Universally Composable Password-Hardened Encryption

Behzad Abdolmaleki[1], Ruben Baecker[2], Paul Gerhart[3], Mike Graf[4], Mojtaba Khalili[⋆5], Daniel Rausch[4], and Dominique Schröder[2,3]

[1] University of Sheffield, Sheffield, England
[2] Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany
[3] TU Wien, Vienna, Austria
[4] University of Stuttgart, Stuttgart, Germany
[5] Isfahan University of Technology, Isfahan, Iran

Abstract. Password-Hardened Encryption (PHE) protects against offline brute-force attacks by involving an external ratelimiter that enforces rate-limited decryption without learning passwords or keys. Threshold Password-Hardened Encryption (TPHE), introduced by Brost et al. (CCS'20), distributes this trust among multiple ratelimiters. Despite its promise, the security foundations of TPHE remain unclear. We make three contributions:

(1) We uncover a flaw in the proof of Brost et al.'s TPHE scheme, which invalidates its claimed security and leaves the guarantees of existing constructions uncertain;
(2) We provide the first universal composability (UC) formalization of PHE and TPHE, unifying previous fragmented models and supporting key rotation, an essential feature for long-term security and related primitives such as updatable encryption;
(3) We present the first provably secure TPHE scheme, which is both round-optimal and UC-secure, thus composable in real-world settings; and we implement and evaluate our protocol, demonstrating practical efficiency that outperforms prior work in realistic WAN scenarios.

1 Introduction

In recent years, the amount of sensitive data collected and processed by online services has grown exponentially, exposing user information to an increasing number of entities. At the same time, data breaches have become more frequent and severe. In 2023, a collection of billions of records were compromised in high-profile incidents involving companies such as T-Mobile, Facebook, and Marriott International.[6] Attackers often exploit weak authentication mechanisms and centralized encryption key management to gain unauthorized access to user data. Once a database is breached, attackers can perform offline brute-force attacks on password hashes or, worse, obtain master keys that provide access to all encrypted records.

Traditional encryption methods are designed to protect stored data from external adversaries, but they often fall short in scenarios involving insider threats or advanced persistent attacks. In particular, once an attacker gains access to a database, he or she can operate independently of authentication rate limits and other online defenses. This fundamental weakness requires new cryptographic solutions that mitigate offline attacks and enforce rate-limited access at the encryption layer.

Password-Hardened Encryption (PHE). To address these challenges, Lai et al. [43] (USENIX'18) introduced password-hardened encryption (PHE), a cryptographic scheme that enhances password-based key derivation by incorporating an external entity called a *ratelimiter* or *crypto service*. Unlike traditional password-based encryption, where the server alone derives the data key, PHE distributes trust by involving the ratelimiter in key derivation. The server computes the encryption key using a combination of the user's password, its own secret key, and the ratelimiter's key. Importantly, the ratelimiter never learns the password or the data key, and its primary function is to enforce rate-limited decryption attempts.

⋆ This work was done while at Friedrich-Alexander-Universität Erlangen-Nürnberg.
[6] https://www.mcafee.com/blogs/internet-security/26-billion-records-released-the-mother-of-all-breaches/

The security guarantees of PHE ensure that neither a malicious database attacker nor the ratelimiter alone can recover user passwords or derived keys. Furthermore, PHE supports key rotation, allowing the server and ratelimiter to update their keys without requiring user intervention. Key rotation is an essential feature mandated by several security standards–including NIST SP 800-57, ISO/IEC-27002:2013, and PCI DSS [47]–and best practices recommended by major industry players like Google and Amazon. Due to its security benefits, PHE has seen real-world adoption, with commercial implementations such as VirgilSecurity's PureKit[7].

Threshold Password-Hardened Encryption (TPHE). While PHE significantly enhances security, it introduces a new availability risk: the ratelimiter becomes a single point of failure. If the ratelimiter is unavailable due to outages or targeted denial-of-service attacks, users may be locked out of their data. Moreover, if an attacker compromises the ratelimiter, they could potentially assist in key recovery, undermining the security guarantees of PHE.

To address these concerns, Brost et al. [10] (CCS'20) proposed *threshold password-hardened encryption* (TPHE), an extension of PHE that distributes the ratelimiter functionality among multiple independent entities. In a (t, m)-TPHE scheme, the ratelimiter role is shared among m independent servers, with a threshold value t specifying the minimum number of ratelimiters required for successful decryption. This design improves the security by ensuring that as long as an adversary does not compromise both the main server and at least t ratelimiters, the security guarantees of PHE remain intact. Furthermore, availability is improved, as the system can tolerate up to $m - t$ ratelimiters becoming unavailable while the remaining ones continue to maintain the service.

Our work uncovers a fundamental gap in the security proof of the threshold password-hardened encryption (TPHE) scheme of Brost et al., which is the so far only TPHE scheme with a formal security analysis. This casts doubt on whether a provably secure construction is currently known. Even if the formal security of their TPHE scheme could be reestablished, it would still only achieve standalone security, making it unsuitable for use in larger cryptographic protocols. In addition, the existing scheme suffers from inefficiency, requiring multiple rounds of communication for encryption and decryption. To address these issues, we propose the first round-optimal TPHE scheme with provable universal composability (UC) guarantees, ensuring both stronger security properties and improved efficiency. In addition, we introduce a unified security definition within the UC framework, resolving inconsistencies and shortcomings in previous game-based models and providing a robust foundation for future research.

1.1 Our Contributions

Our main contributions are the following:

- We identify a gap in the security proof of Brost et al. TPHE scheme, raising doubts about whether a secure threshold PHE construction is currently known.
- PHE's game-based security definitions date back to the early work by Everspaugh *et al.* on password hardening [24]. Each subsequent paper has introduced a new game-based security model [48, 42, 43, 10], leading to a fragmented and incomparable landscape of security definitions and constructions. Such game-based definitions establish only standalone security that is insufficient for use within larger cryptographic protocols. This, however, is the intended use case of PHE and TPHE. These definitions further require passwords to be distributed in specific ways, independently of each other, and to remain fully confidential to guarantee security. This is generally not the case in practice where users tend to re-use (parts of) their passwords and include public information such as birth dates.
  We solve these issues by proposing the first security definition within a universal composability (UC) model to establish a unified notion for both PHE and TPHE with stronger and composable security properties required in practice. Specifically, we propose an ideal functionality $\mathcal{F}_{\text{PHE}}$ for (t, m)-threshold password-hardened (authenticated) encryption.

[7] https://developer.virgilsecurity.com/docs/purekit/fundamentals/password-hardened-encryption/

– We introduce the first provably UC secure round-optimal (authenticated) (t, m)-threshold password-hardened encryption scheme, addressing multiple open questions:
  - We establish the feasibility of provably secure TPHE schemes.
  - Our scheme is the first TPHE (and also PHE) scheme with provable universal composability guarantees.
  - Our scheme improves upon the recent TPHE construction by Brost et al. [10], which requires three rounds for encryption and five for decryption, by reducing both encryption and decryption to a single round. Our benchmarks highlight the practical impact of this improvement, showing that our scheme outperforms that of Brost et al. by more than 200% in realistic WAN settings. More broadly, this result underscores the practical feasibility of round-optimal schemes.

  Our construction can be seen as a reminiscence of Pythia [24], the pioneering password hardening scheme. We adapt the scheme for thresholding and extend it to support both encryption and integrity functionalities. The scheme's security is proved under the Gap One-More Bilinear Computational Diffie-Hellman (Gap-OM-BCDH) assumption in the random oracle model.

1.2 Related Work

PH, PHE, and TPHE. Everspaugh *et al.* [24] pioneered the concept of password-hardening (PH) following its initial introduction by Facebook [46]. Their work identified key rotation as a fundamental property for practical deployment, highlighting it as both an enabler and a core challenge in designing PH and password-hardened encryption (PHE) schemes. Subsequent refinements to PH were made by Schneider *et al.* [48] and Lai *et al.* [42], further solidifying its security guarantees and applicability. Later, Lai *et al.* [43] extended the paradigm to password-hardened encryption (PHE), which not only secures password verification but also enables encryption of associated data under a per-user key. This ensures that the data remains inaccessible without the user's password while maintaining strong security guarantees similar to those of PH. Lai *et al.* further extended PHE to the threshold setting [43].

PPSS. Password-protected secret sharing (PPSS), introduced by Bagherzandi *et al.* [5], is a foundational concept in password-based cryptography. Their work laid the groundwork for securing secrets using passwords, inspiring a sequence of influential advancements [12, 14, 32, 53, 11, 34, 2, 35]. These works refined the cryptographic techniques underlying PPSS, significantly improving its security guarantees and practical applicability. Although PPSS and TPHE share similar core functionality, they operate in orthogonal settings. TPHE is designed for stateless users, allowing them to securely store an encryption key using only a password while offloading all cryptographic operations to a server and a set of ratelimiters. This setting can be thought of as a service provider that employs a set of external ratelimiters without the user noticing. Another requirement is that the ratelimiters can only have a minimal state (i.e., independent of the number of users). In contrast, PPSS merges the roles of the server and the ratelimiters, requiring the client to interact directly with the ratelimiters and perform cryptographic computations. Another key difference between TPHE and PPSS is key rotation. While TPHE is designed with key rotation as an integral feature, PPSS does not support it. Extending PPSS to include key rotation would require modifications to both its secret-sharing structure and security model.

PAKE. Password-Authenticated Key-Exchange (PAKE) and its various extensions ((s)aPAKE, tPAKE, t-saPAKE) [1, 21, 28, 29, 31, 33, 36, 50, 52] operates in a different setting compared to PHE. It is designed to establish a fresh key between two stateless parties where the key exchange is only successful if both parties enter the same password (or, in some cases, an augmented version of the password). In contrast, PHE focuses on protecting a long-term secret key while allowing controlled password-based access. Furthermore, PHE preserves the client-server interface and does not require clients to perform cryptographic computations, making it accessible to all clients.

Distributed and Threshold Single Sign-On. Distributed and threshold-based SSO schemes [3, 6] operate in a different setting than TPHE, as they focus on authentication rather than key management. Jiang *et al.* [38] propose a post-quantum threshold SSO with key rotation but no per-user rate-limiting, relying on costly

cryptographic components. Frederiksen *et al.* [25] extend SSO for privacy-preserving attribute attestation using MPC.

Additional. DPaSE [20] allows users to encrypt data with many object-specific keys derived from a single password, enhancing security against offline and online attacks. While it shares the fundamental concept of deriving symmetric keys from passwords, it is limited to the distributed setting ($t = n$) and does not support key rotation. Password-Authenticated Public-Key Encryption (PAPKE)[9] enables the generation of a password-dependent private key that can decrypt ciphertexts only if the same password was used during encryption. Password-based server-aided signatures [16, 51] allow a user and a server to jointly sign a message under the user's password. The primitive's functionality is distinct to the one of TPHE and does not support key rotation. Password-based key derivation and encryption [39] allow direct encryption of messages using keys derived from passwords but remain susceptible to brute-force attacks. A related concept for key rotation that does not rely on passwords is updatable encryption (UE) [8, 23, 44]. UE focuses on enabling ciphertext updates under evolving keys, but it operates in a different security model. While UE assumes a single party managing the encryption keys, (t, n)-TPHE involves multiple parties–the server and the ratelimiters–each playing a distinct role in access control and security enforcement.

2 Technical Outline

2.1 Gap in the Proof of Brost et al [10]

We begin by describing a gap in the proof of TPHE [10] and refer the reader to the proof of Theorem 3.1 of the full version of [10], in particular the security proof of *hiding*. Intuitively, hiding guarantees that the TPHE encryptions of the two messages M_0^* and M_1^* under a random password are indistinguishable, except when the correct password is known or successfully guessed. We emphasize that our findings reveal a gap in the security proof, not necessarily a flaw in the construction itself. While we have not identified an attack that exploits this gap, the absence of a valid security proof means that the security of the TPHE scheme proposed by Brost *et al.* remains uncertain.

Protocol description For the sake of this overview, we abstract away details related to the encryption of the message and focus solely on the parts relevant to authentication, which is central to the security gap. We further simplify the setting by assuming a single ratelimiter, while emphasizing that the identified gap persists even in configurations with multiple ratelimiters.

The ratelimiter holds a secret key s that it uses to compute a PRF $\mathsf{H}(\cdot)^s$. Furthermore, the server and the ratelimiter hold a shared secret key $k = k_{\mathcal{S}} + k_{\mathcal{R}}$ and the corresponding public key $K = g^k$ used for ElGamal Encryption. During user enrollment, the server and the ratelimiter jointly compute a record (C, n) that stores the user's password as

$$C = \mathsf{H}(pw, n) \cdot \mathsf{H}(n)^s,$$

which can be thought of as the salted password hash $\mathsf{H}(pw, n)$ blinded by the ratelimiter's PRF evaluated on the nonce $\mathsf{H}(n)^s$.

During authentication, the server and the ratelimiter jointly check the correctness of the password in an interactive protocol that can be summarized in three steps:

1. **Encryption of $\frac{\mathsf{H}(pw',n)}{\mathsf{H}(pw,n)}$:** The server computes $C^{-1} \cdot \mathsf{H}(pw', n) = \frac{\mathsf{H}(pw',n)}{\mathsf{H}(pw,n)} \cdot \mathsf{H}(n)^{-s}$ and encrypts the result under the joint public key K, resulting in

$$(U_{\mathcal{S}}, V_{\mathcal{S}}) = (g^{r_{\mathcal{S}}}, K^{r_{\mathcal{S}}} \cdot \frac{\mathsf{H}(pw', n)}{\mathsf{H}(pw, n)} \cdot \mathsf{H}(n)^{-s}).$$

The ratelimiter recomputes the PRF on the nonce $\mathsf{H}(n)^s$ and encrypts the result under K, resulting in

$$(U_{\mathcal{R}}, V_{\mathcal{R}}) = (g^{r_{\mathcal{R}}}, K^{r_{\mathcal{R}}} \cdot \mathsf{H}(n)^s).$$

Once combined, these result in

$$(U, V) = (U_{\mathcal{S}} \cdot U_{\mathcal{R}}, V_{\mathcal{S}} \cdot V_{\mathcal{R}}) = (g^{r_{\mathcal{S}}+r_{\mathcal{R}}}, K^{r_{\mathcal{S}}+r_{\mathcal{R}}} \cdot \frac{\mathsf{H}(pw', n)}{\mathsf{H}(pw, n)}),$$

which is an encryption of the identity element if the entered password matches the enrolled password. For simplicity, we denote the quotient of the two password hashes as Δ and the combined randomness as $r = r_{\mathcal{S}} + r_{\mathcal{R}}$.

2. **Joint Re-randomization:** To ensure that nothing is leaked about the passwords–except if they match– the server and the ratelimiter jointly re-randomize the ciphertext. This re-randomization raises Δ to a random scalar $\tilde{r} = \tilde{r_{\mathcal{S}}} + \tilde{r_{\mathcal{R}}}$, resulting in the encryption of a uniformly random group element. If the passwords match, however, the re-randomized ciphertext still corresponds to an encryption of the identity element. The resulting ciphertext looks as follows:

$$(\tilde{U}, \tilde{V}) = (U^{\tilde{r}}, V^{\tilde{r}}) = (g^{r \cdot \tilde{r}}, K^{r \cdot \tilde{r}} \cdot \Delta^{\tilde{r}}).$$

3. **Joint Decryption:** Finally, the server and the ratelimiter each partially decrypt the ciphertext by raising $\tilde{U}$ to their share of the decryption key k, resulting in $T_{\mathcal{S}} = \tilde{U}^{k_{\mathcal{S}}}$ and $T_{\mathcal{R}} = \tilde{U}^{k_{\mathcal{R}}}$. Now, the server can decrypt the ciphertext to obtain

$$\begin{aligned}
\Delta^{\tilde{r}} \leftarrow \tilde{V}/(T_{\mathcal{S}} \cdot T_{\mathcal{R}}) &= K^{r \cdot \tilde{r}} \cdot \Delta^{\tilde{r}}/(\tilde{U}^{k_{\mathcal{S}}} \cdot \tilde{U}^{k_{\mathcal{R}}}) \\
&= K^{r \cdot \tilde{r}} \cdot \Delta^{\tilde{r}}/\tilde{U}^{k} \\
&= K^{r \cdot \tilde{r}} \cdot \Delta^{\tilde{r}}/(g^{r \cdot \tilde{r}})^{k} \\
&= K^{r \cdot \tilde{r}} \cdot \Delta^{\tilde{r}}/K^{r \cdot \tilde{r}} = \Delta^{\tilde{r}}.
\end{aligned}$$

The server now checks whether the entered password matches the registered one using a simple equality check $\Delta^{\tilde{r}} \stackrel{?}{=} 1$.

At every step, both parties send Schnorr NIZKs along their computations to prove that they adhered to the protocol. For the security proof, it is important that the NIZKs are both simulatable and witness-extractable.

Proof idea. The proof follows the standard procedure of game hopping [7]. In a series of games, they gradually remove the information encrypted in the ciphertexts to show that no information about the passwords is leaked during authentication. They prove that an adversary cannot notice the transitions by reducing the distinguishability of every game from its predecessor to a hardness assumption. In the end, the adversary can only win the experiment with non-negligible probability if it correctly guesses the password.

The gap occurs in the transition from Hybrid $\mathsf{Hyb}_{b,3,q-1}$ to Hybrid $\mathsf{Hyb}'_{b,3,q}$. For the sake of this overview, we assume that the server is honest and the ratelimiter is corrupt, which corresponds to case $0 \notin I$ in their proof. Nevertheless, the gap occurs in the same way when the server is corrupt (case $0 \in I$). The only difference between the two Hybrids is that in Hybrid $\mathsf{Hyb}'_{b,3,q}$, they remove the information encrypted in $(U_{\mathcal{S}}, V_{\mathcal{S}})$ of step 1 described above. In the reduction where they prove that the adversary cannot notice the difference between the ciphertexts, they follow the standard approach for reducing the security of ElGamal to DDH. They replace the server's public key and the ciphertext components with values taken from the DDH tuple $(g^{\alpha}, g^{\beta}, g^{\gamma})$, resulting in

$$K_{\mathcal{S}} = g^{\alpha} \quad U_{\mathcal{S}} = g^{\beta} \quad V_{\mathcal{S}} = g^{\gamma} \cdot (g^{\beta})^{k_{\mathcal{R}}} \cdot C^{-1} \cdot \mathsf{H}(pw', n).$$

The rationale is that if the tuple is a valid DDH tuple (i.e., $\gamma = \alpha \cdot \beta$), this is equivalent to $\mathsf{Hyb}_{b,3,q-1}$. Otherwise, if $\gamma \neq \alpha \cdot \beta$, this is equivalent to $\mathsf{Hyb}'_{b,3,q}$.

It is important to note that the server cannot compute the proof of well-formedness for step 1 as described in the protocol, because it does not know the witness β that is used to compute $U_{\mathcal{S}} = g^{\beta}$. Instead, the reduction has to simulate the proof without knowing β by using the simulator that is guaranteed to exist by the simulatability property of the NIZKs.

