BeaverDam: A Post-Quantum Threshold Key Recovery Protocol from a One-Time Threshold Oblivious Random Function*

Benedikt Auerbach[1] Shuichi Katsumata[1,2] Kaoru Takemure[1,2] Ida Tucker[1]

[1] PQShield [2] AIST

Abstract

Secure backups are a practical necessity of E2EE messaging apps, typically relying on password-protected key-recovery (PPKR) protocols. These allow users to turn a human-memorable password into a cryptographic key with the help of a server. Recently, Signal and Juicebox proposed *threshold* PPKR (tPPKR) protocols, removing the trust on a centralized server. Unfortunately, known solutions are only classically secure, lagging behind the shift toward *post-quantum* (PQ), or to be more precise, *hybrid* secure E2EE.

We present BeaverDam, a novel tPPKR protocol that is PQ secure with efficiency close to existing classical protocols. It also easily extends to hybrid security, standing as the first concretely efficient hybrid-secure tPPKR protocol. BeaverDam relies on a new primitive we introduce — *one-time* threshold oblivious random functions (tORF). Unlike full-fledged OPRFs, which are notoriously difficult to construct in a PQ-secure manner, one-time tORF offers weaker but sufficient security guarantees tailored to the tPPKR setting.

We then construct an efficient one-time tORF called Beaver, inspired by multi-party computation (MPC) techniques. Owing to the one-time feature, we only need to securely compute a 2-wise independent hash function, making our protocol very efficient. We conclude by presenting a practical instantiation of BeaverDam and comparing its performance to state-of-the-art classical tPPKR protocols, namely Signal's SVR3 and Juicebox.

1 Introduction

While secure messaging platforms like Signal, WhatsApp, and Messenger provide end-to-end encryption (E2EE) for billions of users, maintaining data availability — such as message history, account information, and address book — remains a practical necessity when a device is lost. This requirement creates a fundamental tension: users must be able to recover their data securely, yet the service provider must remain strictly unable to access it.

A secure message backup protocol, or its technical core, a *key recovery protocol*, aims to solve this conundrum.

Key recovery protocols. While numerous services offer different forms of key recovery, e.g., WhatsApp [1], Signal [64, 71], Apple [61], Google [76], Messenger [2], they all build on the idea of using a human-memorable (i.e., a low-entropy) secret like a password as something users can rely on when their devices get lost. The cryptographically most sound and well-studied (e.g., [30, 37]) protocol among them is the WhatsApp backup protocol (WBP).

At its core, WBP relies on an *oblivious pseudorandom function* (OPRF) [39], an interactive protocol between a user with an input x and a server with a PRF key sk. An OPRF allows the user to learn only the PRF evaluation $y = f_{\mathsf{sk}}(x)$ without revealing anything to the server. It is easy to see how this is useful for a key recovery protocol; set the password as x and the recovered key as y. The resulting value y can then be used to encrypt and back up user data on the server without compromising E2EE.

Removing the single point of failure. An inherent shortcoming of WBP is that a compromised server can mount a brute-force dictionary attack to find the user's password. While a common mitigation is to use secure hardware to limit the number of password guesses [1, 61, 64, 74, 76], history shows that secure hardware is error prone, e.g., [27, 44, 65, 67, 69]. As E2EE is only as secure as the key recovery protocol, compromising the server (or the hardware) will undermine the system as a whole.

Threshold key recovery protocols, as developed by Signal [28] and Juicebox [73], mitigate this risk by distributing trust across multiple servers under the pragmatic assumption that compromising *all* servers is infeasible.[1] A thresholdized protocol remains immune to offline dictionary attacks as long as *one* server remains uncompromised. Conveniently, such a protocol can be realized from a (threshold) OPRF [4, 52–54], offering stronger security at almost no additional cost compared to its non-thresholdized counterpart.

Post-quantum and hybrid security. As the secure messaging landscape transitions to *post-quantum* (PQ) security, or *hybrid* security to be more precise — as seen by Signal [11, 33, 60] and Apple's PQ3 [10] — (threshold) key recovery protocols must also migrate to PQ security. Unfortunately, known key retrieval protocols all rely on a single type of elliptic-curve based (threshold) OPRF, offering only *classical* security. While PQ OPRFs exist, they either lack the security required for key recover protocols (e.g., no malicious security or UC security) or are prohibitively inefficient. To put this in perspective, the state-of-the-art PQ OPRF with the required security (proven secure only in the *classical* random oracle model (ROM)) [78] has a 3700x (!) communication overhead compared to its classical counterpart. To make matters worse, a recent work by Faller et al. [36] shows that black-box combiners for OPRF do not exist, demonstrating the difficulty of constructing a hybrid secure key recovery protocol even in presence of a PQ secure OPRF.

1.1 Our Contributions

We present BeaverDam, an efficient threshold password-protected key recovery (tPPKR) protocol. It relies solely on symmetric cryptography and secure channels between the user and the servers. By

*This is the full version of the paper published in the Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), https://doi.org/10.1145/3830454.3846580.

[1] In the following, we no longer differentiate between servers and secure hardware. Following [28], different servers are understood to run different types of secure hardware, making a specific vulnerability hard to exploit on different platforms.

leveraging these secure channels, BeaverDam easily achieves post-quantum (PQ) or hybrid security accordingly. Notably, it is the first such protocol proven secure in the quantum random oracle model (QROM), explicitly accounting for the impact of Grover's search on offline dictionary attacks — a security consideration overlooked by existing candidates proven only in the classical ROM. The design is also simple, offering a communication cost comparable to the classical counterparts [28, 73] even in the PQ or hybrid setting. Below, we explain our contributions in more detail.

Key observation. While a (threshold) OPRF is sufficient for constructing a tPPKR protocol, we provide three observations demonstrating that it is not strictly necessary. First (Obs1), in practice, servers are already assumed to operate under a public-key infrastructure (PKI) [1, 61, 64, 71, 76] — an atypical setting for OPRFs. Second (Obs2), although threshold OPRFs give strong statistical user privacy guarantees even if *all* servers are compromised, it does not carry over to tPPKR protocols due to an inherent offline dictionary attack.[2] We can therefore target user privacy only if there exists *at least one* uncompromised server. Finally (Obs3), most deployed password-based protocols enforce *rate-limiting*, where servers cap the number c of failed password attempts, e.g., WhatsApp [1] and Signal [64] use $c = 10$. Once these attempts are exhausted, users are locked out and we can forgo correctness guarantees.

Weakening threshold OPRF. These observations lead us to introduce *one-time threshold oblivious random functions* (tORF) as a core primitive for building tPPKR. Similar to threshold OPRFs, the user inputs x, and each server holds a share of a secret function f. The difference is that tORF only provides security for a *single* evaluation (related to Obs3), and f is *truly* random. Moreover, the user's privacy of the input x relies on the secrecy of the communication with at least one honest server (related to Obs1 and Obs2).

We formalize security through a simulation-based definition, borrowing ideas from UC-based (threshold) OPRF definitions [12, 45, 49, 54, 55]. Unlike UC-based definitions, which provide a single unified functionality, ours provide a clean separation between user and server privacy, which we believe makes it easier to understand and less error-prone. Indeed, UC definitions for OPRFs are often monolithic, abstract, and can be inaccessible to the broader applied cryptography community, which can obscure cryptographic details, essential for modularity, and even lead to subtle errors [12, 36, 45]. (See [40] for an extended discussion.) More details are given in Section 3.

One-time tORF *to key recovery.* We take advantage of the rate-limiting feature to construct a tPPKR protocol, BeaverDam, generically from one-time tORFs. For a rate-limit of c, the user uploads c (shares of) tORF instances $(f_i)_{i\in[c]}$ and corresponding ciphertexts $(\mathsf{ct}_i)_{i\in[c]}$, which encrypt a key K using values $y_i = f_i(\mathsf{pwd})$, to the servers. Every recovery attempt consumes one instance regardless of the outcome. Success allows the user to retrieve y_i and ct_i to recover K, after which they replenish the server-side tORF instances, and reset the rate-limit to c. To prevent unauthorized updates, replenishment is authenticated by a tag derived from y_i.

We prove BeaverDam is *semi-strong* augmented secure, a novel notion bridging standard and strong augmented security across three corruption scenarios: If at least one server is honest, security reduces to unavoidable online password guessing attacks. Under full server corruption, the adversary must perform an offline dictionary attack, as in standard augmented security. However, we provide a stronger guarantee for the system "at rest": if no key recovery is run *after* full server corruption, pre-computations do not help the adversary in its attack, as in strong augmented security. To this end, we restrict the adversary from corrupting the last honest server during the brief window of protocol execution — a pragmatic trade-off for rarely executed recovery protocols.

The proof is notably subtle in the *quantum* ROM, requiring a fine-grained control over the RO queries (i.e., offline dictionary attack) performed by the adversary. Consequently, we introduce an extension of the semi-classical oneway to hiding lemma [9] that may be of independent interest. More details are given in Section 5.

Constructing a one-time tORF. We propose Beaver, an *information theoretic* one-time tORF that can be used to instantiate BeaverDam. Observe that, since we only require *one-time* use, it suffices to obliviously evaluate a 2-wise independent hash function $f(x) = A \cdot x + B$ for random $A, B \xleftarrow{\$} \mathbb{Z}_p$. In an initialization phase, the user secret shares A and B across the servers. Later, to evaluate $f(x)$, the user secret shares x and engages in a multi-party computation (MPC) with the servers.

We optimize this general design in several ways. First, by performing *pre-computation* during initialization, the user can create Beaver triples [16] to make the multi-party multiplication of A and x much lighter. However, done naively, this only provides security against *passive* adversaries, allowing malicious servers to mount online dictionary attacks against tPPKR by deviating from the protocol. While we can rely on generic MPC techniques like GMW [43] to detect malicious behavior, this prohibitively increases the communication and computational costs. To resolve this, we take advantage of the user being able to perform not only pre-computations, but also *part of the multi-party computation of* $f(x)$ in the initialization phase; unlike in a standard MPC setting, during initialization, the user knows both f and x, where recall x is the password. This allows us to tweak the passively secure protocol, giving us the desired security in which malicious servers can no longer perform online attacks. More details are given in Section 4.

Efficiency comparison. We compare BeaverDam to existing tPPKR protocols, which, to our knowledge, all rely on some variant of OPRFs. Specifically, we evaluate against Signal's SVR3 [28], based on a standard OPRF, and Juicebox [74], based on a threshold OPRF. We compare them for different security levels (i.e., classical, PQ, and hybrid security), and conclude that BeaverDam substantially outperforms the others in terms of communication cost in the latter two settings, while remaining competitive in the classical setting. Concretely, key recovery in BeaverDam for 3 servers and $c = 10$ improves on a PQ secure instantiation of SVR3 by a factor in the

[2] Technically, this limitation is only true for a class of tPPKR protocols where there exists a mechanism to check the validity of a password after full corruption. While such a mechanism is not inherent to tPPKR protocols, it will become so when using tPPKR in a larger system such as in a secure message backup protocol.

range of 300x to 1500x while having an increased cost in the classical setting between 2.5x and 13x, the exact value depending on the amount of prior failed recovery attempts. See Section 6 for details.

Limitations and open problems. A limitation of BeaverDam is that it achieves semi-strong rather than strong augmented security. While achieving the latter seems challenging, incorporating user-specific salts in the hash computation for pwd could enhance practical security in multi-user environments as shown in a recent work by Dayanikli and Lehmann [31]. Additionally, our work focuses on the n-out-of-n threshold setting, mirroring the practical instantiation of Signal's SVR3. While we discuss a natural extension to t-out-of-n thresholds for small n in Section 5.5, a more general extension remains an open problem.

A range of password-based protocols have been constructed from tOPRF or password-protected secret sharing, a closely related primitive. Our protocols BeaverDam and Beaver provide similar functionality though they rely on a PKI. As we discuss in more detail in Section 1.3, they may therefore be used to build similar applications assuming a PKI.

1.2 Related Work

Password-protected secret sharing. Password-Protected Secret Sharing (PPSS) [13] is a primitive almost identical to tPPKR. Indeed, our definition for tPPKR (see Section 5.4) is inspired by existing definitions for PPSS. There exist various security definitions for PPSS, for example: in the game-based [13], and in the UC framework [23, 53]; with a secret chosen randomly or by the adversary; with or without interactive initialization, etc. Our game-based definition may be seen as one such variant, where we specifically consider the n-out-of-n setting under a static corruption model. We provide a detailed comparison between our game-based and existing UC-based definitions in Section 5.2.

Many PPSS schemes have appeared in the literature, focusing either on improving efficiency [52, 54, 79], on ensuring PQ security [68], or on providing additional properties [4, 15, 34]. We note that, to the best of our knowledge, the most practical PPSS schemes to date (in terms of efficiency, provided security properties, and ease of implementation), e.g., [28, 30, 73], are built from (threshold) OPRFs [52–55].

(Threshold) OPRFs. Since their first formalization [39], many OPRFs have been proposed, both with classical security (e.g., [47, 48, 53, 55, 56, 75]), and PQ security (e.g. [6–8, 14, 19, 32, 38, 49, 59, 70, 78]). See [40] for a systematic comparison of OPRF security notions. Variants providing novel security properties have also been developed, such as verifiable OPRFs [8, 29, 35, 52], and threshold OPRFs (tOPRF) [12, 15, 29, 45, 54, 72]. Jarecki et al. [54] were the first to formalize tOPRF, and provided a security definition in the UC framework. Recent work [12, 45] identified subtle flaws in their UC definition, to which they provide fixes (for both the definition and the protocol). Game-based definitions for tOPRF were introduced by Agrawal et al. [5] and Jarecki and Nazarian [57]. See [25] for an extensive overview on (threshold) OPRFs.

1.3 Further Potential Applications

A range of password-based protocols have been constructed from PPSS or tOPRF. Our protocols Beaver and BeaverDam provide similar functionality though they rely on a PKI, and may therefore be used to build similar applications assuming a PKI. Note however, that said applications must allow the client to upload new states to servers in a secure manner, since a Beaver state can only be used once.

For example, one could obtain a PQ threshold password authenticated key exchange (PAKE) protocol, using the compiler from Jarecki et al. [52], by substituting BeaverDam for PPSS, and combining it with a PQ key exchange protocol. In this application, it would be possible to upload new states using an established key. Password-based threshold authentication protocols [5, 15] have also been constructed from tOPRF and threshold signatures. One may be able to use Beaver instead of tOPRF, though the existence of a suitable threshold signature scheme remains an open question. Furthermore, if Beaver can be extended to achieve properties such as verifiability or partial obliviousness, it may be possible to build threshold variants of the PRF service of [35], and of the password hardening services (PHS) from [62, 77]. Finally, password-protected threshold signatures (PPTS) [34] have been built from strong augmented PPSS. Since BeaverDam achieves a weaker notion of augmented security, whether PPTS can be constructed from BeaverDam remains an interesting open problem.

2 Preliminaries

Notations. For a set $\mathcal{M}$, we denote $m \xleftarrow{\$} \mathcal{M}$ as the process of sampling uniformly random over $\mathcal{M}$. For a randomized algorithm $f : \mathcal{X} \times \mathcal{R} \to \mathcal{Y}$, $y \xleftarrow{\$} f(x)$ denotes the process of executing f on x with randomness $r \xleftarrow{\$} \mathcal{R}$. We write $y = f(x;r)$ when fixing the randomness r, and write $y \in f(x)$ to denote that $\exists r \in \mathcal{R}$ such that $y = f(x;r)$. We further write $f(\mathcal{X})$ for the set $\{f(x;r) \mid (x,r) \in \mathcal{X} \times \mathcal{R}\}$, where "|" is interpreted as "such that". With an overload of notation, $\{f(x;r) \mid (x,r) \xleftarrow{\$} \mathcal{X} \times \mathcal{R}\}$ denotes a distribution of the value $f(x;r)$ under the randomness of sampling (x,r). For two distributions $\mathcal{D}$ and $\mathcal{E}$, $\mathcal{D} \simeq_s \mathcal{E}$ denotes that they are statistically close. We use n-out-of-n additive secret sharing, denoting $(\llbracket x \rrbracket_i)_{i\in[n]} \leftarrow \mathsf{Share}(x,n)$ as the process of sharing a secret x into n shares $\llbracket x \rrbracket_i$ for $i \in [n]$, and $x := \mathsf{Rec}((\llbracket x \rrbracket_i)_{i\in[n]})$ as the inverse operation of reconstructing the secret x from said shares (i.e., $x := \sum_{i\in[n]} \llbracket x \rrbracket_i$). $\langle a; b\rangle \xleftarrow{\$} M\langle x; y\rangle$ denotes the execution of an interactive protocol M between two parties with inputs x and y, locally receiving outputs a and b. We use a function **require** (b) to check a boolean statement in games and algorithms. If b is false, the game (resp. the algorithm) aborts (resp. terminates) with output 0.

Types of adversaries. When we consider the class of "efficient" algorithms, this includes all *classical* PPT and *quantum* PT adversaries. Background on quantum tools is provided in Section 2.2.

2.1 Symmetric Key Encryption

We review the definition of symmetric key encryption (SKE).

Definition 1 (SKE)**.** A symmetric key encryption (SKE) scheme consists of the following PPT algorithms where $\mathcal{K}$ denotes the secret key space:

$\mathsf{SKE.Enc}(\mathsf{k}, m) \to \mathsf{ct}$: On input a secret key $\mathsf{k} \in \mathcal{K}$ and a message m, outputs a ciphertext ct.

$\mathsf{SKE.Dec}(\mathsf{k}, \mathsf{ct}) \to m/\bot$: On input a secret key k and a ciphertext ct, outputs a m or $\bot$ indicating decryption failure.

Definition 2 (Correctness)**.** An SKE is correct if for all $\mathsf{k} \in \mathcal{K}$ and messages m, we have $\mathsf{SKE.Dec}(\mathsf{k}, \mathsf{SKE.Enc}(\mathsf{k}, m)) = m$.

Definition 3 (IND-CPA)**.** An SKE is IND-CPA *secure* if for any efficient adversary $\mathcal{A}$, the following advantage is $\mathsf{negl}(\lambda)$:

$$\mathsf{Adv}^{\mathsf{IND\text{-}CPA}}_{\mathsf{SKE},\mathcal{A}}(1^\lambda) := \left| 2 \cdot \Pr\left[b = b' \;\middle|\; \begin{array}{c} b \xleftarrow{\$} \{0,1\}, \mathsf{k} \xleftarrow{\$} \mathcal{K}, \\ (m_0, m_1) \xleftarrow{\$} \mathcal{A}(1^\lambda); \\ \mathsf{ct} \xleftarrow{\$} \mathsf{Enc}(\mathsf{k}, m_b), \\ b' \xleftarrow{\$} \mathcal{A}(\mathsf{ct}) \end{array} \right] - 1 \right|.$$

We also require an SKE to be *robust* [3], i.e., no ciphertext should validly decrypt under two distinct secret keys.

Definition 4 (Robustness)**.** An SKE is *robust* if for any efficient adversary $\mathcal{A}$, the following advantage is $\mathsf{negl}(\lambda)$:

$$\mathsf{Adv}^{\mathsf{Robust}}_{\mathsf{SKE},\mathcal{A}}(1^\lambda) := \Pr\left[\begin{array}{c} \mathsf{k} \neq \mathsf{k}' \\ \wedge\, \mathsf{Dec}(\mathsf{k}, \mathsf{ct}) \neq \bot \\ \wedge\, \mathsf{Dec}(\mathsf{k}', \mathsf{ct}) \neq \bot \end{array} \;\middle|\; (\mathsf{ct}, \mathsf{k}, \mathsf{k}') \xleftarrow{\$} \mathcal{A}(1^\lambda) \right].$$

2.2 Quantum Tools

Following [22], we model a quantum random oracle $O : \{0,1\}^n \to \{0,1\}^m$ as a mapping $|x\rangle|y\rangle \to |x\rangle|y \oplus O(y)\rangle$, where O is a uniformly sampled random function. In the quantum random oracle model (QROM), all parties can evaluate O in superposition.

Below, we recall the semi-classical oneway to hiding lemma [9, 20]. For that we first define what a semi-classical oracle $O^{\mathsf{sc}}_{\mathcal{S}} : \mathcal{X} \to \mathcal{Y}$ for a set $\mathcal{S} \subseteq \mathcal{X}$ is. This oracle, on input x, outputs 1 if $x \in \mathcal{S}$ and 0 otherwise. Then, for an oracle $\mathsf{H} : \mathcal{X} \to \mathcal{Y}$, we define the oracle $\mathsf{H}\backslash\mathcal{S}$ as an oracle H *punctured on* $\mathcal{S}$, where on input x, it first queries $O^{\mathsf{sc}}_{\mathcal{S}}(x)$ and then $\mathsf{H}(x)$. We denote the event FIND as the event that $O^{\mathsf{sc}}_{\mathcal{S}}(x)$ returns $|x\rangle$. For more details, we refer to [9]. We then have the following:

Lemma 1 (Oneway to Hiding, simplified, [9, Theorem 1])**.** *Let $\mathcal{X}$ and $\mathcal{Y}$ be finite sets and $\mathcal{S} \subseteq \mathcal{X}$ a random set. Let $\mathsf{G}, \mathsf{H} : \mathcal{X} \to \mathcal{Y}$ be random functions satisfying $\mathsf{G}(x) = \mathsf{H}(x)$ for all $x \notin \mathcal{S}$. Let z be an arbitrary bit string, where $\mathcal{S}, \mathsf{G}, \mathsf{H}, z$ may have an arbitrary joint distribution.*

Let $\mathcal{A}$ be a binary-output algorithm making at most Q oracle queries and define

$$P_{\mathsf{H}} := \Pr_{\mathsf{H},z}\left[\mathcal{A}^{\mathsf{H}}(z) = 1\right], \quad P_{\mathsf{G}} := \Pr_{\mathsf{G},z}\left[\mathcal{A}^{\mathsf{G}}(z) = 1\right],$$
$$P_{\mathsf{FIND}} := \Pr_{\mathsf{H},\mathsf{G},\mathcal{S},z}\left[\mathsf{FIND} : \mathcal{A}^{\mathsf{H}\backslash\mathcal{S}}(z)\right].$$

Then, we have

$$|P_{\mathsf{H}} - P_{\mathsf{G}}| \leq 2 \cdot \sqrt{(Q+1) \cdot P_{\mathsf{FIND}}}.$$

Lemma 2 (Search in Semi-Classical Oracle, simplified, [9, Theorem 2, Corollary 1] and [20, Lemma 4])**.** *Let $\mathcal{A}$ be a binary-output algorithm making at most Q oracle queries. Let $\mathcal{X}$ be a finite set, $\mathcal{S} \subseteq \mathcal{X}$ be a random set, and z be an arbitrary bit string, where $\mathcal{S}$ and z may have an arbitrary joint distribution.*

Let $\mathcal{B}$ be an algorithm that on input z chooses $i \xleftarrow{\$} [Q]$; runs $\mathcal{A}^{O^{\mathsf{sc}}_{\emptyset}}(z)$ until (just before) the i-th query; and then measures all query input registers in the computational basis and outputs the set $\mathcal{T}$ of measured outcomes. Then,

$$\Pr_{\mathcal{S},z}\left[\mathsf{FIND} : \mathcal{A}^{O^{\mathsf{sc}}_{\mathcal{S}}}(z)\right] \leq 4 \cdot (Q+1) \cdot \Pr[\mathcal{S} \cap \mathcal{T} \neq \emptyset : \mathcal{T} \xleftarrow{\$} \mathcal{B}(z)].$$

In particular, if for each $x \in \mathcal{X}$, $\Pr[x \in \mathcal{S}] \leq \epsilon$ (conditioned on z, on other oracles $\mathcal{A}$ has access to, and on other outputs of H), then the right hand side is upper bounded by $4 \cdot (Q+1) \cdot \epsilon$.

Below we extend Lemma 1 to capture a two-phase algorithm that receives a second input after receiving z but can no longer access the oracle. Indeed, we can give $\mathcal{A}$ such an additional input z' by preparing an auxiliary register in the initial state of $\mathcal{A}$. For simplicity, we denote such an algorithm by $\mathsf{st} \xleftarrow{\$} \mathcal{A}^O(z), b \xleftarrow{\$} \mathcal{A}(\mathsf{st}, z')$, using the notation of classical algorithms.

Lemma 3 (Extended Oneway to Hiding)**.** *Let $\mathcal{X}$, $\mathcal{Y}$, G, and H be finite sets and random functions as defined in Lemma 1. Let z and z' be arbitrary bit strings, where $\mathcal{S}, \mathsf{G}, \mathsf{H}, z, z'$ may have an arbitrary joint distribution. Let $\mathcal{A}$ be a binary-output two-phase algorithm making at most Q oracle queries and define*

$$\hat{P}_{\mathsf{H}} := \Pr_{\mathsf{H},z,z'}\left[b = 1 : \mathsf{st} \xleftarrow{\$} \mathcal{A}^{\mathsf{H}}(z), b \xleftarrow{\$} \mathcal{A}(\mathsf{st}, z')\right],$$
$$\hat{P}_{\mathsf{G}} := \Pr_{\mathsf{G},z,z'}\left[b = 1 : \mathsf{st} \xleftarrow{\$} \mathcal{A}^{\mathsf{G}}(z), b \xleftarrow{\$} \mathcal{A}(\mathsf{st}, z')\right],$$
$$\hat{P}_{\mathsf{FIND}} := \Pr_{\mathsf{H},\mathsf{G},\mathcal{S},z,z'}\left[\mathsf{FIND} : \mathcal{A}^{\mathsf{H}\backslash\mathcal{S}}(z)\right].$$

Then, we have

$$\left|\hat{P}_{\mathsf{H}} - \hat{P}_{\mathsf{G}}\right| \leq 2 \cdot \sqrt{(Q+1) \cdot \hat{P}_{\mathsf{FIND}}}.$$

In particular, if for each $x \in \mathcal{X}$, $\Pr[x \in \mathcal{S}] \leq \epsilon$ (conditioned on z, on other oracles $\mathcal{A}$ has access to, and on other outputs of H), then we have

$$\left|\hat{P}_{\mathsf{H}} - \hat{P}_{\mathsf{G}}\right| \leq 4 \cdot (Q+1) \cdot \sqrt{\epsilon}.$$

Proof. We consider a wrapper algorithm $\hat{\mathcal{A}}$ that receives both z and z' and internally runs $\mathcal{A}$ using its oracle access. Specifically, it first runs $\mathcal{A}$ on input z. When $\mathcal{A}$ makes queries to the oracle, it forwards them to its own oracle and returns the answers to $\mathcal{A}$. When $\mathcal{A}$ requests the second input, it gives z' to $\mathcal{A}$. Eventually, $\hat{\mathcal{A}}$ outputs the bit b returned by $\mathcal{A}$. By construction, we have

$$\Pr_{\mathsf{H},z,z'}\left[\hat{\mathcal{A}}^{\mathsf{H}}(z,z') = 1\right] = \hat{P}_{\mathsf{H}} \text{ and } \Pr_{\mathsf{G},z,z'}\left[\hat{\mathcal{A}}^{\mathsf{G}}(z,z') = 1\right] = \hat{P}_{\mathsf{G}}.$$

Applying Lemma 1, we obtain

$$\left|\hat{P}_{\mathsf{H}} - \hat{P}_{\mathsf{G}}\right| \leq 2 \cdot \sqrt{(Q+1) \cdot \Pr_{\mathsf{H},\mathsf{G},\mathcal{S},z,z'}\left[\mathsf{FIND} : \hat{\mathcal{A}}^{\mathsf{H}\backslash\mathcal{S}}(z,z')\right]}.$$

Since $\mathcal{A}$ makes oracle queries only before receiving z', giving z' does not affect the probability of FIND. Thus we have

$$\Pr_{\mathsf{H},\mathsf{G},\mathcal{S},z,z'}\left[\mathsf{FIND} : \hat{\mathcal{A}}^{\mathsf{H}\backslash\mathcal{S}}(z,z')\right] = \Pr_{\mathsf{H},\mathsf{G},\mathcal{S},z,z'}\left[\mathsf{FIND} : \mathcal{A}^{\mathsf{H}\backslash\mathcal{S}}(z)\right] = \hat{P}_{\mathsf{FIND}}.$$

Combining the above arguments, we obtain

$$\left|\hat{P}_{\mathsf{H}} - \hat{P}_{\mathsf{G}}\right| \leq 2 \cdot \sqrt{(Q+1) \cdot \hat{P}_{\mathsf{FIND}}}.$$

Moreover, due to Lemma 2, $\hat{P}_{\mathsf{FIND}}$ is bounded by $4 \cdot (Q+1) \cdot \epsilon$ when $\Pr[x \in \mathcal{S}] \leq \epsilon$ (conditioned on z, on other oracles $\mathcal{A}$ has

access to, and on other outputs of H). Applying this bound to the above inequality, we have

$$\left|\hat{P}_{\mathsf{H}} - \hat{P}_{\mathsf{G}}\right| \leq 4 \cdot (Q+1) \cdot \sqrt{\epsilon}.$$

This completes the proof. □

2.3 Secure Channels

In this work, we require *secure channels* between clients and servers. This ensures that exchanged messages are authenticated and stay hidden. Formally, secure channels can be defined in numerous ways and there is no "one fits all" definition; this is exemplified by the abundant literature on authenticated key exchange (AKE), which only captures the key establishment phase of a secure channel. We therefore provide a minimal definition of secure channel required in our work. In our model, servers are the only entities in possession of a public key and we assume it is known to all parties through a PKI. Effectively, we model a *server-authenticated* secure channel, capturing common key exchange protocols such as the Noise and TLS protocols.

