Oblivious Revocable Functions and Encrypted Indexing

Kevin Lewi klewi@fb.com
Jon Millican jmillican@fb.com
Ananth Raghunathan ananthr@fb.com
Arnab Roy arnabr@fb.com

Abstract

Many online applications, such as online file backup services, support the sharing of indexed data between a set of devices. These systems may offer client-side encryption of the data, so that the stored data is inaccessible to the online host. A potentially desirable goal in this setting would be to protect not just the contents of the backed-up files, but also their identifiers. However, as these identifiers are typically used for indexing, a deterministic consistent mapping across devices is necessary. Additionally, in a multi-device setting, it may be desirable to maintain an ability to revoke a device's access—e.g. through rotating encryption keys for new data.

We present a new primitive, called the Oblivious Revocable Function (ORF), which operates in the above setting and allows identifiers to be obliviously mapped to a consistent value across multiple devices, while enabling the server to permanently remove an individual device's ability to map values. This permits a stronger threat model against metadata, in which metadata cannot be derived from identifiers by a revoked device colluding with the service provider, so long as the service provider was honest at the instant of revocation. We describe a simple Diffie-Hellman-based construction that achieves ORFs and provide a proof of security under the UC framework.

1 Introduction

In a traditional encrypted cloud storage service, a centralized service provider manages a server that holds encrypted files which can be accessed and decrypted only by a set of devices associated with a user. Here, the encryption provides confidentiality for the user's data—since only the user's devices own the decryption keys needed to access the data, the server is unable to recover the files in plaintext.

However, this confidentiality is limited to the file's contents, since the server still has access to all metadata needed in order to properly retrieve these encrypted files for the user. In particular, the *filenames* associated with the user's files need to be used as an index for the server to find the user's files, and are hence leaked to the server. In this work, we consider solutions that address this problem, which we call *encrypted indexing*, by allowing the client to keep human-readable filenames while ensuring that these same filenames are not leaked to the service provider.

In the single-device setting, there is a rather straightforward solution to encrypted indexing: a client can simply prescramble the filenames by selecting a pseudorandom identifier for each file to use as a label for the server, in place of its actual filename. These pseudorandom identifiers could be derived as outputs of a keyed pseudorandom function (PRF) F, where the key K is kept on the device, and the input to the PRF is the filename. As long as the server does not have access to K, confidentiality of the filenames is preserved.

Supporting multiple devices. While the above approach may be sufficient when the user has only one device, it is less secure when the user controls multiple devices, each of which must hold the PRF key. In particular, if any of the devices are compromised and the key leaked, then confidentiality from the server no longer holds.

As a concrete example, consider the scenario in which a user wishes to access the encrypted cloud storage service from a web browser on a public computer. An adversary that compromised the public computer could collude with the service provider in an attempt to break confidentiality.

Enabling revocation. Note that in the multi-device setting, devices can be registered to a user, as well as be revoked. A device revocation allows a user to delete their device's copy of the PRF key. However, this relies on the user to proactively trigger the revocation before being compromised by an adversary. In practical settings, this can often be an unrealistic assumption—akin to expecting that a user will remember to "log out" of their account on a web browser on a public computer. And in the case where a user loses a device that falls into the hands of an adversary, preemptive device revocation would no longer be possible.

Instead of client revocation, another option for maintaining security is server revocation for a device, which can be triggered without the device having to take any action. This could happen either through the user using another device to connect to the server and indicate that one of its other devices should be revoked access, or it could be handled automatically by the server (say, through the expiration of a 90-day session for the registered device). Now, if the user loses a device, triggering a server revocation would make the lost device's secrets obsolete to an adversary. Of course, if the adversary has fully compromised the server, then it cannot be trusted to perform the server-side revocation. However, in practical settings, it is often the case that a service provider is acting honestly in the present, but may be susceptible to a compromise in the future. Providing a meaningful notion of security in this setting is significantly more complicated.

1.1 Oblivious Revocable Functions (ORFs)

In this work, we focus on the use of server revocation as a means for providing security for multi-device encrypted indexing. In Section 2, we introduce an ideal functionality $\mathcal{F}_{\mathsf{ORF}}$ for this setting which captures the scenario in which a user can adaptively register its devices to a server, to jointly compute an encrypted index without revealing the plaintext to the server. The ideal functionality produces server outputs which emulate the outputs of a truly random function on the user's inputs, without needing to expose these inputs to the server. For the same input, the functionality's evaluation outputs are deterministically fixed for each user, and are independent of the device that initiated the function evaluation. Practically speaking, this ensures that the filename index mapping established by one of the user's devices will remain consistent across all of the other user's devices, without requiring the devices to communicate beyond the initial registration of the devices to the server.

The random outputs produced by the function evaluation allows for the private client inputs to take on low-entropy values (e.g. filenames), without adversely affecting the security of the indexing. Furthermore, the function evaluation must be initialized by a client device and completed by the server. This ensures that the server cannot preprocess the function evaluation outputs without interaction with a client device. Similarly, a client device cannot compute the function evaluation by itself without involving the server. As a result, an adversary is unable to predict future outputs of the function evaluation without accessing the device secrets and the server to which it has

registered.

Client input privacy. A natural notion of privacy for the client's evaluated inputs would ensure that nothing about the inputs are leaked to the adversary (in the ideal world, the simulator would not receive any information about the inputs from the ideal functionality). In our formalism, we slightly relax this notion to allow for the functionality to leak a minimal amount of information to the simulator: specifically, whether or not the input has been evaluated by the client before. As a result, this formalism may not be suitable for applications where same-input evaluations need to appear indistinguishable from different-input evaluations. Note however that this information will inevitably be leaked to the server if it is able to complete a server-side evaluation and learn the final function output, since this function output is already deterministic in the client's input.

Modeling device and server compromises. In general, properly capturing the nuances of the adversary's interactions with the clients and server is a non-trivial task. We allow for an adversary interacting with a protocol that realizes the ideal functionality to trigger compromises for each party that leaks all party secrets to the adversary. For instance, in the case where secure channels are used, this also implies that the secrets used to access the secure channel must be leaked—we model this by ensuring that the adversary also has access to any messages that the compromised party has ever received.

When a revocation is triggered for a target device by an honest server, the server-side secrets associated with that device are removed, and the corresponding device's secrets are no longer of value to an adversary. In order to capture this, we restrict the class of adversaries that we consider as distinguishers between a real execution of a protocol and the ideal functionality to be the set of adversaries that do not trigger an "active dual compromise" event; when a compromise for an actively-registered device occurs, along with a compromise for the server it registered to, *before it has been revoked*. Intuitively, when an adversary successfully triggers an active dual compromise for a device-server pair, all secrets used to perform function evaluation are revealed to the adversary, which hinders the ability to use any cryptographic primitives in order to continue indistinguishably simulating the ideal functionality. In Section 2.2, we elaborate on these restrictions, along with a more detailed explanation of these compromise events.

1.2 Realizing ORFs

We discuss several candidate constructions for $\mathcal{F}_{\mathsf{ORF}}$ with various tradeoffs. We conclude with a high-level description of the main construction that we present in Section 3.

Threshold OPRFs. Perhaps the most natural realization of an $\mathcal{F}_{\mathsf{ORF}}$ is through the use of a 2-out-of-2 threshold oblivious PRF [8, 9] (OPRF). In a t-out-of-n threshold OPRF, a client with a private input can jointly compute a PRF output among n servers, of which only t have to participate. If any $t-1$ servers collude, they cannot reconstruct the final PRF output. The following function, often called DH-OPRF, can be thresholdized:

$$F(k, x) = H(x, H(x)^k)$$

This function can be obliviously evaluated by having the client pick a blinding scalar r and sending $a \leftarrow H(x)^r$ to the server, which then returns $b \leftarrow a^k$ to the client, who then computes $H(x, b^{1/r})$ to recover the PRF output. By performing a secret sharing of the key k across the n servers, this can also be made to support a t-out-of-n scheme.

To map the threshold OPRF primitive into the ORF setting, the ORF client simultaneously plays the role of an OPRF client (holding the private input) and also as one of the two threshold OPRF servers, holding a keyshare. The ORF server is then the remaining threshold OPRF servers, and the function evaluation proceeds correspondingly. Device initialization is handled by each party picking one half of the key independently, and new device registration is handled by each party rerandomizing their keyshares and using secure channels to communicate the rerandomization factor in order to ensure consistency across devices.

However, the main issue with the above-described threshold OPRF construction of an ORF is that the client learns the output $F(k, x)$, whereas for ORFs, the output must be unpredictable to the client, even after a successful round of interaction for evaluating the function between the client and server.

Building off of this approach, an alternative construction which resolves the above issue works as follows. Let k_1 be the key held by a registered device, and k_2 the key held by the server it was registered to. Consider the following function:

$$F(k, x) = H(H(x)^{k_1 k_2})$$

The client begins the evaluation by picking a blinding scalar r_1 and sending $a \leftarrow H(x)^{k_1 r_1}$ to the server. The server picks another blinding scalar r_2 responds with $b \leftarrow a^{k_2 r_2}$ to the client. The client then sends $c \leftarrow b^{1/r_1}$ to the server, who then computes $H(c^{1/r_2})$ as the final function output.