Gap in the proof. The problem occurs in the simulation of step 3 of the authentication protocol. The server has to compute a partial decryption $T_{\mathcal{S}} = \tilde{U}^{k_{\mathcal{S}}}$ of $(\tilde{U}, \tilde{V})$ without knowing the decryption key $k_{\mathcal{S}} = \alpha$. To do so, the authors exploit the witness-extractability of the proofs of well-formedness that both the server and the ratelimiter provide with every message that they send. They extract the exponent $\tilde{u}$ from the relation $\tilde{U} = g^{\tilde{u}}$ to compute $T_{\mathcal{S}} = \tilde{U}^{k_{\mathcal{S}}} = \tilde{U}^{\alpha}$ as $T_{\mathcal{S}} = (g^{\alpha})^{\tilde{u}}$. The relation $\tilde{U} = g^{\tilde{u}}$ is actually composed of four relations:

1. $U_{\mathcal{S}} = g^{r_{\mathcal{S}}}$ computed by the server in step 1,
2. $U_{\mathcal{R}} = g^{r_{\mathcal{R}}}$ computed by the ratelimiter in step 1,
3. $\tilde{U}_{\mathcal{S}} = U^{\tilde{r}_{\mathcal{S}}}$ computed by the server in step 2, and
4. $\tilde{U}_{\mathcal{R}} = U^{\tilde{r}_{\mathcal{R}}}$ computed by the ratelimiter in step 2.

While this strategy looks fairly standard when using the extractability of NIZKs, here lies a subtle issue: As mentioned before, the server cannot compute the NIZK of relation 1 as it does not know the witness $r_{\mathcal{S}} = \beta$ and instead uses the simulator to compute the proof. Therefore, the witness extraction from this NIZK cannot be done since extracting from a simulated proof is not possible. If it were possible, this would imply that breaking DDH is easy. The server could extract the witness $r_{\mathcal{S}} = \beta$ and check whether $(g^{\alpha})^{\beta} = g^{\gamma}$, effectively solving DDH. As a result, the indistinguishability between the Hybrids $\mathsf{Hyb}_{b,3,q-1}$ and $\mathsf{Hyb}_{b,3,q}$ is not proven to be non-negligible, and hence, the proof of security fails.

2.2 The Challenge of Formalizing Security

Formalizing the security of sophisticated cryptographic primitives and protocols is challenging in general; password-hardening [24] and password-hardened encryption [43] are no exception here, but rather show that game-based security definitions quickly reach their limits in complicated real-world settings. In the following, we outline the (long) history of the search for adequate security formalizations. This demonstrates not just that defining security is non-trivial but also that a UC definition is sorely needed.

Password-hardening (PH). Everspaugh et al. [24] proposed the first security formalization in terms of cryptographic games. The security notions of Everspaugh et al. followed "one-more" game properties, which were revised by Schneider et al. [48]. Shortly afterward, Lai et al. [42] realized that the scheme of Schneider et al. [48] is vulnerable to offline dictionary attacks. This attack did not result from a flaw in the proof but from a security model that was too weak. Lai et al. showed that a single verification query is sufficient to brute-force the password afterward; queries to the verification oracle were denied in the simple indistinguishable definition due to Schneider et al.

Password-hardened Encryption (PHE). Lai *et al.* [42] generalized PH to password-hardened *encryption*, a cryptographic protocol that involves password-based key derivation and supports the important property of key rotation, going beyond mere password verification. In their work, they enhanced the security properties. At CCS'20, Brost *et al.* [10] generalized PHE to the threshold setting and proposed a novel security notion that unified some previously defined properties. More recently, Chunfu et al. [37] extended the security properties. The current situation is unsatisfactory in many different aspects:

Incomparable Models and Constructions. Each publication in PH, PHE, and TPHE proposed a novel game-based security notion. Not having a unified security model is unsatisfactory, as the models and constructions are partially incomparable. One reason might be the difficulty of covering the security properties of such complex functionality in cryptographic games. The complexity stems from various factors, such as a corruption model, low-entropy security notions, and key rotation.

Standalone Security. Another issue is the usage of game-based notions for a *composable cryptographic primitive* in the first place. The purpose of PHE and its threshold variant is to compute cryptographic keys used in different contexts. Game-based security notions guarantee stand-alone security and provide no security guarantees in composability with other cryptographic primitives.

Assumptions on Passwords. Formalizing password-based security, also beyond the area of password-hardening, is notoriously difficult for game-based security definitions: defining a complete game also entails specifying how passwords are chosen and managed which easily introduces unintended assumptions. For

example, the TPHE notion of [10] specifies a game where plaintexts are proven to be secret if they are encrypted under a uniformly random[8] one-time password that is generated fresh and independent of other passwords, remains entirely secret, and is never used for decryption of a different ciphertext. These assumptions are typically not met by passwords in reality where users do not follow an a-priori fixed password distribution, tend to re-use (parts of) passwords, might include publicly known or easy to guess information such as birth dates, and might accidentically mix up passwords of different ciphertexts during decryption.

In contrast, UC security definitions can formalize statements of the form "as long as an attacker does not obtain/guess the entire password, security holds true" without having to fix how passwords are chosen, managed, used, or how likely an attacker is to guess a specific password. Because such security results apply to all situations that can occur in practice, UC definitions are considered the gold standard for password-based security such as password-authenticated key exchanges [19].

2.3 The Challenge of Round-Optimal TPHE

All prior work on PH and PHE suggested *round optimal* (two moves) constructions [48, 42]. Being round optimal is crucial as PHE is used in practice to secure business data and processes; long latencies may prohibitively slow down these processes. The goal of the threshold variant of Brost *et al.* [10] is to increase security and availability as there is no single point of failure. However, the resulting encryption protocol consists of three rounds, while the decryption protocol consists of six rounds. The authors claim that the throughput of the ratelimiter scales linearly in the number of cores. Thus, the protocol is unsuitable for practical use for at least two reasons. First, the large number of rounds introduces a significant overhead and slows down any practical operation. Second, the protocol does not scale well with the number of ratelimiters, because the message size and the computation per ratelimiter grow linearly with the number of ratelimiters. However, if only a few ratelimiters can be used, the question arises of why the protocol should be used.

3 An Ideal Functionality for TPHE

In this section, we propose the first ideal TPHE functionality $\mathcal{F}_{\text{PHE}}$ for Universal Composability (UC) [18, 17, 45, 41, 13, 30].

3.1 Expected TPHE security properties

Let us start by summarizing which properties a secure TPHE scheme is supposed to provide and which are thus formalized by $\mathcal{F}_{\text{PHE}}$. A $(\mathtt{t}, \mathtt{n})$-TPHE scheme is run among one server $\mathcal{S}$ and $\mathtt{n}$ rate limiters $\mathcal{R}_i$ such that $\mathtt{t}$ rate limiters are necessary to perform a decryption. The server can perform three types of actions: (i) $\mathtt{Store}(id, pw, m)$ encrypts a message m under some password pw and stores the resulting ciphertext in a local database at some position id, (ii) $\mathtt{Retrieve}(id, pw')$ decrypts a ciphertext stored at a database position id using some password pw', and (iii) *key rotation* to (try to) restore security for those parties, i.e., both server and rate limiters, that were under control of an adversary and which therefore must be considered corrupted/malicious. When successful, we say that the affected parties are *decorrupted* and are treated as honest again. All three actions require interaction with rate limiters. Rate limiters are expected to always participate in encryption and key rotation. They are free to decide whether and how often they are willing to help with a decryption operation for any given database position id, thus implementing a rate limiting mechanism. This mechanism can be very fine grained and different for each id, which itself might encode further information such as a username that this ciphertext "belongs" to.

A secure TPHE scheme is expected to provide at least the following five fundamental properties:

1. *Secrecy of encryption:* If $\mathtt{Store}(id, pw, m)$ is performed while the server is honest, then both the plaintext m and the password pw remain secret, even if all ratelimiters are malicious, the resulting ciphertext leaks,

[8] The authors of [10] note that it is straightforward to generalize their notion for any arbitrary but fixed password distribution.

and the server after finishing this operation gets corrupted by the adversary. The plaintext m might be revealed only if the adversary obtains the correct password pw in some other way, e.g., because pw is easy to guess (see below).

2. *Secrecy of decryption:* If $\texttt{Retrieve}(id, pw')$ is performed while the server is honest, then pw' remains secret, even if all ratelimiters are malicious.
3. *Correctness:* Given any database position id, if m is the most recent plaintext stored at this position while the server was honest and the server has since then not yet been corrupted, then retrieving the ciphertext at position id will either yield m or an error $\perp$, even if all ratelimiters are malicious during both encryption and decryption.[9]
4. *Ratelimited decryption:* Decrypting a ciphertext encrypted for id requires at least $\texttt{t}$ ratelimiters that are willing to help with the decryption. This implies that an attacker controlling the server cannot circumvent rate-limiting by running the decryption protocol for id' with a ciphertext that was originally encrypted for id. Note that encryption cannot be abused to perform decryption, because a fresh nonce is included in every encryption.
5. *Online password guessing:* An attacker might try to guess the password used to encrypt a leaked ciphertext c. Verifying a guess and, if correct, obtaining the plaintext m should be possible only via rate-limited online decryption.This requirement is lifted if the attacker at some point controls both the server and at least $\texttt{t}$ ratelimiters since then he has all secret keys to simulate decryption locally and offline.
   In practice, the choice of the threshold parameter $\texttt{t}$ relative to the total number of ratelimiters $\texttt{n}$ has a direct impact on how effectively an attacker can distribute online guessing attempts. If $\texttt{t}$ is chosen significantly smaller than $\texttt{n}$, an adversary can simply exclude a ratelimiter for which the rate limit has already been reached and instead involve another, thereby increasing its effective rate of attack. This consideration must therefore be taken into account when configuring practical rate limits. Nevertheless, since typical deployments are expected to use ratios $\texttt{n}/\texttt{t}$ close to a small constant (often below two), the potential impact of such amplification is not expected to be significant.

All of these properties should hold true for arbitrary rate limiting mechanisms and arbitrary password distributions, including cases where (parts of) some passwords are easy to guess or are re-used to encrypt different plaintexts.

3.2 Definition of $\mathcal{F}_{\text{PHE}}$

The ideal functionality $\mathcal{F}_{\text{PHE}}$ is defined in Figure 1 and explained in what follows.

Internal state. The state of $\mathcal{F}_{\text{PHE}}$ mainly consists of two variables. The variable $\mathsf{storageHistory}$ captures the database of ciphertexts, including all of its history. That is, $\mathsf{storageHistory}(id)$ stores the chronologically ordered sequence of plaintext-password pairs (pw, m) that have been stored at position id. We note that a real TPHE protocols that realizes $\mathcal{F}_{\text{PHE}}$ would of course only keep the most recent ciphertext for each id. However, to define the behavior in cases where previous ciphertexts were leaked the ideal functionality $\mathcal{F}_{\text{PHE}}$ has to keep the full history. The second variable $\mathsf{retrieveRate}(R_i, id)$ tracks how often rate limiter $\mathcal{R}_i$ is still willing to help with a decryption of a ciphertext stored at position id.

Main operations available to higher-level protocols. $\mathcal{F}_{\text{PHE}}$ accepts three types of inputs from higher-level protocols/the environment: They can instruct the server $\mathcal{S}$ to run the $\texttt{Store}(id, pw, m)$ or the $\texttt{Retrieve}(id, pw')$ operations for arbitrary inputs of their choosing. For example, $\texttt{Store}(id, pw, m)$ might be triggered because a client in a higher-level wants to store a message m under a password pw (and id choosen to be an empty database position assigned to this client). Since the password pw is determined externally, $\mathcal{F}_{\text{PHE}}$ indeed formalizes security for arbitrary password distributions where passwords used for different ciphertexts might also depend on each other. Higher-level protocols can further send the command $\texttt{HelpRetrieve}(id)$ to any rate limiter $\mathcal{R}_i$ to instruct them whether and how often they are supposed to help

[9] PHE protocols cannot ensure correct decryption of a stored ciphertext once a server gets corrupted: an adversary can modify the ciphertext database by replacing a ciphertext for m with a valid ciphertext for a different plaintext m'.

$\mathcal{F}_{\text{PHE}}$ keeps a list $\mathsf{storageHistory}$ which tracks, for each $id \in \mathbb{N}$ denoting a database position, all password-message-pairs (pw, m) that have been stored under that ID at some point.

Initialize $\mathsf{retrieveRate}(\mathcal{R}_i, id) \leftarrow 0, i \in \{1, \ldots, \mathsf{n}\}, id \in \mathbb{N}$.

On $(\texttt{Store}, id, pw, m)$ from the server $\mathcal{S}$, do:
- append (pw, m) to $\mathsf{storageHistory}(id)$. Send $(\texttt{Store}, id, |m|)$ to the adversary $\mathcal{A}$.

On $(\texttt{Retrieve}, id, pw')$ from the server $\mathcal{S}$, do:
- store this request with a unique request id $rqid$. Send $(\texttt{Retrieve}, id, rqid)$ to the adversary $\mathcal{A}$.

On $(\texttt{HelpRetrieve}, id)$ from any ratelimiter $\mathcal{R}_i$, do:
- increment $\mathsf{retrieveRate}(\mathcal{R}_i, id)$ and notify $\mathcal{A}$.

On $(\texttt{FinishRetrieve}, rqid)$ from the adversary $\mathcal{A}$, do:
- $\mathcal{A}$ may decide that this request failed and resulted in an error; if so, return $(\texttt{FinishRetrieve}, \perp)$ to the server $\mathcal{S}$. Otherwise, do the following.
- retrieve the request $(\texttt{Retrieve}, id, pw')$ corresponding to $rqid$.
- let $\mathcal{A}$ decide on a set $RLset$ of honest ratelimiters. Ensure that $\mathsf{retrieveRate}(\mathcal{R}_i, id) \geq 1\ \forall \mathcal{R}_i \in RLset$ and $|RLset| + n_c \geq \mathsf{t}$ where n_c is the number of corrupted ratelimiters. Decrement $\mathsf{retrieveRate}(\mathcal{R}_i, id)\ \forall \mathcal{R}_i \in RLset$.
- let (pw, m) be the most recent entry in $\mathsf{storageHistory}(id)$; if this list is still empty, set $(pw, m) \leftarrow (\perp, \perp)$.
- if the server $\mathcal{S}$ has been corrupted at any point since that entry was stored or no entry was stored yet, let $\mathcal{A}$ decide whether decryption should instead use an older entry (pw_i, m_i) stored at some attacker-chosen position i of $\mathsf{storageHistory}(id)$ or an attacker provided message $m_{\mathcal{A}}$ and password $pw_{\mathcal{A}}$. If so and depending on the choice, set $(pw, m) \leftarrow (pw_i, m_i)$ or $(pw, m) \leftarrow (pw_{\mathcal{A}}, m_{\mathcal{A}})$.
- If $pw' == pw$, return $(\texttt{FinishRetrieve}, m)$ to the server $\mathcal{S}$. Otherwise, return $(\texttt{FinishRetrieve}, \perp)$.

On $(\mathsf{ChangeCorruption}, P)$ from the adversary $\mathcal{A}$, do:
- Toggle the corruption status of the party P.
- If this decorrupts a ratelimiter $\mathcal{R}_i$, reset $\mathsf{retrieveRate}(\mathcal{R}_i, id) \leftarrow 0, id \in \mathbb{N}$.

On $(\texttt{PwGuessStart}, id, i)$ from the adversary $\mathcal{A}$, do:
- Check that $\mathsf{storageHistory}(id)$ is non-empty at position i.
- If the server $\mathcal{S}$ and at least t ratelimiters were corrupted simultaneously at some point in time, then proceed. Otherwise, check that the server $\mathcal{S}$ is currently corrupted and then perform the same ratelimiting checks as for $\texttt{FinishRetrieve}$, including decrementing $\mathsf{retrieveRate}$.
- If these checks pass, store a new password guess token for the i-th ciphertext stored under ID id.

On $(\texttt{PwGuessFinish}, id, i, pw_{\mathcal{A}})$ from the adversary $\mathcal{A}$, do:
- Check that there is at least one remaining password guess token for the i-th ciphertext stored under ID id. Delete one such token.
- Let (pw, m) be the i-th entry of $\mathsf{storageHistory}(id)$.
- If $pw == pw_{\mathcal{A}}$, return $(\texttt{PwGuessFinish}, m)$ to the adversary $\mathcal{A}$. Otherwise, return $(\texttt{PwGuessFinish}, \perp)$.

Fig. 1: The ideal TPHE Functionality $\mathcal{F}_{\text{PHE}}$. It models an encryption server $\mathcal{S}$ running with n ratelimiters $\mathcal{R}_i$ using threshold $\mathsf{t} \leq \mathsf{n}$.

with a decryption of a ciphertext at position id. This modeling not only ensures that any realization of $\mathcal{F}_{\text{PHE}}$ will be secure irrespective of which rate limiting algorithm is used (since this decision is left to the environment in the security proof). It also allows protocols composed with $\mathcal{F}_{\text{PHE}}$ to, if desired, fix a specific rate limiting algorithm. The key rotation operation is not expected to be triggered explicitly by a higher-level protocol and hence not part of the inputs that $\mathcal{F}_{\text{PHE}}$ accepts; it is instead captured by the ChangeCorruption command discussed below.

The logic behind `HelpRetrieve`(id) and `Store`(id, pw, m) is straightforward: Each call to `HelpRetrieve` simply increases the state variable retrieveRate by 1, thus permitting one additional decryption. `Store` stores the inputs and then notifies the ideal adversary/simulator $\mathcal{A}$ that a storage operation has been started while only revealing the length of the message m and the position id, modeling part of *encryption secrecy*. In contrast, `Retrieve`(id, pw') has to be more complex. It starts by notifying the adversary $\mathcal{A}$ that a retrieve operation has started but without revealing the password, modeling *decryption secrecy*. $\mathcal{A}$ can decide whether and when this operation finishes by sending a message `FinishRetrieve` to $\mathcal{F}_{\text{PHE}}$. $\mathcal{F}_{\text{PHE}}$ then performs several checks to ensure that *decryption is rate limited* and *outputs are correct* in those cases where these properties are expected. The adversary gets to choose which honest rate limiters take part in decryption, if any, as long as their counters retrieveRate are larger than 0; their counters are then decremented by 1 accordingly. $\mathcal{F}_{\text{PHE}}$ verifies whether the password pw' matches the one previously used to store the current ciphertext. This captures *decryption secrecy* since the check is internal such that no one learns pw'. This is also another part of *encryption secrecy*, i.e., it is impossible to learn a stored plaintext without the correct password, not even when server and rate limiters willingly perform a decryption. Depending on the result of all checks, the server $\mathcal{S}$ then returns an error or the plaintext m that was previously stored.

A seemingly small but non-trivial and surprisingly crucial detail is to define the functional behavior of `Retrieve` in $\mathcal{F}_{\text{PHE}}$ for cases where correctness cannot be guaranteed. Specifically, if the server in a real TPHE protocol was corrupted at some point, then the adversary might have modified the ciphertext database. Among others, he might replace ciphertexts with older ones that were leaked but had since then been deleted by the server. He might also insert new adversarially-generated ciphertexts that decrypt to a message that was never stored (and is hence unknown to $\mathcal{F}_{\text{PHE}}$ in the ideal world). Both cases cannot be prevented yet might result in a successful decryption, namely, when ciphertexts were generated for the id where they are currently stored and the password used for decryption matches the one used for encryption. If the server was corrupted, we hence cannot define $\mathcal{F}_{\text{PHE}}$ to only check against the most recent ciphertext, to return an error, or use a definition that generates any form of output without consulting $\mathcal{A}$; all of these would be impossible to realize. Instead, whenever correctness cannot be guaranteed $\mathcal{F}_{\text{PHE}}$ lets the adversary $\mathcal{A}$ optionally determine either an arbitrary position in storageHistory(id) or a password and plaintext of $\mathcal{A}$'s choosing that is used to check success of decryption against. While this circumvents impossibility of realizations, it also implies that a simulator for a realization must be able to extract plaintexts corresponding to ciphertexts that were maliciously crafted and inserted by the real adversary. Full composability of secure (t)PHE schemes hence requires *plaintext extractability* in addition to the intuitively expected PHE security properties. With our scheme UCPY we show for the first time that this can be achieved.