Definition 5 (Secure Channel)**.** A *secure channel* consists of the following PPT algorithms where $\mathcal{K}$ and $\mathcal{M}$ denote the session key and message spaces, respectively.

$\mathsf{SC.Init_S}(1^\lambda) \to (\mathsf{pk_S}, \mathsf{sk_S})$**:** The server initialization algorithm on input a security parameter 1^λ, outputs a public and secret key pair $(\mathsf{pk_S}, \mathsf{sk_S})$.

$\mathsf{SC.KEX}\left\langle \mathsf{pk_S}; \mathsf{sk_S} \right\rangle \to \langle \mathsf{k_C}; \mathsf{k_S} \rangle$**:** An interactive *key exchange* protocol between a client and a server. Upon termination, client and server output session keys $\mathsf{k_C}, \mathsf{k_S} \in \mathcal{K} \cup \{\perp\}$, respectively, where $\perp$ indicates failure to establish a session key.

$\mathsf{SC.Send}(\mathsf{k}, \mathsf{msg}) \to \mathsf{ct}$**:** The send algorithm, on input a key k and a message msg, outputs a ciphertext ct.

$\mathsf{SC.Receive}(\mathsf{k}, \mathsf{ct}) \to \mathsf{msg}$**:** The receive algorithm on input a key k and a ciphertext ct, outputs a message $\mathsf{msg} \in \mathcal{M} \cup \{\perp\}$, where $\perp$ indicates a decryption failure.

Definition 6 (Correctness)**.** SC is correct if for all $\mathsf{k} \in \mathcal{K}$ and $\mathsf{msg} \in \mathcal{M}$, $\mathsf{SC.Receive}(\mathsf{k}, \mathsf{SC.Send}(\mathsf{k}, \mathsf{msg})) = \mathsf{msg}$.

Security considerations. At a high level, we want to claim that after a *successful* key exchange, the session keys can be used to *securely* exchange messages. For the later, we model two properties: indistinguishability and unforgeability. Indistinguishability requires that, assuming the established session key is hidden, an adversary cannot distinguish whether M_0 or M_1 was sent. Unforgeability requires that, under the same assumption, an adversary cannot forge a ciphertext ct accepted by the client or server.

Additionaly, we require that the channel satisfies forward secrecy. This property ensures that past sessions, for which session keys (and associated states) have been errased, remain secure, even if a server is corrupted, revealing its long term secret key to the adversary.

Capturing successful key establishment. Defining what constitutes a successful key exchange is more subtle. This is illustrated by the numerous technical papers on the formalization of AKEs (e.g., [17, 18, 21, 24, 41, 46, 50, 51, 63]). Informally, the subtlety lies in how to define what a *secure* session key is; this is required to exclude trivial attacks on the security of the exchanged messages. To keep our definitions minimal, we therefore make several simplifications (S1, S2, S3) below which are sufficient for our purpose.

Syntactical details and simplifications. We assume client and server communicate in a ping-pong manner during the interactive key-exchange protocol. More precisely, $(\mathsf{st_C}, \mathsf{mc}) \xleftarrow{\$} \mathsf{SC.KEX_C}(\mathsf{st_C}, \mathsf{ms})$ is followed by $(\mathsf{st_S}, \mathsf{ms}) \xleftarrow{\$} \mathsf{SC.KEX_S}(\mathsf{st_S}, \mathsf{mc})$, and so on. Here, ms and mc denote the protocol messages sent by the server and client, respectively, and $\mathsf{st_S}$ and $\mathsf{st_C}$ denote the server and client states maintained throughout the key exchange protocol, respectively. We restrict ourselves to protocols in which the client (resp. server) talks first (resp. last) unless the protocol ends prematurely (S1). That is, the client initiates the key exchange by sending $(\mathsf{st_C}, \mathsf{mc_{first}}) \xleftarrow{\$} \mathsf{SC.KEX_C}(\mathsf{pk_S})$ (where $\mathsf{st_C}$ is initially empty) and the server ends by running $(\mathsf{st_S}, \mathsf{ms}^*) \xleftarrow{\$} \mathsf{SC.KEX_S}(\mathsf{st_S}, \mathsf{mc})$, and sending $\mathsf{ms_{last}}$, where $\mathsf{ms}^* := (\mathsf{k_S}, \mathsf{ms_{last}})$.

Additionally, we require that the server explicitly authenticates itself to the client. This ensures that once the client establishes a session key, the server with whom it intended to communicate is also in possession of the *same* session key (S2). Finally, as we only consider scenarios where the client sends the first message, we require no security for any messages sent by the server *before* receiving a message from the client (S3).

Formal definition. Security of a secure channel is defined via the game given in Fig. 1, where the sessions a server maintains with distinct clients are tracked via the variable ctr. Simplifications S1 and S2 are covered by the flag β_{sound}. S3 is dictated by the server's state, which is maintained by the game, and can take five values: $\perp$, `started`, `finished`, `failed`, `challenge`. Here, `started` means that the key exchange protocol has started; `finished` (resp. `failed`) means that a session key was (resp. was not) established; and `challenge` means that the server has already received a message from the client, and as such messages sent by the server should be hidden. Namely, while the server does not receive any message from the client, the server's state will not be in the `challenge` state. The indistinguishability and unforgeability of messages are covered by flags β_{ind} and β_{unf}, respectively, while forward secrecy is ensured by the corruption oracle O_{corr}.

Definition 7 (Secure channel)**.** A secure channel SC is *forward secure* if for any efficient adversary $\mathcal{A}$ against $\mathsf{Game}_{\mathsf{SC},\mathcal{A}}$ (cf. Fig. 1), the following advantage is $\mathsf{negl}(\lambda)$:

$$\mathsf{Adv}_{\mathsf{SC},\mathcal{A}}(1^\lambda) := \left|2 \cdot \Pr[\mathsf{Game}_{\mathsf{SC},\mathcal{A}}(1^\lambda) = 1] - 1\right|$$

3 One-time Threshold Oblivious Random Function

In this section, we define *one-time threshold oblivious random functions* (tORF).

3.1 Syntax and Correctness

A one-time tORF is a two phase protocol between a client[3] and a collection of servers $\mathsf{S}_1, \ldots, \mathsf{S}_n$. In the first *initialization* phase, the client generates a function $f \in \mathcal{F}$ along with server states $(\mathsf{st}_{\mathsf{S}_i})_{i \in [n]}$, and uploads the states to the respective servers. Here,

[3] In the technical sections, we mainly use "client" instead of "user".

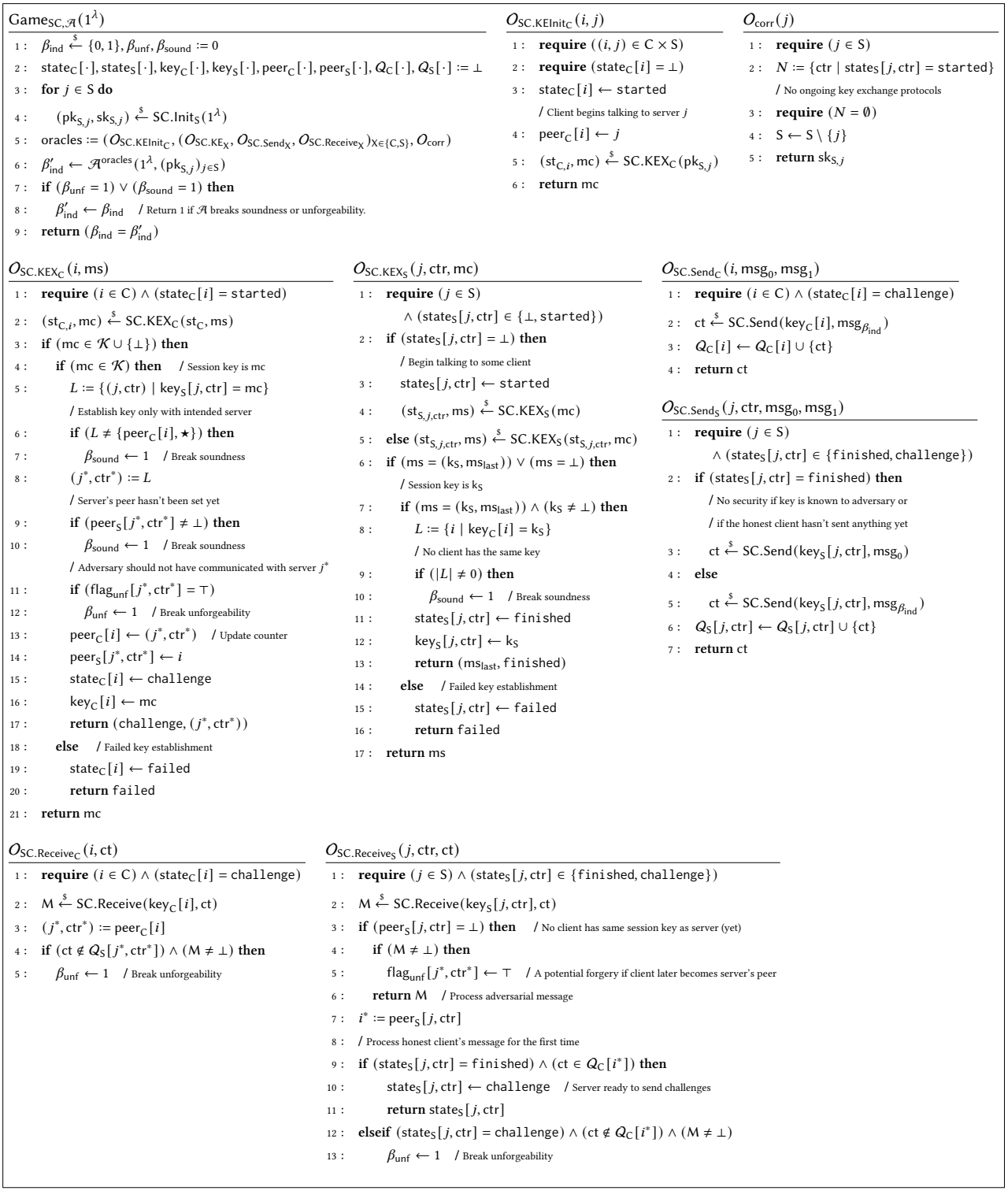

Figure 1: Security game for SC. Above, we assume $O_{\mathsf{SC.Receive}_\mathsf{S}}$ and $O_{\mathsf{SC.Receive}_\mathsf{C}}$ output some empty symbol to indicate that it has terminated. Moreover, we assume the adversary is notified when β_{unf} or β_{sound} are set to 1. Above $\star$ indicates an arbitrary value.

the client may discard any state generated during this phase. In the second *evaluation* phase, the client can interact with the servers to evaluate f on any input $x \in \mathcal{X}$.

Importantly, in the evaluation phase, the client does not learn any information on f besides the output $f(x)$ and the servers do not learn any information on the input x as long as at least one server remains uncompromised. The protocol is one-time-use, meaning that the evaluation phase can only be executed once to retain security.

Below, we provide the syntax of a one-time tORF.

Definition 8 (One time tORF)**.** A one-time *threshold oblivious random function* (tORF) consists of the following PPT algorithms where $\mathcal{F}$ is a subset of all functions from $\mathcal{X}$ to $\mathcal{Y}$.

$\mathsf{Init}(1^\lambda, n) \to (f, (\mathsf{st}_{\mathsf{S}_i})_{i\in[n]})$**:** The initialization algorithm on input the security parameter 1^λ and the number of servers n, outputs a function $f \in \mathcal{F}$ and server states $(\mathsf{st}_{\mathsf{S}_i})_{i\in[n]}$. Each server state $\mathsf{st}_{\mathsf{S}_i}$ is secret to the i-th server.

$\mathsf{Eval}\left\langle x; (\mathsf{st}_{\mathsf{S}_i})_{i\in[n]}\right\rangle \to \langle y; \perp\rangle$**:** An interactive evaluation protocol between a client with input $x \in \mathcal{X}$ and n servers with states $(\mathsf{st}_{\mathsf{S}_i})_{i\in[n]}$. The client terminates by outputting $y = \perp$ indicating failure or $y \in \mathcal{Y}$ indicating successful evaluation. The server always outputs $\perp$.

For simplicity, the servers only interact with the client. When we explicitly write down the evaluation protocol it goes as $(\mathsf{st}_\mathsf{C}, (\mathsf{mc}_i)_{i\in[n]}) \xleftarrow{\$} \mathsf{Eval}_\mathsf{C}(\mathsf{st}_\mathsf{C}, (\mathsf{ms}_i)_{i\in[n]})$ followed by all $i \in [n]$ servers running $(\mathsf{st}_{\mathsf{S}_i}, \mathsf{ms}_i) \xleftarrow{\$} \mathsf{Eval}_\mathsf{S}(\mathsf{st}_{\mathsf{S}_i}, \mathsf{mc}_i)$, and so on, where mc_i and ms_i denote the protocol messages sent by the client and server S_i, respectively, and st_C is the client state only used within the evaluation protocol. We further assume the client (resp. server) to send the first (resp. last) protocol message unless the protocol ends prematurely. Concretely, the client first executes $(\mathsf{st}_\mathsf{C}, (\mathsf{mc}_i)_{i\in[n]}) \xleftarrow{\$} \mathsf{Eval}_\mathsf{C}(x)$, creating a protocol specific state st_C. Eventually, the client state st_C will be the output $y \in \mathcal{Y} \cup \{\perp\}$ of the evaluation protocol and the server states $\mathsf{st}_{\mathsf{S}_i}$ are set to $\perp$.

A one-time tORF is required to be *correct*, meaning that a client learns $y = f(x)$ after an honest execution.

Definition 9 (Correctness)**.** A one-time tORF is *correct* if for all $\lambda \in \mathbb{N}$, $n = \mathsf{poly}(\lambda)$, all $x \in \mathcal{X}$, and $(f, (\mathsf{st}_{\mathsf{S}_i})_{i\in[n]}) \in \mathsf{Init}(1^\lambda, n)$ we have

$$\mathsf{Eval}\left\langle x; (\mathsf{st}_{\mathsf{S}_i})_{i\in[n]}\right\rangle = \langle f(x); \perp\rangle .$$

3.2 Security

We provide two simulation-based definitions for one-time tORF, separately capturing honest-client and honest-server security. Unlike monolithic and abstract UC-based (t)OPRF definitions [12, 45, 54, 55], our approach cleanly separates client and server privacy. We believe this modularity enhances clarity and reduces the risk of the subtle errors often associated with UC-based frameworks [12, 36, 40, 45]. Our model is made intentionally minimalistic, sufficient for our applications, and a detailed comparison with UC-based definitions is given at the end of this section. Note that our definition assumes an adversary cannot eavesdrop or inject messages in the honest client and server communication. This is naturally ensured by establishing a secure channel at the higher-layer application.

3.2.1 Server Privacy. Intuitively, we want to ensure that the output y^* of the evaluation protocol on input x^* looks random, even to an adversary that colludes with $n-1$ compromised servers. This should hold even if an adversary learns one "free" evaluation point $y = f(x)$ for any choice of $x \in \mathcal{X}$, reflecting the fact that a client may immediately start using y after the initialization phase in natural applications of (t)OPRF (see [45] for more discussion).[4]

For a standard, non-oblivious PRF, security is typically defined via a simple real-or-random game: the adversary must distinguish between $y^* = f(x^*)$ and a random $y^* \xleftarrow{\$} \mathcal{Y}$. However, this definition is insufficient for a tORF because it is an *oblivious* and *interactive* protocol; that is, the adversary's input x^* is never explicitly provided and it is unclear what the honest server messages should be in the random case while maintaining consistency with the free evaluation point.

To address this, we employ a simulation-based definition. We require a simulator capable of providing the adversary with a view indistinguishable from the real protocol without knowledge of the function f. To ensure consistency with the evaluation point y obtained by the adversary, we require an extraction algorithm, $\mathsf{Extract}_x$, which extracts a value $\widehat{x} \in \mathcal{X}$ from the adversary's transcripts, intuitively corresponding to the adversary's input x^*. If $\widehat{x} = x$, the simulator is given (x, y) to ensure the simulated transcript remains consistent with the adversary's free evaluation point. Otherwise, the simulator simulates using only $\widehat{x}$. This definition is inspired by the extraction algorithms implicitly found in UC-based (t)OPRF security proofs.

Formally, we define server privacy as follows.

Definition 10. [Server Privacy] A one-time tORF is *server private* if there exist efficient simulators $(\mathsf{SimInit}, \mathsf{SimEval}_\mathsf{S}, \mathsf{Extract}_x)$ such that for any (possibly inefficient) adversary $\mathcal{A}$, we have

$$\Pr[\mathsf{Game}^{\texttt{sp-real}}_{\mathsf{tORF},\mathcal{A}}(1^\lambda, n) = 1] \simeq_s \Pr[\mathsf{Game}^{\texttt{sp-sim}}_{\mathsf{tORF},\mathcal{A}}(1^\lambda, n) = 1],$$

where the two games are depicted in Fig. 2.

3.2.2 (Weak) Client Privacy. We aim to ensure that an adversary — specifically, a set of colluding malicious servers — learns no information (i.e., remains *oblivious*) about the client's input $x \in \mathcal{X}$ during the evaluation protocol. Mirroring our requirements for server privacy, this must hold even if the adversary learns one free evaluation point $y = f(x)$ for any choice of $x \in \mathcal{X}$.

Furthermore, we require client privacy to hold even if the adversary learns whether the client successfully recovered y. This implies that colluding malicious servers cannot mount *online attacks*, where they might attempt to extract non-trivial information about x by observing the protocol's success or failure. Formally, we require a simulator CorrectVal that simulates the validity of the client output. We coin our definition "weak" since client privacy is only required to hold as long as at least one server remains honest.

Formally, we define weak client privacy as follows.

[4] Put differently, in the security proof, the reduction (i.e., the adversary against server privacy) may need to use y before an evaluation protocol is executed.

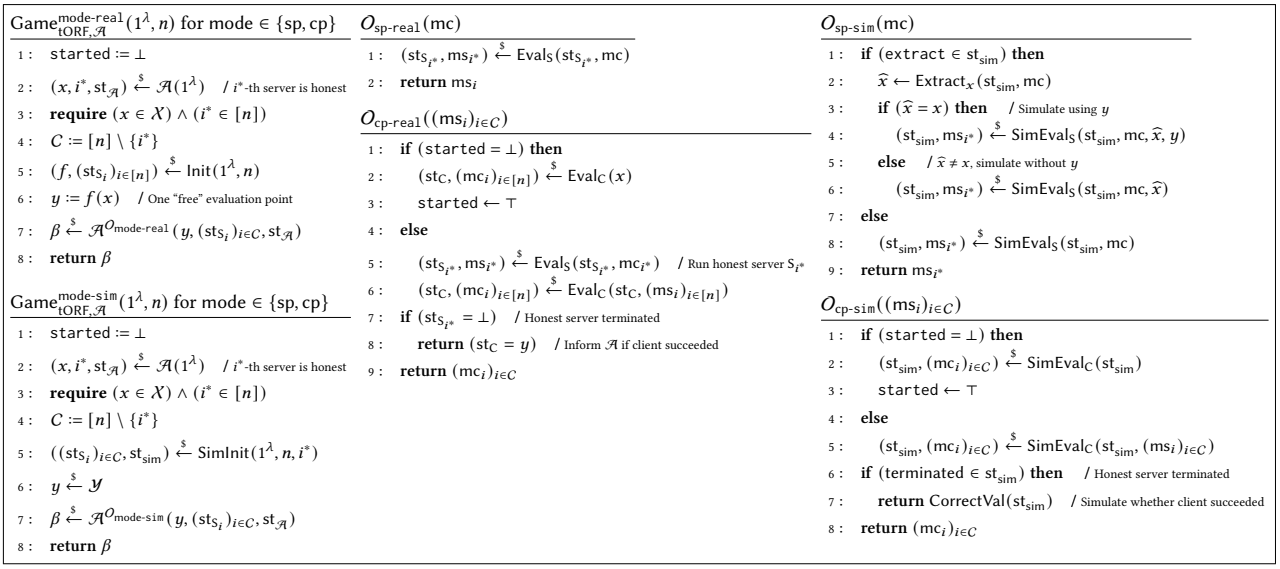

Figure 2: The real and simulated games for server and client privacy. The simulation state is assumed to include special indicator strings extract **and** terminated **in server privacy and client privacy, respectively, to perform certain tasks. We assume the string** extract **is only included in the state once, i.e., $\widehat{x}$ is extracted at most once.**

Definition 11. [Client Privacy] A one-time tORF is weak *client private* if there exist efficient simulators (SimInit, $\mathsf{SimEval}_\mathsf{C}$, CorrectVal) such that for any (possibly inefficient) adversary $\mathcal{A}$, we have

$$\Pr[\mathrm{Game}^{\text{cp-real}}_{\text{tORF},\mathcal{A}}(1^\lambda, n) = 1] \simeq_s \Pr[\mathrm{Game}^{\text{cp-sim}}_{\text{tORF},\mathcal{A}}(1^\lambda, n) = 1],$$

where the two games are depicted in Fig. 2.

Remark 1 (On Simulator SimInit)**.** *Both server and client privacy assume existence of* SimInit*, simulating the corrupted server states. In this paper, we assume the description of* SimInit *is identical for server and client privacy. This is because in a typical proof, the reduction will not know whether to rely on server or client privacy when setting up a* tORF *state. Having the same* SimInit *allows it to prepare these states so that both privacy games can be adaptively invoked.*

3.2.3 Corruption Simulatability. In server and client privacy, the adversary obtains the $n-1$ corrupted server states generated by SimInit along with one evaluation point $y = f(x)$. Corruption simulatability ensures that the state of the last honest server S_{i^*} can also be simulated consistently with this adversary's view. Note that we only consider simulatability of unused tORF states; S_{i^*}'s state need not be simulatable if it is being used in an evaluation protocol. Formally, we define corruption simulatability as follows.

Definition 12. [Corruption Simulatability] Let SimInit be defined as in server and weak client privacy and $\mathcal{D} : \mathcal{X} \to \mathbb{R}_{\geq 0}$ be an arbitrary distribution. A one-time tORF is *corruption simulatable* if there exists an efficient simulator SimSt such that for any $i^* \in [n]$, the following two distributions are identical, where $\mathcal{C} = [n] \setminus \{i^*\}$:

$$\mathcal{D}_{\mathtt{real}}(i^*) = \left\{ (x, f(x), (\mathsf{st}_{\mathsf{S}_i})_{i\in[n]}) \;\middle|\; (f, (\mathsf{st}_{\mathsf{S}_i})_{i\in[n]}) \xleftarrow{\$} \mathsf{Init}(1^\lambda, n), x \xleftarrow{\$} \mathcal{D} \right\},$$

$$\mathcal{D}_{\mathtt{sim}}(i^*) = \left\{ (x, y, (\mathsf{st}_{\mathsf{S}_i})_{i\in[n]}) \;\middle|\; \begin{array}{l} ((\mathsf{st}_{\mathsf{S}_i})_{i\in\mathcal{C}}, \mathsf{st}_{\mathsf{sim}}) \xleftarrow{\$} \mathsf{SimInit}(1^\lambda, n, i^*), \\ (x, y) \xleftarrow{\$} \mathcal{D} \times \mathcal{Y}, \mathsf{st}_{\mathsf{S}_{i^*}} \xleftarrow{\$} \mathsf{SimSt}(\mathsf{st}_{\mathsf{sim}}, x, y) \end{array} \right\}.$$

Corollary 1. *If a one-time* tORF *is corruption simulatable, the marginal distribution of $(\mathsf{st}_{\mathsf{S}_i})_{i\in[n]}$ sampled from $\mathcal{D}_{\mathtt{sim}}(i^*)$ for any $i^* \in [n]$ is independent of x.*

Proof. By definition, the marginal distributions of $(\mathsf{st}_{\mathsf{S}_i})_{i\in[n]}$ are identical for $\mathcal{D}_{\mathtt{real}}(i^*)$ and $\mathcal{D}_{\mathtt{sim}}(i^*)$. Since the former outputs $(\mathsf{st}_{\mathsf{S}_i})_{i\in[n]}$ independent of x, the claim follows. □

Comparison with UC (t)OPRF *Definitions.* Let us comment on how our game-based definition of tORF differs from existing UC definitions of (t)OPRF (e.g., [12, 45, 54, 55]). Because many UC definitions of (t)OPRF are tailored to specific constructions, they are often incomparable, and there is no universally accepted standard. As such, we focus on the high-level design principles typically seen in UC definitions for this comparison.

First, in UC definitions, client and server privacy are defined within a single ideal functionality. While we provide two independent definitions (Definitions 10 and 11) for simplicity, these are implicitly connected through a shared simulator SimInit (cf. Remark 1), making our framework consistent with UC formalizations.

Second, our definition of server privacy is mostly consistent with UC definitions. The $\mathsf{Extract}_x$ algorithm implicitly plays the role of the environment (and by extension, the adversary) specifying the evaluation point x in the UC setting. Specifically, in UC definitions, the adversary commits to x when calling the ideal evaluation functionality — our $\mathsf{Extract}_x$ algorithm mimics this feature by extracting $\widehat{x}$ from the adversary's transcript. Note that such an extraction algorithm implicitly appears in UC security proofs, as the reduction must extract the evaluation point from a real-world adversary.

On the other hand, our definition of client privacy is more limited than UC definitions in two respects. First, the adversary can only run the honest client's evaluation on the declared input x for which it received the "free" evaluation $y = f(x)$. To align this

with UC definitions, the game could allow the adversary to use a different input x' for the evaluation. Second, the adversary only learns whether the client succeeded in recovering a value y that is consistent with the free evaluation point (i.e., $(\mathsf{st}_C = y)$). We can strengthen the definition by simply returning st_C instead in the real game. Accordingly, when $\mathsf{CorrectVal}(\mathsf{st}_{\mathsf{sim}}) = 1$, the ideal game returns the free evaluation y; otherwise, it returns a random value y'. This is then consistent with the UC definition, where the adversary chooses whether the client succeeds, and if not, the ideal functionality outputs a random value to the honest client. We note that our weaker definition suffices for our application, although our construction Beaver in the next section does not satisfy such a strengthened definition.

Finally, in UC definitions, the last server can be corrupted at any point. However, in our game, server corruption is disallowed while the server is executing the evaluation protocol. Put differently, in UC definitions, intermediate states during evaluation can also be simulated, whereas in our game, they cannot. This is not just out of simplification as our protocol Beaver will not be secure under such corruption.

4 Beaver: Our One-time tORF

Our one-time tORF protocol Beaver supports the function class $\mathcal{F}$ of all degree-one affine functions over $\mathbb{Z}_p$ for a prime p. Below, we provide a bottom-up overview of its design.