This construction ensures that the function output is unpredictable to the client as required by $\mathcal{F}_{\mathsf{ORF}}$, and also has the extra property (similar to the threshold OPRF construction) that the messages exchanged between client and server fully hide the input. In most practical scenarios, this construction should be a sufficient solution to the multi-device encrypted indexing problem. However, our definition of an ORF allows for a weaker notion of hiding for the plaintexts, in that a deterministic handle can be leaked which allows for equality comparisons of the corresponding evaluated inputs.

Single-message construction. In the construction that we analyze in Section 3, we present a simpler evaluation of the above function which does allow for equality comparisons to be leaked for evaluated inputs, but with the benefit that the the evaluation only requires a single message to be sent from the client to the server. To evaluate, the client sends $a \leftarrow H(x)^{k_1}$ to the server, who then computes the function output as $H(a^{k_2})$. In Section 3.1, we provide a proof that this construction UC-realizes $\mathcal{F}_{\mathsf{ORF}}$ in the random oracle model under the Decision Diffie Hellman assumption.

1.3 Related Work

As mentioned above, a closely-related primitive to ORF is the notion of an oblivious PRF [6, 7]. In both primitives, a client holds a private input which it wishes to evaluate to produce a pseudorandom output with a server who holds a key. However, in the revocable setting, the user may control multiple devices, each with their own keys, and we want for the server to be able to revoke access for a client device while still allowing it to compute PRF evaluations for the remaining client devices. As a result, the overall setting (from a security perspective) is quite different.

Another related primitive is the Pythia PRF service [5], which also allows for clients to compute PRF outputs with a server while maintaining message privacy and using unique secret keys for distinct clients. Pythia relies on constructions of a partially-oblivious PRF, and consequently offers a more advanced suite of security considerations when compared to ORF.

Parties: A set of devices (each denoted by D) owned by a single user U and a server S.

Initialization:

- **Client Initialize:** On input $(\texttt{ClientInitialize}, \mathsf{uid}, \mathsf{did}, \mathsf{rid})$ from $\mathsf{D}_{\mathsf{did}}$, if the record $\langle \textsc{ClientInitialized}, \mathsf{uid}, \mathsf{did}, \mathsf{rid}\rangle$ has already been stored, then abort. Otherwise, store the record $\langle \textsc{ClientInitialized}, \mathsf{uid}, \mathsf{did}, \mathsf{rid}\rangle$, and send $(\texttt{ClientInitialize}, \mathsf{uid}, \mathsf{did}, \mathsf{rid})$ to $\mathcal{A}$.
- **Server Initialize:** On input $(\texttt{ServerInitialize}, \mathsf{uid}, \mathsf{did}, \mathsf{rid})$ from $\mathsf{S}_{\mathsf{rid}}$, if the record $\langle \textsc{Registered}, \mathsf{uid}, \mathsf{did}, \mathsf{rid}\rangle$ has already been stored, then abort. Otherwise, store the record $\langle \textsc{Registered}, \mathsf{uid}, \mathsf{did}, \mathsf{rid}\rangle$, and send $(\texttt{ServerInitialize}, \mathsf{uid}, \mathsf{did}, \mathsf{rid})$ to $\mathcal{A}$.

Registration:

- **Client Start Registration:** On input $(\texttt{ClientStartRegistration}, \mathsf{uid}, \mathsf{did}_1, \mathsf{did}_2, \mathsf{rid})$ from $\mathsf{D}_{\mathsf{did}_1}$, if the record $\langle \textsc{ClientInitialized}, \mathsf{uid}, \mathsf{did}_1, \mathsf{rid}\rangle$ has not been stored, then abort. Otherwise, store the record $\langle \textsc{ClientRegistrationStarted}, \mathsf{uid}, \mathsf{did}_1, \mathsf{did}_2, \mathsf{rid}\rangle$, and send $(\texttt{ClientStartRegistration}, \mathsf{uid}, \mathsf{did}_1, \mathsf{did}_2, \mathsf{rid})$ to $\mathcal{A}$.
- **Client Finish Registration:** On input $(\texttt{ClientFinishRegistration}, \mathsf{uid}, \mathsf{did}_1, \mathsf{did}_2, \mathsf{rid})$ from $\mathsf{D}_{\mathsf{did}_2}$, if the record $\langle \textsc{ClientInitialized}, \mathsf{uid}, \mathsf{did}_2, \mathsf{rid}\rangle$ has already been stored, then abort. If the record $\langle \textsc{ClientRegistrationStarted}, \mathsf{uid}, \mathsf{did}_1, \mathsf{did}_2, \mathsf{rid}\rangle$ has not been stored, then abort. Otherwise, store the record $\langle \textsc{ClientInitialized}, \mathsf{uid}, \mathsf{did}_2, \mathsf{rid}\rangle$ and send $(\texttt{ClientFinishRegistration}, \mathsf{uid}, \mathsf{did}_1, \mathsf{did}_2, \mathsf{rid})$ to $\mathcal{A}$.
- **Server Accept Registration:** On input $(\texttt{ServerAcceptRegistration}, \mathsf{uid}, \mathsf{did}_1, \mathsf{did}_2, \mathsf{rid})$ from $\mathsf{S}_{\mathsf{rid}}$, if the record $\langle \textsc{Registered}, \mathsf{uid}, \mathsf{did}_2, \mathsf{rid}\rangle$ has already been stored, then abort. If the record $\langle \textsc{ClientRegistrationStarted}, \mathsf{uid}, \mathsf{did}_1, \mathsf{did}_2, \mathsf{rid}\rangle$ or the record $\langle \textsc{Registered}, \mathsf{uid}, \mathsf{did}_1, \mathsf{rid}\rangle$ has not been stored, then abort. Otherwise, store the record $\langle \textsc{Registered}, \mathsf{uid}, \mathsf{did}_2, \mathsf{rid}\rangle$, and send $(\texttt{ServerAcceptRegistration}, \mathsf{uid}, \mathsf{did}_1, \mathsf{did}_2, \mathsf{rid})$ to $\mathcal{A}$.

Revocation:

- **Server Revoke:** On input $(\texttt{Revoke}, \mathsf{uid}, \mathsf{did}, \mathsf{rid})$ from $\mathsf{S}_{\mathsf{rid}}$, delete the record $\langle \textsc{Registered}, \mathsf{uid}, \mathsf{did}, \mathsf{rid}\rangle$ (aborting if it cannot be found), and send $(\texttt{Revoke}, \mathsf{uid}, \mathsf{did}, \mathsf{rid})$ to $\mathcal{A}$.

Figure 1: The ORF ideal functionality $\mathcal{F}_{\mathsf{ORF}}$.

2 The ORF Ideal Functionality

We formulate our definition of the ORF ideal functionality under the universal composability (UC) framework [3], which consists of an environment that attempts to distinguish between the "real world" and an "ideal world". Additionally, we frame our security notions within the (programmable) random oracle model. The ideal functionality is presented in Figures 1 and 2. In the following, we establish the notation and conventions we use in our formalism, and then provide an intuitive description of the interfaces supported by this functionality.

Conventions. We use $\mathcal{A}$ to represent an adversary, U to represent a user controlling a set of devices $\mathsf{D}_1, \ldots, \mathsf{D}_n$, and S for a server. We use angle brackets $\langle \cdots \rangle$ to denote *records* that are kept as state in between operations. Records can be stored or deleted. The first parameter of a record is a label that denotes the type of the record. We use $\star$ to denote a wildcard for use in conditional

Evaluation:

- **Client Evaluate:** On $(\texttt{ClientEvaluate}, \mathsf{sid}, \mathsf{uid}, \mathsf{did}, \mathsf{rid}, x)$ from $\mathsf{D}_{\mathsf{did}}$, if the record $\langle \textsc{Registered}, \mathsf{uid}, \mathsf{did}, \mathsf{rid} \rangle$ has not been stored, then abort. Otherwise, pick a unique deterministic handle $\mu \in \{0,1\}^\star$ based on the tuple $(\mathsf{uid}, \mathsf{rid}, x)$, store the record $\langle \textsc{ClientEvaluation}, \mathsf{sid}, \mathsf{uid}, \mathsf{did}, \mathsf{rid}, x \rangle$, and send $(\texttt{ClientEvaluate}, \mathsf{sid}, \mathsf{uid}, \mathsf{did}, \mathsf{rid}, \mu)$ to $\mathcal{A}$.
- **Server Evaluate:** On $(\texttt{ServerEvaluate}, \mathsf{sid}, \mathsf{uid}, \mathsf{did}, \mathsf{rid})$ from $\mathsf{S}_{\mathsf{rid}}$, if the record $\langle \textsc{Registered}, \mathsf{uid}, \mathsf{did}, \mathsf{rid} \rangle$ does not exist, or if a record of the form $\langle \textsc{ClientEvaluation}, \mathsf{sid}, \mathsf{uid}, \mathsf{did}, \mathsf{rid}, \star \rangle$ does not exist, then abort. Otherwise, set x as the last parameter for the ClientEvaluation record. If there is a record of the form $\langle \textsc{EvaluationInput}, \mathsf{uid}, \mathsf{rid}, x, \star \rangle$, then set z as the last parameter of that record. Otherwise, sample $z \leftarrow_{\mathsf{R}} \mathcal{O}$ (uniformly at random from the output space) and store the record $\langle \textsc{EvaluationInput}, \mathsf{uid}, \mathsf{rid}, x, z \rangle$. Output z to the environment, and send $(\texttt{ServerEvaluate}, \mathsf{sid}, \mathsf{uid}, \mathsf{did}, \mathsf{rid})$ to $\mathcal{A}$.