Finally, the adversary $\mathcal{A}$ is always allowed to return an error for `Retrieve` operations. This covers two cases that can occur in real TPHE protocols. Firstly, `Store`(id, pw, m) might fail, i.e., a ciphertext potentially already stored at id gets deleted but no new ciphertext gets stored, say, because some network messages were lost. The only visible result of such a failure is that following `Retrieve` operations for id will result in an error. Secondly, when $\mathcal{A}$ corrupts a server, he might modify the ciphertext database by inserting invalid ciphertexts or moving ciphertexts to different IDs that they were not generated for, say, to try to circumvent rate limiting which is based on the ID. In a secure TPHE protocol, this will not result in a successful decryption but will yield an error.

Modeling (de-)corruption and key rotation. The adversary $\mathcal{A}$ can, at any point in time, issue a `ChangeCorruption` command to switch the current corruption status (honest vs. malicious/corrupted) of the server or a ratelimiter. This abstracts from the particular mechanism used in a realization to change the corruption status. It hence covers static, epoch-based, and fully adaptive corruption of parties as well

as de-corruption due to a successful key rotation. It also does not impose any conditions on when or how decorruptions are triggered. As a result, $\mathcal{F}_{\text{PHE}}$ supports realizations that use arbitrary key rotation schedules.

After corrupting a server in $\mathcal{F}_{\text{PHE}}$, as usual the adversary gains full control over the server. That is, incoming `Store` and `Retrieve` requests are forwarded to the adversary, including any passwords and plaintexts, and the adversary can freely output arbitrary responses to `Retrieve` requests without any checks. In addition, the adversary is now allowed to perform password guesses (see below). By corrupting a ratelimiter, the adversary can use that ratelimiter to meet the threshold requirement for decryption and password guessing (see below). De-corruption of servers and/or ratelimiters mainly re-establishes security guarantees, such as privacy and correctness, for plaintexts that are stored following the decorruption. We note that $\mathcal{F}_{\text{PHE}}$ resets the counters retrieveRate for rate limiters upon decorruption. This formalizes that an attacker controlling a rate limiter R_i must not be able to influence future rate limiting after losing control over R_i due to a successful key rotation.

It is standard in UC to implicitly assume for modeling purposes that the environment can learn the current corruption status of parties. This is needed for a faithful simulation; otherwise a simulator could corrupt all parties in an ideal protocol, even when they were supposed to be honest, to trivially simulate any real protocol (cf., e.g., [17]). For $\mathcal{F}_{\text{PHE}}$, which allows for de-corruption, we need to slightly strengthen this modeling technique: we implicitly let the environment learn the chronological sequence of corruption and de-corruption events from $\mathcal{F}_{\text{PHE}}$, including the affected party. This is necessary to rule out situations where, e.g., the simulator briefly corrupts a party that should be honest, performs some actions using the additional power he gained (say, corrupting all ratelimiters and the server in $\mathcal{F}_{\text{PHE}}$ to perform unlimited password guesses), and then de-corrupts the party again before an environment with access to only the current corruption status has a chance to notice this change.

Password guessing. Secrecy of plaintexts m in real TPHE protocols is protected by three factors: secret key(s) of the server, secret key(s) of the rate limiters, and the password pw used during encryption. Since pw is chosen and managed outside of the TPHE protocol by some other higher-level protocol, it might have bad entropy or might have been leaked at some point, essentially voiding this factor. We reflect this inherent issue in $\mathcal{F}_{\text{PHE}}$ by offering a password guessing interface to the adversary $\mathcal{A}$: whenever pw is the only remaining protection for a ciphertext, $\mathcal{A}$ can issue a password guess pw' for that ciphertext. If pw' matches the password pw used to store that ciphertext, i.e., if $\mathcal{A}$ somehow manages to obtain or guess pw, then $\mathcal{F}_{\text{PHE}}$ reveals the original plaintext m to the adversary (this completes modeling *encryption secrecy*). Password guessing is available for all ciphertexts, not just the most recent one stored at any given position id, since ciphertexts might have been leaked and stored by the attacker at some point in the past to try to decrypt them later on. This modeling is similar in spirit to ideal functionalities for password-authenticated key exchange, e.g., [19].

More technically, password guessing consists of two steps: `PwGuessStart` first checks whether a password guess is possible for a given ciphertext, i.e., pw is the only remaining protection mechanism, which captures *online password guessing*. In a secure real TPHE protocol, this should be the case only in two situations: (i) the attacker controls the server and enough rate limiters are willing to help with a decryption. Then the attacker can run the interactive online decryption algorithm on a password pw' of his choosing but only as often as rate limiters permit. Or (ii) the attacker managed to obtain all keys from servers and rate limiters. Since key rotation is supposed to break links between keys belonging to different epochs, this information must have been obtained from the same epoch. If this happens, then the attacker has all information to run decryption locally and arbitrarily often for passwords pw' of his choosing. If `PwGuessStart` determines that a guess is possible, it stores a token that can be used for a single password guess query on that ciphertext by calling the second operation `PwGuessFinish`. Notably, `PwGuessFinish` might be called at a much later point in time and the adversary $\mathcal{A}$ only has to commit on the password pw' of his choice during that second step.

The split into two commands `PwGuessStart` and `PwGuessFinish`, rather than a single operation `PwGuess` that does both, allows for a wider range of realizations of $\mathcal{F}_{\text{PHE}}$. Specifically, in a real TPHE protocol including UCPY an attacker might corrupt the server, start running the decryption procedure with the help of some honest rate limiters, and collect their responses (represented by `PwGuessStart`). Instead of finishing decryption, the attacker might store the responses and wait until after all parties have performed a successful key rotation. Only then might the attacker finish decryption which typically involves, e.g., some random

oracle calls that allow a simulator to extract the password guess pw' (represented by `PwGuessFinish`). If $\mathcal{F}_{\text{PHE}}$ were to merge both operations into a single query `PwGuess`(pw'), then there is no simulator for such a protocol even though the protocol can still provide all intuitively expected security properties: The simulator cannot use `PwGuess` before the key rotation because he cannot yet extract pw'. But rate limiters reset their counters $\mathsf{retrieveRate}$ during key rotation, so the simulator also cannot use `PwGuess` when he learns pw' since there might no longer be enough rate limiters willing to help with decryption.

4 UCPY – A Round-Optimal UC-secure TPHE Scheme

In this section, we provide the construction of UCPY, the first round-optimal UC-secure $(t, m)-$PHE scheme. To do so, we first introduce the used notation, cryptographic primitives, and requirements to build this scheme and afterward provide the construction in Section 4.1.

Notation. Let $\lambda \in \mathbb{N}$ be the security parameter. We denote the set $\{1, \ldots, m\}$ by $[m]$. Let $\mathbb{G}$ be an additive cyclic group of prime order p with generator P. We use the implicit representation of group elements as introduced in [22]. For $a \in \mathbb{Z}_{\mathsf{q}}$, we define $[a] = aP \in \mathbb{G}$ as the implicit representation of a in $\mathbb{G}$, which we always use for group elements. Let $\mathsf{BG} = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, \mathsf{q}, [1]_1, [1]_2, e)$ be an asymmetric bilinear group, where $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t$ are cyclic groups of order p. $[1]_1$ and $[1]_2$ are generators of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively, and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$ is an efficiently computable (non-degenerate) bilinear map. For $s \in \{1, 2, T\}$ and $a \in \mathbb{Z}_p$, we define $[a]_s = aP_s \in \mathbb{G}_s$ as the implicit representation of a in $\mathbb{G}_s$. We denote $e([a]_1, [b]_2)$ as $[a]_1 \cdot [b]_2$. So $[a]_1 \cdot [b]_2 = [ab]_t$. Let $r \xleftarrow{\$} \mathcal{S}$ denote a uniformly random sample of an element r from a set $\mathcal{S}$, and let ϵ be any negligible function of λ. Note that from $[a] \in \mathbb{G}$ it is generally hard to compute the value a (discrete logarithm problem in $\mathbb{G}$).

Ingredients. We use the following cryptographic primitives to construct UCPY: (i) five hash functions modeled as random oracles with appropriate range. (ii) a bilinear group $\mathsf{BG} = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, \mathsf{q}, [1]_1, [1]_2, e)$ of prime order q equipped with a pairing e. The algorithm `BGGEN`(η) outputs such a group and additionally generators of $\mathbb{G}_1$ and $\mathbb{G}_2$. (iii) Zero-knowledge proofs $\mathsf{ZK}.\Pi \leftarrow (\mathsf{ZK.KGen}, \mathsf{ZK}.\mathcal{P}, \mathsf{ZK}.\mathcal{V})$ [26, 27] for two different languages which we discuss in a later paragraph.

Setup. UCPY relies on a trusted party to run the setup algorithm, which is a reasonable assumption for any (t, m)-PHE since the trusted server can perform this task. After running the setup algorithm, the server sends key shares to ratelimiters and securely deletes them. This setup is sufficient because ratelimiters only contribute secret key shares in UCPY protocols. If there is mistrust in the server's setup capabilities, we can emulate the setup protocol using a secure multiparty computation (MPC) protocol.

4.1 Construction

Before presenting the formal construction of UCPY, we give a brief overview of how our approach meets the diverse requirements of (t, m)-PHE. Detailed descriptions can be found in Figure 2 for encryption, Figure 3 for decryption, and Figure 4 for key rotation.

A PHE scheme can be considered a distributed key derivation function (KDF) operating on inputs of a password pw, a user identity id, and secret values held by the server and the ratelimiter. These secret values should be updatable with efficient communication between the server and ratelimiter, all achieved without the need for user interaction to satisfy PCI DSS [47] (c.f. Section 1).

In the context of UCPY, we implement this distributed KDF as follows: Upon receiving a user identity and password, the server computes a randomized commitment on the password, denoted as $[p]_2 := r \cdot H_2(pw, n)$, where r is a random number and n is a random nonce generated jointly by the server and t ratelimiters which is freshly drawn for each enrollment. The server then sends this randomized commitment $[p]_2$ and the user identity id to the ratelimiter. Upon receiving these values, the ratelimiter proceeds to compute a commitment to the user identity by evaluating $H_1(id, n)$. The ratelimiter then combines this commitment with the received randomized password commitment by performing a pairing operation on both the password commitment and the user identity commitment. We denote the resulting value as $[o]_t := H_1(id, n) \cdot [p]_2 = r \cdot H_1(id, n) \cdot H_2(pw, n)$.

To further include its key share k_i, the i^{th} ratelimiter calculates $[u_i]_t := k_i \cdot [o]_t$ and sends $[u_i]_t$ value back to the server. By combining all shares from a threshold of t ratelimiters, the server can construct the output of the distributed KDF, where $k_{\mathcal{R}}$ is the shared ratelimiter key via

$$[f]_t = \frac{[u]_t}{r} + k_{\mathcal{S}} \cdot [o]_t = (k_{\mathcal{S}} + k_{\mathcal{R}}) \cdot H_1(id, n) \cdot H_2(pw, n).$$

Since ratelimiters might be malicious, we introduce Schnorr-style well-formedness proofs for the share $[u_i]_t$. The proofs π_i are defined with respect to the language

$$\mathcal{L}_{\pi_i} := \{([o]_t, [u_i]_t, [\mathsf{sk}_i]_t) \mid \exists\ k_i \text{ s.t } [u_i]_t = k_i \cdot [o]_t \wedge [\mathsf{sk}_i]_t = k_i \cdot [1]_t\}.$$

Since the server can locally compute the value $[o]_t$ and knows the public key share pk_i, this language guarantees the well-formedness of $[u_i]_t$.

Setup Phase. The setup algorithm executes $\mathtt{BGGEN}(\eta)$ and $\mathsf{ZK.KGen}(\eta)$ to generate both the group description and the public parameter pp for the Schnorr proof. It samples a random key $k_{\mathcal{R}} \leftarrow \mathbb{Z}_q^*$ and uses Shamir's Secret Sharing to secretly share $k_{\mathcal{R}}$ among the m ratelimiters using coefficients $\delta_{i\in[1,m]}$ and a reconstruction threshold t. When the key is shared, each ratelimiter R_i has k_i as its secret key. The corresponding public key on the server is $\mathsf{pk}_i \leftarrow k_i \cdot [1]_t = [k_i]_t$ for $i \in [1, m]$. The setup algorithm also samples initial nonces n_i for each ratelimiter R_i that the server also knows. It also stores all possible subsets of $[1 \dots m]$ of size t in $\mathsf{T}_{\mathsf{set}}$. Intuitively, the set $\mathsf{T}_{\mathsf{set}}$ contains all possible sets of ratelimiters, which can help to reconstruct $k_{\mathcal{R}}$. Finally, the setup algorithm samples a random key $k_{\mathcal{S}}$, which will be the server's part of protecting the password.

Encryption and Decryption. The intuitive explanations above already include several major ideas for both processes. To encrypt a message M, the server first constructs the token $[f]_t$ by interacting with the ratelimiters using the distributed KDF (as described above) and computes the ciphertext

$$c_1 \leftarrow \mathsf{H}_{\mathsf{OTP}}\left([f]_t, pw, id, n\right) \oplus M$$

and the MAC

$$c_2 \leftarrow \mathsf{H}_{\mathsf{MAC}}\left([f]_t, M, pw, id, n\right).$$

To decrypt the ciphertext (c_1, c_2), the server and the ratelimiters proceed analogously. We provide formal descriptions of the interactive encryption and decryption protocols in Figure 2 and Figure 3, respectively.

Key Rotation. For key rotation in the ratelimiters, the server initiates the process by sampling a random value α and sending Shamir secret shares s_i of α to each ratelimiter. Each ratelimiter updates its partial key $k_i' \leftarrow k_i - s_i$ and responds with the updated public key along with a freshly sampled nonce. The server updates its private key by computing $k_{\mathcal{S}}' \leftarrow k_{\mathcal{S}} + \alpha$. The idea of this update process is, that the added α on the server side is subtracted by all ratelimiters, such that the combined key $k_{\mathcal{S}}' + k_{\mathcal{R}}' = k_{\mathcal{S}} + k_{\mathcal{R}}$ remains constant. Upon completion of the key rotation, the server evaluates which subsets of the public key shares are suitable for reconstructing the public key pk using the $\mathsf{Findset}(.)$ method and stores the identified ratelimiter sets capable of reconstructing $k_{\mathcal{R}}$ in $\mathsf{T}_{\mathsf{set}}$. For a detailed protocol description of key rotation and ciphertext update, we refer to Figure 4.

4.2 Security Model for the TPHE Scheme

We model and analyze UCPY as a hybrid protocol in a model for Universal Composability. From a high-level perspective, we *(i)* model the operations of UCPY during encryption and decryption between server and ratelimiters in the random oracle model. *(ii)* Server and ratelimiters communicate during de- and encryption via authenticated channels which are asynchronous, i.e., do not ensure message order nor eventual message delivery. Key rotations are performed via secure channels to protect key update shares. *(iii)* We consider an epoch-based corruption model where parties can be corrupted at the start of each new epoch but not during one. We classify key rotations into two cases to define epochs: We call a key rotation *honest* if parties were able to restore security, which is the case if the key rotation was successful and the attacker did not actively

Server$(k_{\mathcal{S}}, id, pw, M, \{\mathsf{pk}_i\}_{i\in[1,m]}, \mathsf{T}_{\mathsf{set}}, \{n_i\}_{i\in[1,m]}, \mathsf{pp})$

1 : choose $T \leftarrow\!\$\ \mathsf{T}_{\mathsf{set}}$ ratelimiters for current round.
2 : $n_{\mathcal{S}} \leftarrow\!\$\ \{0,1\}^{\lambda}; r \leftarrow\!\$\ \mathbb{Z}_q^*$
3 : $n \leftarrow \mathsf{H}_{\mathsf{N}}(\{j, n_j\}_{j\in T}, n_{\mathcal{S}})$
4 : $[p]_2 \leftarrow r \cdot H_2(pw, n)$
5 : **send** $(id, [p]_2, \{j, n_j\}_{j\in T}, n_{\mathcal{S}})$ **to** $\mathcal{R}_i$
6 : **upon receiving** $(id, \pi_i, [u_i]_t, n_i)$ **from** $\mathcal{R}_i$
7 : save n_i for next round.
8 : $[o]_t \leftarrow H_1(id, n) \cdot [p]_2$
9 : **if** $!\mathsf{ZK}.\mathcal{V}([1]_t, [o]_t, \mathsf{pk}_i, [u_i]_t, \pi_i, \mathsf{pp})$ **then abort**
10 : $[u]_t \leftarrow \sum_{j\in T} \delta_j \cdot [u_j]_t$ // $[u]_t = k_{\mathcal{R}} \cdot r \cdot H_1(id, n) \cdot H_2(pw, n)$
11 : $[f]_t \leftarrow \frac{[u]_t}{r} + k_{\mathcal{S}} \cdot [o]_t$ // $[f]_t = (k_{\mathcal{S}} + k_{\mathcal{R}}) \cdot H_1(id, n) \cdot H_2(pw, n)$
12 : $c_1 \leftarrow \mathsf{H}_{\mathsf{OTP}}([f]_t, pw, id, n) \oplus M$
13 : $c_2 \leftarrow \mathsf{H}_{\mathsf{MAC}}([f]_t, M, pw, id, n)$
14 : **return** $(id, n, (c_1, c_2))$

i-th Ratelimter(k_i, n_i, pp)

1 : **upon receiving** $(id, [p]_2, \{j, n_j\}_{j\in T}, n_{\mathcal{S}})$ **from** $\mathcal{S}$
2 : **if** $(n_i \neq n_j$ for $j = i$ and $j \in T$)
3 : **then** sample a fresh $n_i \leftarrow\!\$\ \{0,1\}^{\lambda}$
4 : **send** (id, n_i) **to** $\mathcal{S}$, and **abort**
5 : $n \leftarrow \mathsf{H}_{\mathsf{N}}(\{j, n_j\}_{j\in T}, n_{\mathcal{S}})$
6 : $[o]_t \leftarrow H_1(id, n) \cdot [p]_2$
7 : $[u_i]_t \leftarrow k_i \cdot [o]_t$
8 : $\mathsf{pk}_i \leftarrow k_i \cdot [1]_t$
9 : $\pi_i \leftarrow \mathsf{ZK}.\mathcal{P}([1]_t, [o]_t, \mathsf{pk}_i, [u]_t, k_i, \mathsf{pp})$
10 : $n_i \leftarrow\!\$\ \{0,1\}^{\lambda}$
11 : **send** $(id, \pi_i, [u_i]_t, n_i)$ **to** $\mathcal{S}$

Fig. 2: TPHE Encryption

control any parties during the key rotation. All other key rotations are called *dishonest*. Each *honest* key rotation defines the start of a new epoch. The adversary $\mathcal{A}$ can freely decide whether and when which type of key rotation is performed, which captures all cases of key rotations including different rotation schedules that can occur in practice. *(iv)* During an *honest* key rotation, all parties are modeled to honestly execute the protocol via secure and reliable channels. At the end, $\mathcal{A}$ decides which parties are corrupted in the following epoch. $\mathcal{A}$ can corrupt the server and up to $\mathsf{t}-1$ ratelimiters or as many ratelimiters as it likes but not the server. As usual, $\mathcal{A}$ obtains full control over corrupted parties including their internal state after the key rotation. *(v)* During a *dishonest* key rotation, we do not make any assumptions or restrictions besides using the aforementioned secure (but possibly unreliable) channel. $\mathcal{A}$ thus fully orchestrates the dishonest key rotation process including the behavior of those parties that are currently corrupted and hence under his control.