Core Design Idea. The starting point of Beaver is based on a simple observation. As we only evaluate it *one time*, the base function being computed can be a 2-wise independent hash function $f(x) = A \cdot x + B$ for random coefficients $A, B \xleftarrow{\$} \mathbb{Z}_p$. In contrast, standard tOPRFs require f to be a pseudorandom function as it is evaluated on unbounded polynomially many inputs. An idea for the initialization phase is then for the client to additively secret share (A, B) as $(\llbracket A \rrbracket_i, \llbracket B \rrbracket_i)_{i\in[n]}$ and store the i-th share on to the server S_i. In the evaluation phase, the client first secret shares x and sends $\llbracket x \rrbracket_i$ to S_i. The servers can then compute $f(x)$ in a multi-party fashion, talking to each other through the client. While this theoretically works, it is deemed inefficient as $f(x)$ involves secure multiplication.

Instead, we take advantage of the fact that the client is allowed to perform *pre-computation* during the initialization phase. The client uploads Beaver triples $(\llbracket a \rrbracket_i, \llbracket b \rrbracket_i, \llbracket c \rrbracket_i)_{i\in[n]}$ such that $a \cdot b = c$ [16] on to the servers. With this, computing $f(x)$ becomes easy: after receiving $\llbracket x \rrbracket_i$ from the client, S_i sends back $\llbracket d \rrbracket_i = \llbracket A \rrbracket_i + \llbracket a \rrbracket_i$ and $\llbracket e \rrbracket_i = \llbracket x \rrbracket_i + \llbracket b \rrbracket_i$; the client computes $(d, e) = (\mathsf{Rec}((\llbracket d \rrbracket_i)_{i\in[n]}), \mathsf{Rec}((\llbracket e \rrbracket_i)_{i\in[n]}))$ and sends it to the servers; S_i replies with $\llbracket y \rrbracket_i = d \cdot \llbracket b \rrbracket_i + e \cdot \llbracket a \rrbracket_i - \llbracket c \rrbracket_i - \llbracket B \rrbracket_i$; and finally, the client computes $y = d \cdot e - \sum_{i\in[n]} \llbracket y \rrbracket_i$. A routine calculation shows that if all parties are honest, then $y = f(x)$ as desired.

Adding Security. Unfortunately, the above evaluation protocol does not have client privacy since it is not secure against *malicious* servers. As a concrete example of an online password-guessing attack, consider a malicious server S_i that replies with $\llbracket d \rrbracket_i + 1$ and $\llbracket y \rrbracket_i + \widehat{x}$ instead of $\llbracket d \rrbracket_i$ and $\llbracket y \rrbracket_i$, where $\widehat{x}$ is S_i's guess of the client input x. A routine calculation shows that the client computes $y = f(x) + (x - \widehat{x})$. As the adversary learns the bit $(f(x) = y)$ from the client privacy security game, it can check whether its guess $\widehat{x}$ was correct. While we can rely on generic MPC techniques like the GMW compiler [43] to thwart such attacks, this leads to a prohibitive increase in efficiency and achieves a much stronger security guarantee than what we require.[5]

To resolve this issue, we take advantage of the client being able to perform not only pre-computations, but also *part of the multi-party computation of* $f(x)$ during the initialization phase. In particular, notice that once the server shares are fixed, $d = A + a$ is fixed no matter what value of x is used during the evaluation protocol. Using this, we modify the protocol so that the client also gives the pre-computed d to each server during the initialization phase. In the evaluation phase, each server simply sends the stored d_i to the client as opposed to shares of d (i.e., $\llbracket d \rrbracket_i$). If all d_i are not the same, the client notices some servers are being malicious and aborts. Otherwise, since we assume there exists one honest server, the agreed value d must be the one that the client created during the initialization phase. It can be checked that this modification thwarts the above attack. In fact, this simple idea turns out to be powerful enough to prove client privacy.

Removing Beaver Triples. Counterintuitively, we can remove the Beaver triples and optimize the protocol. Notice that $\llbracket B \rrbracket_i$ is only used to mask $\llbracket y \rrbracket_i$, meaning that we can let B "absorb" $c = a \cdot b$. We can thus redefine the function as $f(x) = A \cdot x + B - a \cdot b$ and run the same protocol except that $\llbracket y \rrbracket_i$ is now computed as $d \cdot \llbracket b \rrbracket_i + e \cdot \llbracket a \rrbracket_i - \llbracket B \rrbracket_i$. Moreover, due to the idea of using d above, notice $\llbracket A \rrbracket_i$ is no longer used anywhere. As such, we can further remove $\llbracket A \rrbracket_i$ from the server state and rewrite f as $f(x) = (d - a) \cdot x + B - a \cdot b$. At this point, the server state is just a random string.[6] Below we describe the concrete construction of our tOPRF, Beaver.[7]

4.1 Construction of Beaver

The initialization algorithm $\mathsf{Init}(1^\lambda, n)$, run by the client, samples $a, b, B \xleftarrow{\$} \mathbb{Z}_p$, and secret shares $(\llbracket a \rrbracket_i)_{i\in[n]} \xleftarrow{\$} \mathsf{Share}(a, n)$, $(\llbracket b \rrbracket_i)_{i\in[n]} \xleftarrow{\$} \mathsf{Share}(b, n)$, and $\llbracket B \rrbracket_i \xleftarrow{\$} \mathsf{Share}(B, n)$. It then defines the function

$$f(x) := a \cdot x - B + a \cdot b,$$

and sets the server state $\mathsf{st}_{\mathsf{S}_i}$ as

$$\forall i \in [n], \quad \mathsf{st}_{\mathsf{S}_i} := (\llbracket B \rrbracket_i, \llbracket a \rrbracket_i, \llbracket b \rrbracket_i).$$

The server states do not change throughout the protocol and get deleted after the last server message. The evaluation protocol is given in Fig. 3.

Remark 2 (L-time tORF). *One can easily extend* Beaver *to an L-time* tORF *for any polynomial $L = \mathsf{poly}(\lambda)$ by scaling the communication cost and server states by L. Concretely, one simply computes an L-degree polynomial $f(x) = \sum_{t\in[0:L-1]} A_t \cdot x^t$. The protocol is modified*

[5] Recall that GMW [43] employs zero-knowledge proofs to prove correct computation of each step. This ensures that if the client completes the protocol, the output is the correct value. This is a property stronger than what we require from client privacy.

[6] While technically our protocol no longer relies on Beaver triples, we still name it Beaver because the construction is inspired from them.

[7] While our original construction uniformly samples d, we can optimize it by setting $d = 0$ since we do not leverage the randomness of d to prove the security. This optimization was suggested by a reviewer at ACM CCS. We focus on this optimized scheme hereafter.

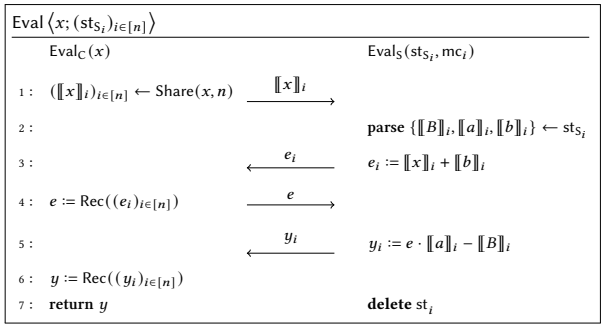

Figure 3: The interactive evaluation protocol of Beaver**.**

so that the client starts by sending secret shares $(\llbracket x^t\rrbracket_i)_{t\in[0:L-1]}$ to the i-th server, and the server states are modified appropriately, allowing to compute $A_t \cdot x^t$ efficiently as per Beaver.

4.2 Security of Beaver

In this section, we formally prove that Beaver is corruption simulatable, server private, and weak client private.

4.2.1 Correctness.

Theorem 1. Beaver *is correct.*

Proof. For honestly generated protocol messages, we have

$$\sum_{i\in[n]} y_i = \sum_{i\in[n]} (e\cdot\llbracket a\rrbracket_i - \llbracket B\rrbracket_i) = (x+b)\cdot a - B = a\cdot x - B + a\cdot b.$$

Thus the client eventually obtains $f(x)$. This completes the proof. □

4.2.2 Corruption Simulatablility.

We first provide the simulated initialization algorithm $\mathsf{SimInit}$ that is also used by both server and weak client privacy. $\mathsf{SimInit}(1^\lambda, n, i^*)$ is identical to Init except that it no longer computes the honest server S_{i^*}'s state. Namely, it samples $\llbracket B\rrbracket_i, \llbracket a\rrbracket_i, \llbracket b\rrbracket_i \xleftarrow{\$} \mathbb{Z}_p$ for $i \in C := [n]\setminus\{i^*\}$, computes $(\llbracket B\rrbracket_C, \llbracket a\rrbracket_C, \llbracket b\rrbracket_C) := \sum_{i\in C}(\llbracket B\rrbracket_i, \llbracket a\rrbracket_i, \llbracket b\rrbracket_i)$, and sets the corrupted server states as

$$\forall i \in C, \quad \mathsf{st}_{\mathsf{S}_i} := (\llbracket B\rrbracket_i, \llbracket a\rrbracket_i, \llbracket b\rrbracket_i).$$

It then sets the simulation state as

$$\mathsf{st}_{\mathsf{sim}} := (\llbracket B\rrbracket_C, \llbracket a\rrbracket_C, \llbracket b\rrbracket_C).$$

Theorem 2. Beaver *is corruption simulatable.*

Proof. Simulator SimSt is depicted in Fig. 4. We show that, with respect to $\mathsf{SimInit}$ described above and this simulator, we have $\mathcal{D}_{\mathtt{real}}(i^*) = \mathcal{D}_{\mathtt{sim}}(i^*)$.

In both cases, $\llbracket a\rrbracket_{i^*}$, $\llbracket b\rrbracket_{i^*}$, $\llbracket B\rrbracket_i$, $\llbracket a\rrbracket_i$, and $\llbracket b\rrbracket_i$ for all $i \in C$ are sampled uniformly at random from $\mathbb{Z}_p$, and x is sampled from $\mathcal{D}$. Thus, the distributions of x, $(\mathsf{st}_{\mathsf{S}_i})_{i\in C}$ and $(\llbracket a\rrbracket_{i^*}, \llbracket b\rrbracket_{i^*})$ are identical. It remains to show that the distributions of $(f(x), \llbracket B\rrbracket_{i^*})$ in the real case and $(y, \llbracket B\rrbracket_{i^*})$ in the simulated case are identical for fixed x, $(\mathsf{st}_{\mathsf{S}_i})_{i\in C}$, and $(\llbracket a\rrbracket_{i^*}, \llbracket b\rrbracket_{i^*})$. To this end we define $b := \sum_{i\in[n]}\llbracket b\rrbracket_i$, $a := \sum_{i\in[n]}\llbracket a\rrbracket_i$, and $\llbracket B\rrbracket_C := \sum_{i\in C}\llbracket B\rrbracket_i$ and denote the tuple $(x, a, b, \llbracket B\rrbracket_C)$ by D.

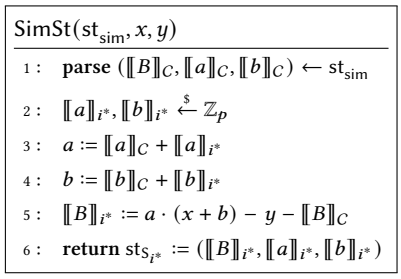

Figure 4: Simulator for corruption simulatability. For readability, $C := [n]\setminus\{i^*\}$.

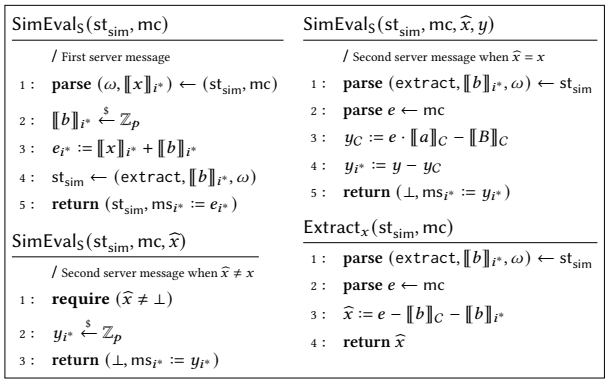

Figure 5: Simulators for server privacy. For readability, $C := [n]\setminus\{i^*\}$ and $\omega := (\llbracket B\rrbracket_C, \llbracket a\rrbracket_C, \llbracket b\rrbracket_C)$. If $\mathsf{SimEval}_{\mathsf{S}}$ and $\mathsf{Extract}_x$ fail in parsing mc **and** $\mathsf{st}_{\mathsf{sim}}$**, it returns $(\bot, \bot)$ and $\bot$, respectively.**

In the real case, $\llbracket B\rrbracket_{i^*}$ is uniformly distributed over $\mathbb{Z}_p$ conditioned on D, and $f(x)$ is uniquely determined from $\llbracket B\rrbracket_{i^*}$ so that $f(x) = a\cdot x - \llbracket B\rrbracket_C + \llbracket B\rrbracket_{i^*} + a\cdot b$. Thus, in this case, $(f(x), \llbracket B\rrbracket_{i^*})$ is uniformly distributed over $\mathbb{Z}_p^2$ conditioned on the event that above equation holds. In the simulated case, y is sampled uniformly from $\mathbb{Z}_p$, and after that, $\llbracket B\rrbracket_{i^*}$ is computed by SimSt so that $y = a\cdot x - \llbracket B\rrbracket_C - \llbracket B\rrbracket_{i^*} + a\cdot b$. Thus $(y, \llbracket B\rrbracket_{i^*})$ is also uniformly distributed over $\mathbb{Z}_p^2$ conditioned on the same event holds as in the real case. Therefore, the distributions of them are identical. This completes the proof. □

4.2.3 Server Privacy.

Theorem 3. Beaver *is server private.*

Proof. The simulators $\mathsf{SimEval}_{\mathsf{S}}$ and $\mathsf{Extract}_x$ for Beaver are depicted in Fig. 5. In the initial phase of both games, an adversary $\mathcal{A}$ obtains $(\mathsf{st}_{\mathsf{S}_i})_{i\in C}$ and y. By Theorem 2, the distributions of $(\mathsf{st}_{\mathsf{S}_i})_{i\in C}$ and y in both games are identical, taking the marginal distribution of them. In the following we argue that all responses to oracle queries by the adversary are equally distributed in the real and in the simulated game. To this end, we consider fixed $(\mathsf{st}_{\mathsf{S}_i})_{i\in C}$ and (x, y). Throughout the proof we use $(\llbracket B\rrbracket_C, \llbracket a\rrbracket_C, \llbracket b\rrbracket_C) := \sum_{i\in C}(\llbracket B\rrbracket_i, \llbracket a\rrbracket_i, \llbracket b\rrbracket_i)$, and define

$$\bar{e} := x + \llbracket b\rrbracket_C + \llbracket b\rrbracket_{i^*},$$

$$\bar{y}_C := \bar{e} \cdot [\![a]\!]_C - [\![B]\!]_C, \text{ and}$$
$$\bar{y}_{i^*} := y - \bar{y}_C.$$

We now evaluate the distribution of $([\![B]\!]_{i^*}, [\![a]\!]_{i^*}, [\![b]\!]_{i^*})$ in the real game conditioned on $(\mathsf{st}_{\mathsf{S}_i})_{i\in C}$ and (x, y). Giving y to $\mathcal{A}$ leaks the information that f satisfies $f(x) = a \cdot x - B + a \cdot b = y$, which by a couple simple transformations is equivalent to

$$\bar{e} \cdot [\![a]\!]_{i^*} - [\![B]\!]_{i^*} = \bar{y}_{i^*}. \tag{1}$$

The tuple $([\![B]\!]_{i^*}, [\![a]\!]_{i^*}, [\![b]\!]_{i^*})$ is sampled uniformly at random from $\mathbb{Z}_p^3$. Thus, when $\mathcal{A}$ is given $(\mathsf{st}_{\mathsf{S}_i})_{i\in C}$ and y then $([\![B]\!]_{i^*}, [\![a]\!]_{i^*}, [\![b]\!]_{i^*})$ is uniformly distributed over $\mathbb{Z}_p^3$ conditioned on the event that Eq. (1) holds. Since for arbitrary $[\![a]\!]_{i^*}$ and $[\![b]\!]_{i^*}$ there exists a unique $[\![B]\!]_{i^*}$ satisfying Eq. (1), this is equivalent to sampling $[\![a]\!]_{i^*}$ and $[\![b]\!]_{i^*}$ uniformly at random from $\mathbb{Z}_p$ and deterministically computing $[\![B]\!]_{i^*}$ from them.

We now evaluate the distributions of the responses to calls to oracle O_{sp}. We first focus on the response to the first call, which is e_{i^*}. In the real game, since $[\![b]\!]_{i^*}$ is uniformly random over $\mathbb{Z}_p$, so is $e_{i^*} = [\![x]\!]_{i^*} + [\![b]\!]_{i^*}$. The same holds for e_{i^*} in the simulated game, since $[\![b]\!]_{i^*}$ is sampled uniformly from $\mathbb{Z}_p$ by $\mathsf{SimEval}_\mathsf{S}$ and e_{i^*} is computed as in the real game. Thus, $\mathcal{A}$'s view after the first oracle call is identical in both games.

Next, we focus on the response y_{i^*} to the second oracle call. Since $\mathcal{A}$ can choose $[\![x]\!]_{i^*}$ in the first round, $[\![b]\!]_{i^*}$ is uniquely determined from $\mathcal{A}$'s view when it receives e_{i^*}. As a consequence, $\bar{e}$ and $\bar{y}_{i^*}$ are also fixed at the same point. In the real game, after $\mathcal{A}$ queries e, the oracle returns $y_{i^*} = e \cdot [\![a]\!]_{i^*} - [\![B]\!]_{i^*}$. If e is equal to $\bar{e}$, then by Eq. (1) we have $y_{i^*} = \bar{y}_{i^*}$. In the case $e \neq \bar{e}$, let α be an element in $\mathbb{Z}_p^*$ such that $e = \bar{e} + \alpha$. Then y_{i^*} satisfies $y_{i^*} = \bar{y}_{i^*} + \alpha \cdot [\![a]\!]_{i^*}$. Since $\bar{y}_{i^*}$ is fixed and $[\![a]\!]_{i^*}$ is uniformly distributed over $\mathbb{Z}_p$, we obtain that y_{i^*} is uniformly distributed over $\mathbb{Z}_p$ for $e \neq \bar{e}$. In the simulated game, because $\mathsf{Extract}_x$ computes $\widehat{x}$ as $\widehat{x} = e - [\![b]\!]_C - [\![b]\!]_{i^*}$, $e = \bar{e}$ holds exactly if $\widehat{x} = x$. Then y_{i^*} computed by $\mathsf{SimEval}_\mathsf{S}$ in the case of $\widehat{x} = x$ is equal to $\bar{y}_{i^*}$. $\mathsf{SimEval}_\mathsf{S}$ in the case of $\widehat{x} \neq x$ outputs y_{i^*} by choosing uniformly at random from $\mathbb{Z}_p$. Thus the distributions of y_{i^*} in both games are identical conditioned on the view of $\mathcal{A}$ so far.

Combining all arguments above, $\mathcal{A}$'s view is identical in both games. This completes the proof. □

4.2.4 *Weak Client Privacy.*

Theorem 4. Beaver *is weak client private.*

Proof. The simulators $\mathsf{SimEval}_\mathsf{C}$ and CorrectVal for Beaver are depicted in Fig. 6. As in the proof of Theorem 3, we obtain that the distributions of $(\mathsf{st}_{\mathsf{S}_i})_{i\in C}$ and y in both games are identical. Also, in the real game, conditioned on $(\mathsf{st}_{\mathsf{S}_i})_{i\in C}$ and (x, y), tuple $([\![a]\!]_{i^*}, [\![b]\!]_{i^*})$ is distributed uniformly random over $\mathbb{Z}_p^2$, and by Eq. (1) $[\![B]\!]_{i^*}$ is uniquely determined from $[\![a]\!]_{i^*}$ and $[\![b]\!]_{i^*}$. Moreover, we use the same notations. In the following we analyze the distribution of responses of oracle calls to O_{cp} conditioned on $(\mathsf{st}_{\mathsf{S}_i})_{i\in C}$ and (x, y) for both games.

The response $([\![x]\!]_i)_{i\in C}$ to the first call is uniformly random over $\mathbb{Z}_p^{|C|}$ in both cases by the security of n-out-of-n additive secret sharing. As response to the second query, in the real game, the oracle returns $e = e_{i^*} + \sum_{i\in C} e_i$ where $e_{i^*} = [\![x]\!]_{i^*} + [\![b]\!]_{i^*}$. Since $[\![b]\!]_{i^*}$, conditioned on $\mathcal{A}$'s view up to this point, is distributed uniformly random in $\mathbb{Z}_p$, so is e_{i^*}. In the simulated game $\mathsf{SimEval}_\mathsf{C}$ samples e_{i^*} uniformly at random and computes $e = e_{i^*} + e_C$ where $e_C = \sum_{i\in C} e_i$. Therefore, responses to the first two queries to O_{cp} are distributed identically in both games.

To complete the proof, we analyze the distributions of the response to the final oracle query conditioned on $\mathcal{A}$'s view up to the third oracle query which has the form $(\mathsf{ms}_i)_{i\in C} = (y_i)_{i\in C}$. We denote this view by $\mathsf{view}_{<b}$. Note that, conditioned on $\mathsf{view}_{<b}$ the share $[\![x]\!]_{i^*}$ has been fixed to $x - \sum_{i\in C}[\![x]\!]_i$ after the first oracle response, and $[\![e]\!]_{i^*}$ to $e - \sum_{i\in C} e_i$ after the second. Similarly, $[\![b]\!]_{i^*}$ is fixed to $[\![e]\!]_{i^*} - [\![x]\!]_{i^*}$ in the second reply.

In the real game, $e = \bar{e}$ is equivalent to

$$\sum_{i\in C} e_i = \sum_{i\in C} [\![x]\!]_i + [\![b]\!]_i. \tag{2}$$

This, in turn, is by Eq. (1) equivalent to $y_{i^*} = \bar{y}_{i^*}$, where y_{i^*} is the value generated by Eval_S. Thus, if Eq. (2) holds, then since Eval_C computes its final output as $\sum_{i\in[n]} y_i$, oracle O_{cp} by definition of $\bar{y}_{i^*}$ returns 1 exactly if

$$\sum_{i\in C} y_i = \bar{y}_C. \tag{3}$$

On the other hand, if Eq. (2) does not hold, y_{i^*} is of the form $y_{i^*} = \bar{y_{i^*}} + \alpha \cdot [\![a]\!]_{i^*}$ where $\alpha = e - \bar{e} \neq 0$. Because $[\![a]\!]_{i^*}$ conditioned on $\mathsf{view}_{<b}$ is still uniformly distributed over $\mathbb{Z}_p$, Eval_C's output y' is uniformly distributed over $\mathbb{Z}_p$ as well. Summing up, if Eq. (2) does not hold, then $y' = y$, i.e., the oracle returns 1, with probability $1/p$. We denote this event by $\mathsf{E}_{\mathsf{bad}}$.

In the simulated game, CorrectVal outputs 1 only when $\mathsf{view}_{<b}$ satisfies both Eq. (2) and Eq. (3). Otherwise, it always returns 0. Therefore, as long as $\mathsf{E}_{\mathsf{bad}}$ does not happen, the distributions of O_{cp}'s response to the third oracle call are identical. Combining all arguments, we have

$$|\Pr[\mathsf{Game}^{\mathtt{cp\text{-}real}}_{\mathsf{tORF},\mathcal{A}}(1^\lambda, n) = 1] - \Pr[\mathsf{Game}^{\mathtt{cp\text{-}sim}}_{\mathsf{tORF},\mathcal{A}}(1^\lambda, n) = 1]| \leq 1/p.$$

□

$\mathsf{SimEval}_\mathsf{C}(\mathsf{st}_{\mathsf{sim}})$

/ First client messsage
1 : **parse** $\omega \leftarrow \mathsf{st}_{\mathsf{sim}}$
2 : **for** $i \in C$ **do**
3 : $\quad [\![x]\!]_i \xleftarrow{\$} \mathbb{Z}_p$
4 : $[\![x]\!]_C := \sum_{i\in C} [\![x]\!]_i$
5 : $\mathsf{st}_{\mathsf{sim}} \leftarrow (1, [\![x]\!]_C, \omega)$
6 : $(\mathsf{mc}_i)_{i\in C} := ([\![x]\!]_i)_{i\in C}$
7 : **return** $(\mathsf{st}_{\mathsf{sim}}, (\mathsf{mc}_i)_{i\in C})$

$\mathsf{CorrectVal}(\mathsf{st}_{\mathsf{sim}})$

1 : **parse** $(\mathsf{terminated}, e_C, y_C, e'_C, y'_C) \leftarrow \mathsf{st}_{\mathsf{sim}}$
2 : **return** $((e_C, y_C) = (e'_C, y'_C))$

$\mathsf{SimEval}_\mathsf{C}(\mathsf{st}_{\mathsf{sim}}, (\mathsf{ms}_i)_{i\in C})$

1 : **parse** $(b, \star) \leftarrow \mathsf{st}_{\mathsf{sim}}$ / $b \in \{1, 2\}, \star \in \{0, 1\}^*$
2 : **if** $(b = 1)$ **then** / Second client message
3 : $\quad$ **parse** $(1, [\![x]\!]_C, \omega) \leftarrow \mathsf{st}_{\mathsf{sim}}$
4 : $\quad$ **parse** $(e_i)_{i\in C} \leftarrow (\mathsf{ms}_i)_{i\in C}$
5 : $\quad e_{i^*} \xleftarrow{\$} \mathbb{Z}_p$
6 : $\quad e_C := \sum_{i\in C} e_i$
7 : $\quad e := e_C + e_{i^*}$
8 : $\quad e'_C := [\![x]\!]_C + [\![b]\!]_C$
9 : $\quad y'_C := e \cdot [\![a]\!]_C - [\![B]\!]_C$
10 : $\quad \mathsf{st}_{\mathsf{sim}} \leftarrow (2, e_C, e'_C, y'_C)$
11 : $\quad$ **return** $(\mathsf{st}_{\mathsf{sim}}, (\mathsf{mc}_i := e)_{i\in C})$
12 : **else** / $b = 2$, Final client output
13 : $\quad$ **parse** $(2, e_C, e'_C, y'_C) \leftarrow \mathsf{st}_{\mathsf{sim}}$
14 : $\quad$ **parse** $(y_i)_{i\in C} \leftarrow (\mathsf{ms}_i)_{i\in C}$
15 : $\quad y_C := \sum_{i\in C} y_i$
16 : $\quad \mathsf{st}_{\mathsf{sim}} \leftarrow (\mathsf{terminated}, e_C, y_C, e'_C, y'_C)$
17 : $\quad$ **return** $(\mathsf{st}_{\mathsf{sim}}, \perp)$

Figure 6: Simulators for client privacy. For readability, $C := [n]\backslash\{i^*\}$ **and** $\omega := ([\![B]\!]_C, [\![a]\!]_C, [\![b]\!]_C)$**. If** $\mathsf{SimEval}_\mathsf{C}$ **fails in parsing** $(\mathsf{ms}_i)_{i\in C}$ **and** $\mathsf{st}_{\mathsf{sim}}$**, it returns** $(\perp, \perp)$

5 BeaverDam: A Threshold Password-Protected Key Recovery Protocol

We present a generic construction of a *threshold password-protected key recovery* (tPPKR) protocol called BeaverDam based on one-time tORFs and a secure channel between the client and servers.

5.1 Syntax and Correctness

A tPPKR protocol involves a client interacting with a collection of servers through a two-phase process. In the *registration* phase, the client stores a secret key K across the servers, secured by a password pwd. Later, in the *key recovery* phase, the client recovers K using only that pwd; notably, the protocol is designed so that the client requires no additional state beyond the remembered pwd. Our formalization assumes the existence of a PKI for the servers, as modeled by the $\mathsf{KeyGen}_\mathsf{S}$ algorithm.

Definition 13 (Threshold PPKR)**.** A *threshold password-protected key recovery* (tPPKR) protocol consists of the following PPT algorithms where $\mathcal{K}$ and $\mathcal{D}$ are the secret key and dictionary spaces, respectively.