Compromise:

- **Server Compromise:** On $(\texttt{CompromiseServer}, \mathsf{rid})$ from $\mathcal{A}$, let V be the set of all tuples (uid, z) for which there exists a stored record of the form $\langle \textsc{EvaluationInput}, \mathsf{uid}, \mathsf{rid}, \star, z \rangle$. Send $(\texttt{CompromiseServer}, \mathsf{rid}, V)$ to $\mathcal{A}$.

Figure 2: The ORF ideal functionality $\mathcal{F}_{\mathsf{ORF}}$ (continued).

statements over stored records. Following the convention of Canetti [3], we assume without loss of generality that $\mathcal{A}$ is a dummy adversary that merely passes all of its messages and computations to the environment $\mathcal{E}$. Therefore, we formulate the real world and ideal world in the context of an environment which triggers all parties and receives all inputs and computations.

Handling concurrent and duplicate invocations. Each of the interfaces of $\mathcal{F}_{\mathsf{ORF}}$ read from (and some modify) a shared storage of records. In our presentation, we assume that each of these interfaces are invoked *sequentially*, so as to avoid issues arising from race conditions that can occur from concurrent reads from the shared state of records. This assumption can be lifted by simply instantiating read-write locks that ensure proper synchronization of the shared records. However, in the interest of an ease of presentation, we omit mentioning these locks in Figure 1. In a similar vein, we also assume that each of these interfaces can only be called at most once on any given set of arguments, so as to avoid double-setting internal state. This assumption can also be lifted by keeping track of and rejecting duplicate invocations of the functions on an identical set of arguments, which we again omit for ease of presentation.

Identifiers. There are four types of identifiers that we use in our model: device identifiers (did), user identifiers (uid), server identifiers (rid), and session identifiers (sid). A device identifier did is a label that uniquely identifies each of a user's devices. These devices may be adaptively registered to and revoked from the user's device list, and maintain the property that the evaluations between any two devices on the same user's device list with a server remain consistent. A user identifier uid identifies a user with the set of devices registered to it. A server identifier rid identifies the server that is associated with a set of registered devices and users. All evaluations for an input x across this user's devices produce the same output for the same server.

A session identifier sid is a label that uniquely identifies an invocation of `ClientEvaluate`

and `ServerEvaluate`. Executions of the protocol across different session identifiers are completely independent of one another. As a result, our UC formulation of the security of ORF is inherently parameterized by the session identifier, which guarantees that concurrent instantiations of these user-server sessions do not interfere with one another with respect to security. In the common setting in which there only exists a single server, the session identifier is equivalent to a "user identifier".

2.1 Intuition for the Ideal Functionality

The ideal functionality $\mathcal{F}_{\mathsf{ORF}}$ consists of users controlling a set of devices that interact with servers. A user can perform the following operations: initialize a device with a server (through `ClientInitialize`), register a new device to a server from an already-registered device (through `ClientStartRegistration` and `ClientFinishRegistration`), or initiate an evaluation between a device and a server by calling `ClientEvaluate` on an input x. A server can perform the following operations: accept a device initialization (through `ServerInitialize`), accept a device registration (through `ServerAcceptRegistration`), revoke a previously-registered device by calling `Revoke`, or participate in an evaluation with a device by calling `ServerEvaluate`.

Initialization and registration. Initialization begins with a client and ends with the server accepting the initialization by storing a REGISTERED record for the device $\mathsf{D}_{\mathsf{did}}$, associating it with the user uid and server rid. Registration is used for when a client wishes to register a new device after an initial device has already been registered. This involves a three-way handshake between a client's old device, a client's new device, and the server (which accepts the registration request). After a successful registration, the server stores a REGISTERED record for the new device, similarly to the end of a successful initialization of the first device. These functions keep track of records to ensure that for both the initialization and registration functions, the adversary $\mathcal{A}$ receives the same input that was used to invoke these functions.

Revocation. For server revocation, the ideal functionality deletes the corresponding registration record as indicated by the requesting server. An indication of the revocation is also forwarded to the adversary.

Evaluation. For client evaluation, the functionality first checks that the requesting device has been registered to a user and server. If so, it associates a unique handle μ for the tuple $(\mathsf{uid}, \mathsf{rid}, x)$, meaning that on subsequent calls to `ClientEvaluate`, if the same uid, rid, and x are supplied, then the same handle μ is picked. A CLIENTEVALUATION record holding x is stored, and this handle is sent *in place of* the input x to the adversary. The intention here is to convey that the adversary does not get access to x, but instead only receives a handle for x, which allows it to compare against other received handles to test for equality of the corresponding inputs x (evaluated from the same uid and rid). For server evaluation, the functionality checks for the appropriate registration record, and also ensures that a CLIENTEVALUATION record exists. Then, it takes the input x supplied to `ClientEvaluate` and evaluates a consistent random function, using z as the output to the environment. Note that z is not sent to the adversary in this step.

Server compromise. Finally, when responding to server compromises triggered from the adversary, the ideal functionality gathers all z outputs produced through `ServerEvaluate` that corresponding with the requested rid, and sends them to the adversary.

2.2 Handling Compromises

In the real world, the adversary is allowed to (adaptively) compromise a subset of the parties participating in the protocol. A "compromise" is a weaker form of the more general notion of party corruption, in which the adversary obtains access to all long-term secrets stored and messages received by the compromised party, but is not able to maliciously control the party's operations. Here, we can think of the real-world representation of a compromise as the scenario in which an attacker has passively gained access to a snapshot of party's secrets and message storage at a certain point in time. Since a compromise represents the leakage of a snapshot of a party's secrets, note that a party can be compromised multiple times by an adversary, with the leakage potentially being different across compromises, based on the party's interactions with other parties in between compromises.

There are two types of compromise events which we model: device compromise and server compromise. We also define the notion of "active" device and server compromises.

Device compromise. When an adversary chooses to compromise a device, all long-term values that must be persisted on the device in order to complete an execution of client evaluation with a server are revealed to the adversary. Additionally, any values that were transmitted to the device from another party must also be leaked to the adversary. In the ideal world, device compromise events are not captured in the ideal functionality, which means that any long-term secret values and messages sent through channels must be produced by a simulator without any additional leakage from $\mathcal{F}_{\mathsf{ORF}}$.

Definition 2.1 (Active Device Compromise)**.** A $(\mathsf{did}, \mathsf{rid})$-*active* device compromise describes the situation in which an adversary $\mathcal{A}$ triggers a device compromise for device did at any point after either $(\texttt{ClientInitialize}, \star, \mathsf{did}, \mathsf{rid})$ or $(\texttt{ClientFinishRegistration}, \star, \star, \mathsf{did}, \mathsf{rid})$ have been invoked by the environment.

Server compromise. When the adversary chooses to compromise a server, all long-term values that must be persisted on the server in order to complete an execution of server evaluation with a device are revealed to the adversary. Additionally, any values that were transmitted to the server from another party must also be leaked to the adversary. In the ideal world, the adversary invokes `CompromiseServer` during a server compromise event for a given rid, and the ideal functionality reveals all evaluation inputs associated with rid that were computed and recorded through previous calls to `ServerEvaluate`.

Definition 2.2 (Active Server Compromise)**.** A $(\mathsf{did}, \mathsf{rid})$-*active* server compromise describes the situation in which an adversary $\mathcal{A}$ triggers a server compromise for server rid at any point after either $(\texttt{ServerInitialize}, \star, \mathsf{did}, \mathsf{rid})$ or $(\texttt{ServerAcceptRegistration}, \star, \star, \mathsf{did}, \mathsf{rid})$ have been invoked by the environment, but *before* $(\texttt{Revoke}, \star, \mathsf{did}, \mathsf{rid})$ has been invoked by the environment.

Restricting active dual compromises. In this work, we specifically restrict our consideration to a class of adversaries that do not execute "active dual compromises"—that is, adversaries that do not simultaneously compromise a device and a server when the device has an active registration (not revoked) with the server.

Definition 2.3 (Active Dual Compromise)**.** We say that an adversary $\mathcal{A}$ has triggered a $(\mathsf{did}, \mathsf{rid})$-active *dual compromise* if $\mathcal{A}$ triggers both a $(\mathsf{did}, \mathsf{rid})$-active device compromise and a $(\mathsf{did}, \mathsf{rid})$-active server compromise.

