\{\mathbb{O}_i\}_{i\in[d]}$ where each $\mathbb{O}_i$ represents the discrete logarithm of the ith handle of $\mathcal{O}$ relative to base γ. Sim_2 then acts as the Mult oracle relative to these variables. On query $(\mathsf{ZeroTest}, h)$ from $\mathcal{A}$, Sim_2 interprets the discrete logarithm of h relative to base g as a linear combination of bases $\{(\mathbb{K}\mathbb{O}_i)_{i\in[d]}, 1\}$:

$$\mathbb{K}\left(\sum_i a_i \mathbb{O}_i\right) + z$$

  If $z = 0$, Sim_2 sends $\boldsymbol{a}$ to its function oracle, and forwards the response to $\mathcal{A}$. If $z \neq 0$, Sim_2 responds with FALSE.

Our proof of indistinguishability will be carried out through a series of hybrids starting from the real interaction $\{\mathcal{O}_x \leftarrow \mathsf{Obf}^G(x); \mathcal{A}^G(\mathcal{O}_x)\}$ and ending in the ideal interaction $\{(\mathcal{O}, \mathsf{state}) \leftarrow \mathsf{Sim}_1; \mathcal{A}^{\mathsf{Sim}_2^{f(x,\cdot)}(\mathsf{state})}(\mathcal{O})\}$.

Real interaction: The real world is characterized by the $\mathcal{A}$'s interactions with the GGM oracle G with public generator g and the real obfuscation $\mathcal{O}_x$ of $\mathcal{F}_p^d$ on input $\boldsymbol{x}$. $\mathcal{O}_x$ is constructed by sampling a generator γ and returning $(\gamma^{x_i})_{i\in[d]}$.

Hybrid 1: (Replacement of the Mult Oracle) In this hybrid, we introduce an algorithm $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$ which plays the role of the game challenger. Sim_1 is identical to the first phase of the real world, *i.e.* it generates $\mathcal{O}_x \leftarrow \mathsf{Obf}^G(x)$ and outputs $\mathcal{O}_x$ to $\mathcal{A}$. Sim_2 now simulates the GGM oracle honestly, keeping track of the information between the interaction with Obf and with $\mathcal{A}$. Sim_2 stores these query responses (handle, discrete logarithm) in a lookup table DL. Additionally, Sim_2 responds with uniform handles from $\mathbf{H}$ upon Mult queries and aborts when a handle sampled by $\mathcal{A}$ gets assigned a discrete logarithm which collides with a known handle's or when a new handle collides with a previously generated handle.

$\mathcal{A}$'s view diverges exactly when Sim_2 aborts which occurs with negligible probability by a birthday bound.

Hybrid 2: (Replacement of the ZeroTest Oracle and Obfuscation) In this and future hybrids, $\boldsymbol{x}$ is no longer used during the computation of $\mathcal{O}_x$; instead, Sim_1 generates a dummy obfuscation by sampling $\mathcal{O} \leftarrow \mathbf{H}^d$.

Sim_2 now simulates the Mult oracle by allowing DL to store polynomials over formal variables. Specifically, Sim_2 introduces formal variables $\mathbb{K}$ and $(\mathbb{O}_i)_{i\in[d]}$, and stores $DL[\gamma] := \mathbb{K}$ and $DL[o_i] := \mathbb{K}\mathbb{O}_i$ (where o_i is the ith handle of $\mathcal{O}_x$). Sim_2 also keeps track of the mapping between $\mathbb{O}_i$ and x_i. When $\mathcal{A}$ makes a $\mathsf{Mult}(h_1, h_2)$ query whose response is h, Sim_2 stores $DL[h] := DL[h_1] + DL[h_2]$.

On query $(\mathsf{ZeroTest}, h)$ from $\mathcal{A}$, Sim_2 references $DL[h] = \mathbf{S}$ where $\mathbf{S}$ is a polynomial of the form $\mathbb{K}\left(\sum_{i\in[d]} a_i \cdot \mathbb{O}_i\right) + z$ for coefficients $a_i, z \in \mathbb{Z}_p$. Sim_2 then retrieves $(x_i)_{i\in[d]}$, and returns TRUE iff $z = 0$ and $\sum_{i\in[d]} a_i x_i = \langle \boldsymbol{x}, \boldsymbol{a}\rangle = 0$. Equivalently,

1. Sim_2 replaces each polynomial variable $\mathbb{O}_i$ with integer $x_i \in \mathbb{Z}_p$.
2. If $z = 0$ and $\sum_{i\in[d]} a_i x_i \cdot \mathbb{K} = \langle \boldsymbol{x}, \boldsymbol{a}\rangle \cdot \mathbb{K}$ is the zero polynomial of variable $\mathbb{K}$, then Sim_2 returns TRUE.
3. Otherwise, Sim_2 returns FALSE.