- $\mathsf{KeyGen}_\mathsf{S}(1^\lambda) \to (\mathsf{pk}_\mathsf{S}, \mathsf{sk}_\mathsf{S})$: The server key generation algorithm, on input the security parameter 1^λ, outputs a public and secret key pair $(\mathsf{pk}_\mathsf{S}, \mathsf{sk}_\mathsf{S})$.
- $\mathsf{Register}(\mathsf{pwd}, \mathsf{K}, (\mathsf{pk}_{\mathsf{S}_i})_{i\in[n]}) \to (\mathsf{st}_{\mathsf{S}_i})_{i\in[n]}$: The registration algorithm, on input a password $\mathsf{pwd} \in \mathcal{D}$, a secret key $\mathsf{K} \in \mathcal{K}$, and n server public keys, outputs initial server states $(\mathsf{st}_{\mathsf{S}_i})_{i\in[n]}$.
- $\mathsf{Recover}\langle(\mathsf{pwd}', (\mathsf{pk}_{\mathsf{S}_i})_{i\in[n]}); (\mathsf{sk}_{\mathsf{S}_i}, \mathsf{st}_{\mathsf{S}_i})_{i\in[n]}\rangle \to \langle \mathsf{K}; (\mathsf{st}_{\mathsf{S}_i})_{i\in[n]}\rangle$: An interactive key recovery protocol between a client with input a password $\mathsf{pwd}' \in \mathcal{D}$ and n server public keys, and n servers with secret keys and states. The client outputs $\mathsf{K} = \perp$ indicating failure, or $\mathsf{K} \in \mathcal{K}$ indicating successful recovery. The server outputs an updated state $(\mathsf{st}_{\mathsf{S}_i})_{i\in[n]}$.

Similarly to the case of tORF, when we explicitly write down the interactive key recovery protocol, we assume the client and server communicate in a ping-pong manner. Concretely, $(\mathsf{st}_\mathsf{C}, (\mathsf{mc}_i)_{i\in[n]}) \xleftarrow{\$} \mathsf{Recover}_\mathsf{C}(\mathsf{st}_\mathsf{C}, (\mathsf{ms}_i)_{i\in[n]})$ followed by all $i \in [n]$ servers running $(\mathsf{st}_{\mathsf{S}_i}, \mathsf{ms}_i) \xleftarrow{\$} \mathsf{Recover}_\mathsf{S}(\mathsf{sk}_{\mathsf{S}_i}, \mathsf{st}_{\mathsf{S}_i}, \mathsf{mc}_i)$, and so on, where mc_i and ms_i denote the protocol messages sent by the client and server S_i, respectively. We also assume the client sends the first and last protocol messages unless the protocol ends prematurely. Concretely, the protocol starts with the client executing $(\mathsf{st}_\mathsf{C}, (\mathsf{mc}_i)_{i\in[n]}) \xleftarrow{\$} \mathsf{Recover}_\mathsf{C}(\mathsf{pwd}', (\mathsf{pk}_{\mathsf{S}_i})_{i\in[n]})$, creating a protocol specific state st_C, and the last execution of $\mathsf{Recover}_\mathsf{S}$ outputs $\mathsf{ms}_i = \perp$.

A tPPKR protocol is required to be *correct*, that is, a client can recover the registered key using the correct password.

Definition 14 (Correctness)**.** A tPPKR protocol is *correct* if for all $i \in [n]$, $(\mathsf{pk}_{\mathsf{S}_i}, \mathsf{sk}_{\mathsf{S}_i}) \in \mathsf{KeyGen}_\mathsf{S}(1^\lambda)$, $\mathsf{pwd} \in \mathcal{D}$, and $\mathsf{K} \in \mathcal{K}$, we have

$$\mathsf{Recover}\langle(\mathsf{pwd}, (\mathsf{pk}_{\mathsf{S}_i})_{i\in[n]}); (\mathsf{sk}_{\mathsf{S}_i}, \mathsf{st}_{\mathsf{S}_i})_{i\in[n]}\rangle = \langle \mathsf{K}; (\mathsf{st}_{\mathsf{S}_i})_{i\in[n]}\rangle,$$

where $\mathsf{st}_{\mathsf{S}_i}$ is either initially generated through the Register protocol or has since been updated through Recover.

Remark 3 (Rate Limiting)**.** *It is standard practice for servers to limit the number of failed password attempts. That is, the* Recover *protocol always outputs $\perp$ after $c > 0$ unsuccessful password tries. For instance, Signal [64] and WhatsApp [1] use $c = 10$. The servers can further decide what to do on a successful client attempt: they may reset the available tries to c [1], or set a hard limit on the number of attempts regardless of the client succeeding [28, 64]. We choose the more general former setting as the latter can be implemented easily assuming the former.*

5.2 Security

We require a tPPKR protocol to satisfy *semi-strong augmented security* against adversaries performing offline corruptions. By offline corruption, we mean the adversary may only corrupt a server when it is not currently executing the key recovery protocol; we believe this restriction to be reasonable, as key recovery is an infrequent operation and its execution time is negligible compared to the duration of key storage. Semi-strong augmented security ensures the registered secret key K remains indistinguishable from random across three corruption scenarios.

First, *as long as* at least one server S_{i^*} remains uncompromised, security is only vulnerable to an inherent password-guessing *online attack*. This is a "baseline" security notion for any tPPKR protocol. Specifically, if an adversary executes the Recover protocol with S_{i^*} for $Q_{\mathtt{online}}$ sessions, the advantage is bounded by $Q_{\mathtt{online}}/|\mathcal{D}|$.

Second, after corrupting *all* servers, an offline dictionary attack is still required to break security. Combined with the above baseline security notion, this is the standard definition of *augmented* security. Concretely, in the *classical* setting, we define the tPPKR protocol relative to a random oracle H and require the adversary's advantage to scale proportionally to $Q^\mathsf{H}/|\mathcal{D}|$, where Q^H is the number of classical random oracle queries.[8] This bound is optimal; upon full server compromise, the adversary can locally simulate the Recover protocol for every password in $\mathcal{D}$ to check for the correct key K.

Third, we require pre-computation to become ineffective if the client does not perform key recovery after full server corruption. Combined with the baseline security notion, we call this *strong-at-rest* augmented security. In this case, the adversary's advantage scales only with $Q^\mathsf{H}_{\mathsf{offline}}/|\mathcal{D}|$, where $Q^\mathsf{H}_{\mathsf{offline}}$ is the number of random oracle queries *after* full compromise. While this is strictly weaker than the *strong* augmented security often found in the literature — which maintains this bound even if the client interacts with corrupted servers, and allows for corruptions to occur at any time — it represents a practical compromise for key recovery protocols that are rarely executed.

To summarize, semi-strong augmented security ensures that it is both augmented *and* strong-at-rest augmented secure.

Finally, our model accounts for *quantum* adversaries capable of running Grover's search to speed up the offline dictionary attack. Accordingly, we adjust the above bounds to $C \cdot Q^\mathsf{H}/\sqrt{|\mathcal{D}|}$ and $C \cdot Q^\mathsf{H}_{\mathsf{offline}}/\sqrt{|\mathcal{D}|}$ for some small constant $C \le 5$ (cf.[9]).

Formally, we define semi-strong augmented security as follows.

Definition 15 (Semi-Strong Augmented Security)**.** A tPPKR protocol with dictionary space $\mathcal{D}$ is *semi-strong augmented secure* if

[8] To formally define augmented security, the adversary must be granted black-box access to some primitive Π, allowing to quantify the amount of work the adversary performs (i.e., an ideal primitive model [42]). We follow the standard convention of using a random oracle for this purpose.

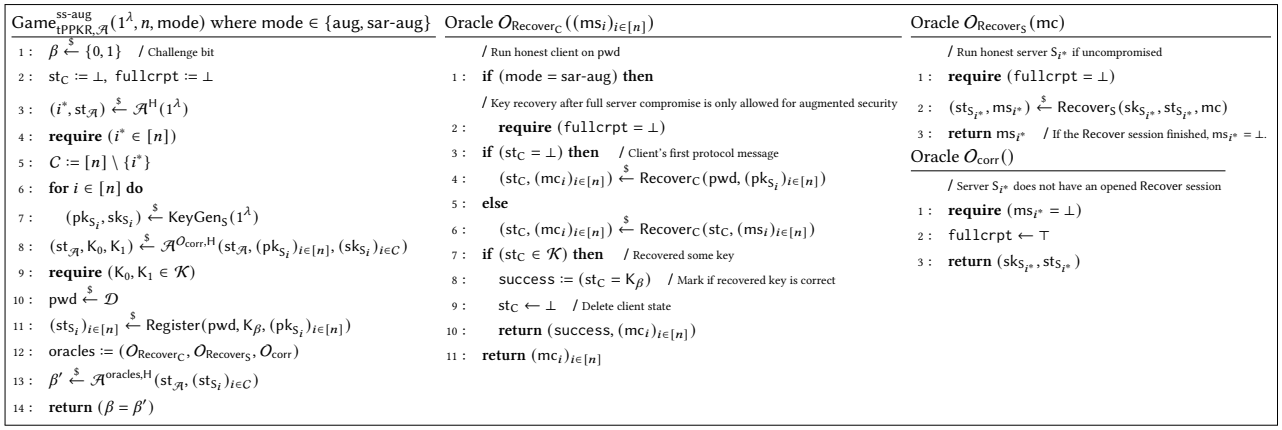

Figure 7: Strong-at-rest augmented security game for tPPKR, where H is a random oracle.

for any efficient adversary $\mathcal{A}$ making at most Q_{online} queries to oracle O_{Recover_S}, $Q^{\mathsf{H}}_{\text{pre}}$ and $Q^{\mathsf{H}}_{\text{post}}$ queries to random oracle H before and after full server compromise, we have the following for mode $\in \{\text{aug}, \text{sar-aug}\}$:

$$\left|2 \cdot \Pr\left[\mathsf{Game}^{\text{ss-aug}}_{\text{tPPKR},\mathcal{A}}(1^\lambda, n, \text{mode}) = 1\right] - 1\right| = \frac{Q_{\text{online}}}{|\mathcal{D}|} + \begin{pmatrix} \frac{b_{\text{mode}} \cdot Q^{\mathsf{H}}_{\text{pre}} + Q^{\mathsf{H}}_{\text{post}}}{|\mathcal{D}|} & \text{if classical ROM} \\ \frac{C \cdot (b_{\text{mode}} \cdot Q^{\mathsf{H}}_{\text{pre}} + Q^{\mathsf{H}}_{\text{post}})}{\sqrt{|\mathcal{D}|}} & \text{if QROM} \end{pmatrix} + \mathsf{negl}(\lambda),$$

where the game is defined in Fig. 7, C is some small universal constant, and b_{mode} is an indicator bit such that $b_{\text{mode}} = 1$ if mode = aug and 0 otherwise.

Above, notice $Q^{\mathsf{H}}_{\text{pre}}$ (i.e., pre-computation) has no impact if the adversary is mounting an attack against strong-at-rest augmented security (i.e., mode = sar-aug). Also, the baseline security notion is implicitly captured since we have $b_{\text{mode}} = 0$ and $Q^{\mathsf{H}}_{\text{post}} = 0$ when mode = sar-aug and the adversary does not make use of O_{corr}. Lastly, in practice, the bound $\frac{Q_{\text{online}}}{|\mathcal{D}|}$ should be notably better due to rate limiting (cf. Remark 3). This is because an adversary that performs $c - 1$ unsuccessful online password guesses must wait for the client to successfully run key recovery before being able to continue with its online attack.

Comparison with UC (t)PPKR *Definitions.* Let us comment on how our game-based definition of (t)PPKR differs from the existing UC definitions, e.g., [23, 34, 53, 54].[9] As there is no single, standardized UC definition for tPPKR, we focus only on the core differences.

First, we do not define *soundness*, which would guarantee that the client cannot recover any key other than the one registered unless the adversary guesses the password correctly. Currently, our definition allows the client to recover a key that was not registered, corresponding to success $= \bot$ (i.e., $\mathsf{st}_C \neq \mathsf{K}_\beta$). To be consistent with UC definitions, we can alter the game so that the adversary wins if the client terminates with a state $\mathsf{st}_C \neq \mathsf{K}_\beta$. We note that our construction in Section 5.3 should satisfy such a soundness

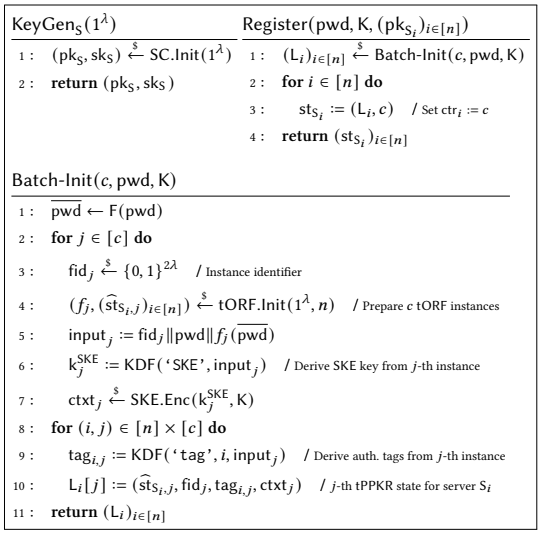

Figure 8: BeaverDam**'s server-key generation, registration, and helper subroutine** Batch-Init **algorithms.**

definition because the underlying SKE scheme is robust and correct, meaning a ciphertext can only be decrypted to a unique message. Further, we note that since ours is an indistinguishability type definition, as opposed to a simulation type UC definition, the game never returns the key to the adversary.

Second, the client is assumed to use the same password for both registration and the key recovery protocol. Notably, unlike existing UC definitions, our current Definition 15 does not capture scenarios where the client uses a password pwd′ that is "close" to pwd during key recovery; any password that is independent from pwd is captured as it can be simulated by the adversary. One way to formally capture this in our game is to additionally endow the adversary with the capability to run key recovery with an arbitrarily "shifted' client password. This can be implemented, for instance, by an adversary-specified efficiently computable function $f : \mathcal{D} \to \mathcal{D}$, e.g., if $\mathcal{D} = \{0,1\}^*$, f can model bit-flips. While we leave this for

[9] Note that tPPKR is also known as threshold password-protected secret sharing and threshold password-authenticated secret sharing.

future work, the security proof for our construction in Section 5.3 seems to naturally extend to such a definition.

Third, the last server can only be corrupted while it has no open Recover session. This is in contrast to UC definitions that allow corruption to occur even midway a Recover session. Put differently, this means that our definition does not guarantee security in case the server leaks intermediate states. This limitation is inherited from our tORF definition (see Section 3.2), and therefore, can be lifted assuming a tORF that allows for stronger corruption simulatability.

Lastly, our notion of strong-at-rest augmented security is novel to our work and can naturally be ported to existing UC definitions. Furthermore, considering that our model operates in a PKI setting, we can make the definition more granular. In our current definition, the adversary obtains both server i^*'s secret key $\mathsf{sk}_{\mathsf{S}_{i^*}}$ and state $\mathsf{st}_{\mathsf{S}_{i^*}}$ upon corruption, and cannot make further key recovery queries. However, given that $\mathsf{sk}_{\mathsf{S}_{i^*}}$ stems from the PKI, it is also natural to consider a setting where only $\mathsf{st}_{\mathsf{S}_{i^*}}$ leaks while still allowing the adversary to perform key recovery queries after full server corruption. Our construction in Section 5.3 conceivably satisfies such a definition, since $\mathsf{sk}_{\mathsf{S}_{i^*}}$ is only used to establish a secure channel with the client.

5.3 BeaverDam: Our tPPKR Protocol

We present our generic construction for c rate-limited tPPKR from a one-time tORF, called BeaverDam. The formal description of the registration and key recovery algorithms are given in Figs. 8 and 9.

The high level design idea is as follows. In registration, the client hashes the password pwd to $\overline{\mathsf{pwd}} := \mathsf{F}(\mathsf{pwd})$ and runs tORF.Init c times. For each instance, it derives $\mathsf{k}^{\mathsf{SKE}}$ from the tORF output $y = f(\overline{\mathsf{pwd}})$ to encrypt K. The servers store the resulting tORF states and ciphertexts. To recover K, the client and servers execute the tORF.Eval protocol. If the client uses the correct pwd, it can derive $\mathsf{k}^{\mathsf{SKE}}$ and decrypt K as desired. Here, importantly, the client establishes a secure channel with each server prior to tORF.Eval (cf. Section 3.2). Moreover, to accommodate the fact tORF is one-time use, the client replenishes consumed instances after successful recovery to maintain rate c (cf. Remark 3). To prevent unauthorized updates, replenishment is authenticated by a tag derived from the tORF output. The construction in Figs. 8 and 9 implements this idea using domain separations when deriving SKE keys and tags.

Let us briefly comment on semi-strong augmented security of BeaverDam. If the client performs key recovery *after* full compromise (i.e., standard augmented setting), the adversary learns $\overline{\mathsf{pwd}}$, as tORF does not protect the client input upon full server compromise. In this setting, an offline dictionary attack to find pwd′ such that $\mathsf{F}(\mathsf{pwd}') = \overline{\mathsf{pwd}}$ is the optimal adversarial strategy. On the other hand, if no key recovery is performed (i.e., strong-at-rest augmented setting), the adversary does not learn $\overline{\mathsf{pwd}}$. The optimal strategy then shifts to searching for pwd′ by computing $y = f(\mathsf{F}(\mathsf{pwd}'))$ and checking if y derives the same SKE ciphertexts and tags on the compromised server state. Crucially, because server privacy ensures the tORF function f is only revealed to the adversary *after* compromise, no pre-computation can speed up this attack.

5.4 Security of BeaverDam

Formally, the following theorem establishes the security of BeaverDam. To our knowledge, BeaverDam is the first tPPKR protocol proven secure in the *quantum* ROM. Since the one-time tORF is information theoretically secure, the following implies *hybrid* security assuming the hybrid security of the secure channel and SKE.

Theorem 5. *Our* tPPKR *protocol* BeaverDam *in Figs. 8 and 9 is semi-strong augmented secure assuming* SC *is forward secure,* SKE *is* IND-CPA *secure and robust,* tORF *is server and weak client private as per Remark 1 and corruption simulatable, and* KDF *and* F *are modeled as a quantum random oracle. Specifically, if there exist an adversary $\mathcal{A}$ against the key-indistinguishability game making at most $Q_{\text{Ret-C}}$* first-round *oracle queries to $O_{\text{Recover}_\mathsf{C}}$ and $Q_{\mathsf{H}}^{\perp}$ and $Q_{\mathsf{H}}^{\mathtt{offline}}$ random oracle queries before and after a corruption query, there exist adversaries $\mathcal{B}_1$ against the secure channel and $\mathcal{B}_2$ against IND-CPA of* SKE *such that*

$$\begin{aligned}&\mathsf{Adv}^{\mathsf{kind}}_{\mathsf{atRet},\mathcal{A}}(\lambda, n, \perp)\\ &\leq \frac{c\cdot(Q_{\text{Ret-C}}+1)}{|\mathcal{D}|} + \mathsf{Adv}_{\mathsf{SC},\mathcal{B}_1}(1^\lambda)\\ &\quad + c\cdot(Q_{\text{Ret-C}}+1)\cdot\mathsf{Adv}^{\text{ind-cpa}}_{\mathsf{SKE},\mathcal{B}_2}(1^\lambda) + \mathsf{negl}(\lambda),\end{aligned}$$

and

$$\begin{aligned}&\mathsf{Adv}^{\mathsf{kind}}_{\mathsf{atRet},\mathcal{A}}(\lambda, n, \mathtt{offline})\\ &\leq \frac{c\cdot(Q_{\text{Ret-C}}+1) + Q_{\mathsf{H}}^{\mathtt{offline}}}{|\mathcal{D}|} + \mathsf{Adv}_{\mathsf{SC},\mathcal{B}_1}(1^\lambda)\\ &\quad + c\cdot(Q_{\text{Ret-C}}+1)\cdot\mathsf{Adv}^{\text{ind-cpa}}_{\mathsf{SKE},\mathcal{B}_2}(1^\lambda) + \mathsf{negl}(\lambda).\end{aligned}$$

where $\mathsf{Time}(\mathcal{A}) \approx \mathsf{Time}(\mathcal{B}_1) \approx \mathsf{Time}(\mathcal{B}_2)$.

Before formally proving this theorem, we provide the proof overview. Our proof consists of carefully removing the dependency on pwd from $\mathsf{k}^{\mathsf{SKE}}$ so that we arrive at a game where we can formally invoke IND-CPA security of SKE, ensuring that the ciphertext is independent of the key K_β.

To this end, we first invoke the security of the secure channel so that all conversations between the honest client and server S_{i^*} are hidden from the adversary. We then simulate the one-time tORF states and invoke weak client and server privacy. Due to the definition of the simulators, unless the extract algorithm $\mathsf{Extract}_x$ in server privacy (cf. Definition 10) outputs $\overline{\mathsf{pwd}}$ (i.e., an online attack succeeds), the view of the adversary is independent of $\overline{\mathsf{pwd}}$ and the tORF outputs y_j^*, where y_j^* is now uniformly random over $\mathcal{Y}$ as opposed to being set to $f_j(\overline{\mathsf{pwd}})$. Importantly, the challenger no longer uses $(\overline{\mathsf{pwd}}, y_j)$ to simulate the key recovery oracle unless an online attack succeeds. Corruption of the last honest server S_{i^*} is simulated using corruption simulatability (cf. Definition 12).[10]

At this point, we can show the "baseline" security, implicit in strong-at-rest augmented security: the adversary's advantage is bounded by $Q_{\mathtt{online}}/|\mathcal{D}|$ if server S_{i^*} remains honest. Observe the tags and SKE keys are derived from y_j^* via a KDF modeled as a random oracle H, and otherwise y_j^* is unused before full corruption or

[10] Note that we can only simulate unused tORF states; this is where we need to restrict the adversary from corrupting the server only when no key recovery is in execution.

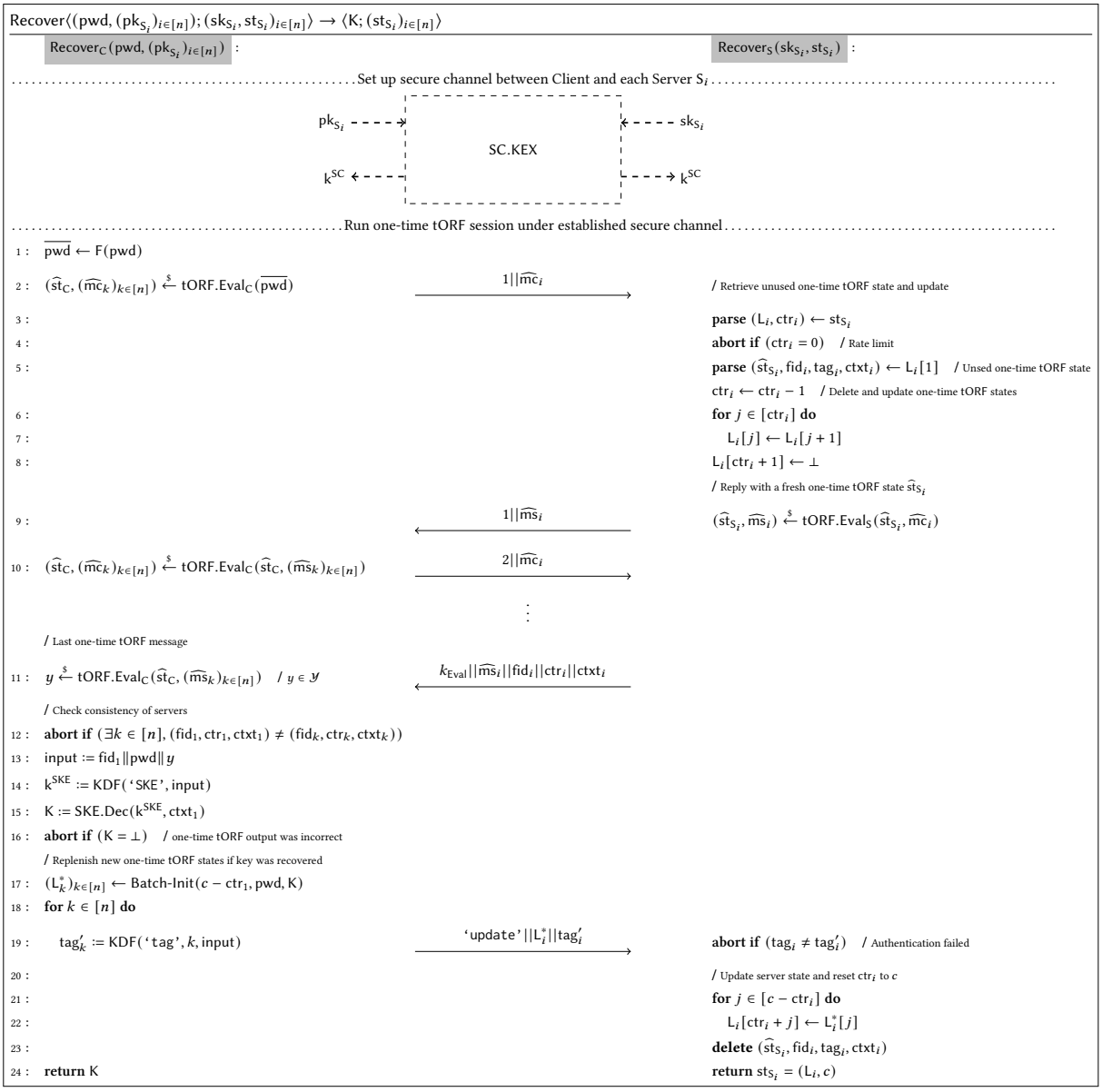

Figure 9: BeaverDam**'s interactive key recovery protocol. A secure channel is established using** SC.KEX**, depicted in a dashed box. Solid arrows indicate that messages are sent over the secure channel via** SC.Send **using the established key** k^{SC}**. For readability, we assume above the client and servers derive the same session key.** k_{Eval} **denotes the round trips of** tORF.Eval**. When abort occurs, the protocol ends by outputting** $\perp$**. While we omit for readability, the parties terminate if their states** $\widehat{st}_C$ **and** $\widehat{st}_{S_i}$ **become** $\perp$ **before completing the** tORF **session.**

a successful online attack. We thus can replace H with a "punctured" oracle G that behaves identically to H except on inputs involving y_j^* (and pwd).[11] This replacement is indistinguishable with all but *negligible* probability $Q^H/|\mathcal{Y}|$ in the classical ROM, and $O(Q^H/\sqrt{|\mathcal{Y}|})$ in the QROM via the O2H lemma [9, 20], where Q^H is the number of random oracle query and $\mathcal{Y}$ is exponentially large. Consequently, the SKE keys are independent of the adversary's view, enabling to invoke IND-CPA security of SKE as desired. Moreover, since the view of the adversary is independent of pwd, we can exactly bound the online attack success probability at $Q_{\mathtt{online}}/|\mathcal{D}|$.

(Strong-at-rest) augmented security is significantly more subtle, especially in the QROM, as we must simulate the adversary's view *after* corruption.[12] The main challenge is that upon fully compromise, the adversary learns the tORF function f_j from the server

[11] Concretely, the challenger derives tags and SKE keys by evaluating H on y_j^* but provides the adversary access only to the punctured oracle G.

[12] As both standard and strong-at-rest augmented security follows a similar proof strategy, we treat them together in our proof sketch.

states, allowing it to mount an offline dictionary attack via queries of the form $(\mathsf{pwd}', f_j(\mathsf{F}(\mathsf{pwd}')))$ to the random oracle. If we replace H with a punctured G, this substitution becomes distinguishable with *non-negligible* probability, as the dictionary attack — which succeeds in the real world with H — would necessarily fail when querying G! This necessitates a more fine-grained argument.

We resolve this issue with a two-step proof. First, we replace H with G only *before* full corruption, restoring access to H *after* corruption to ensure the adversary's post-corruption view remains identical. However, the standard O2H lemma cannot meaningfully bound the distinguishing probability of such replacement. Roughly, the O2H lemma meaningfully bounds $\left|\Pr[1 \xleftarrow{\$} \mathcal{R}^{\mathsf{H}}(z)] - \Pr[1 \xleftarrow{\$} \mathcal{R}^{\mathsf{G}}(z)]\right|$, where $\mathcal{R}$ is a (possibly inefficient) algorithm and z is any string that is *independent* of the punctured set (i.e., the inputs for which H and G differ). Ideally, we would like $\mathcal{R}$ to simulate the security game to the adversary, where z includes the challenger state and the description of the random oracle H; the latter is used so that $\mathcal{R}^{\mathsf{G}}$ can simulate H after full corruption. However, the standard O2H lemma fails here because the challenger state (i.e., y_j^*) contains information on the punctured set. To overcome this, we observe that y_j^* is unnecessary *before* full corruption. We therefore extend the O2H lemma to a two-phase variant where z is provided to $\mathcal{R}$ in stages, allowing the simulation to remain consistent without violating the independence requirement.