Definition 2.4 ($\mathcal{F}_{\mathsf{ORF}}$-Admissibility)**.** We say that an adversary is $\mathcal{F}_{\mathsf{ORF}}$-*admissible* if it never triggers a dual compromise for any device and server pair. That is, there is no such pair of $(\mathsf{did}, \mathsf{rid})$ for which the adversary has triggered a $(\mathsf{did}, \mathsf{rid})$-active dual compromise.

Notably, an $\mathcal{F}_{\mathsf{ORF}}$-admissible adversary excludes the following cases:

- Adversaries which compromise a device and server *after* the device has been registered to the server, but then subsequently revoked by the server, do not constitute as having performed an active dual compromise (since the device compromise is not considered to be an "active" compromise).
- Similarly, adversaries which compromise a device after it has been registered and before it has been revoked, but compromise the server *before* the device was registered to it, also do not constitute as having performed an active dual compromise (since the device is not considered to be "active" when the server was compromised).

Note that there is a distinction to be made between how the ideal functionality responds during a device and server compromise, compared with the information that the adversary receives when triggering a device and server compromise in the real protocol. The ideal functionality does not deal with device compromises in any way, and only leaks the evaluation outputs during the server compromise step. However, in the real world, all device secrets are sent to the adversary for device compromises, and all server secrets are sent to the adversary for server compromises.

3 Construction

We present a construction Π_{DH} which describes a single-message protocol for client and server evaluation (one message sent from the client to the server). The security of Π_{DH} is based on the Decisional Diffie-Hellman (DDH) assumption [1] in a prime-order group, along with the instantiation of a random oracle. The protocol Π_{DH} is described in Figures 3 and 4.

In this construction, each party (device or server) communicates with another party through the use of a pre-established secure channel, which provides confidentiality and authentication for its messages. When an adversary compromises a party (device or server), it gains access to the plaintexts of all messages that the party has received so far, along with any internal state (the secret values needed to perform client and server evaluation) kept by the party.

Building blocks. In this construction, we make use of the following three building blocks: secure channels ($\mathcal{F}_{\mathsf{SC}}$), a global programmable random oracle ($\mathcal{G}_{\mathsf{pRO}}$), and a group $\mathbb{G}$ for which the DDH assumption holds.

- We use the notion of a strong UC-secure channel defined in [4]. We use the notation $\mathcal{F}_{\mathsf{SC}}$ as the ideal functionality for a secure channel, with interfaces $\mathcal{F}_{\mathsf{SC}}$.`EstablishSession`, $\mathcal{F}_{\mathsf{SC}}$.`Send`, $\mathcal{F}_{\mathsf{SC}}$.`Receive`, and $\mathcal{F}_{\mathsf{SC}}$.`ExpireSession` (see Section 5 and Figure 11 of [4]).
- We prove security in the (global) programmable random oracle model [2]. We borrow their notation, $\mathcal{G}_{\mathsf{pRO}}$, which describes the ideal functionality of a global programmable random oracle with interfaces $\mathcal{G}_{\mathsf{pRO}}$.`HashQuery` and $\mathcal{G}_{\mathsf{pRO}}$.`ProgramRO` (see Section 4 of [2]).

- We make use of a group $\mathbb{G}$ of prime order p, where the Decision Diffie-Hellman problem is assumed to be intractable. That is, given a generator $g \in \mathbb{G}$, the tuple (g, g^a, g^b, g^{ab}) is indistinguishable to a computationally bounded adversary from (g, g^a, g^b, g^c), for uniformly sampled scalars $a, b, c \leftarrow_{\mathsf{R}} [p]$.

Construction intuition. In Π_{DH}, a registered client device always has a some key k_{D}, and a server has a key k_{S} corresponding to each device that has been registered with it. We maintain the invariant (through the definitions of the initialization and registration) that for any user uid, the corresponding keys k_{D} and k_{S} are such that $k_{\mathsf{D}} \cdot k_{\mathsf{S}}$ is the same across all devices owned by that user. This is due to the fact that during registration, an old device picks a random scalar r, and sends the new device the key $k_{\mathsf{D}} \cdot r^{-1}$, so that the invariant still holds for the new device-server pair. All message exchanges occur through secure channels so that they are not leaked to the adversary.

Then, when executing client and server evaluations, the client device on an input x computes $p \leftarrow H(x)^{k_{\mathsf{D}}}$ and sends p to the server through a secure channel. The server, upon receiving this p value, computes $H(p^{k_{\mathsf{S}}})$ and uses this as the function output sent to the environment. Note that, by the above-mentioned invariant, this value is consistent across all devices owned by the user that were registered to the same server.

3.1 Proof of Security

We provide a proof of the following theorem:

Theorem 1. The construction Π_{DH} GUC-realizes $\mathcal{F}_{\mathsf{ORF}}$, under the assumption that DDH holds for $\mathbb{G}$, and in the $(\mathcal{F}_{\mathsf{SC}}, \mathcal{G}_{\mathsf{pRO}})$-hybrid model.

We describe a simulator Sim in Figure 5 that interacts with the ideal functionality $\mathcal{F}_{\mathsf{ORF}}$. We define the following series of games which involve Sim attempting to simulate the environment's interactions with the honest parties.

- **Game 0:** Game_0 is identical to the environment's interactions with the real world execution of Π_{DH}.
- **Game 1:** Game_1 differs from Game_0 in that, instead of computing the abort conditions for each function by checking the existence of entries in $\mathtt{St}_{\mathsf{D}}$, $\mathtt{St}_{\mathsf{S}}$, and whether or not a message was received by $\mathcal{F}_{\mathsf{SC}}$, Game_1 instead keeps track of records ClientInitialized, ClientRegistrationStarted, Registered, and ClientEvaluation to compute the corresponding abort conditions, identically to how they are computed in $\mathcal{F}_{\mathsf{ORF}}$.
- **Game 2:** Game_2 differs from Game_1 in that, instead of using the secure channel functionality $\mathcal{F}_{\mathsf{SC}}$ to send messages to parties in `ClientStartRegistration` and `ClientEvaluate` (with their receiving counterparts in `ClientFinishRegistration`, `ServerAcceptRegistration`, and `ServerEvaluate`), the simulator simulates the transmission of these messages and outputs the length $\ell = \lceil \log_2(q) \rceil$ to $\mathcal{A}$, storing the plaintexts in $\mathtt{St}_{\mathtt{N}}$, $\mathtt{St}_{\mathsf{R}}$, and $\mathtt{St}_{\mathtt{P}}$ correspondingly. Also, instead of returning the plaintexts of all messages received through secure channels in `CompromiseDevice` and `CompromiseServer`, Sim additionally outputs to $\mathcal{A}$ the values:
  - $\mathtt{St}_{\mathtt{N}}[\star, \mathsf{did}, \star]$ for `CompromiseDevice`, and
  - $\mathtt{St}_{\mathsf{R}}[\star, \star, \mathsf{rid}]$ and $\mathtt{St}_{\mathtt{P}}[\star, \mathsf{uid}, \mathsf{did}, \mathsf{rid}]$ for `CompromiseServer`.

Parameters: For the security parameter λ, a prime-order group $\mathbb{G}$ with $|\mathbb{G}| = q$, and a hash function $H : \{0,1\}^\star \to \mathbb{G}$, the protocol proceeds as follows.

Initialization:

- **Client Initialize:** On input $(\texttt{ClientInitialize}, \mathsf{uid}, \mathsf{did}, \mathsf{rid})$, device $\mathsf{D}_{\mathsf{did}}$ checks if there is an existing entry for $\texttt{St}_\mathsf{D}[\mathsf{uid}, \mathsf{did}, \mathsf{rid}]$, aborting if so. Otherwise, it samples and stores a scalar $\texttt{St}_\mathsf{D}[\mathsf{uid}, \mathsf{did}, \mathsf{rid}] \leftarrow_\mathsf{R} [q]$.
- **Server Initialize:** On input $(\texttt{ServerInitialize}, \mathsf{uid}, \mathsf{did}, \mathsf{rid})$, server $\mathsf{S}_{\mathsf{rid}}$ checks if there is an existing entry for $\texttt{St}_\mathsf{S}[\mathsf{uid}, \mathsf{did}, \mathsf{rid}]$, aborting if so. Otherwise, it samples and stores a scalar $\texttt{St}_\mathsf{S}[\mathsf{uid}, \mathsf{did}, \mathsf{rid}] \leftarrow_\mathsf{R} [q]$.