More formally, we model UCPY as the hybrid protocol $\mathcal{P}_{\mathrm{PHE}} = (\mathcal{S}, \mathcal{R} \mid \mathcal{F}_{\mathrm{ro}}^{1}, \mathcal{F}_{\mathrm{ro}}^{2}, \mathcal{F}_{\mathrm{ro}}^{\mathsf{OTP}}, \mathcal{F}_{\mathrm{ro}}^{\mathsf{MAC}}, \mathcal{F}_{\mathrm{ro}}^{\mathsf{N}}, \mathcal{F}_{\mathrm{nizk}}, \mathcal{F}_{\mathrm{init\&rotateKey}}, \mathcal{F}_{\mathrm{auth}})$. We illustrate the static structure in Figure 5. The environment $\mathcal{E}$ can send inputs to and receives outputs from the server $\mathcal{S}$ and ratelimiters $\mathcal{R}$. In one session of $\mathcal{P}_{\mathrm{PHE}}$, there is at most one server $\mathcal{S}$ active and at most n ratelimiters $\mathcal{R}$. $\mathcal{S}$ and $\mathcal{R}$ run the protocol as defined in Figure 2 to 4 using the following ideal subroutines:

Random oracles $\mathcal{F}_{\mathrm{ro}}^{i}$ modelling hash functions H_i for $i \in \{1, 2, \mathsf{OTP}, \mathsf{MAC}, \mathsf{N}\}$.

Server$(k_{\mathcal{S}}, pw, \{\mathsf{pk}_i\}_{i\in[1,m]}, \mathsf{T}_{\mathsf{set}}, n, id, (c_1, c_2), \mathsf{pp})$

1 : choose $T \leftarrow\!\$\ \mathsf{T}_{\mathsf{set}}$ ratelimiters for current round.
2 : $r \leftarrow\!\$\ \mathbb{Z}_q^*$
3 : $[p]_2 \leftarrow r \cdot H_2(pw, n)$
4 : **send** $(id, n, [p]_2)$ **to** $\mathcal{R}_i$
5 : **upon receiving** $(id, [u]_t)$ **from** $\mathcal{R}_i$
6 : $[u]_t \leftarrow \sum_{j\in T} \delta_j \cdot [u_j]_t$ // $[u]_t = k_{\mathcal{R}} \cdot r \cdot H_1(id, n) \cdot H_2(pw, n)$
7 : $[f]_t \leftarrow \frac{[u]_t}{r} + k_{\mathcal{S}} \cdot [o]_t$ // $[f]_t = (k_{\mathcal{S}} + k_{\mathcal{R}}) \cdot H_1(id, n) \cdot H_2(pw, n)$
8 : $M \leftarrow \mathsf{H}_{\mathsf{OTP}}([f]_t, pw, id, n) \oplus c_1$
9 : **if** !Check $\left(c_2 \stackrel{?}{=} \mathsf{H}_{\mathsf{MAC}}([f]_t, M, pw, id, n)\right)$ **then abort**
10 : **return** M

i-th Ratelimiter(k_i, pp)

1 : **upon receiving** $(id, n, [p]_2)$ **from** $\mathcal{S}$
2 : **if** !CheckRl(id) **then abort**
3 : $[o]_t \leftarrow H_1(id, n) \cdot [p]_2$
4 : $[u]_t \leftarrow k_i \cdot [o]_t$
5 : **send** $(id, [u]_t)$ **to** $\mathcal{S}$

Fig. 3: TPHE Decryption

Server$(k_{\mathcal{S}}, \mathsf{pk})$

1 : $\alpha \leftarrow\!\$\ \mathbb{Z}_q^*, k_{\mathcal{S},new} \leftarrow k_{\mathcal{S}} + \alpha$
2 : $(s_1, \cdots, s_m) \leftarrow\!\$\ \mathsf{Share}(\alpha)$
3 : **send** s_i **to** $\mathcal{R}_i$
4 : **upon receiving** $(\mathsf{pk}_{i,new}, n_i)$ **from** $\mathcal{R}_i$
5 : $\mathsf{T}_{\mathsf{set}} \leftarrow \mathsf{Findset}(\mathsf{pk}, \{\mathsf{pk}_{i,new}\}_{i\in[1,m]}, k_{\mathcal{S},new} \cdot [1]_t)$
6 : **return** $(\{\mathsf{pk}_{i,new}\}_{i\in[1,m]}, \mathsf{T}_{\mathsf{set}}, \{n_i\}_{i\in[1,m]})$

i-th Ratelimter(k_i)

1 : **upon receiving** s_i **from** $\mathcal{S}$
2 : $k_{i,new} = k_i - s_i$
3 : $\mathsf{pk}_i \leftarrow k_{i,new} \cdot [1]_t$
4 : **send** $\mathsf{pk}_{i,new}, n_i$ **to** $\mathcal{S}$

Fig. 4: TPHE Key Rotation

The ideal NIZK functionality $\mathcal{F}_{\text{nizk}}$ [26, 27] allows for generating non-interactive ZKPs and verifying them. This is used for NIZKPs generated by ratelimiters in their responses.

An authenticated channel $\mathcal{F}_{\text{auth}}$ [40] provides (unreliable) authenticated channels between $\mathcal{S}$ and $\mathcal{R}$ used for encryption and decryption.

The initialization and key rotation functionality $\mathcal{F}_{\text{init\&rotateKey}}$ serves three different purposes: *(i)* it models a trusted execution of UCPY's setup phase. It initially runs $\texttt{BGGEN}(1^\eta)$ and distributes these public parameters to $\mathcal{S}$, $\mathcal{R}$, and $\mathcal{F}_{\text{ro}}^i$ for $i \in \{1, 2, \mathsf{OTP}, \mathsf{MAC}, \mathsf{N}\}$. $\mathcal{F}_{\text{init\&rotateKey}}$ further generates the secret master key sk which is necessary for decryption. It samples a server key $\mathsf{sk}_{\mathcal{S}}$, computes the overall ratelimiter key $\mathsf{sk}_{\mathcal{R}} \leftarrow \mathsf{sk} - \mathsf{sk}_{\mathcal{S}}$, and Shamir-secret shares $\mathsf{sk}_{\mathcal{R}}$ among the ratelimiters. It also generates the corresponding initial public keys of the ratelimiters and distributes them to $\mathcal{S}$. *(ii)* Honest key rotation (changing epochs): models a correct and successful execution of UCPY's key rotation subprotocol. During this process, all participating parties - including those that were previously corrupted but are no longer under the control of the adversary - execute the protocol honestly. In particular, $\mathcal{F}_{\text{init\&rotateKey}}$ computes and distributes the Shamir-secret sharing of α to the ratelimiters and updates the server key by adding α. It collects the public keys computed by the now-honest ratelimiters and provides them to $\mathcal{S}$. The server $\mathcal{S}$ then finalizes the key rotation. We note that this captures reliable communication via secure channels between $\mathcal{S}$ and $\mathcal{R}$. At the start of an honest key

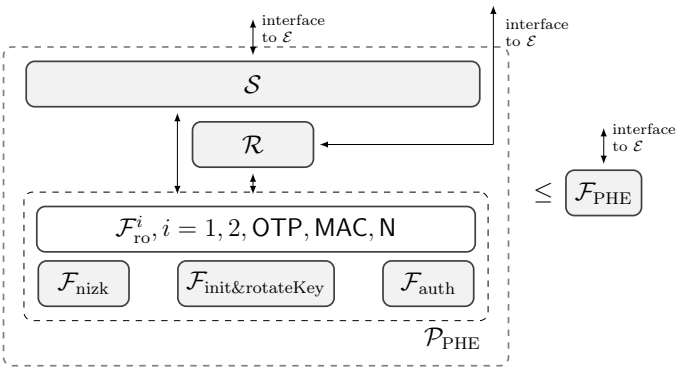

Fig. 5: Schematic representation of Theorem 1. $\mathcal{P}_{\text{PHE}}$ is defined as $(\mathcal{S}, \mathcal{R} \mid \mathcal{F}^1_{\text{ro}}, \mathcal{F}^2_{\text{ro}}, \mathcal{F}^{\mathsf{OTP}}_{\text{ro}}, \mathcal{F}^{\mathsf{MAC}}_{\text{ro}}, \mathcal{F}^{\mathsf{N}}_{\text{ro}}, \mathcal{F}_{\text{nizk}}, \mathcal{F}_{\text{init\&rotateKey}}, \mathcal{F}_{\text{auth}})$. (i) $\mathcal{F}^i_{\text{ro}}$ for $i \in \{1, 2, \mathsf{OTP}, \mathsf{MAC}, \mathsf{N}\}$ denote random oracles, (ii) $\mathcal{F}_{\text{nizk}}$ is an ideal ZKP functionality, (iii) $\mathcal{F}_{\text{init\&rotateKey}}$ is the initialization and key rotation functionality, and (iv) $\mathcal{F}_{\text{auth}}$, an ideal authenticated channel functionality. All machines are additionally connected to the adversary $\mathcal{A}$.

rotation, $\mathcal{A}$ can freely determine the internal state of those parties that were corrupted in the previous epoch. However, $\mathcal{A}$ does *not* have active control over corrupted parties during honest key rotation and does not learn any of the values that are exchanged among parties, capturing that the adversary has lost direct access to the corrupted party, e.g., because that party has re-installed its machine using a fresh software image. At the end of the honest key rotation $\mathcal{A}$ determines a set of parties that are considered to be corrupted in the following epoch (subject to the aforementioned restrictions) and learns their current internal state. All other parties are considered to be honest in the new epoch, even if they were corrupted in the previous epoch. *(iii)* Dishonest key rotation with potentially malicious parties: This models all remaining cases of key rotation where the server or some ratelimiters might be corrupted such that $\mathcal{A}$ might try to, e.g., extract some information from key rotations. $\mathcal{A}$ can freely orchestrate the execution of these key rotations, including the behavior of corrupted parties, but communication between two honest parties is protected via secure (unreliable) channels. The corruption status of parties does not change.

4.3 Security Proof

In this section, we give a high-level idea of our proof and defer to Appendix C for the formal proof of security. Even though we took Pythia [24] as a starting point for our construction, the security proof in [24] is insufficient for our purposes. Pythia was proposed by Everspaugh *et al.* as a partially oblivious pseudo-random function (pOPRF) service. A pOPRF is a two-party protocol that allows a user and a server to compute a PRF. The user holds the input consisting of a private and a public part, and the server holds the key. The user learns the result of the computation, while the server learns the public but not the private part of the input. Pythia has been proven secure for a game-based definition of one-more unpredictability under the one-more bilinear computational Diffie-Hellman (OM-BCDH) assumption. No further security properties were formally proven in [24]. We prove our UCPY construction to be UC-secure and show that the UCPY protocol $\mathcal{P}_{\text{PHE}}$ (see Appendix B) realizes the ideal TPHE functionality $\mathcal{F}_{\text{TPHE}}$ (see Appendix A). As part of the security proof, we construct a simulator such that – as common – an environment can not distinguish whether it interacts with the real protocol $\mathcal{P}_{\text{PHE}}$ or the ideal functionality $\mathcal{F}_{\text{TPHE}}$ (connected to the simulator). In other words, we show that an adversary learns at most as much from the real protocol as it learns from the ideal protocol. We prove the indistinguishability of $\mathcal{P}_{\text{PHE}}$ and $\mathcal{F}_{\text{TPHE}}$ under the Gap-OM-BCDH assumption.

Theorem 1. *Let $\eta \in \mathbb{N}$ be the security parameter. Let $\mathcal{P}_{PHE}$ be the UCPY protocol as introduced in Section B and $\mathcal{F}_{TPHE}$ be the ideal TPHE functionality as defined in Section A. Let $\mathsf{R}_{\mathcal{R}}$ be a relation for a language $\mathsf{L}_{\mathcal{R}} := \{([1]_t, [o]_t, \mathsf{pk}, [u]_t) \mid \exists \mathsf{sk} : \mathsf{pk} = \mathsf{sk} \cdot [1]_t \wedge [u]_t = \mathsf{sk} \cdot [o]_t\}$. Let $\mathsf{n}, \mathsf{t} \in \mathbb{N}$ be the number of ratelimiters,*

resp. the number of ratelimiters necessary to enc- and decrypt in $\mathcal{P}_{PHE}$. Let $\mathsf{ng} \in \mathbb{N}$ *be the number of nonces generated during a batch request. Suppose the* Gap-OM-BCDH *assumption holds true, then*

$$\mathcal{P}_{PHE} \leq \mathcal{F}_{TPHE}$$

in the $(\mathcal{F}_{\text{auth}}, \mathcal{F}_{\text{nizk}}, \mathcal{F}_{\text{ro}}^{i \in \{1,2,\mathsf{OTP},\mathsf{MAC},\mathsf{N}\}})$*-hybrid world.*

The simulator Sim operates as a single machine connected to the ideal functionality $\mathcal{F}_{\text{TPHE}}$ and the environment $\mathcal{E}$ through their network interfaces. During execution, Sim accepts and processes all incoming messages. Internally, it mostly emulates the real-world protocol $\mathcal{P}_{\text{TPHE}}$ and uses the information leaked by $\mathcal{F}_{\text{TPHE}}$ to construct messages for the environment indistinguishable from those originating from real parties. In the event of server compromise, Sim outputs a dummy database, adequately responds to adversarial messages, and consistently answers queries to the random oracle. We only consider the behavior of honest parties, forwarding messages addressed to corrupt parties to the environment. The complete simulator strategy of Sim is depicted in Figures 24 to 26. We focus on the main challenges and how the simulator solves them in the following.

(1) Failure events. There are some special edge cases where the simulation is distinguishable from the real protocol execution. Those include hash collisions, the reuse of nonces, a random oracle outputting the neutral element of a group, and the environment correctly guessing uniformly random values. We ignore those edge cases and show in the formal proof that they happen with negligible probability.

(2) Key generation. The simulator samples a public key pair $(\mathsf{pk}, \mathsf{sk})$ and shares it between the server and the ratelimiters. Consequently, Sim always knows all keys and can give key shares to corrupted parties.

(3) Generating dummy records. In the ideal world, when a user with ID id wants to encrypt a message m under a password pw, it calls the store function of the ideal functionality $\mathcal{F}_{\text{TPHE}}$. $\mathcal{F}_{\text{TPHE}}$ stores the data in its storage and leaks id and $|m|$ to the simulator Sim. Sim must mimic the real protocol's behavior without knowing the password and the message by following the steps independent of the unknown data and faking steps dependent on the unknown data. Instead of sending a blinded password hash $r \cdot \mathsf{H}_2(pw, n_{\mathcal{S}})$ to the ratelimiters, it samples a uniformly random $a \leftarrow\!\!\$\ \mathbb{Z}_q^*$ and sends the element $[p]_2 \leftarrow a \cdot [1]_2$ which is indistinguishable for the environment. The ratelimiters are either honest and simulated by Sim or corrupt and controlled by the environment. Simulated ratelimiters simply follow the protocol, as no information known to them in the real world is hidden in the ideal world. Once receiving all ratelimiter responses, the server would de-blind and use the resulting value as a key to (i) compute a keystream used to encrypt the message and to (ii) compute a message authentication code (MAC). Because Sim cannot de-blind the value and both computations rely on unknown data, it samples both values uniformly at random. The resulting record is, at this point, indistinguishable from a real record.

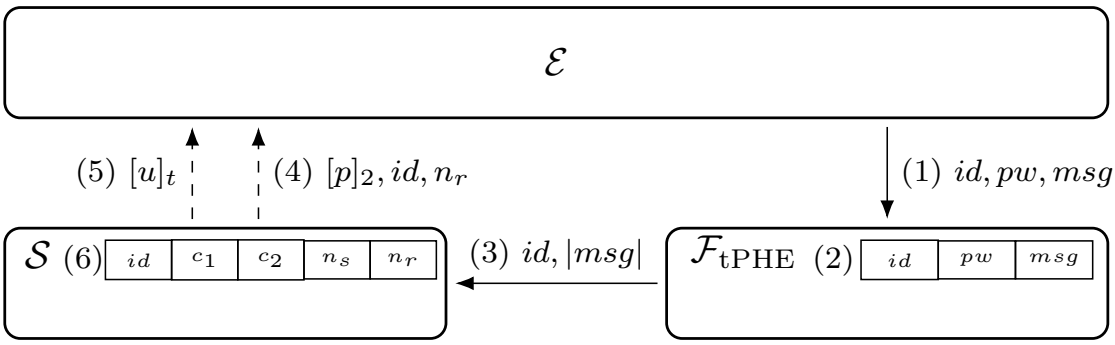

Fig. 6: Store operaton in the ideal world (dashed arrows represent leakage via $\mathcal{F}_{\text{auth}}$)

(4) Ratelimiter quota. The environment can allow a ratelimiter to participate in a retrieval request by incrementing the ratelimiters quota for a specific id. The quota of a ratelimiter is decremented every time it answers a retrieval request. This prevents a corrupt server from brute forcing a user's password, as it must interact with ratelimiters to test a password. Note that the quota of a ratelimiter in both $\mathcal{F}_{\text{TPHE}}$ and Sim is incremented when instructed by the environment.

(5) Correct decryption of dummy records. When a user wants to decrypt the message, it calls the retrieve function of $\mathcal{F}_{\text{TPHE}}$ with a candidate password pw'. $\mathcal{F}_{\text{TPHE}}$ leaks id to Sim, which must mimic the real protocol's behavior. In this case, the simulator's task is to simulate the network traffic expected from the real protocol, while $\mathcal{F}_{\text{TPHE}}$ takes care of the actual functionality, including a password check and the delivery of the stored message. The messages to the ratelimiters are generated as before, and the ratelimiters respond as before but also decrement their quota for that specific user. Once receiving all ratelimiter responses, Sim instructs $\mathcal{F}_{\text{TPHE}}$ to continue with the retrieval. Furthermore, it has to provide ratelimiters to $\mathcal{F}_{\text{TPHE}}$ with quota left, which are exactly the ones it interacted with in the simulation. $\mathcal{F}_{\text{TPHE}}$ checks whether the entered password matches the recorded one and outputs the stored message if they do. This behavior is indistinguishable from the real protocol.

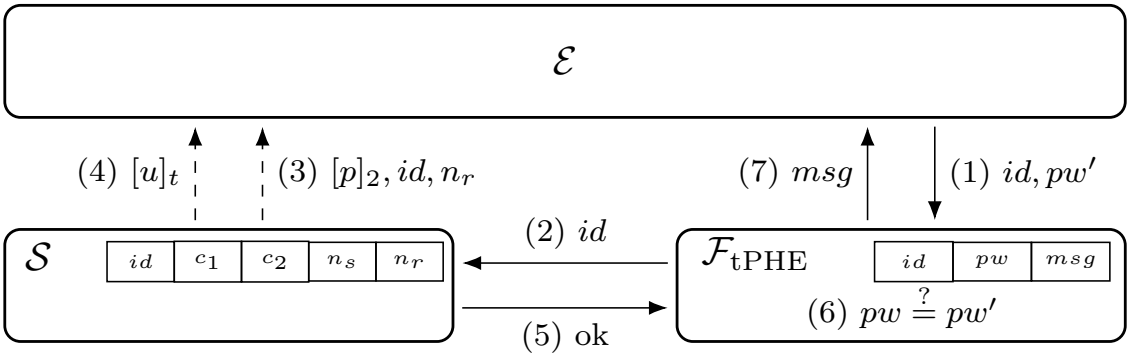

Fig. 7: Retrieval operation in the ideal world (dashed arrows represent leakage via $\mathcal{F}_{\text{auth}}$).