In the second step, we replace H given to the adversary *after* full corruption with G — at this point, since the adversary is only given access to G, we can invoke the IND-CPA security of SKE as desired. While this step relies on the standard O2H lemma, a subtle dependency arises: z must not contain information on the punctured point (i.e., pwd in this case), yet pwd is required to verify the success of an online attack via $\mathsf{Extract}_x$ and $\overline{\mathsf{pwd}} = \mathsf{F}(\mathsf{pwd})$. To resolve this, we carefully define z by sampling it from the distribution of challenger and adversary states *conditioned on* the absence of a successful online attack. This ensures that z remains independent of the punctured set while maintaining a consistent simulation.

Below, we provide the formal proof of Theorem 5.

Proof. We show semi-strong augmented security by a sequence of games and denote the advantage of $\mathcal{A}$ in Game_i as ϵ_i. In the proof, we assume the KDF and F are instantiated by a random oracle H. We use the domain separation to distinguish KDF and F. Specifically, $\mathsf{H}(0, \mathsf{input})$ (resp, $\mathsf{H}(1, \mathsf{input})$) returns $\mathsf{KDF}(\mathsf{input})$ (resp. $\mathsf{F}(\mathsf{input})$).

Game_0: This is the original game of semi-strong augmented security, that is, $\epsilon_0 = \mathsf{Adv}^{\text{ss-aug}}_{\mathsf{tPPKR},\mathcal{A}}(1^\lambda, n)$.

Game_1: In this game, the challenger aborts the game if the same fid_j is generated while executing Batch-Init. Since each fid_j is chosen uniformly from $\{0,1\}^{2\lambda}$ and at most $Q_{\mathtt{online}} + c$ tORF instances are made, we have

$$|\epsilon_0 - \epsilon_1| \leq \frac{(Q_{\mathtt{online}} + c)^2}{2^{2\lambda}}.$$

Game_2: In this game, the challenger modifies the ciphertexts exchanged between the honest client and server S_{i^*} *after* a secure channel is set up. Namely, it replaces all messages sent between the client and server S_{i^*} through SC.Send by a zero string; instead of running SC.Receive to decrypt the ciphertext, it simply keeps a record of what was sent via SC.Send to recover the message. Through a simple but tedious argument, we can construct an adversary $\mathcal{B}_1$ against the security of the secure channel such that

$$|\epsilon_1 - \epsilon_2| \leq \mathsf{Adv}_{\mathsf{SC},\mathcal{B}_1}(1^\lambda).$$

A formal description of $O_{\mathsf{Recover_C}}$ and $O_{\mathsf{Recover_S}}$ in this game along with its proof is given in Lemma 4. We highlight that in this game, any messages sent to the honest server S_{i^*} after a secure channel is set up can be identified as a message sent by an honest client or an adversary.

Game_3: In this game, the challenger changes how to check if the client obtains the correct tORF value in the client oracle. Specifically, the challenger tracks if an adversary succeeds in an *online attack* via a flag $\mathsf{flag}_{\mathsf{online}}$, initially set to $\perp$. This is set to $\top$ when S_{i^*} completes an entire execution of the tPPKR protocol with an adversary. Note that once an online attack succeeds, the server state $\mathsf{st}_{\mathsf{S}_{i^*}}$ will contain adversarial tORF states as a consequence of the final `'update'` message. Even after $\mathsf{flag}_{\mathsf{online}}$ is set to $\top$, an adversary $\mathcal{A}$ can force the client to update honest tORF states to the honest server. We also call these states adversarial tORF states even though they are honestly generated.

In the client oracle, the challenger no longer decrypts the SKE ciphertext ctxt when it communicates with the honest server using an honest state. Instead, it first checks if the tORF value y received by the honest client matches the tORF value y^* that was previously computed in Batch-Init. If not, it aborts with outputting $\perp$ as in the previous game, and otherwise, it uses the challenge key K_β as the retrieved key to generat new servers' states and sets $\mathsf{st}_{\mathsf{C}} = \mathsf{K}_\beta$. If it interacts with the honest server whose state is an adversarial one, it decrypts ctxt as in the previous game. Note that even if $\mathsf{flag}_{\mathsf{online}} = \top$ (i.e., online attack succeeds), we cannot immediately apply this modification since there could still be honest tORF states on the honest server S_{i^*}, e.g., if the adversary's online attack succeeds at the outset of the game, S_{i^*} will still contain $c - 1$ unused simulated tORF states.

Thanks to the consistency check in Fig. 9, line 12 and the modification we made in Game_2, the ctxt the client receives is guaranteed to be generated during Batch-Init. Thus, the only way to distinguish between the two games is for the client to compute some $y \neq y^*$ such that $\mathsf{SKE.Dec}(\mathsf{k}^{\mathsf{SKE}}, \mathsf{ctxt}) \neq \perp$ where $\mathsf{k}^{\mathsf{SKE}} := \mathsf{KDF}(\text{`SKE'}, \mathsf{fid} \| \mathsf{pwd} \| y)$. However, such a y can be used to immediately break the collision resistance of the KDF or the robustness of the SKE. Namely, there exist $\mathcal{B}_2$ and $\mathcal{B}_3$ such that

$$|\epsilon_2 - \epsilon_3| \leq \mathsf{Adv}^{\mathsf{CR}}_{\mathsf{H},\mathcal{B}_2}(1^\lambda) + \mathsf{Adv}^{\mathsf{Robust}}_{\mathsf{SKE},\mathcal{B}_3}(1^\lambda).$$

Note that since H is a random oracle, collision resistance holds unconditionally.

Game_4: In this game, the challenger simulates the honest client, the hoenst server S_{i^*}, and S_{i^*}'s unused states using tORF's simulation algorithms. Concretely, the tORF.Init run inside algorithm Batch-Init by the client is replaced by tORF.SimInit (cf. Definitions 10 and 11 and Remark 1, simulator for corrupted servers' states) unless $\mathsf{flag}_{\mathsf{online}} = \top$. Since the simulation function tORF.SimInit does not output functions f_j, the challenger samples a random $y_j^* \xleftarrow{\$} \mathcal{Y}$, which it uses in place of $f_j(\overline{\mathsf{pwd}})$. Moreover, it sets the

simulated tORF state $\widehat{\mathsf{st}}_{\mathsf{S}_{i^*},j}$ of the honest server S_{i^*} to an empty string as it no longer requires it to run the game — even if $\widehat{\mathsf{st}}_{\mathsf{S}_{i^*},j}$ is an empty string, we call it the *simulated* tORF *state* for the sake of explanation. When $\mathsf{flag}_{\mathsf{online}} = \top$, it runs tORF.Init as in the previous game.

Next, $\mathsf{tORF.Eval}_{\mathsf{C}}$, run by the honest client, is replaced by $(\mathsf{tORF.SimEval}_{\mathsf{C}}, \mathsf{tORF.CorrectVal})$ (cf. Definition 11, client privacy) while there are simulated tORF states on the honest server S_{i^*}. Otherwise, as corresponding S_{i^*}'s state is an adversarial tORF state, it runs $\mathsf{tORF.Eval}_{\mathsf{C}}$ as in the prior game. Moreover, recall that in the previous game, the challenger checked if the tORF value y received by the honest client matches the tORF value y^* that was previously computed in Batch-Init when corresponding servers's state is not adversarial. In this game, the challenger uses tORF.CorrectVal to check this; if $b = 0$, then abort and otherwise, it continues as in the previous game. Looking ahead, this modification guarantees that the view of the adversary is independent of $\overline{\mathsf{pwd}}$ and y^*.

Finally, $\mathsf{tORF.Eval}_{\mathsf{S}}$ run by the honest server S_{i^*} is modified in two ways while there are simulated tORF states; if there are none, it runs $\mathsf{tORF.Eval}_{\mathsf{S}}$ as in the prior game. If S_{i^*} is communicating with an honest client, then it does not compute anything related to tORF and simply sends the zero string over the secure channel as in Game_2. This is consistent with the fact that the tORF state $\widehat{\mathsf{st}}_{\mathsf{S}_{i^*},j}$ of the honest server S_{i^*} is no longer used when talking to an honest client (cf. Definition 11). In contrast, if S_{i^*} is communicating with an adversary, it runs $(\mathsf{tORF.SimEval}_{\mathsf{S}}, \mathsf{tORF.Extract}_x)$ (cf. Definition 10, server privacy). Here, recall $\mathsf{tORF.Extract}_x$ outputs $\overline{\mathsf{pwd}}'$, intuitively extracting $\mathcal{A}$'s guess of the hash of password.

Using a simple hybrid argument and corruption simulatability of tORF (cf. Definition 12), it is easy to check that there exists $\mathcal{B}_4$ and $\mathcal{B}_5$ against the server and client privacy, respectively, such that

$$|\epsilon_3 - \epsilon_4| \le Q_{\mathsf{online}} \cdot \left(\mathsf{Adv}^{\mathsf{sp}}_{\mathsf{tORF},\mathcal{B}_4}(1^\lambda, n) + \mathsf{Adv}^{\mathsf{cp}}_{\mathsf{tORF},\mathcal{B}_5}(1^\lambda, n)\right).$$

A formal description of $O_{\mathsf{Recover}_{\mathsf{C}}}$ and $O_{\mathsf{Recover}_{\mathsf{S}}}$ in this game is given in Lemma 5.

Game_5: In this game, the challenger aborts the game when either of the two events OnlineAtk and ExtractPWD hold. Both events can only be triggered when the honest server S_{i^*} is communicating with an adversary. Formally, event OnlineAtk is triggered when $\mathsf{flag}_{\mathsf{online}} = \top$ (i.e., online attack succeeds), and ExtractPWD is triggered when $\mathsf{tORF.Extract}_x$ outputs the correct hash of the password $\overline{\mathsf{pwd}}$. As the two games are identical until these events occur, we have

$$|\epsilon_4 - \epsilon_5| \le \Pr[\mathsf{OnlineAtk}] + \Pr[\mathsf{ExtractPWD}]. \tag{4}$$

Since we cannot bound the right-hand side in this game, we move on to the next game and bound it later.

For the next game, we prepare some notations and define a subset of the domain of H. Let us denote H in Game_5 by H_1, and define $\mathcal{S}_1$ as the set of all tuples of the form $(0, \texttt{'mode'}, \mathsf{fid}\|\mathsf{pwd}\|y_j^*)$ where $\texttt{mode} \in \{\texttt{SKE}, \texttt{tag}\}$, pwd is the challenge password, and fid and y_j^* are used in Batch-Init.

Game_6: In this game, we replace the random oracle H_1 with another oracle H_2 for queries made before full corruption. This removes the dependency on (pwd, y_j^*) from tags and SKE keys until full corruption occurs. H_2 is defined exactly as the random oracle H_1 except on inputs in $\mathcal{S}_1$. In other words, for $x \notin \mathcal{S}_1$, $\mathsf{H}_2(x) = \mathsf{H}_1(x)$, and otherwise, $\mathsf{H}_2(x)$ is random. Specifically, in this game, the challenger answers random oracle queries before full corruption by using H_2. But it still uses H_1 when it runs Batch-Init and after full corruption. Also, in $O_{\mathsf{Recover}_{\mathsf{C}}}$ using simulated tORF states, it uses the same tags used in Batch-Init to prepare $(\mathsf{mc}_i)_{i\in C}$. Note that this change in $O_{\mathsf{Recover}_{\mathsf{C}}}$ is conceptual as the tags are computed by H_1.

This game is indistinguishable from the previous as long as the adversary does not query an input in $\mathcal{S}_1$ *before* S_{i^*} *is corrupted*. Here, note that we use the argument made in Game_1 — as all fid are unique, each tag and SKE key can be sampled independently. Thanks to how the challenger simulates the tORF queries, and to the abort conditions added in the prior game, no information on y is leaked to the adversary before full corruption. Thus, in the classical ROM, the probability that an adversary notices a difference between Game_5 and Game_6 can be easily bounded by $Q^{\mathsf{H}}_{\mathsf{pre}}/|\mathcal{Y}|$. In the quantum ROM, due to the extended semi-classical O2H lemma (cf. Lemma 3) for random functions $\mathsf{H}_1, \mathsf{H}_2$, and the punctured set $\mathcal{S}_1$, we can bound that probability by $4\cdot(Q^{\mathsf{H}}_{\mathsf{pre}}+1)/\sqrt{|\mathcal{Y}|}$. The formal proof in the QROM is given in Lemma 6. Thus, we have

$$|\epsilon_5 - \epsilon_6| \le \begin{pmatrix} \frac{Q^{\mathsf{H}}_{\mathsf{pre}}}{|\mathcal{Y}|} & \text{if classical ROM} \\ \frac{4\cdot(Q^{\mathsf{H}}_{\mathsf{pre}}+1)}{\sqrt{|\mathcal{Y}|}} & \text{if QROM} \end{pmatrix}. \tag{5}$$

Denoting the abort conditions in Game_6 as $\mathsf{OnlineAtk}^*$ and $\mathsf{ExtractPWD}^*$, we can further upper bound Eq. (4) by

$$\begin{aligned} &\Pr[\mathsf{OnlineAtk}] + \Pr[\mathsf{ExtractPWD}] \\ &\le \Pr[\mathsf{OnlineAtk}^*] + \Pr[\mathsf{ExtractPWD}^*] \\ &\quad + \begin{pmatrix} \frac{Q^{\mathsf{H}}_{\mathsf{pre}}}{|\mathcal{Y}|} & \text{if classical ROM} \\ \frac{4\cdot(Q^{\mathsf{H}}_{\mathsf{pre}}+1)}{\sqrt{|\mathcal{Y}|}} & \text{if QROM} \end{pmatrix}. \end{aligned} \tag{6}$$

It remains to upper bound the abort probability in Game_6. Recall that these abort events are only triggered before full corruption. Also, in this game, the tORF outputs no longer leak any information on the hash of password $\overline{\mathsf{pwd}}$ and evaluation outputs y^*. Moreover, the tags and SKE keys are sampled from H_1 to which the adversary has no access before full corruption. First, let's consider event $\mathsf{OnlineAtk}^*$. For $\mathsf{flag}_{\mathsf{online}}$ to be set to $\top$, the adversary must be able to provide the honest server S_{i^*} the correct tag tag_{i^*}. However, the adversary can only access H_2 before full corruption, while tag_{i^*} is sampled via H_1. Thus, the probability of guessing the correct tag_{i^*} is bounded by $Q_{\mathsf{online}}/\left|2^{2\lambda}\right|$. Next, for the case of event $\mathsf{ExtractPWD}^*$, since $\mathsf{tORF.Extract}_x$ extracts a single guess $\overline{\mathsf{pwd}}'$ per query to the honest server S_{i^*}, $\Pr[\mathsf{ExtractPWD}^*]$ is bounded by $Q_{\mathsf{online}}/|\mathcal{D}|$. We thus obtain the bound:

$$\Pr[\mathsf{OnlineAtk}^*] + \Pr[\mathsf{ExtractPWD}^*] \le \frac{Q_{\mathsf{online}}}{|\mathcal{D}|} + \frac{Q_{\mathsf{online}}}{2^{2\lambda}}$$

A formal description of $O_{\mathsf{Recover}_{\mathsf{C}}}$ and $O_{\mathsf{Recover}_{\mathsf{S}}}$ in Game_5 and Game_6 and a more detailed proof on the abort probability are given in Fig. 12 and in Lemma 7, respectively.

Strong-at-Rest Augmented Security. Let us now consider the case mode = sar-aug, i.e., the adversary makes no query to the oracle $O_{\mathsf{Recover}_{\mathsf{C}}}$ after full corruption. We denote the i-th game in this case

by $\mathsf{Game}_i^{\text{sar-aug}}$ and the advantage in $\mathsf{Game}_i^{\text{sar-aug}}$ by $\epsilon_i^{\text{sar-aug}}$. We will analyze augmented security later.

$\mathsf{Game}_7^{\text{sar-aug}}$: In this game, we replace H_1 by H_2 after full corruption to make the adversary's view independent of pwd. Concretely, after S_{i^*} is corrupted, the challenger uses random oracle H_2 to answer random oracle queries instead of H_1. Note that the challenger still uses H_1 to generate tags and SKE keys in Batch-Init.

The adversary only can detect this change if it queries pwd to the random oracle after full corruption. In the strong-at-rest augmented security game, after full corruption, the adversary obtains only simulated S_{i^*}'s unused states since it may no longer access $O_{\mathsf{Recover_C}}$. To bound the probability that the adversary makes such a query, the main observation is that the revealed tORF states $\widehat{\mathsf{st}}_{\mathsf{S}_{i^*}j} \xleftarrow{\$} \mathsf{SimSt}(\mathsf{st}_{\mathsf{sim}}, \overline{\mathsf{pwd}}, y_j^*)$ are independent of $\overline{\mathsf{pwd}}$ due to Corollary 1. This implies that the revealed states do not leak any information on $\overline{\mathsf{pwd}}$, and thus pwd. Therefore, due to the modifications we made, even after corrupting all servers, the adversary gains no more information on pwd than what it had already gained via performing online attacks prior to the full corruption. Through a rigorous probability analysis, we have

$$\left|\epsilon_6 - \epsilon_7^{\text{sar-aug}}\right| \leq \begin{pmatrix} \frac{Q^{\mathsf{H}}_{\text{post}}}{|\mathcal{D}|} & \text{if classical ROM} \\ \frac{4\cdot(Q^{\mathsf{H}}_{\text{post}}+1)}{\sqrt{|\mathcal{D}|}} & \text{if QROM} \end{pmatrix}.$$

We formally prove these bounds in Lemma 8.

$\mathsf{Game}_8^{\text{sar-aug}}$: Lastly, in this game, the challenger replaces the SKE ciphertexts ctxt to always decrypt to a zero string as opposed to the challenge key K_β. Thanks to the modifications in Game_6 and $\mathsf{Game}_7^{\text{sar-aug}}$, an adversary only has access to H_2, although the SKE key $\mathsf{k}^{\mathsf{SKE}}$ is derived via H_1. This game is equivalent to the game in which the SKE key $\mathsf{k}^{\mathsf{SKE}}$ is sampled uniformly at random. Thus, we can construct an adversary $\mathcal{B}_6$ against the IND-CPA security of the SKE such that

$$\left|\epsilon_7^{\text{sar-aug}} - \epsilon_8^{\text{sar-aug}}\right| \leq Q_{\mathsf{online}} \cdot \mathsf{Adv}^{\text{ind-cpa}}_{\mathsf{SKE},\mathcal{B}_6}(1^\lambda),$$

where note that there can be at most Q_{online} SKE ciphertexts.

In $\mathsf{Game}_8^{\text{sar-aug}}$, the adversary's view is independent of the challenge bit β. Therefore $\epsilon_8^{\text{sar-aug}} = 0$. Combining all the arguments completes the proof of strong-at-rest augmented security.

Augmented Security. Now we move to the case of mode = aug, namely, the adversary is allowed to make use of the key recovery protocol after full corruption. Similarly to the prior case, let us denote the i-th game in this case by $\mathsf{Game}_i^{\text{aug}}$ and the advantage in $\mathsf{Game}_i^{\text{aug}}$ by ϵ_i^{aug}. The main difference from the case of strong-at-rest augmented security is that the adversary can obtain $\overline{\mathsf{pwd}}$ from $O_{\mathsf{Recover_C}}$ after full corruption. This requires us to remove the dependency on pwd from $\overline{\mathsf{pwd}}$. To this end, we consider a fresh random oracle which is defined as H_2 except on input $(1, \mathsf{pwd})$.

After replacing the random oracles, the rest of the proof will follow the same arguments as in the proof of strong-at-rest augmented security. We will thus omit it, and in the following, we focus only on $\mathsf{Game}_7^{\text{aug}}$ in which the random oracles are replaced.

$\mathsf{Game}_7^{\text{aug}}$: In this game, we replace both H_1 and H_2 with H_2' which has the same outputs as H_2 except on the input $(1, \mathsf{pwd})$. Specifically, the challenger uses H_2' instead of H_1 and H_2 to answer the adversary's random oracle queries before and after full corruption, respectively. Everything else about this game is identical to Game_6.

The adversary can distinguish between the current and previous games only if it queries $(1, \mathsf{pwd})$ or $x \in \mathcal{S}_1$, which includes pwd, to the random oracles. To bound the probability that the adversary makes such a query, the main two observations are that (i) leaking whether $\overline{\mathsf{pwd}}' \neq \overline{\mathsf{pwd}}$ via $O_{\mathsf{Recover_S}}$, (i.e., the result of an online attack) provides no additional advantage when $\overline{\mathsf{pwd}}$ is leaked, and (ii) the corruption oracle and server and client oracles do not leak any information on pwd after full corruption. Indeed, the adversary can perform this check locally by using its random oracle access and $\overline{\mathsf{pwd}}$. Also, S_{i^*}'s unused states are computed from $\overline{\mathsf{pwd}}$ and y_j^*, not pwd. Moreover, pwd is used in both games only when the challenger computes $\overline{\mathsf{pwd}}$, tags, and SKE keys using random oracles. Therefore, in the classical ROM, the probability that the adversary queries pwd is bounded by $(Q^{\mathsf{H}}_{\text{pre}} + Q^{\mathsf{H}}_{\text{post}})/|\mathcal{D}|$. In the QROM, relying on the semi-classical O2H lemma (cf. Lemmata 1 and 2), this probability is bounded by $4 \cdot (Q^{\mathsf{H}}_{\text{pre}} + Q^{\mathsf{H}}_{\text{post}} + 1)/\sqrt{|\mathcal{D}|}$, as shown in Lemma 9. Thus we have

$$\left|\epsilon_6 - \epsilon_7^{\text{aug}}\right| \leq \begin{pmatrix} \frac{Q^{\mathsf{H}}_{\text{pre}}+Q^{\mathsf{H}}_{\text{post}}}{|\mathcal{D}|} & \text{if classical ROM} \\ \frac{4\cdot(Q^{\mathsf{H}}_{\text{pre}}+Q^{\mathsf{H}}_{\text{post}}+1)}{\sqrt{|\mathcal{D}|}} & \text{if QROM} \end{pmatrix}.$$

Combining all arguments above, we can conclude that our protocol BeaverDam is semi-strong augmented secure.

□

5.4.1 *Supporting Lemmas.*

Game_1 *to* Game_2.

Lemma 4. *In the proof of Theorem 5, there exists an adversary $\mathcal{B}_1$ against the security of the secure channel* SC *such that*

$$|\epsilon_1 - \epsilon_2| \leq \mathsf{Adv}_{\mathsf{SC},\mathcal{B}_1}(1^\lambda).$$

Proof. The formal description of $O_{\mathsf{Recover_C}}$ and $O_{\mathsf{Recover_S}}$ in Game_2 is given in Fig. 10. At the beginning of the game, the challenger initializes the following:

- Counters n_{C} and n_{S}. These count the number of secure channel sessions the honest client and server S_{i^*}, respectively, initialize.
- Lists $\mathsf{peer_C}[\cdot]$ and $\mathsf{peer}_{\mathsf{S}_{i^*}}[\cdot]$. $\mathsf{peer_C}[n_{\mathsf{C}}]$ and $\mathsf{peer}_{\mathsf{S}_{i^*}}[n_{\mathsf{S}}]$ store the counter of the secure channel used by the honest server S_{i^*} and client, respectively. In case S_{i^*} is communicating with an adversary $\mathsf{peer}_{\mathsf{S}_{i^*}}[n_{\mathsf{S}}]$ will remain set to $\perp$.
- Lists $\mathsf{key_C}[\cdot]$ and $\mathsf{key}_{\mathsf{S}_{i^*}}[\cdot]$. Once the secure channel is established these lists will store the session key.
- List $\mathsf{SentMsg}[\cdot]$. This manages all the messages sent under the secure channel (i.e., messages sent via SC.Send). Note that these messages are of the form $r\|m$ for $r \in [k_{\mathsf{Eval}}] \cup \{\texttt{'update'}\}$ (cf. Fig. 9).

The key modification from the previous game is that all honest client and server S_{i^*} messages that were sent via SC.Send are

no longer sent. The challenger instead sends a zero string using SC.Send, and maintains all the sent messages using the list SentMsg[·]. When SC.Receive is run, it first checks if there exists a unique entry in SentMsg[·] to retrieve the message. Due to correctness, this has the same effect as running SC.Receive. Moreover, this implies that an adversary cannot inject messages once a secure channel is set up. The challenger further adds relevant abort conditions induced by the secure channel, e.g., if the client established a session key with the server S_{i^*}, then S_{i^*} must have established the same session key.

One notable abort condition is the one related to $\mathsf{flag}_{\mathsf{unf}}$ on line 10 in $O_{\mathsf{Recover_C}}$ and on line 15 in $O_{\mathsf{Recover_S}}$. As the honest server S_{i^*} is the one that talks last in the secure channel key exchange (SC.KEX) protocol, it does not know who it is communicating to right after it sets up the n_S-th session, i.e., $\mathsf{peer}_{S_{i^*}}[n_S] = \perp$. As such when the server receives a message through this channel, there could be two cases. The first case is when this is simply a secure channel established by the adversary. The second case is when this is a secure channel that an honest client would later establish, but the adversary was able to break it before the server SC.KEX message was delivered to the client. The flag $\mathsf{flag}_{\mathsf{unf}}$ is used to capture the second case: it is first set to $\top$ on line 15 in $O_{\mathsf{Recover_S}}$ when $\mathsf{peer}_{S_{i^*}}[n_S] = \perp$; later when the client establishes the same key as $\mathsf{key}_{S_{i^*}}[n_S]$, then the game aborts as on line 10 in $O_{\mathsf{Recover_C}}$.

As Game_2 is set up to exactly match the winning conditions of the forward security game of the secure channel (cf. Fig. 1), the lemma statement follows. □

Game_3 *to* Game_4.

Lemma 5. *In the proof of Theorem 5, there exist adversaries $\mathcal{B}_4$ and $\mathcal{B}_5$ against the server and weak client privacy of* tORF*, respectively, such that*

$$|\epsilon_3 - \epsilon_4| \le Q_{\mathsf{online}} \cdot \left(\mathsf{Adv}^{\mathsf{sp}}_{\mathsf{tORF},\mathcal{B}_4}(1^\lambda, n) + \mathsf{Adv}^{\mathsf{cp}}_{\mathsf{tORF},\mathcal{B}_5}(1^\lambda, n)\right).$$

Proof. The formal description of $O_{\mathsf{Recover_C}}$ and $O_{\mathsf{Recover_S}}$ in Game_4 is given in Fig. 11. As there are at most Q_{online} queries to the honest server S_{i^*}, and the challenger may generate c tORF instance in Q_{online}-th query, it generates at most $Q_{\mathsf{online}} + c$ tORF instances. We will go over $Q_{\mathsf{online}} + c$ hybrid games to prove the lemma's statement. We define $\mathsf{Game}_{3,t}$ for $t \in [0 : Q_{\mathsf{online}} + c]$ as the game where the first t tORF instances are simulated; namely, $\mathsf{Game}_{3,0} = \mathsf{Game}_3$ and $\mathsf{Game}_{3,Q_{\mathsf{online}}+c} = \mathsf{Game}_4$. Recall once $\mathsf{flag}_{\mathsf{online}} = \top$ (i.e., successful online attack) occurs, the honest server state $\mathsf{st}_{S_{i^*}}$ will contain adversarial tORF instances. In this case, the game simply uses them.