Registration:

- **Client Start Registration:** On input $(\texttt{ClientStartRegistration}, \mathsf{uid}, \mathsf{did}_1, \mathsf{did}_2, \mathsf{rid})$, device $\mathsf{D}_{\mathsf{did}_1}$ retrieves $k \leftarrow \texttt{St}_\mathsf{D}[\mathsf{uid}, \mathsf{did}_1, \mathsf{rid}]$ (aborting if this entry cannot be found), and samples a scalar $r \leftarrow_\mathsf{R} [q]$, computes $k' \leftarrow k \cdot r$. Then, it invokes $\mathcal{F}_\mathsf{SC}.\texttt{EstablishSession}(\mathsf{ssid}_1, \mathsf{D}_{\mathsf{did}_2}, \texttt{initiator})$ and $\mathcal{F}_\mathsf{SC}.\texttt{Send}(\mathsf{ssid}_1, k')$, along with $\mathcal{F}_\mathsf{SC}.\texttt{EstablishSession}(\mathsf{ssid}_2, \mathsf{S}_{\mathsf{rid}}, \texttt{initiator})$ and $\mathcal{F}_\mathsf{SC}.\texttt{Send}(\mathsf{ssid}_2, r)$.
- **Client Finish Registration:** On input $(\texttt{ClientFinishRegistration}, \mathsf{uid}, \mathsf{did}_1, \mathsf{did}_2, \mathsf{rid})$, device $\mathsf{D}_{\mathsf{did}_2}$ checks if there is an existing entry for $\texttt{St}_\mathsf{D}[\mathsf{uid}, \mathsf{did}_2, \mathsf{rid}]$ (aborting if so). Then, it invokes $\mathcal{F}_\mathsf{SC}.\texttt{EstablishSession}(\mathsf{ssid}_1, \mathsf{D}_{\mathsf{did}_1}, \texttt{responder})$, waits to receive $(\texttt{Receive}, \mathsf{ssid}_1, k')$ from $\mathcal{F}_\mathsf{SC}$ (aborting if never received), and invokes $\mathcal{F}_\mathsf{SC}.\texttt{ExpireSession}(\mathsf{ssid}_1)$. Finally, it stores $\texttt{St}_\mathsf{D}[\mathsf{uid}, \mathsf{did}_2, \mathsf{rid}] \leftarrow k'$.
- **Server Accept Registration:** On input $(\texttt{ServerAcceptRegistration}, \mathsf{uid}, \mathsf{did}_1, \mathsf{did}_2, \mathsf{rid})$, server $\mathsf{S}_{\mathsf{rid}}$ checks if there is an existing entry for $\texttt{St}_\mathsf{S}[\mathsf{uid}, \mathsf{did}_2, \mathsf{rid}]$ (aborting if so), and retrieves $k \leftarrow \texttt{St}_\mathsf{S}[\mathsf{uid}, \mathsf{did}_1, \mathsf{rid}]$ (aborting if this entry cannot be found). Then, it invokes $\mathcal{F}_\mathsf{SC}.\texttt{EstablishSession}(\mathsf{ssid}_2, \mathsf{D}_{\mathsf{did}_1}, \texttt{responder})$, waits to receive $(\texttt{Receive}, \mathsf{ssid}_2, r)$ from $\mathcal{F}_\mathsf{SC}$ (aborting if never received), and invokes $\mathcal{F}_\mathsf{SC}.\texttt{ExpireSession}(\mathsf{ssid}_2)$. Finally, it sets $k' \leftarrow k \cdot r^{-1}$, and stores $\texttt{St}_\mathsf{S}[\mathsf{uid}, \mathsf{did}_2, \mathsf{rid}] \leftarrow k'$.

Revocation:

- **Server Revoke:** On input $(\texttt{Revoke}, \mathsf{uid}, \mathsf{did}, \mathsf{rid})$, $\mathsf{S}_{\mathsf{rid}}$ deletes the entry $\texttt{St}_\mathsf{S}[\mathsf{uid}, \mathsf{did}, \mathsf{rid}]$ (aborting if it cannot be found), executes $\mathcal{F}_\mathsf{SC}.\texttt{ExpireSession}(\mathsf{ssid}_2)$ and $\mathcal{F}_\mathsf{SC}.\texttt{ExpireSession}(\mathsf{ssid}_3)$, and deletes the plaintexts for any messages sent to $\mathsf{S}_{\mathsf{rid}}$.

Evaluation:

- **Client Evaluate:** On input $(\texttt{ClientEvaluate}, \mathsf{sid}, \mathsf{uid}, \mathsf{did}, \mathsf{rid}, x)$, $\mathsf{D}_{\mathsf{did}}$ retrieves the scalar $k \leftarrow \texttt{St}_\mathsf{D}[\mathsf{uid}, \mathsf{did}, \mathsf{rid}]$ (aborting if it cannot be found), sets $h \leftarrow \mathcal{G}_{\mathsf{pRO}}.\texttt{HashQuery}(\text{“}\texttt{client}\text{”} \,||\, x \,||\, \mathsf{uid} \,||\, \mathsf{rid})$, and sets $p \leftarrow h^k$. Then, it invokes $\mathcal{F}_\mathsf{SC}.\texttt{EstablishSession}(\mathsf{ssid}_3, \mathsf{S}_{\mathsf{rid}}, \texttt{initiator})$ and $\mathcal{F}_\mathsf{SC}.\texttt{Send}(\mathsf{ssid}_3, p)$.
- **Server Evaluate:** On input $(\texttt{ServerEvaluate}, \mathsf{sid}, \mathsf{uid}, \mathsf{did}, \mathsf{rid})$, $\mathsf{S}_{\mathsf{rid}}$ first retrieves the scalar $k \leftarrow \texttt{St}_\mathsf{S}[\mathsf{uid}, \mathsf{did}, \mathsf{rid}]$ (aborting if this entry cannot be found). Then, it invokes $\mathcal{F}_\mathsf{SC}.\texttt{EstablishSession}(\mathsf{ssid}_3, \mathsf{D}_{\mathsf{did}}, \texttt{responder})$, waits to receive $(\texttt{Receive}, \mathsf{ssid}_3, p)$ (aborting if never received), and invokes $\mathcal{F}_\mathsf{SC}.\texttt{ExpireSession}(\mathsf{ssid}_3)$. Finally, it computes $z \leftarrow \mathcal{G}_{\mathsf{pRO}}.\texttt{HashQuery}(\text{“}\texttt{server}\text{”} \,||\, p^k \,||\, \mathsf{uid} \,||\, \mathsf{rid})$, and outputs z to the environment.

Figure 3: The protocol Π_{DH}.

Compromise:

- **Device Compromise:** On $(\texttt{CompromiseDevice}, \mathsf{uid}, \mathsf{did})$ from $\mathcal{A}$, return to $\mathcal{A}$ all entries of the form $\mathtt{St}_{\mathsf{D}}[\mathsf{uid}, \mathsf{did}, \star]$ and the plaintexts of all messages received through secure channels with $\mathsf{D}_{\mathsf{did}}$ as the recipient.
- **Server Compromise:** On $(\texttt{CompromiseServer}, \mathsf{rid})$ from $\mathcal{A}$, return to $\mathcal{A}$ all entries of the form $\mathtt{St}_{\mathsf{S}}[\star, \star, \mathsf{rid}]$ and the plaintexts of all messages received through secure channels with $\mathsf{S}_{\mathsf{rid}}$ as the recipient.

Figure 4: The protocol Π_{DH} (continued).

- **Game 3:** Game_3 differs from Game_2 only in `ClientEvaluate`: instead of computing $h \leftarrow \mathcal{G}_{\mathsf{pRO}}.\texttt{HashQuery}(\text{“}\texttt{client}\text{”} \,||\, x \,||\, \mathsf{uid} \,||\, \mathsf{rid})$ and $p \leftarrow h^k$, it checks if there is a p_μ value previously associated with the received μ. If so, it retrieves the did' associated with μ (when it was first encountered), sets $k^\star \leftarrow \mathtt{St}_{\mathtt{N}}[\mathsf{uid}, \star, \mathsf{did}, \mathsf{rid}]$, retrieves $k' \leftarrow \mathtt{St}_{\mathsf{D}}[\mathsf{uid}, \mathsf{did}', \mathsf{rid}]$, and sets $r^\star \leftarrow k^\star / k'$, and sets $p^\star_\mu \leftarrow p_\mu \cdot r^\star$. If not, it samples a $p^\star_\mu \leftarrow_{\mathsf{R}} \mathbb{G}$ and stores the association between μ and $p^\star_\mu$.
- **Game 4:** Game_3 differs from Game_3 in `ServerEvaluate` and `CompromiseServer`:
  - In `ServerEvaluate`, instead of computing $z \leftarrow \mathcal{G}_{\mathsf{pRO}}.\texttt{HashQuery}(\text{“}\texttt{server}\text{”} \,||\, p^k \,||\, \mathsf{uid} \,||\, \mathsf{rid})$, it relies on the ideal functionality $\mathcal{F}_{\mathsf{ORF}}$ to check for an existing record of the form $\langle \textsc{ClientEvaluation}, \mathsf{sid}, \mathsf{did}, \mathsf{uid}, \mathsf{rid}, \star \rangle$, aborting if it does not exist, and setting x as the last argument if it does exist. Then, it checks for an existing record of the form $\langle \textsc{EvaluationInput}, \mathsf{uid}, \mathsf{rid}, x, \star \rangle$, setting z as the last parameter if it exists, and uniformly sampling $z \leftarrow_{\mathsf{R}} \mathcal{O}$ otherwise, storing $\langle \textsc{EvaluationInput}, \mathsf{uid}, \mathsf{rid}, x, z \rangle$. Then, z is output to the environment, and $\mathtt{Prog}[\mathsf{uid} \,||\, \mathsf{rid}] \leftarrow p^k$ is set.
  - In `CompromiseServer`, it first computes the set of all tuples (uid, z) for which there exists a stored record of the form $\langle \textsc{EvaluationInput}, \mathsf{uid}, \mathsf{rid}, \star, z \rangle$, sets $y \leftarrow \mathtt{Prog}[\mathsf{uid}, \mathsf{rid}]$, and executes $\mathcal{G}_{\mathsf{pRO}}.\texttt{ProgramRO}(\text{“}\texttt{server}\text{”} \,||\, y \,||\, \mathsf{uid} \,||\, \mathsf{rid}, z)$, aborting if unsuccessful.