Clearly, adding the bookkeeping in Mult queries does not change $\mathcal{A}$'s view. $\mathcal{A}$'s view diverges when Sim_2 responds to $\mathsf{ZeroTest}$ with a different result than in the previous hybrid. This happens only when $\langle \boldsymbol{x}, \boldsymbol{a}\rangle \cdot \mathbb{K} + z$ is not the zero polynomial of $\mathbb{K}$, but the polynomial evaluates to 0 when $\mathbb{K}$ is instantiated with $\log_g \gamma$ — in which case the current hybrid returns FALSE but the previous hybrid returns TRUE. Note that $\log_g \gamma$ is uniform in $\mathbb{Z}_q \setminus \{0\}$, so for a single $\mathsf{ZeroTest}$ query, if $z = 0$, the bad event above cannot happen; if $z \neq 0$, the probability of the bad event is $\frac{1}{p}$. Overall, the distinguishing advantage between Hybrid 2 and Hybrid 1 is upper-bounded by $\frac{Q_A}{p}$, where Q_A is the number of $\mathcal{A}$'s $\mathsf{ZeroTest}$ queries.

Hybrid 3: (Removing x from the ZeroTest Oracle) We may now note that the only step requiring the x_i in the previous hybrid is the computation of $\langle \boldsymbol{x}, \boldsymbol{a}\rangle \stackrel{?}{=} 0$ in the simulation of $\mathsf{ZeroTest}$. Instead of computing the inner product locally, the simulator will now asks the function oracle to compute the result: When $\mathcal{A}$ makes a $(\mathsf{ZeroTest}, h)$ query, Sim_2 follows the same strategy as the previous hybrid, but instead of replacing $\mathbb{O}_i$ with x_i and calculating $\langle \boldsymbol{x}, \boldsymbol{a}\rangle$, Sim_2 queries $f(\boldsymbol{x}, \boldsymbol{a})$ against the function oracle $f(\boldsymbol{x}, \cdot)$ and returns the oracle's response. Sim_2 response to the zero-test does not change as the only way $\langle \boldsymbol{x}, \boldsymbol{a}\rangle$ was used was to check whether or not the result was the zero polynomial.

Ideal Interaction: Hybrid 3 is exactly the ideal world as in both cases:

- Sim_1 samples $\mathcal{O} \leftarrow \mathbf{H}^d$.
- Sim_2 answers multiplication queries symbolically and without collisions.
- Sim_2 answers all $\mathsf{ZeroTest}(h)$ queries by extracting the discrete logarithm $\sum_i a_i \mathbb{O}_i$ relative to a symbolic γ — this process is independent of $\boldsymbol{x}$ — and querying against the function oracle on $\boldsymbol{a}$.
- When Sim_2 fails to extract $\boldsymbol{a}$, it returns FALSE.

□

6.2.3 Extractability

The construction in Figure 13 is extractable (Definition 4) through the following algorithm:

- $\mathsf{Extract}$ on input $(\mathcal{O}, \mathcal{H})$ iterates through all handles in $\mathcal{H}$ and catalogs their discrete logarithms relative to g in a list DL.
  - If any handles h were sampled by $\mathcal{A}$, $\mathsf{Extract}$ samples a uniform discrete logarithm $DL[h] \leftarrow \mathbb{Z}_p$.
- $\mathsf{Extract}$, interprets $\mathcal{O}$ as $(o_i)_{i\in[d]} \in \mathbf{H}^d$, and for each o_i:
  - If $DL[o_i]$ is defined, $\mathsf{Extract}$ sets $x_i := DL[o_i]$.
  - Otherwise, $\mathsf{Extract}$ samples $x_i \leftarrow \mathbb{Z}_p$.
- $\mathsf{Extract}$ finally returns $\boldsymbol{x} = (x_i)_{i\in[d]}$.

References

[ADDS21] 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, Part II*, volume 12711 of *LNCS*, pages 261–289. Springer, Heidelberg, May 2021.