Let us now bound $\left|\epsilon_{3,t-1} - \epsilon_{3,t}\right|$ for $t \in [Q_{\mathsf{online}} + c]$. The only difference between $\mathsf{Game}_{3,t-1}$ and $\mathsf{Game}_{3,t}$ is whether the t-th tORF instance is simulated or not. For $t \in [Q_{\mathsf{online}}]$, observe that this t-th tORF state generated by tORF.SimInit can be used in only three ways. The first case is when the honest client runs ($\mathsf{tORF.SimEval_C}$, tORF.CorrectVal) (cf. Definition 11, client privacy), the second case is when the honest server S_{i^*} communicating with an adversary runs ($\mathsf{tORF.SimEval_S}$, $\mathsf{tORF.Extract}_x$) (cf. Definition 10, server privacy), and the third case is when it is not used and is revealed due to the corruption. For $t \in [Q_{\mathsf{online}} + 1, Q_{\mathsf{online}} + c]$, since $\mathcal{A}$ makes at most Q_{online} queries to the honest server, the t-th tORF instance falls into the third case. Here, importantly, as we require server and weak client privacy and corruption simulatability to hold with respect to the same simulator tORF.SimInit for the corrupted servers' states, the challenger does not have to guess which case occurs when setting up the tORF state.

For the third case, due to corruption simulatability of the tORF states (cf. Definition 12), the view of the adversary remains unchanged. Thus, we obtains $\epsilon_{3,t-1} = \epsilon_{3,t}$ in this case. For the first and second cases, it is easy to check that we can construct adversaries $\mathcal{B}_4$ and $\mathcal{B}_5$ against the server and client privacy of tORF, such that

$$\left|\epsilon_{3,t-1} - \epsilon_{3,t}\right| \le \mathsf{Adv}^{\mathsf{sp}}_{\mathsf{tORF},\mathcal{B}_4}(1^\lambda, n) + \mathsf{Adv}^{\mathsf{cp}}_{\mathsf{tORF},\mathcal{B}_5}(1^\lambda, n).$$

Going through the $Q_{\mathsf{online}} + c$ hybrid games, we arrive at the lemma statement. □

5.4.2 Game_5 *to* Game_6 *in the QROM.*

Lemma 6. *In the proof of Theorem 5, in the QROM, we have*

$$|\epsilon_5 - \epsilon_6| \le \frac{4 \cdot (Q^{\mathsf{H}}_{\mathsf{pre}} + 1)}{\sqrt{|\mathcal{Y}|}}.$$

Proof. To prove this, we rely on the extended semi-classical O2H lemma (cf. Lemma 3). The non-trivial aspects are in how the reduction replaces the oracle H_2 with H_1 after the adversary corrupts all servers, and in how it simulates the game after full corruption. At a high level, the strategy is to provide the reduction with all the information required to simulate the game after full corruption.

We construct the reduction as follows: it receives all states of corrupted servers, the challenge password pwd, and the tags and SKE keys as the first input z. We note that z does not contain the simulated tORF outputs y^*_j. Since this reduction — via z — has access to all the information required to simulate Game_5 and Game_6 *before full corruption*, it can perfectly simulate the games until S_{i^*} is corrupted by using its oracle access to answer the adversary's random oracle queries. When the adversary corrupts S_{i^*}, it requests an additional input z' which contains the simulated tORF outputs y^*_j and the description of H_1. We can give the description of H_1 to the reduction since the reduction is statistical, and as such can be inefficient. Using z', the reduction can perfectly simulate S_{i^*}'s unused states and the random oracle H_1 after the full corruption. Notice that the reduction no longer accesses its oracle after receiving z'.

Since the input z does not leak any information on any y^*_j, $\Pr[x \in \mathcal{S}_1]$ conditioned on z is at most $1/|\mathcal{Y}|$. Thus, due to Lemma 3, we obtain the bound above. This completes the proof. □

Probability of an Abort.

Lemma 7. *In the proof of Theorem 5, we have*

$$\Pr[\mathsf{OnlineAtk}^*] + \Pr[\mathsf{ExtractPWD}^*] \le \frac{Q_{\mathsf{online}}}{|\mathcal{D}|} + \frac{Q_{\mathsf{online}}}{2^{2\lambda}}.$$

Proof. We first evaluate $\Pr[\mathsf{OnlineAtk}^*]$. Due to the modification we made in Game_6, all tag_{i^*} stored on the honest server S_{i^*}'s state are information theoretically hidden from $\mathcal{A}$. Therefore, the probability that $\mathcal{A}$ sends the correct tag_{i^*} to the server is at most $1/2^{2\lambda}$ per query to the oracle $O_{\mathsf{Recover_S}}$. Thus, we have

$$\Pr[\mathsf{OnlineAtk}^*] \le \frac{Q_{\mathsf{online}}}{2^{2\lambda}}.$$

Next, we move to $\Pr[\mathsf{ExtractPWD}^*]$. For $t \in Q_{\mathsf{online}}$, let Abort_t be the event where the game aborts during the t-th query to the oracle O_{Recovers} owing to the check $\overline{\mathsf{pwd}}' \neq \overline{\mathsf{pwd}}$. Since the adversary can access only H_2 before full corruption, tags and SKE keys in corrupted servers' states, which are derived from H_1, do not leak information on pwd and $\overline{\mathsf{pwd}}$ until it corrupts S_{i^*}. As the challenger does not use pwd and $\overline{\mathsf{pwd}}$ except for the generation of tags and SKE keys and checking whether $\overline{\mathsf{pwd}}'$ is equal to $\overline{\mathsf{pwd}}$ in O_{Recovers}, we have $\Pr[\mathsf{Abort}_1] = 1/|\mathcal{D}|$ on the first query. In general, we have

$$\Pr[\mathsf{ExtractPWD}^*] = \Pr[\mathsf{Abort}_1] + \sum_{t\in[2:Q_{\mathsf{online}}]} \Pr\left[\bigwedge_{\ell\in[t-1]} \neg\mathsf{Abort}_\ell \wedge \mathsf{Abort}_t\right]. \tag{7}$$

Let us now bound $\Pr[\bigwedge_{\ell\in[t-1]} \neg\mathsf{Abort}_\ell \wedge \mathsf{Abort}_t]$ for $t \in [2 : Q_{\mathsf{online}}]$. Let S_t be the set of distinct hashes of passwords $\overline{\mathsf{pwd}}'$ extracted using $\mathsf{tORF.Extract}_x$ up till (and including) the t-th query to oracle O_{Recovers}. For instance, $|S_1| = 1$ as there has only been one $\overline{\mathsf{pwd}}'$ extracted in the first invocation of oracle O_{Recovers}. We then claim the following:

$$\Pr\left[\bigwedge_{\ell\in[t-1]} \neg\mathsf{Abort}_\ell \wedge \mathsf{Abort}_t\right] = \begin{cases} 0 & \text{if } S_t = S_{t-1} \\ \frac{1}{|\mathcal{D}|} & \text{else} \end{cases} \tag{8}$$

$$\Pr\left[\bigwedge_{\ell\in[t]} \neg\mathsf{Abort}_\ell\right] = \frac{|\mathcal{D}| - |S_t|}{|\mathcal{D}|}. \tag{9}$$

The first case in Eq. (8) is immediate since if $S_t = S_{t-1}$, the extracted hash of the password $\overline{\mathsf{pwd}}'$ on the t-th invocation of oracle O_{Recovers} cannot cause an abort owing to $\overline{\mathsf{pwd}}' \in S_{t-1}$. We show the second case via induction assuming $S_{t+1} \neq S_t$ and the case for t holds. Indeed we have

$$\begin{aligned}
&\Pr\left[\bigwedge_{\ell\in[t]} \neg\mathsf{Abort}_\ell \wedge \mathsf{Abort}_{t+1}\right] \\
&= \Pr\left[\mathsf{Abort}_{t+1} \mid \bigwedge_{\ell\in[t]} \neg\mathsf{Abort}_\ell\right] \cdot \Pr\left[\bigwedge_{\ell\in[t]} \neg\mathsf{Abort}_\ell\right] \\
&= \frac{1}{|\mathcal{D}| - |S_t|} \cdot \frac{|\mathcal{D}| - |S_t|}{|\mathcal{D}|} = \frac{1}{|\mathcal{D}|}.
\end{aligned}$$

Above, note that conditioning on $\bigwedge_{\ell\in[t]} \neg\mathsf{Abort}_\ell$ and $S_{t+1} \neq S_t$, the probability of Abort_{t+1} occurring is $1/(|\mathcal{D}| - |S_t|)$. Moreover, the probability of $\bigwedge_{\ell\in[t]} \neg\mathsf{Abort}_\ell$ follows from Eq. (9).

Moreover, we have

$$\begin{aligned}
&\Pr\left[\bigwedge_{\ell\in[t+1]} \neg\mathsf{Abort}_\ell\right] \\
&= \Pr\left[\bigwedge_{\ell\in[t]} \neg\mathsf{Abort}_\ell\right] - \Pr\left[\bigwedge_{\ell\in[t]} \neg\mathsf{Abort}_\ell \wedge \mathsf{Abort}_{t+1}\right] \\
&= \frac{|\mathcal{D}| - |S_t|}{|\mathcal{D}|} - \Pr\left[\bigwedge_{\ell\in[t]} \neg\mathsf{Abort}_\ell \wedge \mathsf{Abort}_{t+1}\right].
\end{aligned}$$

Plugging in Eq. (8), if $S_{t+1} = S_t$, then the above is equal to $(|\mathcal{D}| - |S_{t+1}|)/|\mathcal{D}|$. Otherwise, since $|S_{t+1}| = |S_t| + 1$, the above is equal to $(|\mathcal{D}| - |S_{t+1}|)/|\mathcal{D}|$. This proves Eqs. (8) and (9).

Plugging all the bounds into Eq. (7), we have the following bound as desired:

$$\Pr[\mathsf{ExtractPWD}^*] \leq \frac{Q_{\mathsf{online}}}{|\mathcal{D}|}.$$

□

Game_6 *to* $\mathsf{Game}_7^{\mathsf{sar\text{-}aug}}$.

Lemma 8. *In the proof of Theorem 5, we have*

$$\left|\epsilon_6 - \epsilon_7^{\mathsf{sar\text{-}aug}}\right| \leq \begin{pmatrix} \frac{Q^{\mathsf{H}}_{\mathsf{post}}}{|\mathcal{D}|} & \text{if classical ROM} \\ \frac{4\cdot(Q^{\mathsf{H}}_{\mathsf{post}}+1)}{\sqrt{|\mathcal{D}|}} & \text{if QROM} \end{pmatrix}.$$

Proof. The proof relies on the critical observation that the revealed tORF states, $\widehat{\mathsf{st}}_{\mathsf{S}_{i^*},j} \xleftarrow{\$} \mathsf{SimSt}(\mathsf{st}_{\mathsf{sim}}, \overline{\mathsf{pwd}}, y_j^*)$, are independent of $\overline{\mathsf{pwd}}$ and pwd. Consequently, even after corrupting all servers, the adversary gains no additional information about $\overline{\mathsf{pwd}}$ and pwd beyond what was already gleaned from online attacks performed prior to the full corruption. We detail the analysis below.

Recall that $\mathsf{Game}_7^{\mathsf{sar\text{-}aug}}$ differs from Game_6 only in that the random oracle H_1 is replaced by H_2 *after* the full corruption occurs. The rest of the game remains unchanged. Let Abort denote the event that the game aborts before full corruption. Since no abort condition is triggered when Abort does not occur, we have $\Pr[\neg\mathsf{Abort}] \leq \Pr[\neg\mathsf{ExtractPWD}^*]$. Then we have

$$\begin{aligned}
\left|\epsilon_6 - \epsilon_7^{\mathsf{sar\text{-}aug}}\right| &= \Pr[\neg\mathsf{Abort}] \cdot \left|\bar{\epsilon}_6 - \bar{\epsilon}_7^{\mathsf{sar\text{-}aug}}\right| \\
&\leq \Pr[\neg\mathsf{ExtractPWD}^*] \cdot \left|\bar{\epsilon}_6 - \bar{\epsilon}_7^{\mathsf{sar\text{-}aug}}\right|
\end{aligned} \tag{10}$$

where $\bar{\epsilon}_6$ and $\bar{\epsilon}_7^{\mathsf{sar\text{-}aug}}$ denote the conditional advantages given that the game does not abort. Following the argument in Lemma 7, we have $\Pr[\neg\mathsf{ExtractPWD}^*] = (|\mathcal{D}| - |S_q|)/|\mathcal{D}|$, where S_q is the set of distinct password hashes $\overline{\mathsf{pwd}}' \neq \overline{\mathsf{pwd}}$ obtained by $\mathcal{A}$ via queries to oracle O_{Recovers}, with $|S_q| \leq Q_{\mathsf{online}}$.

We now bound the conditional distance $\left|\bar{\epsilon}_6 - \bar{\epsilon}_7^{\mathsf{sar\text{-}aug}}\right|$. The adversary can distinguish between the two games only if it queries $(1, \mathsf{pwd})$ after full corruption. Thus, this distance is bounded by the probability that the adversary makes such a query after full corruption conditioned on the game not aborting.

Analysis in the Classical ROM. Due to Corollary 1, the simulated unused states of S_{i^*} leak no additional information on $\overline{\mathsf{pwd}}$. Also, due to the modification made in Game_6, tags and SKE keys leak no information on pwd before full corruption. Thus, conditioned on the failure of previous online attacks (i.e., $\overline{\mathsf{pwd}} \notin S_q$), the probability that $\mathcal{A}$ queries pwd after full corruption is bounded by $Q^{\mathsf{H}}_{\mathsf{post}}/(|\mathcal{D}| - |S_q|)$. Substituting this into Eq. (10), we obtain:

$$\left|\epsilon_6 - \epsilon_7^{\mathsf{sar\text{-}aug}}\right| \leq \frac{|\mathcal{D}| - |S_q|}{|\mathcal{D}|} \cdot \frac{Q^{\mathsf{H}}_{\mathsf{post}}}{|\mathcal{D}| - |S_q|} = \frac{Q^{\mathsf{H}}_{\mathsf{post}}}{|\mathcal{D}|}.$$

Analysis in the Quantum ROM. We rely on the semi-classical One-Way-to-Hiding (O2H) lemma (cf. Lemmata 1 and 2). Here, we conceptually split the adversary $\mathcal{A}$ into two phases: $\mathcal{A}_1$ (before full corruption) and $\mathcal{A}_2$ (after full corruption). To bound the conditional advantage above, we construct a reduction from the O2H game which internally runs $\mathcal{A}_2$ only. Let $\tilde{\mathcal{Z}}$ be the set of all possible challenger states just before an adversary receives the response to the corruption query for S_{i^*} such that the game does not abort. For convenience, we denote $\tilde{z} = (\mathsf{pwd}, \overline{\mathsf{pwd}}, (y_j^*)_j, \mathsf{H}_1, \mathsf{H}_2, z) \in \tilde{\mathcal{Z}}$ where $(y_j^*)_j$ is a tuple of simulated tORF outputs corresponding to unused states. In other words, z is a challenger state excluding pwd, $\overline{\mathsf{pwd}}$, tORF outputs $(y_j^*)_j$, and the descriptions of H_1 and H_2. We note that z includes the state of $\mathcal{A}_1$ and S_{i^*}'s unused states computed by SimSt. We assume that the challenger erases information that is no longer needed, such as the simulator's state and y_j^* corresponding to used tORF states.

We construct a reduction. It receives the value z contained in $\tilde{z}$ randomly chosen from $\tilde{\mathcal{Z}}$ and runs $\mathcal{A}_2$ on the appropriate inputs derived from z, including the adversary's state and S_{i^*}'s unused states. To simulate the game after full corruption, it uses the information in z and answers random oracle queries by accessing its own external oracle (which is either H_1 or H_2). Crucially, $\overline{\mathsf{pwd}}$ and pwd are not required to simulate this. Moreover, this simulation does not require the tuple $(y_j^*)_j$ of simulated tORF outputs since z includes S_{i^*}'s unused states. Thus, this reduction perfectly simulates either Game_6 or $\mathsf{Game}_7^{\text{sar-aug}}$ using only z.

Now we evaluate the advantage of the reduction relying on Lemmata 1 and 2. Specifically, we invoke these lemmas with respect to random functions H_1 and H_2 and the punctured set $\mathcal{S}_1$. Due to Corollary 1, S_{i^*}'s unused states leak no information on $\overline{\mathsf{pwd}}$, and thus pwd. Also, tags and SKE keys in z do not leak information on pwd since they are derived from the random function H_1. Finally, z only leaks what $\mathcal{A}$ can learn by performing online attacks, conditioned on their failure. Thus pwd still has the entropy $|\mathcal{D}| - |S_q|$ even though z is given. By applying Lemmata 1 and 2, we have:

$$\left|\bar{\epsilon}_6 - \bar{\epsilon}_7^{\text{sar-aug}}\right| \leq \frac{4 \cdot (Q_{\text{post}}^{\mathsf{H}} + 1)}{\sqrt{|\mathcal{D}| - |S_q|}}.$$

Finally, substituting this into Eq. (10):

$$\begin{aligned}\left|\epsilon_6 - \epsilon_7^{\text{sar-aug}}\right| &= \Pr[\neg\mathsf{ExtractPWD}^*] \cdot \left|\bar{\epsilon}_6 - \bar{\epsilon}_7^{\text{sar-aug}}\right| \\ &\leq \frac{|\mathcal{D}| - |S_q|}{|\mathcal{D}|} \cdot \frac{4 \cdot (Q_{\text{post}}^{\mathsf{H}} + 1)}{\sqrt{|\mathcal{D}| - |S_q|}} \\ &= \frac{4 \cdot (Q_{\text{post}}^{\mathsf{H}} + 1)}{\sqrt{|\mathcal{D}|}} \cdot \sqrt{\frac{|\mathcal{D}| - |S_q|}{|\mathcal{D}|}}.\end{aligned}$$

Since $\sqrt{(|\mathcal{D}| - |S_q|)/|\mathcal{D}|} \leq 1$, we conclude:

$$\left|\epsilon_6 - \epsilon_7^{\text{sar-aug}}\right| \leq \frac{4 \cdot (Q_{\text{post}}^{\mathsf{H}} + 1)}{\sqrt{|\mathcal{D}|}}.$$

This completes the proof. □

Game_6 to $\mathsf{Game}_7^{\text{aug}}$ in the QROM. .

Lemma 9. *In the proof of Theorem 5, in the QROM, we have*

$$\left|\epsilon_6 - \epsilon_7^{\text{aug}}\right| \leq \frac{4 \cdot (Q_{\text{pre}}^{\mathsf{H}} + Q_{\text{post}}^{\mathsf{H}} + 1)}{\sqrt{|\mathcal{D}|}}.$$

Proof. We show this lemma relying on the semi-classical O2H lemma (cf. Lemmata 1 and 2). Recall that H_2 and H_2' are random oracles which are exactly the same as H_1 and H_2 except on inputs in $\mathcal{S}_1$ and $(1, \mathsf{pwd})$, respectively, where $\mathcal{S}_1$ is a set of $(0, \texttt{'mode'}, \mathsf{fid} \| \mathsf{pwd} \| y_j^*)$ with $\mathtt{mode} \in \{\mathtt{SKE}, \mathtt{tag}\}$, pwd being the challenge password, and fid and y_j^* being used in Batch-Init. Now we define random functions M and M' to be used when we apply the O2H lemma. The input of M and M' is of the form $(a, b, \mathsf{input}) \in \{0,1\}^2 \times \{0,1\}^*$, and $\mathsf{M}(a, b, \mathsf{input})$ and $\mathsf{M}'(a, b, \mathsf{input})$ respectively return

$$\mathsf{M}(a, b, \mathsf{input}) = \begin{cases} \mathsf{H}_2(b, \mathsf{input}) & \text{if } a = 0 \\ \mathsf{H}_1(b, \mathsf{input}) & \text{if } a = 1 \end{cases}$$

$$\text{and} \quad \mathsf{M}'(a, b, \mathsf{input}) = \mathsf{H}_2'(b, \mathsf{input}).$$

Then, the punctured set $\mathcal{S}_2$ for the O2H lemma is

$$\mathcal{S}_2 = \{(1, b, \mathsf{input}) | (b, \mathsf{input}) \in \mathcal{S}_1\} \cup \{(a, 1, \mathsf{pwd}) | a \in \{0, 1\}\},$$

where pwd is the challenge password. Looking ahead, the reduction exploits the first bit a to switch random functions before and after full corruption by appending 0 or 1 to the adversary's random oracle queries.

Now we construct the reduction from the O2H game. The reduction receives $\overline{\mathsf{pwd}}$, all tags, SKE keys, fid, and y_j^* used in Batch-Init. Recall that $\overline{\mathsf{pwd}}$, tags, and SKE keys are derived from H_1. It simulates Game_6 or $\mathsf{Game}_7^{\text{aug}}$ by behaving as the challenger, except for the following three points. The first point is that, instead of sampling pwd and computing $\overline{\mathsf{pwd}}$ from pwd, it uses $\overline{\mathsf{pwd}}$ given as input. The second point is that it uses the received tags, SKE keys, fid, y_j^* in Batch-Init, instead of computing them honestly. The final point is that it answers the random oracle queries by using its oracle access. Specifically, upon receiving a random oracle query, it prepends 0 to the query if the query is made before full corruption, otherwise, it prepends 1. After that, it forwards it to its oracle and returns the response to the adversary. In both games, pwd is used only when the challenger computes tags and SKE keys used in Batch-Init and $\overline{\mathsf{pwd}}$, which are given via z. Thus by the definition of M and M', this reduction perfectly simulates Game_6 or $\mathsf{Game}_7^{\text{aug}}$.

Finally, we evaluate the advantage of the reduction. Since $\overline{\mathsf{pwd}}$, tags, and SKE keys in z are derived from the random function H_1, they perfectly hide the information on pwd. Also, fid and y_j^* are independent of pwd. Thus, since no information about pwd is leaked even if z is given, any punctured input $x \in \mathcal{S}_2$ has at least entropy $|\mathcal{D}|$. Therefore, due to Lemmata 1 and 2, we obtain

$$\left|\epsilon_6 - \epsilon_7^{\text{aug}}\right| \leq \frac{4 \cdot (Q_{\text{pre}}^{\mathsf{H}} + Q_{\text{post}}^{\mathsf{H}} + 1)}{\sqrt{|\mathcal{D}|}}.$$

This completes the proof. □

5.5 Adapting to General Thresholds

Extending tPPKR protocols to general thresholds turns out to be subtle. If the servers do not regularly communicate with each other, enforcing a strict rate limit (cf. Remark 3) is difficult; this is true for

known tPPKR protocols [28, 74]. Indeed, an adversary compromising $t-1$ servers can make c attempts with each of the remaining honest servers, making a total of $(n-t+1)\cdot c$ attempts.

Unique to BeaverDam is that the Beaver states must be refreshed after a successful key recover. Notably, if only $t < n$ servers are online, the client cannot refresh the states of offline servers.

Protocol variant for small t and n. The latter issue can be resolved for small n and t by running a t-out-of-t BeaverDam instance for every size-t subset of $[n]$. Server S_i holds a BeaverDam state $\mathsf{st}_{\mathsf{S}_i,S}$ for every size-t set $S \subseteq [n]$ containing i. During recovery, the client, after establishing a secure channel, includes the set of used servers in its first message (comp. Fig. 9, line 2) to indicate the BeaverDam state to be used. Each protocol instance uses different key-derivation and hash functions by means of domain separation. Note that Beaver states can be refreshed as usual, since the uploaded values only pertain to the set of servers involved in key recovery.

While we consider a formal analysis of the resulting protocol a question for future work, we anticipate it achieving the same security as n-out-of-n BeaverDam, i.e., semi-strong augmented security with respect to a security game adapted to the t-out-of-n setting, making use of the same advantage function as in Definition 15. Intuitively, this is due to every online guessing attempt consuming a single Beaver instance, and domain separation of the random oracle. Note, however, that the servers will only be able to enforce a rate limit of $c\cdot(n-t+1)$ online guesses as discussed above.

The downside of the approach is a blow up of each server's storage by a factor of $\binom{n-1}{t-1}$, the amount of sets of size t containing its index, which for general t and n is exponential. We stress that the solution should still be viable in practice, as n is typically small (e.g., Signal's SVR3 using $n=3$).

6 Instantiation and Efficiency Comparison

Lastly, we compare BeaverDam with existing tPPKR protocols. We first provide the comparison methodology below and then analyze different protocols.

6.1 Comparison Methodology

Common setup. Many (threshold) PPKR protocols establish *server*-authenticated secure channels between the client and the servers at the beginning of the protocol, e.g., [1, 2, 61, 64, 76]. This is a common practice, even though the construction of PPKR may not technically require it. They may further enforce non-cryptographic *client*-authentication (e.g., via an SMS code) to prevent malicious outsiders from exhausting a client's key recovery attempts. As the cost of setting up these communication channels is comparable for all protocols, we omit it from our efficiency analysis.

Compared protocols. To the best of our knowledge, all efficient (t)PPKR protocols rely on some variant of OPRFs. The two representative tPPKR protocols are Signal's SVR3 protocol [28] based on an OPRF and the Juicebox protocol [74] based on a tOPRF. Our BeaverDam protocol, based on a one-time tORF, also falls under this abstraction. In this section, we compare these three types of protocols by instantiating them with different concrete choices of OPRFs. For completeness, details on how these are used to construct tPPKRs and those three protocols are provided in Section 6.2.

Compared metrics. We first consider the per-server communication bandwidth, i.e., the total message size sent and received by a single server during key recovery (ignoring small protocol components like round-indicators, version numbers, or counter values). In an abuse of notation we denote this by |tPPKR.Recover|. For all analyzed protocols, the total communication bandwidth is roughly $n\cdot|\mathsf{tPPKR.Recover}|$ where n is the number of servers. The second metric is the tPPKR protocol's round complexity. We count one round when a party sends information to the other party. We instantiate each tPPKR protocol targeting 128 bits of classical and, if possible, PQ, and hybrid security. An overview is given in Table 1.

6.2 High-Level Structure of tPPKR Protocols

We give an overview on the high level structure common to practical tPPKR protocols. Recall that according to the definition of Section 5.1 each server's state after key registration consists of a server secret key sk_S and a client specific server state st_S.

Secure channel. During key recovery the client will first use the server public key pk_S corresponding to sk_S to set up a secure server-authenticated channel with each server. During this step the protocol might further employ a non-cryptographic form of client authentication to prevent malicious parties from exhausting a client's key recovery attempts. Concretely, Signal's SVR3 protocol requires the client first interact with a authentication server to obtain, by means of an SMS code, an authentication token. In turn, the token is used to establish a secure channel to the key-recovery servers. The Juicebox protocol makes use of JSON Web Tokens [58] in a similar manner. While in this work we do not model this form of weak client authentication, we note that one could augment BeaverDam in an analogous manner.

Key-recovery subprotocol. After having established secure channels with every server the client runs an interactive protocol defined with respect to the server states $\mathsf{st}_{\mathsf{S}_i}$ to recover the registered key K. Conceptually, in all protocols the servers store an encryption of K under a password-dependent key $\mathsf{k}_{\mathsf{pwd}}$ recoverable by means of an interactive protocol. Depending on the particular protocol, $\mathsf{k}_{\mathsf{pwd}}$ is derived using OPRF, tOPRF, or one-time tORF instances. The protocols enforce honest server behavior and/or explicitly authenticate the client to the servers using tags. Finally, to enforce rate limiting (cf. Remark 3) $\mathsf{st}_{\mathsf{S}_i}$ contains a counter ctr tracking the remaining recovery attempts. It is initialized to c and, upon reaching 0, leads to S_i deleting $\mathsf{st}_{\mathsf{S}_i}$. Ignoring minor components like version identifiers we can view a server S_i's state $\mathsf{st}_{\mathsf{S}_i}$ as consisting of the following (potentially empty) components.

ctr	amount of recovery attempts left
ctxt	encryption of K under $\mathsf{k}_{\mathsf{pwd}}$
st	state to recover $\mathsf{k}_{\mathsf{pwd}}$ in int. protocol
$\mathsf{tag}_{\text{S-hon}}$	enforce honest server bahaviour
$\mathsf{tag}_{\text{C-auth}}$	explicitly authenticate C to S

We now discuss the structure of tPPKR.Recover. After having established a secure channel to each server the client proceeds with the recovery of K by executing the following steps.

protocol	underl. primitive	instantiation	assumption	PQ | hybrid secure	communication cost	round compl.
SVR3 [28, 53]	OPRF	[52, 53]	omDH	× | ×	64B + $(n+1)\cdot 32$B	2
	–	[19]	DSLS, MLWE	✓ | ×	748KB + $(n+1)\cdot 32$B	9
	–	[78]	DSPRS, MLWE	✓ | ×	970KB + $(n+1)\cdot 32$B	5
Juicebox [74]	tOPRF	[52, 54]	tomDH	× | ×	368B	4
BeaverDam	one-time tORF	Beaver	none	✓✓ | ✓✓	256B + $k\cdot 224$B	5

Table 1: Comparison of tPPKR instantiations targeting 128 bit security. In column "PQ | hybrid secure", ✓ indicates security in the classical ROM, ✓✓ security in the QROM. For BeaverDam, integer k is less than the rate-limit c (typically, $c = 10$ in practice), indicating the number of prior failed key recovery attempts. The comparison ignores the communication and round complexity of establishing secure channels between the client and the servers. All protocols require the use of symmetric cryptography and the establishment of a secure channel.