Note that Game_4 is identical to the interactions between the environment triggering the simulator and the ideal functionality in the ideal world.

In the following series of lemmas, let $\mathcal{E}$ be the environment, and let $\mathsf{Game}_i(\mathcal{E})$ for $i \in [0, 4]$ to represent the environment interacting with Game_i, producing a bit output after its interactions are complete. We write $\Pr[\mathsf{Game}_i(\mathcal{E}) = 1]$ to represent the probability that the environment outputs 1 after interacting with Game_i.

Lemma 1. The quantity $|\Pr[\mathsf{Game}_0(\mathcal{E}) = 1] - \Pr[\mathsf{Game}_1(\mathcal{E}) = 1]|$ is 0.

Proof. Since the only difference between Game_0 and Game_1 is in how the abort conditions for each of the functions are computed, we enumerate them below and show (by inspection) that the triggering of each condition remains the same in between these games:

- For $(\texttt{ClientInitialize}, \mathsf{uid}, \mathsf{did}, \mathsf{rid})$, the abort condition in Game_0 is based on if there exists an entry for $\mathtt{St}_{\mathsf{D}}[\mathsf{uid}, \mathsf{did}, \mathsf{rid}]$, whereas the abort condition in Game_1 is based on if the record

Parameters: For the security parameter λ, a prime-order group $\mathbb{G}$ with $|\mathbb{G}| = q$, the simulator Sim proceeds as follows.

Initialization:

- **Client Initialize:** On input $(\texttt{ClientInitialize}, \mathsf{uid}, \mathsf{did}, \mathsf{rid})$, Sim samples and stores a scalar $\texttt{St}_\mathsf{D}[\mathsf{uid}, \mathsf{did}, \mathsf{rid}] \leftarrow_\mathsf{R} [q]$.
- **Server Initialize:** On input $(\texttt{ServerInitialize}, \mathsf{uid}, \mathsf{did}, \mathsf{rid})$, Sim samples and stores a scalar $\texttt{St}_\mathsf{S}[\mathsf{uid}, \mathsf{did}, \mathsf{rid}] \leftarrow_\mathsf{R} [q]$.

Registration:

- **Client Start Registration:** On input $(\texttt{ClientStartRegistration}, \mathsf{uid}, \mathsf{did}_1, \mathsf{did}_2, \mathsf{rid})$, Sim retrieves $k \leftarrow \texttt{St}_\mathsf{D}[\mathsf{uid}, \mathsf{did}_1, \mathsf{rid}]$, samples a scalar $r \leftarrow_\mathsf{R} [q]$, computes $k' \leftarrow k \cdot r$, and stores $\texttt{St}_\mathtt{N}[\mathsf{uid}, \mathsf{did}_1, \mathsf{did}_2, \mathsf{rid}] \leftarrow k'$ and $\texttt{St}_\mathsf{R}[\mathsf{uid}, \mathsf{did}_1, \mathsf{did}_2, \mathsf{rid}] \leftarrow r$. For $\ell = \lceil \log_2(q) \rceil$, Sim simulates the transmission of a message of length ℓ for $\mathcal{F}_\mathsf{SC}$ while outputting $(\mathsf{ssid}_1, \mathsf{D}_{\mathsf{did}_1}, \ell)$ to $\mathcal{A}$, along with the transmission of another message of length ℓ for $\mathcal{F}_\mathsf{SC}$ while outputting $(\mathsf{ssid}_2, \mathsf{S}_\mathsf{rid}, \ell)$ to $\mathcal{A}$.
- **Client Finish Registration:** On input $(\texttt{ClientFinishRegistration}, \mathsf{uid}, \mathsf{did}_1, \mathsf{did}_2, \mathsf{rid})$, Sim retrieves $k' \leftarrow \texttt{St}_\mathtt{N}[\mathsf{uid}, \mathsf{did}_1, \mathsf{did}_2, \mathsf{rid}]$, and sets $\texttt{St}_\mathsf{D}[\mathsf{uid}, \mathsf{did}_2, \mathsf{rid}] \leftarrow k'$.
- **Server Accept Registration:** On input $(\texttt{ServerAcceptRegistration}, \mathsf{uid}, \mathsf{did}_1, \mathsf{did}_2, \mathsf{rid})$, Sim retrieves $k \leftarrow \texttt{St}_\mathsf{S}[\mathsf{uid}, \mathsf{did}_1, \mathsf{rid}]$, retrieves $r \leftarrow \texttt{St}_\mathsf{R}[\mathsf{uid}, \mathsf{did}_1, \mathsf{did}_2, \mathsf{rid}]$, and stores $\texttt{St}_\mathsf{S}[\mathsf{uid}, \mathsf{did}_2, \mathsf{rid}] \leftarrow k \cdot r^{-1}$.

Revocation:

- **Server Revoke:** On input $(\texttt{Revoke}, \mathsf{uid}, \mathsf{did}, \mathsf{rid})$, Sim deletes the entry $\texttt{St}_\mathsf{S}[\mathsf{uid}, \mathsf{did}, \mathsf{rid}]$, all entries of the form $\texttt{St}_\mathsf{R}[\mathsf{uid}, \star, \mathsf{did}, \mathsf{rid}]$, and all entries of the form $\texttt{St}_\mathtt{P}[\star, \mathsf{uid}, \mathsf{did}, \mathsf{rid}]$.

Evaluation:

- **Client Evaluate:** On input $(\texttt{ClientEvaluate}, \mathsf{sid}, \mathsf{uid}, \mathsf{did}, \mathsf{rid}, \mu)$, Sim checks if there is a p_μ value previously associated with the received μ. If so, it retrieves the did' associated with μ (when it was first encountered), sets $k^\star \leftarrow \texttt{St}_\mathtt{N}[\mathsf{uid}, \star, \mathsf{did}, \mathsf{rid}]$, retrieves $k' \leftarrow \texttt{St}_\mathsf{D}[\mathsf{uid}, \mathsf{did}', \mathsf{rid}]$, and sets $r^\star \leftarrow k^\star / k'$, and sets $p^\star_\mu \leftarrow p_\mu \cdot r^\star$. If not, it samples a $p^\star_\mu \leftarrow_\mathsf{R} \mathbb{G}$ and stores the association between μ and $p^\star_\mu$. Then, it also stores $\texttt{St}_\mathtt{P}[\mathsf{sid}, \mathsf{uid}, \mathsf{did}, \mathsf{rid}] \leftarrow p^\star_\mu$. For $\ell = \lceil \log_2(q) \rceil$, Sim simulates the transmission of a message of length ℓ for $\mathcal{F}_\mathsf{SC}$ and outputs $(\mathsf{ssid}_3, \mathsf{D}_\mathsf{did}, \ell)$ to $\mathcal{A}$.
- **Server Evaluate:** On input $(\texttt{ServerEvaluate}, \mathsf{sid}, \mathsf{uid}, \mathsf{did}, \mathsf{rid})$, Sim retrieves $k \leftarrow \texttt{St}_\mathsf{S}[\mathsf{uid}, \mathsf{did}, \mathsf{rid}]$ and $p \leftarrow \texttt{St}_\mathtt{P}[\mathsf{sid}, \mathsf{uid}, \mathsf{did}, \mathsf{rid}]$, and stores $\texttt{Prog}[\mathsf{uid} \,||\, \mathsf{rid}] \leftarrow p^k$.