(6) Programming the Random Oracles. As we have seen, the decryption of dummy records is possible with an honest server. To ensure the correct decryption of dummy records with a corrupt server, Sim monitors all queries to the random oracles $\mathcal{F}_{\text{ro}}^{\mathsf{OTP}}$ and $\mathcal{F}_{\text{ro}}^{\mathsf{MAC}}$, which are used to compute the keystream and MAC, respectively. Once it detects a query that might be consistent with a dummy record, it checks that the key $[u]_t$ is computed according to the protocol ($[u]_t \leftarrow \mathsf{sk} \cdot \mathsf{H}_1(id, n) \cdot \mathsf{H}_2(pw, n)$). This is possible because id, pw, and n are part of the query and Sim knows sk. Sim sends the extracted pw in a password guess to $\mathcal{F}_{\text{TPHE}}$. This function enables Sim to test a password if it can provide $t - n_c$ honest ratelimiters with enough quota (n_c is the number of currently corrupted ratelimiters). We assume now that the environment computed the key with honest ratelimiters and show later that this is the case with overwhelming probability. Sim finds the ratelimiters involved in the computation of the key because ratelimiters track retrieval requests they helped to answer. These ratelimiters have quota left in $\mathcal{F}_{\text{TPHE}}$ as they were only used in the simulation to compute the key but not in cooperation with $\mathcal{F}_{\text{TPHE}}$ so far. If the password is the one recorded in $\mathcal{F}_{\text{TPHE}}$, Sim gets the correct message m back and programs the random oracles accordingly. This is possible now as it was able to extract the information that was hidden before during the generation of the dummy record. Once the random oracles are programmed correctly, the dummy records are indistinguishable from real ones.

(7) Correct decryption of injected records. When the environment corrupts the server, it has full control over it. This includes full read and write access to the server's storage. Therefore, it can inject records that it can interact with once the server is honest again. There are two cases we have to consider:

- *Injection of old dummy records:* The environment can extract dummy records from the storage and re-inject them in a later epoch. It is also possible that in between, new messages are stored for the same *id* as described in (3), overwriting the current storage of Sim. To ensure that the programming of the random oracles for those dummy records works as described in (6), Sim keeps a history of all dummy records. Furthermore, correct decryption of those re-injected dummy records with an honest server has to be guaranteed. Therefore, $\mathcal{F}_{\text{TPHE}}$ also keeps a history of previously stored messages such that Sim can instruct $\mathcal{F}_{\text{TPHE}}$ to use a previous record for decryption. Otherwise, the retrieval works as described in (5).
- *Injection of honestly generated records:* The environment can corrupt the server and generate records according to the protocol. Of course, these records must be decrypted correctly with an honest server. The challenge in this case is that these records are not generated as described in (3); consequently, $\mathcal{F}_{\text{TPHE}}$ has no corresponding entry in its storage. Furthermore, the environment only leaks the *id* to Sim for a retrieval query, not the entered password. To model this scenario, $\mathcal{F}_{\text{TPHE}}$ provides a function that allows Sim to give a password message pair such that $\mathcal{F}_{\text{TPHE}}$ outputs the message to the user if the entered password matches the one given by Sim. Still, Sim has to come up with the message encrypted in the injected record and the password under which it was encrypted. Because the record was generated according to the protocol, the random oracles $\mathcal{F}_{\text{ro}}^{\mathsf{OTP}}$ and $\mathcal{F}_{\text{ro}}^{\mathsf{MAC}}$ must have been queried to generate the key stream, respectively, the MAC. The input to these random oracles includes the password and, for $\mathcal{F}_{\text{ro}}^{\mathsf{MAC}}$, also the message. This allows Sim to extract the information needed from the random oracles. Therefore, the correct decryption of the injected records is assured.

(8) Injecting the $\mathsf{Gap\text{-}OM\text{-}BCDH}$ challenge (single ratelimiter). We have seen in (6) that correctly programming the random oracles is possible if the environment computed the used key in cooperation with at least $t - n_c$ honest ratelimiters. We prove that it is hard for the environment to come up with the correct key without these interactions by showing that, in this case, we can break the $\mathsf{Gap\text{-}OM\text{-}BCDH}$ assumption. For simplicity, we assume from now on that we only have one ratelimiter and show in the next step how we transfer this setting to the multi-ratelimiter setting. We give a high-level introduction to the $\mathsf{Gap\text{-}OM\text{-}BCDH}$ assumption and then show how we embed the challenges into Sim.

The $\mathsf{Gap\text{-}OM\text{-}BCDH}$ game is parameterized by three groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t$ and a bilinear pairing that maps an element from $\mathbb{G}_1$ and an element from $\mathbb{G}_2$ to an element from $\mathbb{G}_t$. The game internally samples a secret scalar sk and gives two oracles $\mathsf{Targ}_1, \mathsf{Targ}_2$ for sampling challenge group elements from $\mathbb{G}_1$ and $\mathbb{G}_2$. Furthermore, a Help oracle is provided that multiplies a given target group element $[o]_t$ with the secret scalar $[u]_t \leftarrow \mathsf{sk} \cdot [o]_t$. To win the game, one has to provide $Q + 1$ distinct tuples (i, j, σ) such that $\sigma = \mathsf{sk} \cdot [x_i]_1 \cdot [y_j]_2$ while querying Help only Q times, where $[x_i]_1$ is the i-th query to Targ_1 and $[y_j]_2$ the j-th query to Targ_2. Note that the $\mathsf{Gap\text{-}OM\text{-}BCDH}$ game also provides a DDH oracle that outputs 1 if, for given target group elements $[x]_t, [y]_t, [z]_t, [v]_t$, it holds $xy = zv$.

Instead of sampling a key pair in (2), Sim implicitly uses the random scalar of the $\mathsf{Gap\text{-}OM\text{-}BCDH}$ game as the overall secret key. If the server is honest, Sim samples a random ratelimiter secret key. If the ratelimiter is honest, Sim samples a random server secret key and sets the ratelimiter public key as $\mathsf{pk}_{\mathcal{R}} \leftarrow \mathsf{pk} - \mathsf{pk}_{\mathcal{S}}$.

We follow the proof technique introduced by [4] and guess for which tuple (id^*, n^*) the environment will provide an additional solution $[u]_t$. Our guess is correct with probability $\frac{1}{q_{\text{ro1}}}$ with q_{ro1} being the number of queries to $\mathcal{F}_{\text{ro}}^1$. Because the environment runs in polynomial time, the probability is non-negligible. To embed the challenges into Sim, we replace random sampling of group elements with queries to Targ_1 and Targ_2. This includes outputs of $\mathcal{F}_{\text{ro}}^1$ for $(id, n) = (id^*, n^*)$ and $\mathcal{F}_{\text{ro}}^2$ for all queries. For queries $(id, n) \neq (id^*, n^*)$ to $\mathcal{F}_{\text{ro}}^1$, we continue injecting trapdoors. Note that programming $\mathcal{F}_{\text{ro}}^1$ and $\mathcal{F}_{\text{ro}}^2$ in that way leads to a one-to-one correspondence between: (i) i and id, n, (ii) j and pw, n, and (iii) σ and $\mathsf{sk} \cdot \mathsf{H}_1(id, n) \cdot \mathsf{H}_2(pw, n)$. The simulator computes $[u_i]_t \leftarrow \mathsf{sk}_i \cdot [o]_t$ in (3) and (5) for unknown keys with a query to Help. In (6), Sim checks the correctness of the key by recomputing it. This is no longer possible as the secret key is no longer known to Sim. Therefore, it uses the DDH oracle on inputs $\mathsf{pk}, \mathsf{H}_1(id, n) \cdot \mathsf{H}_2(pw, n), [u]_t, [1]_t$ to check the correctness of the key.

(9) From single-ratelimiter to multi-ratelimiter. We use polynomial interpolation in the exponent to replicate Shamir's secret sharing [49]. As a reminder, t points determine a unique polynomial of degree $t-1$, and all remaining points of this polynomial can be obtained with Lagrange polynomial interpolation. This technique also works in the exponent as it only requires the addition of points and multiplication with scalars. Sim uses the random scalar of the Gap-OM-BCDH game as the underlying overall key sk that gets split into a server key $\mathsf{sk}_{\mathcal{S}}$ and an overall ratelimiter key $\mathsf{sk}_{\mathcal{R}}$. The overall ratelimiter key is shared amongst all ratelimiters with Shamir's secret sharing. At the beginning of an epoch, Sim distinguishes between two cases:

- **Honest Server** If the server is honest, Sim can sample the overall ratelimiter key at random and share it between the ratelimiters. This case is straightforward as Sim knows all ratelimiter keys such that scalar multiplications with a ratelimiter key can be done as before.
- **Corrupt Server** If the server is corrupt, Sim samples a random server key $\mathsf{sk}_{\mathcal{S}}$ and chooses $t-1$ random shares sk_i of the overall ratelimiter key such that together with $\mathsf{sk}_{\mathcal{R}} \leftarrow \mathsf{sk} - \mathsf{sk}_{\mathcal{S}}$, they determine the whole polynomial used for the sharing. The known key shares are chosen such that they belong to corrupted ratelimiters, enabling Sim to give sk_i to the environment for all corrupted ratelimiters. The public keys $[\mathsf{sk}_i]_t$ for parties with unknown key shares are computed using polynomial interpolation in the exponent. To answer queries $[o]_t$ in the en-/decryption protocol for ratelimiters with unknown keys, Sim distinguishes two cases:
  - $(id, n) = (id^*, n^*)$: The simulator (i) queries Help to obtain $\mathsf{sk} \cdot [o]_t$ and computes $\mathsf{sk}_{\mathcal{R}} \cdot [o]_t \leftarrow \mathsf{sk} \cdot [o]_t - \mathsf{sk}_{\mathcal{S}} \cdot [o]_t$, (ii) computes $\mathsf{sk}_j \cdot [o]_t$ for all $t-1$ known key shares, and (iii) uses polynomial interpolation to compute $\mathsf{sk}_i \cdot [o]_t$.
  - $(id, n) \neq (id^*, n^*)$: The simulator uses the trapdoor a injected in $\mathcal{F}_{\text{ro}}^1$ and computes $\mathsf{sk}_i \cdot [h_1]_1 \cdot [p]_2 \leftarrow a \cdot [\mathsf{pk}_i]_1 \cdot [p]_2$ which is correct as $a \cdot [1]_1 = [h_1]_1$.

The only issue left is to ensure that Sim gathered $Q+1$ solution tuples once the environment was able to compute a key $[u]_t$ for (id, pw, n) without the interaction of $t - n_c$ honest ratelimiters in (6). This is the case if Sim has not queried Help for $[x_i]_1 \cdot [y_j]_2$ with (i, j) corresponding to (id, pw, n). Note that even for these (id, pw, n), the environment can query up to $t - n_c - 1$ honest ratelimiters. Sim can only answer those queries without a Help query if the corresponding ratelimiters are exactly the ones Sim knows the key shares for. Therefore, at the beginning of each epoch, Sim has to guess the ratelimiter set the environment will query for these inputs. The probability of guessing right is non-negligible as long as $\binom{t-n_c-1}{n-n_c}$ is polynomial in the security parameter, which is the case for parameters relevant in practice.

4.4 Prototype and Evaluation

We implemented a prototype[10] to assess the performance of UCPY and to compare it with the scheme of Brost *et al.* [10]. In the following, we describe our setup and present the evaluation results.

Setup. Our prototype is implemented in Rust (rustc 1.66.1) using the pairing-friendly curve `bls12_381` for bilinear groups and `sha512` as the hash function. The server is based on Rocket 0.5.0 and the client on `reqwest`, with data exchanged via POST requests and JSON serialization through Serde 1.0. For benchmarking, we deployed the prototype on AWS `c5.2xlarge` instances (8 vCPUs, Intel Xeon Platinum 8000 series, 16 GiB RAM, up to 10 Gbps bandwidth) running Ubuntu 20.04. The main server was located in Oregon, while the ratelimiter was hosted in Northern California.

Evaluation. We evaluated UCPY for the case $t = m = 1$, which allows for a direct comparison with the evaluation of Brost *et al.* [10]. We consider both a high-latency (WAN) and a low-latency (LAN) setup. In the WAN setting, our instances were placed in Northern California and Oregon with an average ping of 21.2 ms, closely matching the setup of Brost *et al.*. The results demonstrate the round-optimal advantage of UCPY: in this configuration, our scheme achieves more than 200% faster encryption and over 300% faster decryption (Table 1). By contrast, our LAN results are somewhat slower, which we attribute to our use of pairing-friendly groups, whereas Brost *et al.* rely on standard elliptic curves.

[10] https://github.com/uctphe/uc-tphe-src

Setup	Brost *et al.* [10]	UCPY
LAN - Encrypt	8.431 ms	21.255 ms
LAN - Decrypt	18.763 ms	21.488 ms
WAN - Encrypt	94.911 ms	44.022 ms
WAN - Decrypt	147.970 ms	44.207 ms

Table 1: The latency of UCPY and [10] in a $t = m = 1$ setting.

Acknowledgments

This work was partially supported by Deutsche Forschungsgemeinschaft as part of the Research and Training Group 2475 "Cybercrime and Forensic Computing" (grant number 393541319/GRK2475/1-2019) and through grant 442893093, and by the Smart Networks and Services Joint Undertaking (SNS JU) under the European Union's Horizon Europe research and innovation program in the scope of the CONFIDENTIAL6G project under Grant Agreement 101096435. The contents of this publication are the sole responsibility of the authors and do not in any way reflect the views of the EU. This work is also partially supported by SBA Research (SBA-K1 NGC), which is a COMET Center within the COMET – Competence Centers for Excellent Technologies Programme and funded by BMIMI, BMWET, and the federal state of Vienna. The COMET Programme is managed by FFG. We thank Fritz Schmid for implementing the prototype of our scheme.

References

[1] Michel Abdalla, Fabrice Benhamouda, and Philip MacKenzie. "Security of the J-PAKE Password-Authenticated Key Exchange Protocol". In: *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2015, pp. 571–587. DOI: 10.1109/SP.2015.41.

[2] Michel Abdalla et al. "Robust Password-Protected Secret Sharing". In: *ESORICS 2016, Part II*. Ed. by Ioannis G. Askoxylakis et al. Vol. 9879. LNCS. Springer, Cham, Sept. 2016, pp. 61–79. DOI: 10.1007/978-3-319-45741-3_4.

[3] Shashank Agrawal et al. "PASTA: PASsword-based Threshold Authentication". In: *ACM CCS 2018*. Ed. by David Lie et al. ACM Press, Oct. 2018, pp. 2042–2059. DOI: 10.1145/3243734.3243839.

[4] Ruben Baecker et al. "A Fully-Adaptive Threshold Partially-Oblivious PRF". In: *CRYPTO 2025*. LNCS. Springer, 2025.

[5] Ali Bagherzandi et al. "Password-protected secret sharing". In: *ACM CCS 2011*. Ed. by Yan Chen, George Danezis, and Vitaly Shmatikov. ACM Press, Oct. 2011, pp. 433–444. DOI: 10.1145/2046707.2046758.

[6] C. Baum et al. "PESTO: Proactively Secure Distributed Single Sign-On, or How to Trust a Hacked Server". In: *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. Los Alamitos, CA, USA: IEEE Computer Society, 2020, pp. 587–606. DOI: 10.1109/EuroSP48549.2020.00044. URL: https://doi.ieeecomputersociety.org/10.1109/EuroSP48549.2020.00044.

[7] Mihir Bellare and Phillip Rogaway. "The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs". In: *EUROCRYPT 2006*. Ed. by Serge Vaudenay. Vol. 4004. LNCS. Springer, Berlin, Heidelberg, 2006, pp. 409–426. DOI: 10.1007/11761679_25.

[8] Dan Boneh et al. "Key Homomorphic PRFs and Their Applications". In: *CRYPTO 2013, Part I*. Ed. by Ran Canetti and Juan A. Garay. Vol. 8042. LNCS. Springer, Berlin, Heidelberg, Aug. 2013, pp. 410–428. DOI: 10.1007/978-3-642-40041-4_23.

[9] Tatiana Bradley et al. "Password-Authenticated Public-Key Encryption". In: *ACNS 19International Conference on Applied Cryptography and Network Security*. Ed. by Robert H. Deng et al. Vol. 11464. LNCS. Springer, Cham, June 2019, pp. 442–462. DOI: 10.1007/978-3-030-21568-2_22.

[10] Julian Brost et al. "Threshold Password-Hardened Encryption Services". In: *ACM CCS 2020*. Ed. by Jay Ligatti et al. ACM Press, Nov. 2020, pp. 409–424. DOI: 10.1145/3372297.3417266.

[11] Jan Camenisch, Robert R. Enderlein, and Gregory Neven. "Two-Server Password-Authenticated Secret Sharing UC-Secure Against Transient Corruptions". In: *PKC 2015*. Ed. by Jonathan Katz. Vol. 9020. LNCS. Springer, Berlin, Heidelberg, 2015, pp. 283–307. DOI: 10.1007/978-3-662-46447-2_13.

[12] Jan Camenisch, Anna Lysyanskaya, and Gregory Neven. "Practical yet universally composable two-server password-authenticated secret sharing". In: *ACM CCS 2012*. Ed. by Ting Yu, George Danezis, and Virgil D. Gligor. ACM Press, Oct. 2012, pp. 525–536. DOI: 10.1145/2382196.2382252.

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

[14] Jan Camenisch et al. "Memento: How to Reconstruct Your Secrets from a Single Password in a Hostile Environment". In: *CRYPTO 2014, Part II*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8617. LNCS. Springer, Berlin, Heidelberg, Aug. 2014, pp. 256–275. DOI: 10.1007/978-3-662-44381-1_15.

[15] Jan Camenisch et al. "Universal Composition with Responsive Environments". In: *ASIACRYPT 2016, Part II*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10032. LNCS. Springer, Berlin, Heidelberg, Dec. 2016, pp. 807–840. DOI: 10.1007/978-3-662-53890-6_27.

[16] Jan Camenisch et al. "Virtual Smart Cards: How to Sign with a Password and a Server". In: *SCN 16*. Ed. by Vassilis Zikas and Roberto De Prisco. Vol. 9841. LNCS. Springer, Cham, 2016, pp. 353–371. DOI: 10.1007/978-3-319-44618-9_19.

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

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

[19] Ran Canetti et al. "Efficient Password Authenticated Key Exchange via Oblivious Transfer". In: *PKC 2012*. Ed. by Marc Fischlin, Johannes Buchmann, and Mark Manulis. Vol. 7293. LNCS. Springer, Berlin, Heidelberg, May 2012, pp. 449–466. DOI: 10.1007/978-3-642-30057-8_27.

[20] Poulami Das, Julia Hesse, and Anja Lehmann. "DPaSE: Distributed Password-Authenticated Symmetric-Key Encryption, or How to Get Many Keys from One Password". In: *ASIACCS 22*. Ed. by Yuji Suga et al. ACM Press, 2022, pp. 682–696. DOI: 10.1145/3488932.3517389.

[21] Edward Eaton and Douglas Stebila. "The "Quantum Annoying" Property of Password-Authenticated Key Exchange Protocols". In: *Post-Quantum Cryptography - 12th International Workshop, PQCrypto 2021*. Ed. by Jung Hee Cheon and Jean-Pierre Tillich. Springer, Cham, 2021, pp. 154–173. DOI: 10.1007/978-3-030-81293-5_9.