(1) C recovers $\mathsf{k}_{\mathsf{pwd}}$ by running the interactive protocol, $(\mathsf{S}_i)_{i\in[n]}$ decrement ctr.
(2) $(\mathsf{S}_i)_{i\in[n]}$ send $(\mathsf{ctxt}, \mathsf{tag}_{\text{S-hon}})$, C recovers K from ctxt.
(3) C verifies $\mathsf{tag}_{\text{S-hon}}$ for all S_i.
(4) C authenticates to all S_i by recomputing $\mathsf{tag}_{\text{C-auth}}$, and replenishes st. If successful, S_i resets ctr.
(5) If all steps above were successful, C returns K. If $\mathsf{ctr} = 0$ then S_i deletes $\mathsf{st}_{\mathsf{S}_i}$.

While the compared protocols fit the presented structure, they might slightly reorder or skip some steps. In the following subsections we briefly discuss BeaverDam, SVR3, and Juicebox fit into the structure discussed above.

6.2.1 BeaverDam.

Server states. The protocol is based on one-time tORF instances. Following the abstraction above we state $\mathsf{st}_{\mathsf{S}_i}$ of server S_i is as follows. st consists of c one-time tORF states $= (\widehat{\mathsf{st}}_{\mathsf{S}_i,j})_{j\in[c]}$, and ctxt of ciphertexts $(\mathsf{ctxt}_j)_{j\in[c]}$. Here, ctxt_j is an SKE encryption of K under a key derived from the evaluation $f_j(\mathsf{pwd})$ of the j-th one-time tORF instance on pwd.

$\mathsf{tag}_{\text{S-hon}}$ for each one-time tORF instance j contains a randomly sampled function identity fid_j, the attempt counter ctr, and the ciphertext ctxt_j. Similarly, for server S_i the tag $\mathsf{tag}_{\text{C-auth}}$ contains $(\mathsf{tag}_{i,j})_{j\in[c]}$ each of which is derived from the corresponding $f_j(\mathsf{pwd})$.

Key recovery. Step 1 executes tORF.Eval with respect to one of the c one-time tORF states. Here, to save a round trip $(\mathsf{ctxt}, \mathsf{tag}_{\text{S-hon}})$ (cf. step 2) is sent alongside the last message. Step 3 corresponds to checking that the values in $\mathsf{tag}_{\text{S-hon}}$ match for all S_i. Finally, to refresh the servers' states and reset the attempt counter the client recomputes and sends to all S_i tag $\mathsf{tag}_{i,j}$. st is replenished with additional one-time tORF instances, ciphertexts, function identities, and tags and ctr is reset.

6.2.2 SVR3.

Server states. The protocol makes use of an OPRF based PPSS by Jarecki et al. [53] to handle key recovery. Following our abstraction, server state $\mathsf{st}_{\mathsf{S}_i}$ contains $\mathsf{st} = \mathsf{k}^{\mathsf{OPRF}_i}$, the key of the i-th (server-specific) OPRF. The registered key K is deterministically derived from a seed that is secret shared into n parts. The ciphertext ctxt contains the n shares s_i, each of which is encrypted using $\mathsf{OPRF}_i.\mathsf{Eval}(\mathsf{k}^{\mathsf{OPRF}_i}, \mathsf{pwd})$ as a one-time pad. The tag $\mathsf{tag}_{\text{S-hon}}$ takes the form of a commitment com to pwd, ctxt, and the secret shares encrypted by in ctxt. SVR3 does not allow for resetting the authentication-attempt counter and thus does not use tag $\mathsf{tag}_{\text{C-auth}}$.

Key recovery. Step 1 executes $\mathsf{OPRF}_i.\mathsf{Eval}$ to be able to recover the secret shares contained in ctxt and, in turn, K. The values ctxt and $\mathsf{tag}_{\text{S-hon}}$ are sent alongside the last message of $\mathsf{OPRF}_i.\mathsf{Eval}$. After verifying the commitment in step 3, the protocol returns K. Since the protocol does not reset the attempt counter, it does not require explicit client authentication and thus omits step 4.

6.2.3 Juicebox.

Server states. Juicebox makes use of a tOPRF. Following our abstraction, the server state $\mathsf{st}_{\mathsf{S}_i}$ contains the following. st consists of $\mathsf{k}_i^{\mathsf{tOPRF}}$, the i-th key-share for tOPRF, as well as a secret share s_i of a salt s. $\mathsf{k}_{\mathsf{pwd}}$ essentially is a salted hash of tOPRF.Eval(pwd). The ciphertext ctxt is an SKE encryption of K under $\mathsf{k}_{\mathsf{pwd}}$. The tag $\mathsf{tag}_{\text{S-hon}}$ essentially takes the form of general and server specific commitments com and com′ to all values involved in the recovery of K, a signature σ, and the corresponding verification key vk. Tag $\mathsf{tag}_{\text{C-auth}}$ is essentially a tag tag derived from $\mathsf{k}_{\mathsf{pwd}}$.

Key recovery. In step 1 tOPRF.Eval is executed. The protocol slightly breaks with our abstraction, only partially performing steps 2 and 3 at this point. I.e., the servers send com, σ and vk along the last message, which are in turn verified by the client. The client authenticates to the servers recomputing and sending tag (step 4). If authentication is successful the servers reset the attempt counter and complete step 2 by sending ctxt, s_i and the missing part com′ of $\mathsf{tag}_{\text{S-hon}}$, which allows the client to recover K and complete the verification of $\mathsf{tag}_{\text{S-hon}}$ (step 3). Summing up, Juicebox.Recover requires rounds(tOPRF.Eval) + 2 rounds and communication

$$|\mathsf{Juicebox.Recover}| = |\mathsf{tOPRF.Eval}| + |\mathsf{ctxt}| + |\mathsf{com}| + |s_i| + |\mathsf{com}'| + |\sigma| + |\mathsf{vk}| + |\mathsf{tag}|.$$

6.3 Efficiency of Various tPPKR Protocols

6.3.1 BeaverDam. BeaverDam's key recovery protocol is based purely on symmetric cryptography once the secure channel is set up. That is, we can switch between the desired security level (i.e., classical, PQ, or hybrid security) by instantiating the secure channels accordingly. The protocol (cf. Fig. 9) is 5 rounds with a communication cost of

$$\begin{aligned}&|\mathsf{BeaverDam.Recover}|\\ =&|\mathsf{Beaver.Recover}| + |\mathsf{ctxt}| + |\mathsf{fid}| + |\mathsf{tag}| + k\cdot\left(\left|\widehat{\mathsf{st}}_{\mathsf{S}}\right| + |\mathsf{ctxt}| + |\mathsf{fid}| + |\mathsf{tag}|\right),\end{aligned}$$

where $k \in [c]$ indicates the number of prior failed key recovery attempts. Targeting 128 bits of security, we instantiate the degree-one affine function supported by Beaver with a 32B prime p, use SHA-256 [66] to derive 32B tags $\text{tag}_{i,j}$, 32B function identities fid, and the key-committing AEAD [26] as the robust and IND-CPA secure SKE for encryption resulting in 64B ciphertexts for 32B plaintexts. Combining everything, the communication cost is 256B+ $k \cdot 224$B in 5 rounds.

6.3.2 Signal's SVR3. This is a tPPKR protocol designed to be used as part of the Signal secure messaging application. At its core, the protocol uses a password-protected secret sharing scheme (PPSS) by Jarecki et al. [53]. While [28] instantiates the scheme with the classically secure OPRF called 2HashDH[52], for $n = 3$ servers, we will describe its efficiency for general n and additionally discuss instantiations targeting PQ and hybrid security.

The protocol's round complexity is inherited from the underlying OPRF and its communication cost is $|\text{SVR3.Recover}| = |\text{OPRF.Eval}| + n \cdot |s_i| + |\text{com}|$, where $n \cdot |s_i|$ stems from the ciphertext encrypting K. It essentially consist of secret shares s_i of K each of which is one-time padded by $\text{OPRF}(\text{k}_{\text{S}_i}, \text{pwd})$. The shares s_i and the commitment com are of size 32B.

Unlike our information theoretical one-time tORF Beaver, SVR3 relies on a *computational* OPRF, determining the level of security attained by SVR3. We discuss several instantiations below.

Classical security. The non-verifiable version of 2HashDH [52, 53] based on the one-more Diffie Hellman (omDH) assumption achieves classical security. This is used by [28], with a round complexity 2 and communication cost $|\text{2HashDH.Eval}| = 64$B.

PQ security. Regarding PQ secure instantiations, [53] requires an OPRF that is *UC secure* against malicious servers. The only known PQ OPRFs with such security are 2H-OPRF [19] and Gold-OPRF [78]. These are based on the MLWE and Decisional Shifted Legendre Symbol (DSLS), or MLWE and Decisional Shifted Power-Residue Symbol (DSPRS) assumptions, respectively. The former scheme has round complexity 9 and communication cost 748KB, the latter round complexity 5 and communication cost 970KB.

Hybrid security. Recently, Faller et al. [36] proposed hybrid secure OPRFs by combining 2HashDH with variants of either 2H-OPRF or Gold-OPRF. In contrast to [19, 78] the schemes have to make use of a *statistically secure* oblivious transfer. While [36] does not state concrete parameter sizes, the resulting OPRFs will suffer from larger communication costs due to the use of this stronger primitive.

6.3.3 Juicebox. The Juicebox protocol [74] implements a distributed storage and recovery of secrets mechanism from PIN authentication. It builds upon a tOPRF, a signature scheme, and symmetric cryptography. The classical instantiation proposed in [74] has round complexity 4 and communication cost $|\text{Juicebox.Recover}| = 368$B

Regarding a PQ or hybrid secure instantiation of Juicebox, while no formal security analysis of the protocol is given (even in the classical setting), it seems that a UC secure tOPRF is required, similarly to SVR3. To our knowledge, there are no proposals for PQ secure tOPRFs with UC security, and therefore, we leave a PQ instantiation of Juicebox as an interesting open problem.

6.4 Conclusion

For an overview on all discussed instantiation see Table 1. Opposed to the other protocols, BeaverDam requires no additional assumptions beyond symmetric cryptography and the establishment of a secure channel. As a nice byproduct, BeaverDam can be made hybrid secure by relying on a hybrid secure channel. This, however, comes at the cost of achieving semi-strong augmented security instead of strong augmented security (see Section 5.4).

Targeting PQ security, BeaverDam substantially outperforms other protocols in terms of communication cost with the concrete improvement depending on k, the amount of Beaver states that need to be replenished. This value is bounded by the rate-limit c, which e.g., SVR3 sets to $c = 10$. In concrete terms, BeaverDam has communication cost smaller by a factor in the range of 300x to 1500x. It also performs as well as, or better, in terms of round complexity compared to other PQ secure tPPKR instantiations.

Finally, also taking classical security into account, 2HashDH based SVR3 performs best in terms of round complexity and communication cost. For $n = 3$, as proposed in [28], and not too large values of $k \in [10]$ BeaverDam has an increased communication complexity by a factor of 2.5x to 13x. We note, that in practice one would not expect users to mistype their password too often, so typically BeaverDam's communication cost should stay closer to the lower end.

Acknowledgments

We thank the anonymous reviewers of ACM CCS for their valuable feedback. In particular, we are grateful to the reviewer who suggested the way to optimize our original construction (cf. Footnote 7). This work was supported in part by JST CREST Grant Number JPMJCR22M1 and JST K Program Grant Number JPMJKP24U3.

References

[1] 2021. *Security of End-to-End Encrypted Backups.* Technical Report. WhatsApp. Accessed August 17, 2025.
[2] Meta 2024. *End-to-End Encryption on Messenger Explained.* Meta. https://about.fb.com/news/2024/03/end-to-end-encryption-on-messenger-explained/
[3] Michel Abdalla, Mihir Bellare, and Gregory Neven. 2010. Robust Encryption. In *TCC 2010 (LNCS, Vol. 5978)*, Daniele Micciancio (Ed.). Springer, Berlin, Heidelberg, 480–497. doi:10.1007/978-3-642-11799-2_28
[4] Michel Abdalla, Mario Cornejo, Anca Nitulescu, and David Pointcheval. 2016. Robust Password-Protected Secret Sharing. In *ESORICS 2016, Part II (LNCS, Vol. 9879)*, Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows (Eds.). Springer, Cham, 61–79. doi:10.1007/978-3-319-45741-3_4
[5] Shashank Agrawal, Peihan Miao, Payman Mohassel, and Pratyay Mukherjee. 2018. PASTA: PASsword-based Threshold Authentication. In *ACM CCS 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, 2042–2059. doi:10.1145/3243734.3243839
[6] Navid Alamati, Guru-Vamsi Policharla, Srinivasan Raghuraman, and Peter Rindal. 2024. Improved Alternating-Moduli PRFs and Post-quantum Signatures. In *CRYPTO 2024, Part VIII (LNCS, Vol. 14927)*, Leonid Reyzin and Douglas Stebila (Eds.). Springer, Cham, 274–308. doi:10.1007/978-3-031-68397-8_9
[7] Martin R. Albrecht, Alex Davidson, Amit Deo, and Daniel Gardham. 2024. Crypto Dark Matter on the Torus - Oblivious PRFs from Shallow PRFs and TFHE. In *EUROCRYPT 2024, Part VI (LNCS, Vol. 14656)*, Marc Joye and Gregor Leander (Eds.). Springer, Cham, 447–476. doi:10.1007/978-3-031-58751-1_16
[8] Martin R. Albrecht, Alex Davidson, Amit Deo, and Nigel P. Smart. 2021. Round-Optimal Verifiable Oblivious Pseudorandom Functions from Ideal Lattices. In *PKC 2021, Part II (LNCS, Vol. 12711)*, Juan Garay (Ed.). Springer, Cham, 261–289. doi:10.1007/978-3-030-75248-4_10
[9] Andris Ambainis, Mike Hamburg, and Dominique Unruh. 2019. Quantum Security Proofs Using Semi-classical Oracles. In *CRYPTO 2019, Part II (LNCS, Vol. 11693)*, Alexandra Boldyreva and Daniele Micciancio (Eds.). Springer, Cham, 269–295. doi:10.1007/978-3-030-26951-7_10

[10] Apple Security Engineering and Architecture (SEAR). 2024. iMessage with PQ3: The new state of the art in quantum-secure messaging at scale. Available at https://security.apple.com/blog/imessage-pq3/.

[11] Benedikt Auerbach, Yevgeniy Dodis, Daniel Jost, Shuichi Katsumata, and Rolfe Schmidt. 2025. How to Compare Bandwidth Constrained Two-Party Secure Messaging Protocols: A Quest for A More Efficient and Secure Post-Quantum Protocol. In *USENIX Security 2025*, Lujo Bauer and Giancarlo Pellegrino (Eds.). USENIX Association, 6717–6736. https://www.usenix.org/conference/usenixsecurity25/presentation/auerbach

[12] Ruben Baecker, Paul Gerhart, Daniel Rausch, and Dominique Schröder. 2025. A Fully-Adaptive Threshold Partially-Oblivious PRF. In *CRYPTO 2025, Part V (LNCS, Vol. 16004)*, Yael Tauman Kalai and Seny F. Kamara (Eds.). Springer, Cham, 569–597. doi:10.1007/978-3-032-01901-1_18

[13] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. 2011. Password-protected secret sharing. In *ACM CCS 2011*, Yan Chen, George Danezis, and Vitaly Shmatikov (Eds.). ACM Press, 433–444. doi:10.1145/2046707.2046758

[14] Andrea Basso. 2024. POKE: A Framework for Efficient PKEs, Split KEMs, and OPRFs from Higher-dimensional Isogenies. Cryptology ePrint Archive, Report 2024/624. https://eprint.iacr.org/2024/624

[15] Carsten Baum, Tore Kasper Frederiksen, Julia Hesse, Anja Lehmann, and Avishay Yanai. 2020. PESTO: Proactively Secure Distributed Single Sign-On, or How to Trust a Hacked Server. In *2020 IEEE European Symposium on Security and Privacy*. IEEE Computer Society Press, 587–606. doi:10.1109/EuroSP48549.2020.00044

[16] Donald Beaver. 1992. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO'91 (LNCS, Vol. 576)*, Joan Feigenbaum (Ed.). Springer, Berlin, Heidelberg, 420–432. doi:10.1007/3-540-46766-1_34

[17] Mihir Bellare, David Pointcheval, and Phillip Rogaway. 2000. Authenticated Key Exchange Secure against Dictionary Attacks. In *EUROCRYPT 2000 (LNCS, Vol. 1807)*, Bart Preneel (Ed.). Springer, Berlin, Heidelberg, 139–155. doi:10.1007/3-540-45539-6_11

[18] Mihir Bellare and Phillip Rogaway. 1994. Entity Authentication and Key Distribution. In *CRYPTO'93 (LNCS, Vol. 773)*, Douglas R. Stinson (Ed.). Springer, Berlin, Heidelberg, 232–249. doi:10.1007/3-540-48329-2_21

[19] Ward Beullens, Lucas Dodgson, Sebastian H. Faller, and Julia Hesse. 2025. The 2Hash OPRF Framework and Efficient Post-quantum Instantiations. In *EUROCRYPT 2025, Part VIII (LNCS, Vol. 15608)*, Serge Fehr and Pierre-Alain Fouque (Eds.). Springer, Cham, 332–362. doi:10.1007/978-3-031-91101-9_12

[20] Nina Bindel, Mike Hamburg, Kathrin Hövelmanns, Andreas Hülsing, and Edoardo Persichetti. 2019. Tighter Proofs of CCA Security in the Quantum Random Oracle Model. In *TCC 2019, Part II (LNCS, Vol. 11892)*, Dennis Hofheinz and Alon Rosen (Eds.). Springer, Cham, 61–90. doi:10.1007/978-3-030-36033-7_3

[21] Simon Blake-Wilson, Don Johnson, and Alfred Menezes. 1997. Key Agreement Protocols and Their Security Analysis. In *6th IMA International Conference on Cryptography and Coding (LNCS, Vol. 1355)*, Michael Darnell (Ed.). Springer, Berlin, Heidelberg, 30–45. doi:10.1007/bfb0024447

[22] Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. 2011. Random Oracles in a Quantum World. In *ASIACRYPT 2011 (LNCS, Vol. 7073)*, Dong Hoon Lee and Xiaoyun Wang (Eds.). Springer, Berlin, Heidelberg, 41–69. doi:10.1007/978-3-642-25385-0_3

[23] Jan Camenisch, Anja Lehmann, Anna Lysyanskaya, and Gregory Neven. 2014. Memento: How to Reconstruct Your Secrets from a Single Password in a Hostile Environment. In *CRYPTO 2014, Part II (LNCS, Vol. 8617)*, Juan A. Garay and Rosario Gennaro (Eds.). Springer, Berlin, Heidelberg, 256–275. doi:10.1007/978-3-662-44381-1_15

[24] Ran Canetti and Hugo Krawczyk. 2001. Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels. In *EUROCRYPT 2001 (LNCS, Vol. 2045)*, Birgit Pfitzmann (Ed.). Springer, Berlin, Heidelberg, 453–474. doi:10.1007/3-540-44987-6_28

[25] Sílvia Casacuberta, Julia Hesse, and Anja Lehmann. 2022. SoK: Oblivious Pseudorandom Functions. In *2022 IEEE European Symposium on Security and Privacy*. IEEE Computer Society Press, 625–646. doi:10.1109/EuroSP53844.2022.00045

[26] John Chan and Phillip Rogaway. 2022. On Committing Authenticated-Encryption. In *ESORICS 2022, Part II (LNCS, Vol. 13555)*, Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng (Eds.). Springer, Cham, 275–294. doi:10.1007/978-3-031-17146-8_14

[27] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia. 2021. VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface. In *USENIX Security 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 699–716. https://www.usenix.org/conference/usenixsecurity21/presentation/chen-zitai

[28] Graeme Connell, Vivian Fang, Rolfe Schmidt, Emma Dauterman, and Raluca Ada Popa. 2024. Secret Key Recovery in a Global-Scale End-to-End Encryption System. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 703–719. https://www.usenix.org/conference/osdi24/presentation/connell

[29] Poulami Das, Julia Hesse, and Anja Lehmann. 2022. DPaSE: Distributed Password-Authenticated Symmetric-Key Encryption, or How to Get Many Keys from One Password. In *ASIACCS 22*, Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako (Eds.). ACM Press, 682–696. doi:10.1145/3488932.3517389

[30] Gareth T. Davies, Sebastian H. Faller, Kai Gellert, Tobias Handirk, Julia Hesse, Máté Horváth, and Tibor Jager. 2023. Security Analysis of the WhatsApp End-to-End Encrypted Backup Protocol. In *CRYPTO 2023, Part IV (LNCS, Vol. 14084)*, Helena Handschuh and Anna Lysyanskaya (Eds.). Springer, Cham, 330–361. doi:10.1007/978-3-031-38551-3_11

[31] Dennis Dayanikli and Anja Lehmann. 2024. (Strong) aPAKE Revisited: Capturing Multi-User Security and Salting. In *2024 IEEE European Symposium on Security and Privacy*. IEEE Computer Society Press, 415–439. doi:10.1109/EuroSP60621.2024.00030

[32] Cyprien Delpech de Saint Guilhem and Robi Pedersen. 2024. New Proof Systems and an OPRF from CSIDH. In *PKC 2024, Part II (LNCS, Vol. 14603)*, Qiang Tang and Vanessa Teague (Eds.). Springer, Cham, 217–251. doi:10.1007/978-3-031-57725-3_8

[33] Yevgeniy Dodis, Daniel Jost, Shuichi Katsumata, Thomas Prest, and Rolfe Schmidt. 2025. Triple Ratchet: A Bandwidth Efficient Hybrid-Secure Signal Protocol. In *EUROCRYPT*. Springer, 302–331.

[34] Stefan Dziembowski, Stanislaw Jarecki, Pawel Kedzior, Hugo Krawczyk, Chan Nam Ngo, and Jiayu Xu. 2024. Password-Protected Threshold Signatures. In *ASIACRYPT 2024, Part III (LNCS, Vol. 15486)*, Kai-Min Chung and Yu Sasaki (Eds.). Springer, Singapore, 174–206. doi:10.1007/978-981-96-0891-1_6

[35] Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. 2015. The Pythia PRF Service. In *USENIX Security 2015*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 547–562. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/everspaugh

[36] Sebastian Faller, Marc Fischlin, Julius Hardt, and Julia Hesse. 2025. Combining Oblivious Pseudorandom Functions. Cryptology ePrint Archive, Report 2025/1084. https://eprint.iacr.org/2025/1084

[37] Sebastian H. Faller, Tobias Handirk, Julia Hesse, Máté Horváth, and Anja Lehmann. 2024. Password-Protected Key Retrieval with(out) HSM Protection. In *ACM CCS 2024*, Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie (Eds.). ACM Press, 2445–2459. doi:10.1145/3658644.3690358

[38] Sebastian H. Faller, Astrid Ottenhues, and Johannes Ottenhues. 2023. Composable Oblivious Pseudo-random Functions via Garbled Circuits. In *LATINCRYPT 2023 (LNCS, Vol. 14168)*, Abdelrahaman Aly and Mehdi Tibouchi (Eds.). Springer, Cham, 249–270. doi:10.1007/978-3-031-44469-2_13

[39] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. 2005. Keyword Search and Oblivious Pseudorandom Functions. In *TCC 2005 (LNCS, Vol. 3378)*, Joe Kilian (Ed.). Springer, Berlin, Heidelberg, 303–324. doi:10.1007/978-3-540-30576-7_17

[40] Karla Friedrichs, Anja Lehmann, and Cavit Özbay. 2025. Game Changer: A Modular Framework for OPRF Security. To appear in ASIACRYPT 2025. https://eprint.iacr.org/2025/1565

[41] Atsushi Fujioka, Koutarou Suzuki, Keita Xagawa, and Kazuki Yoneyama. 2012. Strongly Secure Authenticated Key Exchange from Factoring, Codes, and Lattices. In *PKC 2012 (LNCS, Vol. 7293)*, Marc Fischlin, Johannes Buchmann, and Mark Manulis (Eds.). Springer, Berlin, Heidelberg, 467–484. doi:10.1007/978-3-642-30057-8_28

[42] Peter Gaži and Stefano Tessaro. 2015. Secret-key cryptography from ideal primitives: A systematic overview. In *2015 IEEE Information Theory Workshop (ITW)*. IEEE, 1–5.

[43] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *19th ACM STOC*, Alfred Aho (Ed.). ACM Press, 218–229. doi:10.1145/28395.28420

[44] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. 2018. Another Flip in the Wall of Rowhammer Defenses. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 245–261. doi:10.1109/SP.2018.00031

[45] Yanqi Gu, Stanislaw Jarecki, Pawel Kedzior, Phillip Nazarian, and Jiayu Xu. 2024. Threshold PAKE with Security Against Compromise of All Servers. In *ASIACRYPT 2024, Part V (LNCS, Vol. 15488)*, Kai-Min Chung and Yu Sasaki (Eds.). Springer, Singapore, 66–100. doi:10.1007/978-981-96-0935-2_3

[46] Keitaro Hashimoto, Shuichi Katsumata, Kris Kwiatkowski, and Thomas Prest. 2022. An Efficient and Generic Construction for Signal's Handshake (X3DH): Post-quantum, State Leakage Secure, and Deniable. *Journal of Cryptology* 35, 3 (July 2022), 17. doi:10.1007/s00145-022-09427-1

[47] Carmit Hazay. 2015. Oblivious Polynomial Evaluation and Secure Set-Intersection from Algebraic PRFs. In *TCC 2015, Part II (LNCS, Vol. 9015)*, Yevgeniy Dodis and Jesper Buus Nielsen (Eds.). Springer, Berlin, Heidelberg, 90–120. doi:10.1007/978-3-662-46497-7_4

[48] Carmit Hazay and Yehuda Lindell. 2010. Efficient Protocols for Set Intersection and Pattern Matching with Security Against Malicious and Covert Adversaries. *Journal of Cryptology* 23, 3 (July 2010), 422–456. doi:10.1007/s00145-008-9034-x

[49] Lena Heimberger, Daniel Kales, Riccardo Lolato, Omid Mir, Sebastian Ramacher, and Christian Rechberger. 2025. Leap: A Fast, Lattice-Based OPRF with Application to Private Set Intersection. In *EUROCRYPT 2025, Part VII (LNCS,*

Vol. 15607), Serge Fehr and Pierre-Alain Fouque (Eds.). Springer, Cham, 254–283. doi:10.1007/978-3-031-91098-2_10

[50] Kathrin Hövelmanns, Eike Kiltz, Sven Schäge, and Dominique Unruh. 2020. Generic Authenticated Key Exchange in the Quantum Random Oracle Model. In *PKC 2020, Part II (LNCS, Vol. 12111)*, Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas (Eds.). Springer, Cham, 389–422. doi:10.1007/978-3-030-45388-6_14

[51] Tibor Jager, Eike Kiltz, Doreen Riepel, and Sven Schäge. 2021. Tightly-Secure Authenticated Key Exchange, Revisited. In *EUROCRYPT 2021, Part I (LNCS, Vol. 12696)*, Anne Canteaut and François-Xavier Standaert (Eds.). Springer, Cham, 117–146. doi:10.1007/978-3-030-77870-5_5

[52] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. 2014. Round-Optimal Password-Protected Secret Sharing and T-PAKE in the Password-Only Model. In *ASIACRYPT 2014, Part II (LNCS, Vol. 8874)*, Palash Sarkar and Tetsu Iwata (Eds.). Springer, Berlin, Heidelberg, 233–253. doi:10.1007/978-3-662-45608-8_13

[53] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. 2016. Highly-Efficient and Composable Password-Protected Secret Sharing (Or: How to Protect Your Bitcoin Wallet Online). In *2016 IEEE European Symposium on Security and Privacy*. IEEE Computer Society Press, 276–291. doi:10.1109/EuroSP.2016.30

[54] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. 2017. TOPPSS: Cost-Minimal Password-Protected Secret Sharing Based on Threshold OPRF. In *ACNS 2017 (LNCS, Vol. 10355)*, Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi (Eds.). Springer, Cham, 39–58. doi:10.1007/978-3-319-61204-1_3

[55] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. 2018. OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-computation Attacks. In *EUROCRYPT 2018, Part III (LNCS, Vol. 10822)*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer, Cham, 456–486. doi:10.1007/978-3-319-78372-7_15

[56] Stanislaw Jarecki and Xiaomin Liu. 2009. Efficient Oblivious Pseudorandom Function with Applications to Adaptive OT and Secure Computation of Set Intersection. In *TCC 2009 (LNCS, Vol. 5444)*, Omer Reingold (Ed.). Springer, Berlin, Heidelberg, 577–594. doi:10.1007/978-3-642-00457-5_34

[57] Stanislaw Jarecki and Phillip Nazarian. 2025. Adaptively Secure Threshold Blind BLS Signatures and Threshold Oblivious PRF. *IACR Cryptol. ePrint Arch.* (2025), 483. https://eprint.iacr.org/2025/483

[58] Michael Jones, John Bradley, and Nat Sakimura. 2015. Rfc 7519: Json web token (jwt).