Compromise:

- **Device Compromise:** On input $(\texttt{CompromiseDevice}, \mathsf{uid}, \mathsf{did})$ from $\mathcal{A}$, return all entries of the form $\texttt{St}_\mathsf{D}[\mathsf{uid}, \mathsf{did}, \star]$ and $\texttt{St}_\mathtt{N}[\mathsf{uid}, \star, \mathsf{did}, \star]$ to $\mathcal{A}$.
- **Server Compromise:** On input $(\texttt{CompromiseServer}, \mathsf{rid}, V)$ from $\mathcal{A}$, do the following: For each tuple (uid, z) of V, let $y \leftarrow \texttt{Prog}[\mathsf{uid}, \mathsf{rid}]$, and execute $\mathcal{G}_{\mathsf{pRO}}.\texttt{ProgramRO}(\text{“}\texttt{server}\text{”} \,||\, y \,||\, \mathsf{uid} \,||\, \mathsf{rid}, z)$, aborting if unsuccessful. Then, return all entries of the form $\texttt{St}_\mathsf{S}[\mathsf{uid}, \star, \mathsf{rid}]$, $\texttt{St}_\mathsf{R}[\mathsf{uid}, \star, \star, \mathsf{rid}]$, and $\texttt{St}_\mathtt{P}[\star, \mathsf{uid}, \mathsf{did}, \mathsf{rid}]$ to $\mathcal{A}$.

Figure 5: The simulator Sim for Π_{DH}.

$\langle$ClientInitialized, uid, did, rid$\rangle$ exists. This record is only stored in `ClientInitialize` and `ClientFinishRegistration` for Game_1, which exactly matches the storage pattern for $\mathtt{St}_\mathsf{D}$ in Game_0 on the same input parameters.

- For (`ServerInitialize`, uid, did, rid), the abort condition in Game_0 is based on if there exists an entry for $\mathtt{St}_\mathsf{S}[\mathsf{uid}, \mathsf{did}, \mathsf{rid}]$, whereas the abort condition in Game_1 is based on if the record $\langle$Registered, uid, did, rid$\rangle$ exists. This record is stored in `ServerInitialize` and `ServerAcceptRegistration`, and deleted in `Revoke`, which exactly matches the storage (and deletion) pattern for $\mathtt{St}_\mathsf{S}$ in Game_0 in the same input parameters.

- For (`ClientStartRegistration`, uid, did_1, did_2, rid), the abort condition in Game_0 is based on if there exists an entry for $\mathtt{St}_\mathsf{D}[\mathsf{uid}, \mathsf{did}_1, \mathsf{rid}]$, whereas the abort condition in Game_1 is based on if the record $\langle$ClientInitialized, uid, did_1, rid$\rangle$ exists. As covered in the case for `ClientInitialize`, the storage patterns for $\mathtt{St}_\mathsf{D}$ and `ClientInitialize` match each other across the games.

- For (`ClientFinishRegistration`, uid, did_1, did_2, rid), the abort conditions in Game_0 are based on if there exists an entry for $\mathtt{St}_\mathsf{D}[\mathsf{uid}, \mathsf{did}_2, \mathsf{rid}]$ and if a message was received from $\mathcal{F}_{\mathsf{SC}}$, invoked by $\mathsf{D}_{\mathsf{did}_1}$ through `ClientStartRegistration`, whereas the abort conditions in Game_1 are based on if the record $\langle$ClientInitialized, uid, did_2, rid$\rangle$ and $\langle$ClientRegistrationStarted, uid, did_1, did_2, rid$\rangle$ exists. The former is covered as in the case for `ClientInitialize`. For the latter, note that the creation of the ClientRegistrationStarted record in Game_1 occurs precisely when $\mathsf{D}_{\mathsf{did}_1}$ sends a message through `ClientStartRegistration` in Game_0.

- For (`ServerAcceptRegistration`, uid, did_1, did_2, rid), the abort conditions in Game_0 are based on if there exists an entry for $\mathtt{St}_\mathsf{S}[\mathsf{uid}, \mathsf{did}_2, \mathsf{rid}]$, $\mathtt{St}_\mathsf{S}[\mathsf{uid}, \mathsf{did}_1, \mathsf{rid}]$, and if a message was received from $\mathcal{F}_{\mathsf{SC}}$, sent by $\mathsf{D}_{\mathsf{did}_1}$ through `ClientStartRegistration`, whereas the abort conditions in Game_1 are based on if the records $\langle$Registered, uid, did_2, rid$\rangle$, $\langle$Registered, uid, did_1, rid$\rangle$, and the record $\langle$ClientRegistrationStarted, uid, did_1, did_2, rid$\rangle$ exist. The first two are covered as in the case for `ServerInitialize`, and the last is covered as in the case for `ClientFinishRegistration`.

- For (`Revoke`, uid, did, rid), the abort condition in Game_0 is based on if there exists an entry for $\mathtt{St}_\mathsf{S}[\mathsf{uid}, \mathsf{did}, \mathsf{rid}]$, whereas the abort condition in Game_1 is based on if the record $\langle$Registered, uid, did, rid$\rangle$ exists. This is covered as in the case for `ServerInitialize`.

- For (`ClientEvaluate`, sid, uid, did, rid, x), the abort condition in Game_0 is based on if there exists an entry for $\mathtt{St}_\mathsf{D}[\mathsf{uid}, \mathsf{did}, \mathsf{rid}]$, whereas the abort condition in Game_1 is based on if the record $\langle$ClientInitialized, uid, did, rid$\rangle$ exists. This is covered as in the case for `ClientInitialize`.

- For (`ServerEvaluate`, sid, uid, did, rid), the abort conditions in Game_0 are based on if there exists an entry for $\mathtt{St}_\mathsf{S}[\mathsf{uid}, \mathsf{did}, \mathsf{rid}]$ and if a message was received from $\mathcal{F}_{\mathsf{SC}}$, sent by $\mathsf{D}_{\mathsf{did}}$ through `ClientEvaluate`, whereas the abort conditions in Game_1 are based on if the records $\langle$Registered, uid, did, rid$\rangle$ and $\langle$ClientEvaluation, sid, uid, did, rid, $\star\rangle$ exist. The former is covered as in the case for `ServerInitialize`. For the latter, note that the creation of the ClientEvaluation record in Game_1 occurs precisely when $\mathsf{D}_{\mathsf{did}}$ sends a message through `ClientEvaluate` in Game_0.

This covers all cases of abort conditions that are triggered based on the existence of entries in $\mathtt{St}_{\mathsf{D}}$, $\mathtt{St}_{\mathsf{S}}$, or messages being sent through $\mathcal{F}_{\mathsf{SC}}$ in Game_0, being replaced by those same abort conditions triggered instead of the existence of records, which concludes the proof. □

Lemma 2. The quantity $|\Pr[\mathsf{Game}_1(\mathcal{E}) = 1] - \Pr[\mathsf{Game}_2(\mathcal{E}) = 1]|$ is negligible in the $\mathcal{F}_{\mathsf{SC}}$-hybrid model.

Proof. Observe that there exists a simulator for each instantiation of a secure channel that interacts with the ideal $\mathcal{F}_{\mathsf{SC}}$ functionality and simply outputs the length $\ell = \lceil \log_2(q) \rceil$ to the adversary $\mathcal{A}$. Note that all messages are sent across secure channels in Game_1 are of length ℓ, which matches the leakage to the adversary in Game_2. Hence, we can invoke the UC-security of secure channels to conclude that Game_1 and Game_2 behave indistinguishably for the functions in both games that deal with secure channels.

Note, by inspection, that the changes to `CompromiseDevice` and `CompromiseServer` across the two games are purely syntactic. Indeed, the state variable $\mathtt{St}_{\mathsf{N}}[\mathsf{uid}, \mathsf{did}, \mathsf{rid}]$ in Game_2 keeps track of the messages received by device $\mathsf{D}_{\mathsf{did}}$ in a call to `ClientFinishRegistration` in Game_1. Similarly, the state variable $\mathtt{St}_{\mathsf{R}}[\mathsf{uid}, \mathsf{did}, \mathsf{rid}]$ in Game_2 keeps track of the messages received by server $\mathsf{S}_{\mathsf{rid}}$ in a call to `ServerAcceptRegistration` in Game_1. And finally, the state variable $\mathtt{St}_{\mathsf{P}}[\mathsf{sid}, \mathsf{uid}, \mathsf{did}, \mathsf{rid}]$ keeps track of the messages received by server $\mathsf{S}_{\mathsf{rid}}$ in a call to `ServerEvaluate` in Game_1. □

Lemma 3. The quantity $|\Pr[\mathsf{Game}_2(\mathcal{E}) = 1] - \Pr[\mathsf{Game}_3(\mathcal{E}) = 1]|$ is negligible for all $\mathcal{F}_{\mathsf{ORF}}$-admissible adversaries in the $\mathcal{G}_{\mathsf{pRO}}$-hybrid model, under the DDH assumption in $\mathbb{G}$.

Proof. Let Q be the total number of user-server registered pairs made throughout Game_2. We consider a series of hybrid games, $\mathsf{Game}_{2,i}$ for each $i \in [Q+1]$, where in $\mathsf{Game}_{2,i}$, if for invocation of $(\mathtt{ClientEvaluate}, \mathsf{sid}, \mathsf{uid}, \mathsf{rid})$ it is the case that uid is one of the first $i-1$ user-server registered pairs, then it uniformly samples $p \leftarrow_{\mathsf{R}} \mathbb{G}$ under the conditions of Game_3; otherwise, it compute $h \leftarrow \mathcal{G}_{\mathsf{pRO}}.\mathtt{HashQuery}(\text{“}\mathtt{client}\text{”} \mid\mid x \mid\mid \mathsf{uid} \mid\mid \mathsf{rid})$ and $p \leftarrow h^k$. Note that $\mathsf{Game}_{2,1} = \mathsf{Game}_2$ and $\mathsf{Game}_{2,Q+1} = \mathsf{Game}_3$. Hence, it suffices to show that $\mathsf{Game}_{2,i}$ and $\mathsf{Game}_{2,i+1}$ are indistinguishable.