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

[23] Adam Everspaugh et al. "Key Rotation for Authenticated Encryption". In: *CRYPTO 2017, Part III*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10403. LNCS. Springer, Cham, Aug. 2017, pp. 98–129. DOI: 10.1007/978-3-319-63697-9_4.

[24] Adam Everspaugh et al. "The Pythia PRF Service". In: *USENIX Security 2015*. Ed. by Jaeyeon Jung and Thorsten Holz. USENIX Association, Aug. 2015, pp. 547–562. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/everspaugh.

[25] Tore Kasper Frederiksen et al. "Attribute-based Single Sign-On: Secure, Private, and Efficient". In: *PoPETs* 2023.4 (Jan. 2023), pp. 35–65. DOI: 10.56553/popets-2023-0097.

[26] Jens Groth. "Simulation-Sound NIZK Proofs for a Practical Language and Constant Size Group Signatures". In: *ASIACRYPT 2006*. Ed. by Xuejia Lai and Kefei Chen. Vol. 4284. LNCS. Springer, Berlin, Heidelberg, Dec. 2006, pp. 444–459. DOI: 10.1007/11935230_29.

[27] Jens Groth, Rafail Ostrovsky, and Amit Sahai. "Perfect Non-interactive Zero Knowledge for NP". In: *EUROCRYPT 2006*. Ed. by Serge Vaudenay. Vol. 4004. LNCS. Springer, Berlin, Heidelberg, 2006, pp. 339–358. DOI: 10.1007/11761679_21.

[28] Yanqi Gu et al. "Threshold PAKE with Security Against Compromise of All Servers". In: *ASIACRYPT 2024, Part V*. Ed. by Kai-Min Chung and Yu Sasaki. Vol. 15488. LNCS. Springer, Singapore, Dec. 2024, pp. 66–100. DOI: 10.1007/978-981-96-0935-2_3.

[29] Julia Hesse. "Separating Symmetric and Asymmetric Password-Authenticated Key Exchange". In: *SCN 20*. Ed. by Clemente Galdi and Vladimir Kolesnikov. Vol. 12238. LNCS. Springer, Cham, Sept. 2020, pp. 579–599. DOI: 10.1007/978-3-030-57990-6_29.

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

[31] Jung Yeon Hwang et al. "Round-Reduced Modular Construction of Asymmetric Password-Authenticated Key Exchange". In: *SCN 18*. Ed. by Dario Catalano and Roberto De Prisco. Vol. 11035. LNCS. Springer, Cham, Sept. 2018, pp. 485–504. DOI: 10.1007/978-3-319-98113-0_26.

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

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

[34] Stanislaw Jarecki et al. *Highly-Efficient and Composable Password-Protected Secret Sharing (Or: How to Protect Your Bitcoin Wallet Online)*. Cryptology ePrint Archive, Report 2016/144. 2016. URL: https://eprint.iacr.org/2016/144.

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

[36] Stanislaw Jarecki et al. *Two-Factor Password-Authenticated Key Exchange with End-to-End Password Security*. Cryptology ePrint Archive, Paper 2018/033. 2018. URL: https://eprint.iacr.org/2018/033.

[37] Chunfu Jia, Shaoqiang Wu, and Ding Wang. "Reliable Password Hardening Service with Opt-Out". In: *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*. 2022, pp. 250–261. DOI: 10.1109/SRDS55811.2022.00031.

[38] Jingwei Jiang et al. "Quantum-Resistant Password-Based Threshold Single-Sign-On Authentication with Updatable Server Private Key". In: *ESORICS 2022, Part II*. Ed. by Vijayalakshmi Atluri et al. Vol. 13555. LNCS. Springer, Cham, Sept. 2022, pp. 295–316. DOI: 10.1007/978-3-031-17146-8_15.

[39] John Kelsey et al. "Secure Applications of Low-Entropy Keys". In: *ISW'97*. Ed. by Eiji Okamoto, George I. Davida, and Masahiro Mambo. Vol. 1396. LNCS. Springer, Berlin, Heidelberg, Sept. 1998, pp. 121–134. DOI: 10.1007/bfb0030415.

[40] Ralf Küsters and Max Tuengerthal. "Composition theorems without pre-established session identifiers". In: *ACM CCS 2011*. Ed. by Yan Chen, George Danezis, and Vitaly Shmatikov. ACM Press, Oct. 2011, pp. 41–50. DOI: 10.1145/2046707.2046715.

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

[42] Russell W. F. Lai et al. "Phoenix: Rebirth of a Cryptographic Password-Hardening Service". In: *USENIX Security 2017*. Ed. by Engin Kirda and Thomas Ristenpart. USENIX Association, Aug. 2017, pp. 899–916. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lai.

[43] Russell W. F. Lai et al. "Simple Password-Hardened Encryption Services". In: *USENIX Security 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, Aug. 2018, pp. 1405–1421. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lai.

[44] Anja Lehmann and Björn Tackmann. "Updatable Encryption with Post-Compromise Security". In: *EUROCRYPT 2018, Part III*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10822. LNCS. Springer, Cham, 2018, pp. 685–716. DOI: 10.1007/978-3-319-78372-7_22.

[45] Ueli Maurer. "Constructive Cryptography - A Primer (Invited Paper)". In: *FC 2010*. Ed. by Radu Sion. Vol. 6052. LNCS. Springer, Berlin, Heidelberg, Jan. 2010, p. 1. DOI: 10.1007/978-3-642-14577-3_1.

[46] Alec Muffett. *Facebook Password Hashing & Authentication*. [Online; accessed 21-February-2023]. URL: https://www.youtube.com/watch?v=7dPRFoKteIU.

[47] PCI. *Requirements and security assessment procedures*. http://pcicompliance.stanford.edu/sites/default/files/pci_dss_v3-2.pdf. [Online, accessed 09/12/17]. 2016.

[48] Jonas Schneider et al. "Efficient Cryptographic Password Hardening Services from Partially Oblivious Commitments". In: *ACM CCS 2016*. Ed. by Edgar R. Weippl et al. ACM Press, Oct. 2016, pp. 1192–1203. DOI: 10.1145/2976749.2978375.

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

[50] SeongHan Shin, Kazukuni Kobara, and Hideki Imai. "A Secure Threshold Anonymous Password-Authenticated Key Exchange Protocol". In: *IWSEC 07*. Ed. by Atsuko Miyaji, Hiroaki Kikuchi, and Kai Rannenberg. Vol. 4752. LNCS. Springer, Berlin, Heidelberg, Oct. 2007, pp. 444–458. DOI: 10.1007/978-3-540-75651-4_30.

[51] Shouhuai Xu and Ravi S. Sandhu. "Two Efficient and Provably Secure Schemes for Server-Assisted Threshold Signatures". In: *CT-RSA 2003*. Ed. by Marc Joye. Vol. 2612. LNCS. Springer, Berlin, Heidelberg, Apr. 2003, pp. 355–372. DOI: 10.1007/3-540-36563-X_25.

[52] Xun Yi, Raylin Tso, and Eiji Okamoto. "ID-Based Group Password-Authenticated Key Exchange". In: *IWSEC 09*. Ed. by Tsuyoshi Takagi and Masahiro Mambo. Vol. 5824. LNCS. Springer, Berlin, Heidelberg, Oct. 2009, pp. 192–211. DOI: 10.1007/978-3-642-04846-3_13.

[53] Xun Yi et al. "Practical Threshold Password-Authenticated Secret Sharing Protocol". In: *ESORICS 2015, Part I*. Ed. by Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl. Vol. 9326. LNCS. Springer, Cham, Sept. 2015, pp. 347–365. DOI: 10.1007/978-3-319-24174-6_18.

Supplementary Material

A Full Details: An Ideal Functionality for TPHE

For completeness and reference purposes, we additionally provide full formal specifications of the ideal TPHE functionality $\mathcal{F}_{\text{PHE}}$ (Figures 8 to 10 in this appendix) and of the protocol model used in our security proof (Appendix B) cast into a concrete model for universal composability. These full specifications spell out all technical details, including minor ones such as the exact implementation of the corruption model, that are typically left implicit as they are not required to understand the overarching ideas of definitions and constructions.

Our specifications are given in the iUC model [13], a highly general model for universal composability by Camenisch et al. The notation used in these full specifications is mostly self explanatory. For interested readers we also provide an overview of iUC and its notation in Appendices D and E. While the choice of a model is to some degree a matter of preference, we chose iUC for our formal specifications over Canetti's UC model [18, 17] as it provides several useful features including a much simpler runtime notion that reduces clutter in such full specifications; we refer to [13] for a detailed comparison.

We note that these full formal specification are entirely optional supplementary material only intended for reference purposes and are not required to understand our paper. Importantly, the main body and our results, including the definition of our ideal TPHE functionality $\mathcal{F}_{\text{PHE}}$ in Figure 1, are independent of a specific model for universal composability.

Description of the protocol $\mathcal{F}_{\text{PHE}} = (\mathcal{S}, \mathcal{R})$:

Participating roles: $\{\mathcal{S}, \mathcal{R}\}$
Corruption model: custom
Protocol parameters:
- $\mathsf{n} \in \mathbb{N}$. {*Number of ratelimiters.*
- $1 \leq \mathsf{t} \leq \mathsf{n}$. {*Threshold of ratelimiters needed for decryption.*

Description of $M_{\mathcal{S},\mathcal{R}}$:

Implemented role(s): $\{\mathcal{S}, \mathcal{R}\}$
Internal state:
- $\mathsf{corrLog} = \emptyset$. {*Chronologically ordered sequence of corruption updates. Each entry is of the form $(entity, b)$ denoting an entity $= (pid, sid, role)$ whose corruption status has been changed to $b \in \{\texttt{true}, \texttt{false}\}$.*
- $\mathsf{storageHistory} : \{0,1\}^* \times \mathbb{N} \to (\{0,1\}^*)^2 \cup \{\bot\}$. {*For each id contains the chronological list of all pairs (pw, m) encrypted at some point (referenced by a counter $\in \mathbb{N}$). Initially all entries are $\bot$.*
- $\mathsf{correctMessageIDs} \subset \{0,1\}^* = \emptyset$. {*Message IDs that are guaranteed to decrypt to the correct plaintext.*
- $\mathsf{retrieveCounter} \in \mathbb{N} = 0$. {*Counter to reference different* Dec *requests.*
- $\mathsf{reqQueue}_{\mathsf{Dec}} : \mathbb{N} \to (\{0,1\}^*)^3 \cup \{\bot\}$. {*Storage for* Dec *requests, referenced by a unique number. Initially $\bot$.*
- $\mathsf{retrieveRate} : (\{0,1\}^*)^2 \to \mathbb{N}$ {*Maps a pid of a ratelimiter and an id to the number of times the ratelimiter can still be used to decrypt the message belonging to id. Initially 0.*
- $\mathsf{registeredGuesses} \subset \{0,1\}^* \times \mathbb{N}$ {*Multiset. Stores pairs (id, i) to indicate that the adversary may guess (once per pair) a password for the i-th ciphertext with ID id. Initially $\emptyset$.*

CheckID$(pid, sid, role)$**:**
Check that $pid = 0 \wedge role = \mathcal{S} \vee pid \in \{1, \ldots, \mathsf{n}\} \wedge role = \mathcal{R}$.
If this check fails, output `reject`.
Otherwise, accept all entities with the same SID.[a]

EntityInitialization:
send responsively `InitialCorruption`? **to** `NET`;
wait for (`InitialCorruption`?, b) **s.t.** $b \in \{\texttt{true}, \texttt{false}\}$. {*Allow adversary to choose initial corruption status. Response must be immediate, i.e., no other interactions with the functionality may be performed in the meantime.*
if $b = \texttt{true}$:
Add $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ to the set CorruptionSet.[b]

Main:
// Meta request: Allow environment to check correct simulation of corruption and decorruption//

recv `CorruptionLog`? **from** `I/O`: {*Provide full chronological history of all corruption changes.*
reply (`CorruptionLog`, corrLog).
// Storing and Retrieving //

recv $(\texttt{Store}, id, pw, m)$ **from** `I/O` **to** $(\_, \_, \mathcal{S})$:
if $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}}) \notin$ CorruptionSet:
For $i \in \mathbb{N}$ minimal such that $\mathsf{storageHistory}[id][i] = \bot$ set $\mathsf{storageHistory}[id][i] \leftarrow (pw, m)$.
Add id to the set correctMessageIDs. {*Ciphertexts encrypted at an honest server are guaranteed to decrypt to the correct plaintext (as long as the server remains honest).*
send $(\texttt{Store}, id, |m|)$ **to** `NET`. {*Reveal only ID and length of the message to the adversary if server is honest. Can be relaxed by also leaking $|pw|$.*
else:
send $(\texttt{Store}, id, pw, m)$ **to** `NET`. {*Reveal all information to the adversary if server is corrupted*

recv $(\texttt{Retrieve}, id, pw')$ **from** `I/O` **to** $(0, \_, \mathcal{S})$: {*Try to retrieve the message stored under ID id using password pw'.*
$\mathsf{retrieveCounter} = \mathsf{retrieveCounter} + 1$
$\mathsf{reqQueue}_{\mathsf{Dec}}[\mathsf{retrieveCounter}] \leftarrow (id, pw', \mathsf{entity}_{\mathsf{call}})$.
if $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}}) \notin$ CorruptionSet:
send $(\texttt{Retrieve}, \mathsf{retrieveCounter}, id)$ **to** `NET`. {*If server is honest, give reference to* Dec *request and ID to adversary. Can be relaxed by also leaking $|pw|$.*
else:
send $(\texttt{Retrieve}, \mathsf{retrieveCounter}, id, pw', \mathsf{entity}_{\mathsf{call}})$ **to** `NET`. {*If server is corrupted, give full input to adversary.*

Continue with Figure 10.

[a] That is, one instance of this ideal functionality models one protocol session with $\mathsf{n} + 1$ participants, namely, the server with PID 0 and ratelimiters with PIDs $1, \ldots, \mathsf{n}$.
[b] CorruptionSet is the set of all corrupted entities. This variable is already defined by the iUC framework and thus not explicitly listed in **Internal state**.

Fig. 8: The ideal password hardened encryption protocol $\mathcal{F}_{\mathsf{TPHE}}$ (Part 1).

Description of $M_{\mathcal{S},\mathcal{R}}$ (continued):

Main:

recv $(\texttt{FinishRetrieve}, \mathit{reqid}, \mathit{RLset}, b, (\mathit{origPlaintext}, i, pw_{adv}, m_{adv}))$ **from** NET **to** $(\_, \_, \mathcal{S})$
s.t. $(0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S}) \notin \mathsf{CorruptionSet} \wedge \mathsf{reqQueue}_{\mathsf{Dec}}[\mathit{reqid}] \neq \bot \wedge \mathit{RLset} \subseteq \{1, \ldots, \mathsf{n}\}$
$\wedge\ b \in \{\texttt{success}, \texttt{failed}\}\ \wedge\ \mathit{origPlaintext} \in \{\texttt{true}, \texttt{false}\}\ \wedge\ i \in \mathbb{N}$:
{*Finish the* Dec *request stored under ID reqid using (honest) ratelimiters RLset.*

$(id, pw', \mathit{caller}) \leftarrow \mathsf{reqQueue}_{\mathsf{Dec}}[\mathit{reqid}]$ and $\mathsf{reqQueue}_{\mathsf{Dec}}[\mathit{reqid}] \leftarrow \bot$.
if $b = \texttt{failed}$: **send** $(\texttt{Retrieve}, id, \bot)$ **to** *caller*
{*Adversary can always let decryption fail.*
else:

For each $rl \in \mathit{RLset}$ check that $(rl, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R}) \notin \mathsf{CorruptionSet}$ and $\mathsf{retrieveRate}[rl, id] \geq 1$. Furthermore, check that $|\mathit{RLset}| + n_c \geq \mathsf{t}$, where n_c is the number of currently corrupted ratelimiters. If $\mathit{origPlaintext} = \texttt{true}$, then additionally verify that $\mathsf{storageHistory}[id, i] \neq \bot$.
{*Verify that enough ratelimiters are willing to help with decryption. If attacker wants to decrypt a previous plaintext, verify that this actually exists.*

if any of the above checks fail: **reply** $(\texttt{FinishRetrieve}, \texttt{Error})$. {*Return error to adversary.*
else:
For each $rl \in \mathit{RLset}$: $\mathsf{retrieveRate}[rl, id] \leftarrow \mathsf{retrieveRate}[rl, id] - 1$.
if $id \in \mathsf{correctMessageIDs}$:
$(pw, m) \leftarrow \mathsf{storageHistory}[id, j]$, where $j \in \mathbb{N}$ is the position of the most recent ciphertext encrypted for ID id, i.e., j is maximal such that $\mathsf{storageHistory}[id, j] \neq \bot$
if $pw = pw'$: **send** $(\texttt{Retrieve}, id, m)$ **to** *caller*.
else: send $(\texttt{Retrieve}, id, \bot)$ **to** *caller*.
{*Decryption of IDs with correctness guarantees must be for the most recent ciphertext and can only yield a correct plaintext.*

else if $\mathit{origPlaintext} = \texttt{true}$:
$(pw, m) \leftarrow \mathsf{storageHistory}[id, i]$
if $pw = pw'$: **send** $(\texttt{Retrieve}, id, m)$ **to** *caller*.
else: send $(\texttt{Retrieve}, id, \bot)$ **to** *caller*.
{*For ids without correctness guarantees, adversary can choose to decrypt to a previously stored (and unknown to him) plaintext.*

else:
if $pw_{adv} = pw'$: **send** $(\texttt{Retrieve}, id, m_{adv})$ **to** *caller*.
else: send $(\texttt{Retrieve}, id, \bot)$ **to** *caller*.
{*For ids without correctness guarantees, adversary can also choose to decrypt to a message and under a password of his choice (but without learning pw').*

recv $(\texttt{CorruptedRetrieve}, id, \mathit{out}, \mathit{caller})$ **from** NET **to** $(\_, \_, \mathcal{S})$ **s.t.** $(0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S}) \in \mathsf{CorruptionSet}$ and *caller* is a higher-level protocol:
send $(\texttt{Retrieve}, id, \mathit{out})$ **to** *caller*. {*If server is corrupted, adversary can send a* Dec *output out for message ID id to caller.*

// Ratelimiters //

recv $(\texttt{HelpRetrieve}, id)$ **from** I/O **to** $(\_, \_, \mathcal{R})$:
$\mathsf{retrieveRate}[\mathsf{pid}_{\mathsf{cur}}, id] \leftarrow \mathsf{retrieveRate}[\mathsf{pid}_{\mathsf{cur}}, id] + 1$.
send responsively $(\texttt{GetRetrieveRate}, \mathsf{retrieveRate}[\mathsf{pid}_{\mathsf{cur}}, id])$ **to** NET; **wait for** _ {*Leak retrieve rate*
reply $(\texttt{HelpRetrieve}, \texttt{OK})$

// Corruption and decorruption (due to key rotation) //

recv $(\texttt{ChangeCorruption}, \mathit{corrupt})$ **from** NET **s.t.** $\mathit{corrupt} \in \{\texttt{true}, \texttt{false}\}$:
Append $(\mathsf{entity}_{\mathsf{cur}}, \mathit{corrupt})$ at the end of corrLog.
if $\mathsf{role}_{\mathsf{cur}} = \mathcal{S}$: Set all entries in $\mathsf{reqQueue}_{\mathsf{Dec}}$ to $\bot$.
if $\mathit{corrupt} = \texttt{true}$: Add $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ to the set CorruptionSet.
else: Remove $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ from the set CorruptionSet (if it was in this set).
If $\mathsf{role}_{\mathsf{cur}} = \mathcal{S}$ and $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ was previously in CorruptionSet, then set $\mathsf{correctMessageIDs} \leftarrow \emptyset$.
If $\mathsf{role}_{\mathsf{cur}} = \mathcal{R}$, then set $\mathsf{retrieveRate}[\mathsf{pid}_{\mathsf{cur}}, id] \leftarrow 0$ for all IDs id.
{*No correctness guarantees for ciphertexts that were under adversarial control*
reply $(\texttt{ChangeCorruption}, \texttt{OK})$

Fig. 9: The ideal password hardened encryption protocol $\mathcal{F}_{\mathsf{TPHE}}$ (Part 2).