[BGI+14] Amos Beimel, Ariel Gabizon, Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, and Anat Paskin-Cherniavsky. Non-interactive secure multiparty computation. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 387–404. Springer, Heidelberg, August 2014.

[BLMZ19] James Bartusek, Tancrède Lepoint, Fermi Ma, and Mark Zhandry. New techniques for obfuscating conjunctions. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 636–666. Springer, Heidelberg, May 2019.

[BM93] Steven M. Bellovin and Michael Merritt. Augmented encrypted key exchange: A password-based protocol secure against dictionary attacks and password file compromise. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93*, pages 244–250. ACM Press, November 1993.

[BR13] Zvika Brakerski and Guy N. Rothblum. Obfuscating conjunctions. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 416–434. Springer, Heidelberg, August 2013.

[BR14] Zvika Brakerski and Guy N. Rothblum. Virtual black-box obfuscation for all circuits via generic graded encoding. In Yehuda Lindell, editor, *TCC 2014*, volume 8349 of *LNCS*, pages 1–25. Springer, Heidelberg, February 2014.

[Can97] Ran Canetti. Towards realizing random oracles: Hash functions that hide all partial information. In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 455–469. Springer, Heidelberg, August 1997.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[CD08] Ran Canetti and Ronny Ramzi Dakdouk. Obfuscating point functions with multibit output. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 489–508. Springer, Heidelberg, April 2008.

[CRV10] Ran Canetti, Guy N. Rothblum, and Mayank Varia. Obfuscation of hyperplane membership. In Daniele Micciancio, editor, *TCC 2010*, volume 5978 of *LNCS*, pages 72–89. Springer, Heidelberg, February 2010.

[FIPR05] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Heidelberg, February 2005.

[GMR06] Craig Gentry, Philip MacKenzie, and Zulfikar Ramzan. A method for making password-based key exchange resilient to server compromise. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 142–159. Springer, Heidelberg, August 2006.

[GMRW13] S. Dov Gordon, Tal Malkin, Mike Rosulek, and Hoeteck Wee. Multi-party computation of polynomials and branching programs without simultaneous interaction. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 575–591. Springer, Heidelberg, May 2013.

[Hes20] Julia Hesse. Separating symmetric and asymmetric password-authenticated key exchange. In Clemente Galdi and Vladimir Kolesnikov, editors, *SCN 20*, volume 12238 of *LNCS*, pages 579–599. Springer, Heidelberg, September 2020.

[HIJ+17] Shai Halevi, Yuval Ishai, Abhishek Jain, Ilan Komargodski, Amit Sahai, and Eylon Yogev. Non-interactive multiparty computation without correlated randomness. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part III*, volume 10626 of *LNCS*, pages 181–211. Springer, Heidelberg, December 2017.

[HLP11] Shai Halevi, Yehuda Lindell, and Benny Pinkas. Secure computation on the web: Computing without simultaneous interaction. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 132–150. Springer, Heidelberg, August 2011.

[JKK14] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 233–253. Springer, Heidelberg, December 2014.

[JKX18] 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, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 456–486. Springer, Heidelberg, April / May 2018.

[JL09] Stanislaw Jarecki and Xiaomin Liu. Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 577–594. Springer, Heidelberg, March 2009.

[JS08] Tibor Jager and Jörg Schwenk. On the equivalence of generic group models. In Joonsang Baek, Feng Bao, Kefei Chen, and Xuejia Lai, editors, *ProvSec 2008*, volume 5324 of *LNCS*, pages 200–209. Springer, Heidelberg, October / November 2008.

[LPS04] Ben Lynn, Manoj Prabhakaran, and Amit Sahai. Positive results and techniques for obfuscation. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 20–39. Springer, Heidelberg, May 2004.

[Mau05] Ueli M. Maurer. Abstract models of computation in cryptography (invited paper). In Nigel P. Smart, editor, *10th IMA International Conference on Cryptography and Coding*, volume 3796 of *LNCS*, pages 1–12. Springer, Heidelberg, December 2005.

[Sho97] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 256–266. Springer, Heidelberg, May 1997.

[TPY+19] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. Protecting accounts from credential stuffing with password breach alerting. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1556–1571, Santa Clara, CA, August 2019. USENIX Association.

[Wee05] Hoeteck Wee. On obfuscating point functions. In Harold N. Gabow and Ronald Fagin, editors, *37th ACM STOC*, pages 523–532. ACM Press, May 2005.