[59] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. 2016. Efficient Batched Oblivious PRF with Applications to Private Set Intersection. In *ACM CCS 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, 818–829. doi:10.1145/2976749.2978381

[60] Ehren Kret and Rolfe Schmidt. 2023. The PQXDH Key Agreement Protocol. Available at https://signal.org/docs/specifications/pqxdh/.

[61] Ivan Krstić. 2016. Behind the Scenes of iOS Security. In *Black Hat USA*. https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf Presentation Slides.

[62] Russell W. F. Lai, Christoph Egger, Dominique Schröder, and Sherman S. M. Chow. 2017. Phoenix: Rebirth of a Cryptographic Password-Hardening Service. In *USENIX Security 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 899–916. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lai

[63] Brian A. LaMacchia, Kristin Lauter, and Anton Mityagin. 2007. Stronger Security of Authenticated Key Exchange. In *ProvSec 2007 (LNCS, Vol. 4784)*, Willy Susilo, Joseph K. Liu, and Yi Mu (Eds.). Springer, Berlin, Heidelberg, 1–16. doi:10.1007/978-3-540-75670-5_1

[64] Joshua Lund. 2019. *Technology Preview for Secure Value Recovery*. Signal. https://signal.org/blog/secure-value-recovery/

[65] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1466–1482. doi:10.1109/SP40000.2020.00057

[66] National Institute of Standards, Technology (NIST), and Quynh Dang. 2015. Secure Hash Standard. doi:10.6028/NIST.FIPS.180-4

[67] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2021. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1852–1867. doi:10.1109/SP40001.2021.00020

[68] Partha Sarathi Roy, Sabyasachi Dutta, Willy Susilo, and Reihaneh Safavi-Naini. 2021. Password Protected Secret Sharing from Lattices. In *ACNS 2021, Part I (LNCS, Vol. 12726)*, Kazue Sako and Nils Ole Tippenhauer (Eds.). Springer, Cham, 442–459. doi:10.1007/978-3-030-78372-3_17

[69] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, 753–768. doi:10.1145/3319535.3354252

[70] István András Seres, Máté Horváth, and Péter Burcsi. 2021. The Legendre Pseudorandom Function as a Multivariate Quadratic Cryptosystem: Security and Applications. Cryptology ePrint Archive, Report 2021/182. https://eprint.iacr.org/2021/182

[71] Signal Foundation. 2019. SecureValueRecovery2. https://github.com/signalapp/SecureValueRecovery2. GitHub repository; Accessed August 20, 2025.

[72] Animesh Singh, Sikhar Patranabis, and Debdeep Mukhopadhyay. 2024. Threshold OPRF from Threshold Additive HE. Cryptology ePrint Archive, Report 2024/1032. https://eprint.iacr.org/2024/1032

[73] Nora Trapp. 2024. *Key to Simplicity: Squeezing the hassle out of encryption key recovery*. https://juicebox.xyz/blog/key-to-simplicity-squeezing-the-hassle-out-of-encryption-key-recovery

[74] Nora Trapp and Diego Ongaro. 2025. Juicebox Protocol: Distributed Storage and Recovery of Secrets Using Simple PIN Authentication. Cryptology ePrint Archive, Report 2025/348. https://eprint.iacr.org/2025/348

[75] Nirvan Tyagi, Sofía Celi, Thomas Ristenpart, Nick Sullivan, Stefano Tessaro, and Christopher A. Wood. 2022. A Fast and Simple Partially Oblivious PRF, with Applications. In *EUROCRYPT 2022, Part II (LNCS, Vol. 13276)*, Orr Dunkelman and Stefan Dziembowski (Eds.). Springer, Cham, 674–705. doi:10.1007/978-3-031-07085-3_23

[76] Shabsi Walfish. 2018. *Google Cloud Key Vault Service*. Google. https://developer.android.com/about/versions/pie/security/ckv-whitepaper

[77] Shaoqiang Wu and Ding Wang. 2024. Universally Composable and Reliable Password Hardening Services. Cryptology ePrint Archive, Paper 2024/1912. https://eprint.iacr.org/2024/1912

[78] Yibin Yang, Fabrice Benhamouda, Shai Halevi, Hugo Krawczyk, and Tal Rabin. 2025. Gold OPRF: Post-Quantum Oblivious Power-Residue PRF. In *2025 IEEE Symposium on Security and Privacy*, Marina Blanton, William Enck, and Cristina Nita-Rotaru (Eds.). IEEE Computer Society Press, 259–278. doi:10.1109/SP61157.2025.00116

[79] Xun Yi, Feng Hao, Liqun Chen, and Joseph K. Liu. 2015. Practical Threshold Password-Authenticated Secret Sharing Protocol. In *ESORICS 2015, Part I (LNCS, Vol. 9326)*, Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl (Eds.). Springer, Cham, 347–365. doi:10.1007/978-3-319-24174-6_18

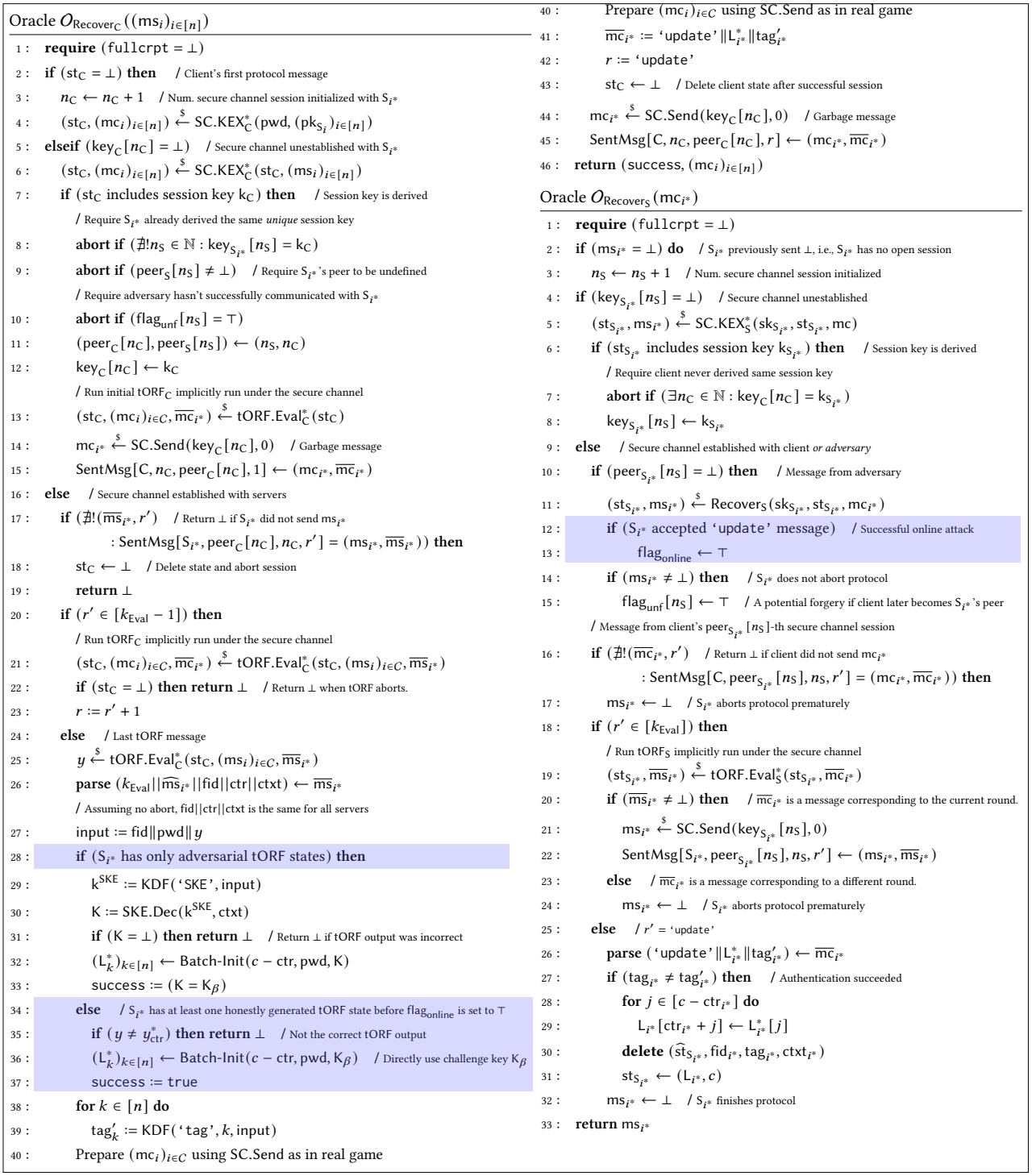

Figure 10: The descriptions of O_{Recover_C} and O_{Recover_S} in Game_2 and Game_3. Lines highlighted in blue appear only in Game_3. SC.KEX^*_C and tORF.Eval^*_C (resp., SC.KEX^*_S and tORF.Eval^*_S) check the validity of inputs, internally run SC.KEX_C and tORF.Eval_C (resp., SC.KEX_S and tORF.Eval_S), and appropriately manage st_C (resp., $\text{st}_{S_{i^*}}$). tORF.Eval^*_S additionally checks if $\text{ctr}_i \neq 0$ before the one-time tORF session. For the definitions of counters and lists, see the proof of Lemma 4. The value y^*_{ctr} is the ctr-th value of $f(\overline{\text{pwd}})$ that is honestly generated by the challenger before $\text{flag}_{\text{online}}$ is set to ⊤.

Oracle $O_{\mathsf{Recover_C}}((\mathsf{ms}_i)_{i\in[n]})$

/ Same as Game_3 up to line 12

/ Run initial $\mathsf{tORF_C}$ implicitly run under the secure channel

13 : **if** (S_{i^*} has simulated tORF states) **then**

14 : $(\mathsf{st_C}, (\mathsf{mc}_i)_{i\in C}) \xleftarrow{\$} \mathsf{tORF.SimEval}^*_{\mathsf{C}}(\mathsf{st_C})$

15 : $\overline{\mathsf{mc}}_{i^*} := \bot$

16 : **else**

17 : $(\mathsf{st_C}, (\mathsf{mc}_i)_{i\in C}, \overline{\mathsf{mc}}_{i^*}) \xleftarrow{\$} \mathsf{tORF.Eval}^*_{\mathsf{C}}(\mathsf{st_C})$

18 : $\mathsf{mc}_{i^*} \xleftarrow{\$} \mathsf{SC.Send}(\mathsf{key_C}[n_\mathsf{C}], 0)$ / Garbage message

19 : $\mathsf{SentMsg}[\mathsf{C}, n_\mathsf{C}, \mathsf{peer_C}[n_\mathsf{C}], 1] \leftarrow (\mathsf{mc}_{i^*}, \overline{\mathsf{mc}}_{i^*})$

20 : **else** / Secure channel established with servers

21 : **if** $(\nexists!(\overline{\mathsf{ms}}_{i^*}, r')$ / Return $\bot$ if S_{i^*} did not send ms_{i^*}

: $\mathsf{SentMsg}[\mathsf{S}_{i^*}, \mathsf{peer_C}[n_\mathsf{C}], n_\mathsf{C}, r'] = (\mathsf{ms}_{i^*}, \overline{\mathsf{ms}}_{i^*}))$ **then**

22 : $\mathsf{st_C} \leftarrow \bot$ / Delete state and abort session

23 : **return** $\bot$

/ Only simulate while S_{i^*} has simulated tORF states

24 : **if** $(r' \in [k_{\mathsf{Eval}} - 1]) \wedge (\mathsf{S}_{i^*}$ has simulated tORF states) **then**

/ Run $\mathsf{tORF_C}$ implicitly run under the secure channel

25 : $(\mathsf{st_C}, (\mathsf{mc}_i)_{i\in C}) \xleftarrow{\$} \mathsf{tORF.SimEval}^*_{\mathsf{C}}(\mathsf{st_C}, (\mathsf{ms}_i)_{i\in C}, \overline{\mathsf{ms}}_{i^*})$

26 : **if** $(\mathsf{st_C} = \bot)$ **then return** $\bot$ / Return $\bot$ when tORF aborts.

27 : $r := r' + 1$

28 : $\overline{\mathsf{mc}}_{i^*} := \bot$

29 : **elseif** $(r' = k_{\mathsf{Eval}}) \wedge (\mathsf{S}_{i^*}$ has simulated tORF states) **then**

30 : $(\mathsf{st_C}, (\mathsf{mc}_i)_{i\in C}) \xleftarrow{\$} \mathsf{tORF.SimEval}^*_{\mathsf{C}}(\mathsf{st_C}, (\mathsf{ms}_i)_{i\in C}, \overline{\mathsf{ms}}_{i^*})$

31 : $b \xleftarrow{\$} \mathsf{tORF.CorrectVal}^*(\mathsf{st_C})$ / Check validity of tORF output without ctxt

32 : **parse** $(k_{\mathsf{Eval}}||\widehat{\mathsf{ms}}_{i^*}||\mathsf{fid}||\mathsf{ctr}||\mathsf{ctxt}) \leftarrow \overline{\mathsf{ms}}_{i^*}$

/ Assuming no abort, fid||ctr||ctxt is the same for all servers

33 : $\mathsf{input} := \mathsf{fid}\|\mathsf{pwd}\|y$

34 : **if** $(b = 0)$ **then return** $\bot$ / Not the correct tORF output

35 : $(\mathsf{L}^*_k)_{k\in[n]} \leftarrow \mathsf{Batch\text{-}Init}^*(c - \mathsf{ctr}, \mathsf{pwd}, \mathsf{K}_\beta)$

36 : $\mathsf{success} := \mathsf{true}$

37 : **for** $k \in [n]$ **do**

38 : $\mathsf{tag}'_k := \mathsf{KDF}(\text{`tag'}, k, \mathsf{input})$

39 : Prepare $(\mathsf{mc}_i)_{i\in C}$ using SC.Send as in real game

40 : $\overline{\mathsf{mc}}_{i^*} := \text{`update'}\|\mathsf{L}^*_{i^*}\|\mathsf{tag}'_{i^*}$

41 : $r := \text{`update'}$

42 : $\mathsf{st_C} \leftarrow \bot$ / Delete client state after successful session

43 : **else** / S_{i^*} message ms_{i^*} is from an adversarial tORF state

44 : Same as line 20 to line 43 of previous Game_3

45 : $\mathsf{mc}_{i^*} \xleftarrow{\$} \mathsf{SC.Send}(\mathsf{key_C}[n_\mathsf{C}], 0)$ / Garbage message

46 : $\mathsf{SentMsg}[\mathsf{C}, n_\mathsf{C}, \mathsf{peer_C}[n_\mathsf{C}], r] \leftarrow (\mathsf{mc}_{i^*}, \overline{\mathsf{mc}}_{i^*})$

47 : **return** $(\mathsf{success}, (\mathsf{mc}_i)_{i\in[n]})$

Oracle $O_{\mathsf{Recover_S}}(\mathsf{mc}_{i^*})$

/ Same as Game_3 up to line 8

9 : **else** / Secure channel established with client *or adversary*

10 : **if** $(\mathsf{peer}_{\mathsf{S}_{i^*}}[n_\mathsf{S}] = \bot)$ **then** / Message from adversary

11 : **if** $(\mathtt{extract} \in \mathsf{st}_{\mathsf{S}_{i^*}})$ **then**

12 : $\overline{\mathsf{pwd}}' := \mathsf{tORF.Extract}^*_x(\mathsf{st}_{\mathsf{S}_{i^*}}, \mathsf{mc}_{i^*})$

13 : **if** $(\overline{\mathsf{pwd}}' = \overline{\mathsf{pwd}})$ **then** / Simulate using y^*_{ctr}

14 : $(\mathsf{st}_{\mathsf{S}_{i^*}}, \mathsf{ms}_{i^*}) \xleftarrow{\$} \mathsf{tORF.SimEval}^*_{\mathsf{S}}(\mathsf{st}_{\mathsf{S}_{i^*}}, \mathsf{mc}_{i^*}, \overline{\mathsf{pwd}}', y^*_{\mathsf{ctr}})$

15 : **else** / Simulate without y^*_{ctr}

16 : $(\mathsf{st}_{\mathsf{S}_{i^*}}, \mathsf{ms}_{i^*}) \xleftarrow{\$} \mathsf{tORF.SimEval}^*_{\mathsf{S}}(\mathsf{st}_{\mathsf{S}_{i^*}}, \mathsf{mc}_{i^*}, \overline{\mathsf{pwd}}')$

17 : **elseif** ($\mathsf{st}_{\mathsf{S}_{i^*}}$ includes simulated tORF state)

18 : $(\mathsf{st}_{\mathsf{S}_{i^*}}, \mathsf{ms}_{i^*}) \xleftarrow{\$} \mathsf{tORF.SimEval}^*_{\mathsf{S}}(\mathsf{st}_{\mathsf{S}_{i^*}}, \mathsf{mc}_{i^*})$

19 : **else** / Process 'update' message or adversarial state

20 : $(\mathsf{st}_{\mathsf{S}_{i^*}}, \mathsf{ms}_{i^*}) \xleftarrow{\$} \mathsf{Recover_S}(\mathsf{sk}_{\mathsf{S}_{i^*}}, \mathsf{st}_{\mathsf{S}_{i^*}}, \mathsf{mc}_{i^*})$

21 : **if** (S_{i^*} accepted 'update' message) / Successful online attack

22 : $\mathsf{flag}_{\mathsf{online}} \leftarrow \top$

23 : **if** $(\mathsf{ms}_{i^*} \neq \bot)$ **then** / S_{i^*} does not abort protocol

24 : $\mathsf{flag}_{\mathsf{unf}}[n_\mathsf{S}] \leftarrow \top$ / A potential forgery if client later becomes S_{i^*}'s peer

/ Message from client's $\mathsf{peer}_{\mathsf{S}_{i^*}}[n_\mathsf{S}]$-th secure channel session

25 : **if** $(\nexists!(\overline{\mathsf{mc}}_{i^*}, r')$ / Return $\bot$ if client did not send mc_{i^*}

: $\mathsf{SentMsg}[\mathsf{C}, \mathsf{peer}_{\mathsf{S}_{i^*}}[n_\mathsf{S}], n_\mathsf{S}, r'] = (\mathsf{mc}_{i^*}, \overline{\mathsf{mc}}_{i^*}))$ **then**

26 : $\mathsf{ms}_{i^*} \leftarrow \bot$ / S_{i^*} aborts protocol prematurely

/ Only simulate while S_{i^*} has simulated tORF states

27 : **if** $(r' \in [k_{\mathsf{Eval}}]) \wedge (\mathsf{S}_{i^*}$ has simulated tORF states) **then**

/ Run $\mathsf{tORF_S}$ implicitly run under the secure channel

28 : **if** (This is r'-th round of tORF evaluation protocol) **then**

29 : $\mathsf{ms}_{i^*} \xleftarrow{\$} \mathsf{SC.Send}(\mathsf{key}_{\mathsf{S}_{i^*}}[n_\mathsf{S}], 0)$

30 : **if** $(r' = k_{\mathsf{Eval}})$ **then**

31 : $\overline{\mathsf{ms}}_{i^*} \leftarrow (k_{\mathsf{Eval}}||\bot||\mathsf{fid}||\mathsf{ctr}||\mathsf{ctxt})$

32 : **else**

33 : $\overline{\mathsf{ms}}_{i^*} \leftarrow r'$

34 : $\mathsf{SentMsg}[\mathsf{S}_{i^*}, \mathsf{peer}_{\mathsf{S}_{i^*}}[n_\mathsf{S}], n_\mathsf{S}, r'] \leftarrow (\mathsf{ms}_{i^*}, \overline{\mathsf{ms}}_{i^*})$

35 : **else**

36 : $\mathsf{ms}_{i^*} \leftarrow \bot$ / S_{i^*} aborts protocol prematurely

37 : **elseif** $(r' = \text{'update'}) \wedge (\mathsf{S}_{i^*}$ has simulated tORF states)

38 : **parse** $(\text{'update'}\|\mathsf{L}^*_{i^*}\|\mathsf{tag}'_{i^*}) \leftarrow \overline{\mathsf{mc}}_{i^*}$

39 : **if** $(\mathsf{tag}_{i^*} \neq \mathsf{tag}'_{i^*})$ **then** / Authentication succeeded

40 : **for** $j \in [c - \mathsf{ctr}_{i^*}]$ **do**

41 : $\mathsf{L}_{i^*}[\mathsf{ctr}_{i^*} + j] \leftarrow \mathsf{L}^*_{i^*}[j]$

42 : **delete** $(\widehat{\mathsf{st}}_{\mathsf{S}_{i^*}}, \mathsf{fid}_{i^*}, \mathsf{tag}_{i^*}, \mathsf{ctxt}_{i^*})$

43 : $\mathsf{st}_{\mathsf{S}_{i^*}} \leftarrow (\mathsf{L}_{i^*}, c)$

44 : $\mathsf{ms}_{i^*} \leftarrow \bot$ / S_{i^*} finishes protocol

45 : **else** / S_{i^*} has only adversarial tORF states

46 : Same as line 18 to line 32 of previous Game_3

47 : **return** ms_{i^*}

Figure 11: The description of $O_{\mathsf{Recover_C}}$ and $O_{\mathsf{Recover_S}}$ in Game_4. The differences are highlighted in blue. $\mathsf{tORF.SimEval}^*_{\mathsf{C}}$ **and** $\mathsf{tORF.SimEval}^*_{\mathsf{S}}$ **check the validity of inputs, internally run** $\mathsf{tORF.SimEval_C}$ **and** $\mathsf{tORF.SimEval_S}$**, and appropriately manage** $\mathsf{st_C}$ **and** $\mathsf{st}_{\mathsf{S}_{i^*}}$**, respectively.** $\mathsf{tORF.SimEval}^*_{\mathsf{S}}$ **additionally checks if** $\mathsf{ctr}_i \neq 0$ **before the one-time** tORF **session.** $\mathsf{tORF.CorrectVal}^*(\mathsf{st_C})$ **returns** $b = 0$ **when** $\mathsf{st_C}$ **is** $\bot$**.** $\mathsf{tORF.Extract}^*_x$ **returns** $\bot$ **as** $\overline{\mathsf{pwd}}'$ **when received** mc_{i^*} **is invalid. In** Batch-Init**, the challenger generates** tORF **states by** tORF.SimInit **and uniformly picks** $y^*_j \xleftarrow{\$} \mathcal{Y}$ **if** $\mathsf{flag}_{\mathsf{online}} = \bot$**, otherwise, it generates them honestly.**

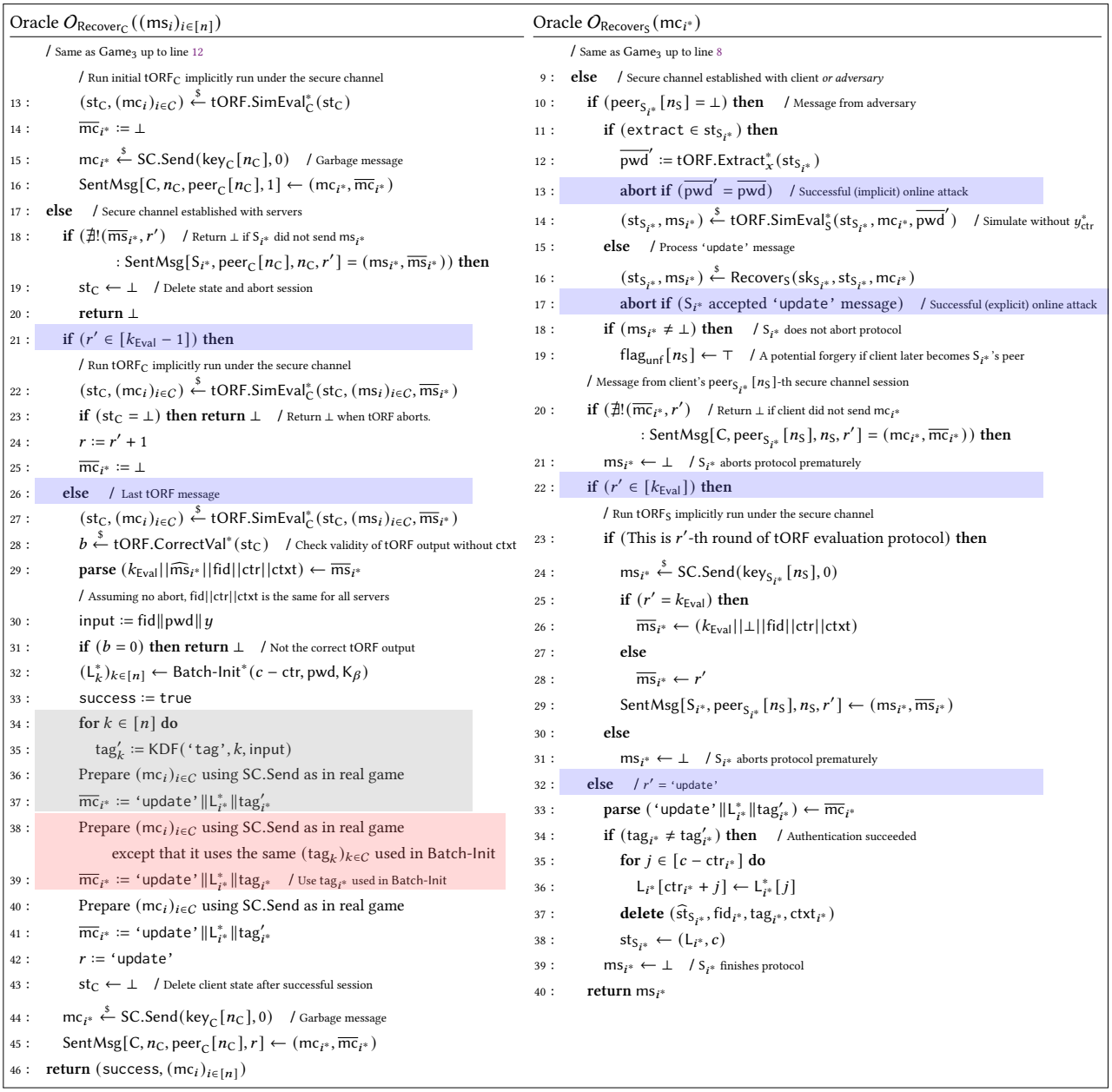

Figure 12: The descriptions of O_{Recover_C} and O_{Recover_S} in Game_5 and Game_6. Lines replaced in Game_5 are highlighted in blue. Lines highlighted in gray and in red appear only in Game_5 and Game_6, respectively. When the game aborts the challenger terminates the game and returns a random bit. Game_6 uses the random oracle H_2 defined in the proof of Theorem 5 before full corruption instead of H_1.