Description of $M_{\mathcal{S},\mathcal{R}}$ (continued):

// Password guessing //

recv $(\texttt{PwGuessStart}, id, i, RLset)$ **from** $\texttt{NET}$ **to** $(\_, \_, \mathcal{S})$
s.t. $i \in \mathbb{N} \ \wedge \ RLset \subseteq \{1, \ldots, \mathsf{n}\} \ \wedge \ \mathsf{storageHistory}[id][i] \neq \bot$:

{Allow adversary to perform a password guess for the i-th message that was encrypted under ID id with the help of honest ratelimiters specified in RLset. In this first step, the adversary indicates his intention to guess a password and the functionality verifies that he is indeed able to do so at this moment.

Verify that for each $j \in RLset$ it holds $(j, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R}) \notin \mathsf{CorruptionSet}$ and $\mathsf{retrieveRate}[j, id] \geq 1$. Further verify that at least one of the following two cases applies:

- At some point in time the server as well as at least t ratelimiters were corrupted simultaneously.

  {No protection against offline password guessing can be given in this case.

- We currently have $(0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S}) \in \mathsf{CorruptionSet}$ and it holds that $|RLset| + n_c \geq \mathsf{t}$ where n_c is the number of currently corrupted ratelimiters.

  {Password guessing is otherwise restricted to be online, i.e., only possible if the attacker currently has access to the server and at least t ratelimiters are helping.

if verification fails:
  reply $(\texttt{PwGuessStart}, \texttt{Error})$ *{Notify adversary that the current request did not succeed.*
else: *{One password guess is possible. Store this permission for the next step.*
  For each $rl \in RLset$: $\mathsf{retrieveRate}[rl, id] \leftarrow \mathsf{retrieveRate}[rl, id] - 1$.
  Add (id, i) to $\mathsf{registeredGuesses}$. *{Note that now there might be several occurrences of (id, i) in this multiset.*
  reply $(\texttt{PwGuessStart}, \texttt{OK})$

recv $(\texttt{PwGuessFinish}, id, i, pw')$ **from** $\texttt{NET}$ **to** $(\_, \_, \mathcal{S})$ **s.t.** $i \in \mathbb{N} \ \wedge \ (id, i) \in \mathsf{registeredGuesses}$:

{Second step: After permission has been checked, the adversary can provide a password guess pw'. If the guess is correct, he learns the plaintext that was encrypted in the i-th ciphertext with ID id.

Remove one occurrence of (id, i) from $\mathsf{registeredGuesses}$.
$(pw, m) \leftarrow \mathsf{storageHistory}[id][i]$.
if $pw = pw'$:
  reply $(\texttt{PwGuessFinish}, \texttt{Correct}, m)$
else:
  reply $(\texttt{PwGuessFinish}, \texttt{Incorrect})$

Fig. 10: The ideal password hardened encryption protocol $\mathcal{F}_{\mathsf{TPHE}}$ (Part 3).

B Full Details: Security Model for the TPHE Scheme

In this section, we provide a low-level explanation of the UCPY model $\mathcal{P}_{\text{PHE}}$. As depicted in Figure 5, $\mathcal{P}_{\text{PHE}}$ is defined as the protocol $\mathcal{P}_{\text{PHE}} = (\mathcal{P}^{\mathcal{S}}_{\text{PHE}}, \mathcal{P}^{\mathcal{R}}_{\text{PHE}} \mid \mathcal{F}^{1}_{\text{ro}}, \mathcal{F}^{2}_{\text{ro}}, \mathcal{F}^{\mathsf{OTP}}_{\text{ro}}, \mathcal{F}^{\mathsf{MAC}}_{\text{ro}}, \mathcal{F}^{\mathsf{N}}_{\text{ro}}, \mathcal{F}_{\text{nizk}}, \mathcal{F}_{\text{init\&rotateKey}}, \mathcal{F}_{\text{auth}})$. We provide a full formal specification of the different ITMs in Figure 11 to Figure 23.

In what follows, we start with a high-level explanation of the different ITMs and their role in $\mathcal{P}_{\text{PHE}}$ and modeling aspects of $\mathcal{P}_{\text{PHE}}$. Then, we explain the various ITMs in detail.

B.1 Model Overview

Here, we explain several high-level aspects of the UCPY model $\mathcal{P}_{\text{PHE}}$.

Model Components In $\mathcal{P}_{\text{PHE}}$, the main components of UCPY are $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$ and $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$, modeling the server of (a) UCPY (instance) and the connected ratelimiters. Both ITMs closely follow the specifications of the server, resp. ratelimiters, of UCPY during de- and encryption requests. However, key rotation is orchestrated by $\mathcal{F}_{\text{init\&rotateKey}}$, which models three different variants of a key rotation:

1. Honest key rotation where all parties honestly execute the code of the key rotation in a synchronous network without message dropping and no network latency. $\mathcal{A}$ may solely change the corruption status of parties during honest key rotation. However, *honest* key rotation ensures that all parties, server, and ratelimiters, honestly execute the key rotation protocol.
2. Dishonest key rotation with an honest server, where the server honestly runs a key rotation, but ratelimiters can be malicious. During this type of key rotation, the honest server declines de- and encryption requests until the key rotation has finished. $\mathcal{A}$ controls the fully asynchronous network (which additionally provides message secrecy for honest parties). $\mathcal{A}$ decided whether corrupted ratelimiters participate in the key rotation and which public key share they provide to $\mathcal{S}$. Honest ratelimiters execute key rotation according to the protocol. However, $\mathcal{A}$ may decide whether honest ratelimiters get informed about the ongoing key rotation. $\mathcal{A}$ also decides whether the network delivers the honest ratelimiter's public key shares to the server.
3. During dishonest key rotation with a malicious server, solely honest ratelimiters follow the key rotation protocol. $\mathcal{A}$ decides whether/which honest ratelimiters are involved in the key rotation.

We emphasize that our corruption model is round-based (see below). Corruption and decorruption are only possible during honest key rotation.

$\mathcal{F}_{\text{init\&rotateKey}}$ initializes one session of the UCPY scheme. It honestly generates the necessary cryptographic material, resp. parameters, according to the UCPY specification. That is, $\mathcal{F}_{\text{init\&rotateKey}}$ generates the secret server key $k_{\mathcal{S}}$, computes the shares for the ratelimiters (including the public key shares), handles initial corruption of server and ratelimiters, and distributes parameters and corruption state among server and ratelimiters. Additionally, $\mathcal{F}_{\text{init\&rotateKey}}$ sets the codomain for the random oracles.

In $\mathcal{P}_{\text{PHE}}$, we further use five different random oracles $\mathcal{F}^{1}_{\text{ro}}$, $\mathcal{F}^{2}_{\text{ro}}$, $\mathcal{F}^{\mathsf{OTP}}_{\text{ro}}$, $\mathcal{F}^{\mathsf{MAC}}_{\text{ro}}$, and $\mathcal{F}^{\mathsf{N}}_{\text{ro}}$ to model hash generation during the five different purposes for hashing in UCPY. The random oracles slightly deviate from common definitions of random oracles in two ways: (i) The random oracles do not randomly choose values from $\{0,1\}^{\eta}$, they randomly select values from a defined codomain (which is $\{0,1\}^{\eta}$, $\mathbb{G}_1$, or $\mathbb{G}_2$ in UCPY). (ii) $\mathcal{F}^{\mathsf{OTP}}_{\text{ro}}$ allows to requests hashes to be bit strings of a desired length.

The other ideal subroutines in the hybrid $\mathcal{P}_{\text{PHE}}$ model are standard. (i) The ratelimiters use the (ideal) non-interactive zero-knowledge functionality $\mathcal{F}_{\text{nizk}}$ [26, 27] to generate ZKPs and the server uses $\mathcal{F}_{\text{nizk}}$ to verify the ratelimiter's proofs, and (ii) $\mathcal{F}_{\text{auth}}$ [40] provides an authenticated channel among the protocol parties. $\mathcal{F}_{\text{auth}}$ does neither provide message secrecy nor eventual message delivery.

We note that $\mathcal{F}_{\text{init\&rotateKey}}$, as well as the other subroutines, are private and thus not directly accessible by $\mathcal{E}$.

Corruption in $\mathcal{P}_{\text{PHE}}$ $\mathcal{P}_{\text{PHE}}$, similarly to $\mathcal{F}_{\text{PHE}}$, does not implement a standard corruption model. As already skimmed above, $\mathcal{P}_{\text{PHE}}$ allows a round-wise de-/corruption. During each honest key rotation, $\mathcal{A}$ determines for each "round" or "epoch" which parties are corrupted (and which honestly follow the protocol). In particular, $\mathcal{A}$ may corrupt the server and up to $\mathsf{t}-1$ ratelimiters or not the server and any amount of ratelimiters. As common, $\mathcal{A}$ has full control over corrupted parties. On decorruption, we require $\mathcal{A}$ to provide values for the relevant state in the protocol, i.e., the public key for validating the ratelimiters public key shares, and the complete stored (encrypted) database for the server. For a decorrupted ratelimiter, $\mathcal{A}$ determines its secret key (share), the public key (share), the nonce counter, and a set of nonces. This models that $\mathcal{A}$ may manipulate the physical data of servers and ratelimiters during corruption.

The different key rotation variants, as introduced above, essentially cover the possible key rotation scenarios. An honest key rotation models that all parties restore to the original code of UCPY and run the protocol honestly but maybe on manipulated data. After the key rotation, $\mathcal{A}$ may corrupt entities for the current round. In $\mathcal{P}_{\text{PHE}}$, we use the number of honest key rotations as a time notion. During honest key rotation, $\mathcal{F}_{\text{init\&rotateKey}}$ ensures that all messages are delivered to their recipients without any network latency.

The two dishonest key rotations model the cases without additional security guarantees (no message delivery guarantee and no guarantee of correct execution of the key rotation protocol). If the server is honest, $\mathcal{F}_{\text{init\&rotateKey}}$ ensures that the server honestly executes the protocol (and stops other processing).

In what follows, we now provide a detailed description of the different ITMs in $\mathcal{P}_{\text{PHE}}$.

B.2 The Server ITM $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$

As already mentioned, the server ITM $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ (cf. Figure 11 to 14) closely follows the protocol description of a server according to the UCPY specification. Per UCPY instance, there is at most one instance of $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ (identified by $(0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S})$). In more detail, $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ works as follows:

Initialization. When an entity invokes an instance of $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ for the first time, $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ registers at the adequate $\mathcal{F}_{\text{auth}}$ session for communicating via authenticated channels with the ratelimiters. It then queries $\mathcal{F}_{\text{init\&rotateKey}}$ for initialization. $\mathcal{F}_{\text{init\&rotateKey}}$ provides the necessary UCPY parameters to $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$, i.e., $q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t$, and g_t including $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$'s corruption state. $\mathcal{F}_{\text{init\&rotateKey}}$ also executes the "intentional" server part for initialization on behalf of $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ and provides the server's secret key share sk, pk, and the ratelimiter share $\mathsf{pk}_1, \ldots, \mathsf{pk}_\mathsf{n}$ to $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$. Thereupon, $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ computes the possible reconstruction vectors (for Shamir's secret sharing) and verifies which subsets of ratelimiters allow reconstructing the public key pk. If $\mathcal{A}$ instructed $\mathcal{F}_{\text{init\&rotateKey}}$ that $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ is corrupted, $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ forwards its full internal state to $\mathcal{A}$ (and acts as message forwarder for $\mathcal{A}$ until decorruption.)

Message Processing. We use message processing to ensure that $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ processes some received messages before entering the common main part of the ITM definition.

corrLog is part of the definition of $\mathcal{P}_{\text{PHE}}$'s custom corruption modeling. In this case, $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ queries $\mathcal{F}_{\text{init\&rotateKey}}$, which stores the corruption history of all parties and forwards the output to the requestor.

Stop processing during key rotation: During a dishonest key rotation with honest server, $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ declines all enc- and decryption requests.

`RotateKey` finalizes the honest key rotation process which started at $\mathcal{F}_{\text{init\&rotateKey}}$. With this call, $\mathcal{F}_{\text{init\&rotateKey}}$ provides whether $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ is corrupted after the key rotation or not, the new server secret key $\mathsf{sk}_{\mathcal{S}}$ and the correct public key pk including the new pk shares $\mathsf{pk}_1, \ldots, \mathsf{pk}_\mathsf{n}$. It also provides (new) nonces for each ratelimiter, $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ should use during the execution of UCPY. When $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ was corrupted before the ongoing key rotation, it requires $\mathcal{A}$ to provide the (possibly manipulated) public key pk and $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ (potentially manipulated) database *storage*.
Based on the (potentially manipulated) data, $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ continues the key rotation. It computes the reconstruction vectors for Shamir's secret sharing and stores them. Afterward, it checks which subsets of ratelimiters can support $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ during enc- and decryption. If $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ is corrupted after key rotation, $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ records itself to be corrupted and forwards its internal state (including sk, pk, the public key shares, nonces, and $\mathsf{storage}$) to *Asys*. Otherwise, it solely leaks pk, the public key shares, and the ratelimiter sets $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ will use for enc- and decryption until the next key rotation.

Message Forwarding: If $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ is corrupted, it acts (mainly) as a message forwarder to $\mathcal{A}$. The code in this section specifies this behavior. Please note that message preprocessing code blocks before this one are *not* forwarded to $\mathcal{A}$, and instead, $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ executes its "honest" code if, e.g., a party calls via I/O `corrLog`, `RotateKey`, ...

Main.

`Store` models an encryption request from a higher-level entity. The code closely follows the definition of the first call of the server in Figure 2. Additionally, we allow $\mathcal{A}$ to decide which subset of ratelimiters $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ will ask for support during encryption. $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ marks used nonces as consumed and stores the encryption request (including all necessary details) for asynchronous processing. To send the necessary messages to the involved ratelimiters, $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ uses the authenticated channel $\mathcal{F}_{\text{auth}}$.

Receiving `FinalizeEnc` from a ratelimiter via $\mathcal{F}_{\text{auth}}$: When receiving $(\texttt{Received}, (rl, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R}), (\texttt{FinalizeEnc}, \ldots))$ from $\mathcal{F}_{\text{auth}}$ send by the ratelimiter rl, $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ mainly executes the second part of the UCPY protocol as depicted in Figure 2. In more detail, it verifies that the ratelimiters are involved in the particular encryption request id request. $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ stores the response if the response is valid, i.e., when ZKP proofs are valid.
$\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ keeps collecting responses until *all* ratelimiters involved in the encryption request have validly answered the request. In this case, $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ finishes the encryption request (cf. Figure 2) and locally stores the encrypted data in its database $\mathsf{storage}$.

`Retrieve` models a decryption request from a higher-level entity. Again, the code closely follows the specification of the UCPY protocol (cf. Figure 3). $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ processes the request if there exists a matching entry in the database. Similarly to encryption requests, $\mathcal{A}$ determines the set of ratelimiters that will support the decryption. $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ stores the necessary data for asynchronous processing and sends the appropriate message for each involved ratelimiter via $\mathcal{F}_{\text{auth}}$ to the ratelimiter.

Receiving `FinalizeDec` from a ratelimiter via $\mathcal{F}_{\text{auth}}$: This
case models the second part of the server specification of UCPY's decryption protocol. The processing in `FinalizeDec` works similarly to the one in `FinalizeEnc`.

`GetNonces` allows $\mathcal{A}$ to trigger that $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$ queries a dedicated ratelimiter pid to provide new nonces to $\mathcal{P}_{\text{PHE}}^{\mathcal{S}}$. The message is sent via $\mathcal{F}_{\text{auth}}$.

Receiving `GetNonces` from a ratelimiter via $\mathcal{F}_{\text{auth}}$: This models the ratelimiter response to the aforementioned request. $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$ stores the nonces.

`stealDB` allows $\mathcal{A}$ to get a copy of $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$'s entire database $\mathsf{storage}$.

`RotationOngoing` is used in the case of a dishonest key rotation with the honest server. With this message, $\mathcal{F}_{\text{init\&rotateKey}}$ indicates to $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$ to stop the execution of de- and encryption requests until key rotation finishes.

B.3 The Ratelimter ITM $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$

The ratelimiter ITM $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ (cf. Figure 15 to 16) closely follows the protocol description of a ratelimiter according to the UCPY specification. In a UCPY session, there is (at most) one instance of $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$ per ratelimiter. That means., there are typically n instances of $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ identified by $(i, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R}), i \in [1..\mathsf{n}]$. In more detail, $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ works as follows:

Initialization. Initialization in $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ works conceptually similar to the $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$ initialization. Firstly, $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ registers at the adequate $\mathcal{F}_{\text{auth}}$ session for communicating with the server via an authenticated channel. It then queries $\mathcal{F}_{\text{init\&rotateKey}}$ for initialization. $\mathcal{F}_{\text{init\&rotateKey}}$ provides the necessary UCPY parameters to $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$, i.e., $q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t$, and g_t including $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$'s corruption state. $\mathcal{F}_{\text{init\&rotateKey}}$ also provides the ratelimiter's secret key share sk and the connected public key (share) pk to $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$. If $\mathcal{A}$ instructed $\mathcal{F}_{\text{init\&rotateKey}}$ that $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ is corrupted, $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ forwards its full internal state to $\mathcal{A}$ (and acts as message forwarder for $\mathcal{A}$ until decorruption.)

Message Processing. Analogously to the server case, we use message processing to ensure that $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ processes some received messages before it enters the main part of the ITM definition.

corrLog works analogously to the server case (cf. Item "corrLog" above).

`RotateKey`**:** $\mathcal{F}_{\text{init\&rotateKey}}$ invokes this interface to allow the ratelimiter to participate in a key rotation. With this call, the functionality $\mathcal{F}_{\text{init\&rotateKey}}$ provides whether $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ is corrupted after the key rotation or not and the secret sharing of $\mathsf{sk}'_{\mathcal{S}} - \mathsf{sk}_{\mathcal{S}}$ as generated by $\mathcal{F}_{\text{init\&rotateKey}}$. If $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ was corrupted before the activation, $\mathcal{A}$ determines the current state of $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$, more specifically, $\mathcal{A}$ provides a secret key share sk, a public key (share) pk, the current nonce counter, and a set of nonces to $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ – all data may be manipulated by $\mathcal{A}$. $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ uses the input from $\mathcal{A}$ as its local state.
$\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ then computes its new secret key share and the connected public key (share) pk. If $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ is corrupted in the next time epoch, $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ leaks its full internal state, in particular sk and pk to $\mathcal{A}$. Otherwise, it generates a new *nonce*, stores it for later usage, and sends the public key (share) pk together with the *nonce* back to $\mathcal{F}_{\text{init\&rotateKey}}$.

Message Forwarding: Analogously to Item "Message Forwarding": if $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ is corrupted, it acts (mainly) as a message forwarder to $\mathcal{A}$.

Main.

Receiving `EncRequest` from the server via $\mathcal{F}_{\text{auth}}$:
When $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ receives an encryption request, it executes the UCPY protocol as specified in Figure 2. It sends the reply to the server via $\mathcal{F}_{\text{auth}}$.