Let $(\mathsf{uid}, \mathsf{rid}_i)$ represent the i^{th} pair of user-server registered pairs made throughout Game_2. Given that the adversary $\mathcal{A}$ is $\mathcal{F}_{\mathsf{ORF}}$-admissible, it either never triggers an active server compromise of the form $(\mathtt{CompromiseServer}, \mathsf{rid}_i)$, or it never triggers an active device compromise of the form $(\mathtt{CompromiseDevice}, \mathsf{uid}_i, \star)$.

In the former case, if $\mathcal{A}$ never triggers an active server compromise on server $\mathsf{S}_{\mathsf{rid}_i}$, then the only information the adversary can receive are the entries of $\mathtt{St}_{\mathsf{D}}$ and the plaintexts of messages sent to devices. In particular, the p values computed in `ClientEvaluate` involving rid_i are never revealed to the adversary, and the k value used in `ServerEvaluate` involving rid_i is also kept hidden from the adversary. Furthermore, in `ServerEvaluate` of $\mathsf{Game}_{2,i}$, the z values output to the environment are sampled uniformly from the output space $\mathcal{O}$, and while there is a dependence between z and p^k established by the invocation of $\mathcal{G}_{\mathsf{pRO}}.\mathtt{HashQuery}$ in $\mathsf{Game}_{2,i}$, the fact that k is hidden means that p^k (and by extension, z) is independent of p. Since the adversary's view is completely independent of p in this case, it follows that $\mathsf{Game}_{2,i}$ and $\mathsf{Game}_{2,i+1}$ are indistinguishable to $\mathcal{A}$, unconditionally.

In the latter case, if $\mathcal{A}$ never triggers an active device compromise on device $\mathsf{D}_{\mathsf{did}}$ associated with uid, then we show that $\mathsf{Game}_{2,i}$ and $\mathsf{Game}_{2,i+1}$ are indistinguishable based on the DDH assumption in $\mathbb{G}$. Let $(g, \alpha, \beta, \gamma)$ be the DDH tuple provided by the DDH challenger for the group $\mathbb{G}$, where g is a generator for $\mathbb{G}$. We present a simulator $\mathcal{B}$ that interacts with a DDH challenger as follows:

- For each unique invocation of $\mathcal{G}_{\mathsf{pRO}}.\texttt{HashQuery}$ on an input x (triggered either directly by $\mathcal{A}$ or indirectly through `ClientEvaluate` or `ServerEvaluate`), $\mathcal{B}$ will associate a uniformly random scalar $r_x \leftarrow_{\mathsf{R}} [p]$ for input x, and invoke $\mathcal{G}_{\mathsf{pRO}}.\texttt{ProgramRO}(x, \beta^{r_x})$.
- For each invocation of $(\texttt{ClientEvaluate}, \mathsf{sid}, \mathsf{uid}, \mathsf{did}, \mathsf{rid}, x)$, if $\mathsf{uid} = \mathsf{uid}_i$ and $\mathsf{rid} = \mathsf{rid}_i$, it sets $p \leftarrow \gamma^{r_x}$; otherwise, it computes p as dictated by $\mathsf{Game}_{2,i}$.

Observe that if the DDH challenger issued a "real" tuple, with $(g, \alpha = g^a, \beta = g^b, \gamma = g^{ab})$ for uniformly random scalars $a, b \in [p]$, then the random oracle outputs on input x are distributed as a uniformly random group element $h_x \in \mathbb{G}$, and the respective `ClientEvaluate` output (on the same input x, and for user uid_i and server rid_i) is $\gamma^{r_x} = (h_x)^a$, which exactly matches the behavior of $\mathsf{Game}_{2,i}$. On the other hand, if the DDH challenger issued a "random" tuple, then γ^{r_x} would be distributed uniformly at random and independent across different inputs x, which thus matches the behavior of $\mathsf{Game}_{2,i+1}$.

We conclude that an adversary which can distinguish between $\mathsf{Game}_{2,i}$ and $\mathsf{Game}_{2,i+1}$ can be used to construct an adversary that distinguishes between the real or random tuple from the DDH challenger. Taking a union bound over the Q device-server registered pairs across all games $\mathsf{Game}_{2,i}$, the claim follows. □

Lemma 4. The quantity $|\Pr[\mathsf{Game}_3(\mathcal{E}) = 1] - \Pr[\mathsf{Game}_4(\mathcal{E}) = 1]|$ is negligible for all $\mathcal{F}_{\mathsf{ORF}}$-admissible adversaries in the $\mathcal{G}_{\mathsf{pRO}}$-hybrid model.

Proof. In Game_4 of `ServerEvaluate`, the z value sent to the environment is the output of calling $\mathcal{G}_{\mathsf{pRO}}.\texttt{HashQuery}(\text{"}\texttt{server}\text{"} \mid\mid p^{k_{\mathsf{S}}} \mid\mid \mathsf{uid} \mid\mid \mathsf{rid})$, where $k_{\mathsf{S}} \leftarrow \texttt{St}_{\mathsf{S}}[\mathsf{uid}, \mathsf{did}, \mathsf{rid}]$, and where $p \leftarrow h^{k_{\mathsf{D}}}$, where h is a uniformly random group element consistently tied to the tuple $(\mathsf{uid}, \mathsf{rid}, x)$, and $k_{\mathsf{D}} \leftarrow \texttt{St}_{\mathsf{D}}[\mathsf{uid}, \mathsf{did}, \mathsf{rid}]$. Putting this together, this means that $p^{k_{\mathsf{S}}} = h^{k_{\mathsf{S}} \cdot k_{\mathsf{D}}}$. Based on the construction of the registration steps, note that $k_{\mathsf{S}} \cdot k_{\mathsf{D}}$ is a consistent value for any user, regardless of the device that initiated the corresponding `ClientEvaluate` call. Furthermore, since $\mathcal{A}$ is $\mathcal{F}_{\mathsf{ORF}}$-admissible, it cannot perform an active device compromise (which would reveal k_{S}) and an active server compromise (which would reveal k_{D}) for a registered device. In other words, observe that the value $k_{\mathsf{S}} \cdot k_{\mathsf{D}}$ is independent of the adversary's view, despite the individual compromises it performs. As a result, the random oracle preimage for z in Game_4 is also independent of the adversary's view, and so the output z is also independent of the adversary's view. Similarly, in Game_4, when the output z is truly uniformly randomly sampled from the output space $\mathcal{O}$. Therefore, the distribution of z as output to the environment between Game_2 and Game_3 are indistinguishable.

Next, we consider the differences between Game_2 and Game_3 for `CompromiseServer`. The only difference to the adversary here is that there is a chance that in Game_3, when executing $\mathcal{G}_{\mathsf{pRO}}.\texttt{ProgramRO}$, the operation can fail. It suffices to bound the probability of this failure for the remainder of the proof.

The random oracle operation will fail if $\mathcal{G}_{\mathsf{pRO}}$ has already assigned a value to $p^k \leftarrow \texttt{Prog}[\mathsf{uid}, \mathsf{rid}]$ before this invocation was made. Note that the p values from $\texttt{St}_{\mathsf{P}}$ are only exposed upon a call to `CompromiseServer`, and k is only exposed upon a call to `CompromiseDevice` from $\texttt{St}_{\mathsf{D}}$. Using the fact that $\mathcal{A}$ must be $\mathcal{F}_{\mathsf{ORF}}$-admissible, the value p^k is independent of the adversary's view. If $\mathcal{A}$ makes a total of Q queries to the random oracle, the probability of triggering the failure event is Q/p (where p is the size of the group $\mathbb{G}$), which is negligible in the security parameter. □

References

[1] Dan Boneh. The decision Diffie-Hellman problem. In *Third Algorithmic Number Theory Symposium (ANTS)*, volume 1423 of *LNCS*. Springer, Heidelberg, 1998. Invited paper.

[2] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 280–312. Springer, Heidelberg, April / May 2018.

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

[4] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 337–351. Springer, Heidelberg, April / May 2002.

[5] Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The pythia PRF service. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 547–562. USENIX Association, August 2015.

[6] 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.

[7] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). Cryptology ePrint Archive, Report 2016/144, 2016. https://eprint.iacr.org/2016/144.

[8] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17*, volume 10355 of *LNCS*, pages 39–58. Springer, Heidelberg, July 2017.

[9] Stanislaw Jarecki, Hugo Krawczyk, and Jason Resch. Threshold partially-oblivious PRFs with applications to key management. Cryptology ePrint Archive, Report 2018/733, 2018. https://eprint.iacr.org/2018/733.