Receiving `DecRequest` from the server via $\mathcal{F}_{\text{auth}}$:
Analogously, $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ handles decryption requests from the server (cf. Figure 3).

Receiving `GetNonces` from the server via $\mathcal{F}_{\text{auth}}$:
Upon receiving the request to generate new nonces from the server, $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ generates ng new nonces, stores them, and sends them back to the server (via $\mathcal{F}_{\text{auth}}$).

`HelpRetrieve` allows the environment to add "support tokens" for a message ID. $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ only supports a decryption, if there are sufficient "support tokens" for the particular message ID.

`GetRetrieveRate` allows $\mathcal{A}$ to query for the $\mathsf{retrieveRate}$ for a specific ID.

B.4 The Initialization and Key Rotation Functionality $\mathcal{F}_{\text{init\&rotateKey}}$

The initialization and key rotation ITM $\mathcal{F}_{\text{init\&rotateKey}}$ (cf. Figure 17 to 19) takes care of the initialization of the UCPY scheme, models several assumptions for analyzing the UCPY protocol, and is used to orchestrate key rotations. There is one instance of $\mathcal{F}_{\text{init\&rotateKey}}$ per UCPY session.

$\mathcal{F}_{\text{init\&rotateKey}}$ is incorruptible by definition.

Initialization. During initialization, $\mathcal{F}_{\text{init\&rotateKey}}$ honestly generates the parameters for the UCPY scheme and generates an overall secret key sk, the server secret key $\mathsf{sk}_{\mathcal{S}}$, and computes the overall ratelimiter secret key $\mathsf{sk}_{\mathcal{R}} \leftarrow \mathsf{sk} - \mathsf{sk}_{\mathcal{S}}$. Furthermore, it generates a sharing of $\mathsf{sk}_{\mathcal{R}}$ (one share per ratelimiter). The ITM also queries $\mathcal{A}$ for the initial corruption state of the involved server and ratelimiters and also leaks UCPY's public parameters to $\mathcal{A}$. It stores the corruption state in a log to match the interfaces of $\mathcal{F}_{\text{PHE}}$.

Main.

`InitMe` **Messages:** When an $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$ or $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ ITM initializes, it calls $\mathcal{F}_{\text{init\&rotateKey}}$ via $\texttt{InitMe}_{\mathcal{S}}$, resp. $\texttt{InitMe}_{\mathcal{R}}$. $\mathcal{F}_{\text{init\&rotateKey}}$ provides the necessary initial parameters to the ITM including its initial corruption status.

`getCodomain`**:** When a random oracle $\mathcal{F}^{i}_{\text{ro}}, i \in \{1, 2, \mathsf{MAC}, \mathsf{N}\}$ initializes it queries $\mathcal{F}_{\text{init\&rotateKey}}$ for the codomain of the hash function, i.e., the set of values from which $\mathcal{F}^{i}_{\text{ro}}$ randomly samples its outputs.

`RotateKey` models a honest key rotation triggered by $\mathcal{A}$. In this case, $\mathcal{A}$ provides the corruption state of the server and ratelimiters for the next epoch. $\mathcal{F}_{\text{init\&rotateKey}}$ records the corruption status in its log. It then executes the (honest) key rotation code of the server (cf. Figure 4) and samples an α to compute the new server secret key $\mathsf{sk}_{\mathcal{S}} \leftarrow \mathsf{sk}_{\mathcal{S}} + \alpha$. It generates a secret sharing of α (one share $s_i, i \in [1..\mathsf{n}]$ per ratelimiter) such that every ratelimiter obtains its new secret key by subtracting its share of α from its secret key. It calls all ratelimiters (modeling a secret channel without latency) and provides them with their s_i and their upcoming corruption state. Each ratelimiter returns a public key (share) pk_i and a $nonce_i$. After $\mathcal{F}_{\text{init\&rotateKey}}$ has collected this data, it forwards the data to $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$ including its upcoming corruption status.

$\texttt{RotateKey}^{\mathcal{S}}_{\mathcal{A}}$**:** models that in an epoch with a corrupted server, $\mathcal{A}$ triggers (for some ratelimiters) a key rotation. $\mathcal{F}_{\text{init\&rotateKey}}$ acts in this case as message forwarder and allows $\mathcal{A}$ to send a (manipulated) share s to an (honest) ratelimiter of his choice. $\mathcal{F}_{\text{init\&rotateKey}}$ returns the answer of the ratelimiter to $\mathcal{A}$.

$\texttt{RotateKey}_{\mathcal{A}}$**:** models that in an epoch with honest server. $\mathcal{A}$ triggers a key rotation but without changing the corruption status of parties. Also, in this variant of the key rotation, $\mathcal{A}$ fully controls the network and also corrupted ratelimiters (which is different from the honest key rotation `RotateKey`). Firstly, $\mathcal{F}_{\text{init\&rotateKey}}$ informs $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$ that a key rotation is ongoing, and he should stop executing enc- and decryption requests. Similar to `RotateKey`, $\mathcal{F}_{\text{init\&rotateKey}}$ generates a secret sharing $s_i, i \in [1..\mathsf{n}]$ of α (one share per ratelimiter). $\mathcal{F}_{\text{init\&rotateKey}}$ leaks s_i for all corrupted ratelimiter to $\mathcal{A}$. Thereupon, $\mathcal{A}$ may instruct whether/which shares are delivered to honest ratelimiter. After an honest ratelimiter returns a public key share and a nonce, $\mathcal{A}$ may decide whether this message is dropped. When $\mathcal{A}$ indicates that the key rotation has finished (via `finishRotation`), $\mathcal{F}_{\text{init\&rotateKey}}$ sends the available public key shares and nonces to $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$.

corrLog provides the corruption log to $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$ and $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$.

B.5 Further Subroutines in $\mathcal{P}_{\text{PHE}}$

We omit a detailed specification of $\mathcal{F}^{1}_{\text{ro}}, \mathcal{F}^{2}_{\text{ro}}, \mathcal{F}^{\mathsf{OTP}}_{\text{ro}}, \mathcal{F}^{\mathsf{MAC}}_{\text{ro}}, \mathcal{F}^{\mathsf{N}}_{\text{ro}}$ (cf. Figures 21 and 22), $\mathcal{F}_{\text{nizk}}$ (Figure 23), and $\mathcal{F}_{\text{auth}}$ (Figure 20) as they mainly match there definitions from literature or already sufficiently detailed explained above.

Description of the protocol $\mathcal{P}^{\mathcal{S}}_{\text{PHE}} = (\mathcal{S})$:

Participating roles: $\{\mathcal{S}\}$
Corruption model: custom
Protocol parameters:
- $\mathsf{n} \in \mathbb{N}$. {*Number of ratelimiters.*
- $1 \leq \mathsf{t} \leq \mathsf{n}$. {*Threshold of ratelimiters needed for decryption.*

Description of $M_{\mathcal{S}}$:

Implemented role(s): $\{\mathcal{S}\}$
Subroutines:
$\mathcal{F}^{i}_{\text{ro}}$: `randomOracle`$(i \in \{1, 2, \mathsf{OTP}, \mathsf{MAC}, \mathsf{N}\})$, $\mathcal{F}_{\text{nizk}}$: `nizk`, $\mathcal{F}_{\text{init\&rotateKey}}$: `init`, $\mathcal{F}_{\text{auth}}$: `auth`
Internal state:
- $\mathsf{storage} : \{0,1\}^* \to (\{0,1\}^*)^3 \cup \{\bot\}$. {*Maps id to (ct, n_s, n_r). Initially $\bot$.*
- $\mathsf{reqQueue}_{\mathsf{Enc}} : \mathbb{N} \to (\{0,1\}^*)^2 \times \{0,1\}^{2\eta} \times \mathbb{N}$ {*Currently active encryption requests, id is mapped to (pw, m, r, n_s, n_r, T) where r is randomness, n_s a nonce, n_r ratelimiters' joint randomness, and $T \subset \mathbb{N}$ the set of involved ratelimiters*
- $\mathsf{q} \in \mathbb{N}$ {q*: the group size of used groups $\mathbb{G}_1, \mathbb{G}_2$, and $\mathbb{G}_t$*
- $\mathsf{nonces} : [1, \mathsf{n}] \to \{0,1\}^{\eta}$. {*Buffered nonces from each ratelimiter in a totally ordered set, initially all entries $\bot$*
- $\mathsf{sk} \in \{0,1\}^{\eta}$. {*$\mathcal{S}$'s secret key, initially set by $\mathcal{F}_{\text{init\&rotateKey}}$*
- $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t \subset \{0,1\}^*$, initially $\emptyset$. {*Used groups, initially set by $\mathcal{F}_{\text{init\&rotateKey}}$*
- $\mathsf{g}_t \in \{0,1\}^*$, resp. $\mathbb{G}_t$, initially ε. {*Generator of $\mathbb{G}_t$, initially set by $\mathcal{F}_{\text{init\&rotateKey}}$*
- $\mathsf{T}_{\mathsf{set}} \subset 2^{[1,\mathsf{n}]}$ {*Ratelimter sets to be accepts to help enc/dec, initially set by $\mathcal{F}_{\text{init\&rotateKey}}$*
- $\mathsf{pk}, \mathsf{pk}_1, \ldots, \mathsf{pk}$ all in $\mathbb{G}_t \cup \{\bot\}$ {*Public keys of ratelimiters per round, initially set by $\mathcal{F}_{\text{init\&rotateKey}}$*
- $\mathsf{limiterResponses}_{\mathsf{Enc}} : \mathbb{N} \times [1, \mathsf{n}] \to \{0,1\}^*$, initially all $\bot$. {*Buffer for ratelimiter responses during encryption. (id, rl) (rl means ratelimiter) map to u*
- $\mathsf{retrieveCounter} \in \mathbb{N} = 0$. {*Counter to reference different* Dec *requests.*
- $\mathsf{reqQueue}_{\mathsf{Dec}} : \mathbb{N} \to (\{0,1\}^*)^{10} \cup \{\bot\}$. {*Storage for* Dec *requests $(id, pw, r, n_s, n_r, T, p, caller)$, $caller = (pid, sid, role)$. Initially $\bot$.*
- $\mathsf{limiterResponses}_{\mathsf{Dec}} : \mathbb{N} \times [1, \mathsf{n}] \to \{0,1\}^*$, initially all $\bot$. {*Buffer for ratelimiter responses during decryption. (id, rl) map to u*
- $\mathsf{reconVectors} : \{A \mid A \in 2^{[1,\mathsf{n}]} \wedge |A| = t\} \to (\{0,1\})^{\mathsf{n}}$, initially all entries $\bot$ {*The set of initial reconstruction vectors*
- $\mathsf{rotationOngoing} \in \{\mathtt{true}, \mathtt{false}\}$, initially `false` {*Flag whether $\mathcal{S}$ is currently in a (malicious) key rotation*

CheckID$(pid, sid, role)$**:**
Check that $pid = 0 \wedge role = \mathcal{S}$ and that we always have the same/one sid.
If this check fails, output `reject`.

Initialization:

send $(\mathtt{Establish}, \varepsilon)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{auth}} : \mathtt{auth})$ {*Establish authenticated channel*
wait for `ack`
send $\mathtt{InitMe}_{\mathcal{S}}$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{init\&rotateKey}} : \mathtt{init})$
wait for $(\mathtt{InitMe}_{\mathcal{S}}, q, G_1, G_2, G_t, g_t, sk, pk, pk_1, \ldots, pk_{\mathsf{n}}, c)$
$\mathsf{q} \leftarrow q; \mathbb{G}_1 \leftarrow G_1; \mathbb{G}_2 \leftarrow G_2; \mathbb{G}_t \leftarrow G_t; \mathsf{g}_t \leftarrow g_t; \mathsf{sk} \leftarrow sk; \mathsf{pk} \leftarrow pk; \mathsf{pk}_1 \leftarrow pk_1, \ldots \mathsf{pk}_{\mathsf{n}} \leftarrow pk_{\mathsf{n}}$;
for all $T \in \{A \mid A \in 2^{[1,\mathsf{n}]} \wedge |A| = \mathsf{t}\}$ **do:** Let δ be an array/sequence of length n (each entry accessible via $\delta[i], i \in [1, \mathsf{n}]$).
  for all $rl \in T$ **do:** $\delta[rl] \leftarrow \prod_{j \in T, j \neq rl} \frac{-j}{rl - j}$ {*Compute part of the reconstruction vector for T*
  $\mathsf{reconVectors}[T] \leftarrow \delta$
for all $T \in \{A \mid A \in 2^{[1,\mathsf{n}]} \wedge |A| = \mathsf{t}\}$ **do:** Load δ as array of length n from $\mathsf{reconVectors}[T]$. {*Generate updated* $\mathsf{T}_{\mathsf{set}}$
  $pk' = \sum_{i \in T} \delta[i] \cdot \mathsf{pk}_i$
  if $pk' + \mathsf{sk} \cdot [1] = \mathsf{pk}$:
    $\mathsf{T}_{\mathsf{set}}.\mathsf{add}(T)$ {*Record that $\mathcal{S}$ can use T to reconstruct* pk*, resp.* sk
if c: Add $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ to CorruptionSet.
  Leak $\mathsf{storage}, \mathsf{reqQueue}_{\mathsf{Enc}}, \mathsf{reqQueue}_{\mathsf{Dec}}, \mathsf{nonces}, \mathsf{sk}, \mathsf{pk}, \mathsf{pk}_1, \ldots, \mathsf{pk}_{\mathsf{n}}, \mathsf{retrieveCounter}, \mathsf{limiterResponses}, \mathsf{reconVectors}$, and $\mathsf{T}_{\mathsf{set}}$ responsively to `NET`.[a]

MessagePreprocessing:
// Allow environment to check correct simulation of corruption and decorruption//

recv `CorruptionLog`? **from** `I/O`: {*Provide full chronological history of all corruption changes.*
  send `CorruptionLog`? **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{init\&rotateKey}} : \mathtt{init})$
  wait for $(\mathtt{CorruptionLog}, corrLog)$
  reply $(\mathtt{CorruptionLog}, corrLog)$

recv msg **from** `NET` or `I/O` **s.t.** $\mathsf{rotationOngoing} = \mathtt{true} \wedge m \neq (\mathtt{RotateKey}, corr_{\mathcal{S}}, sk, pk, pk_1, \ldots, pk_{\mathsf{n}}, nonces)$:
  break {*$\mathcal{S}$ declines further requests during key rotation*

Continue with Figure 12.

[a] The notation "send this responsively to `NET`" means that the ITM sends a restricting message via **send responsively to** $\mathcal{A}$ leaking the data here and waits for its reactivation, e.g., by **wait for** `ack`.

Fig. 11: The real server protocol $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$ of the TPHE scheme (Part 1).

Description of $M_{\mathcal{S}}$ (continued):

MessagePreprocessing (cont.):

// (Honest) key rotation //

recv $(\texttt{RotateKey}, corr_{\mathcal{S}}, sk, pk, pk_1, \ldots, pk_{\mathsf{n}}, nonces)$ **from** $(\_, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{init\&rotateKey}})$**:**
- $\mathsf{sk} \leftarrow sk, \mathsf{pk} \leftarrow pk, \mathsf{pk}_1 \leftarrow pk_1, \ldots \mathsf{pk}_{\mathsf{n}} \leftarrow pk_{\mathsf{n}}; \mathsf{nonces} \leftarrow nonces.$
- **if** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ in CorruptionSet**:**
  - **send responsively** `getState` **to** `NET` {*Get manipulated storage from* $\mathcal{A}$
  - **wait for** $(\texttt{getState}, storage)$ **s.t.** data formats match the specification of $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$
  - $\mathsf{storage} \leftarrow storage.$
- Remove $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ from CorruptionSet.
- Set all entries in $\mathsf{limiterResponses}_{\mathsf{Enc}}, \mathsf{reqQueue}_{\mathsf{Enc}}, \mathsf{reqQueue}_{\mathsf{Dec}}, \mathsf{limiterResponses}_{\mathsf{Dec}}$, and $\mathsf{T}_{\mathsf{set}}$ to $\perp$, resp. back to their initial state. {*Clear all caches as secret key* sk *changed and cannot be used after this activation.*
- **for all** $T \in \{A \mid A \in 2^{[1,\mathsf{n}]} \wedge |A| = \mathsf{t}\}$ **do:**
  - Let δ be an array/sequence of length n (each entry accessible via $\delta[i], i \in [1, \mathsf{n}]$).
  - **for all** $rl \in T$ **do:**
    - $\delta[rl] \leftarrow \prod_{j \in T, j \neq rl} \frac{-j}{rl - j}$ {*Compute part of the reconstruction vector for* T
  - $\mathsf{reconVectors}[T] \leftarrow \delta$
- $\mathsf{T}_{\mathsf{set}} = \emptyset$
- **for all** $T \in \{A \mid A \in 2^{[1,\mathsf{n}]} \wedge |A| = t \wedge \forall i \in A : \mathsf{pk}_i \neq \perp\}$ **do:** {*Update* $\mathsf{T}_{\mathsf{set}}$
  - Load δ as array of length n from $\mathsf{reconVectors}[T]$.
  - $pk' = \sum_{i \in T} \delta[i] \cdot \mathsf{pk}_i$
  - **if** $pk' + \mathsf{sk} \cdot [1] = \mathsf{pk}$**:** $\mathsf{T}_{\mathsf{set}}.\mathsf{add}(T)$ {*Record that* $\mathcal{S}$ *can use* T *to reconstruct* pk*, resp.* sk
- $\mathsf{rotationOngoing} \leftarrow \texttt{false}$ {*Record that key rotation finished*
- **if** $corr_{\mathcal{S}}$**:** {*Server is honest in current round*
  - Add $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ to CorruptionSet.
  - Leak $\mathsf{storage}, \mathsf{sk}, \mathsf{pk}, \mathsf{pk}_1, \ldots, \mathsf{pk}_{\mathsf{n}}, \mathsf{nonces}, \mathsf{T}_{\mathsf{set}}, \mathsf{limiterResponses}_{\mathsf{Enc}}, \mathsf{reqQueue}_{\mathsf{Enc}}, \mathsf{reqQueue}_{\mathsf{Dec}}$, and $\mathsf{limiterResponses}_{\mathsf{Dec}}$ to `NET`.
- **else:**
  - Leak $\mathsf{pk}, \mathsf{pk}_1, \ldots, \mathsf{pk}_{\mathsf{n}}, \mathsf{T}_{\mathsf{set}}$ to `NET`.

// Forwarding during corruption //

recv m' **from** `NET`**:** {*Forwarding of messages received via* `NET`
- **if** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}}) \in$ CorruptionSet**:** {*Forward non-modeling related calls to* `NET`
  - **if** $m' = (\texttt{Fwd}, (pid, sid, role), m) \wedge$ *role* is a subroutine of $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$**:**
    - **send** m **to** $(pid, sid, role)$ {*Forward messages to subroutines*
  - **if** $m' = (\texttt{Fwd}, (pid, sid, role), m) \wedge$ *role* is not a subroutine of $\mathcal{P}^{\mathcal{S}}_{\text{PHE}} \wedge m$ can be parsed as $(\texttt{Retrieve}, \_, \_)$**.:**
    - **send** m **to** $(pid, sid, role)$ {*Forward messages to I/O*

recv m' **from** `I/O`**:**
- **if** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}}) \in$ CorruptionSet**:** {*Forward non-modeling related calls to* `NET`
  - **if** $m' \in \{(\texttt{Store}, \_, \_, \_), (\texttt{Retrieve}, \_, \_)\}$ or `Received`**:**
    - Forward message to `NET`.

Fig. 12: The real server protocol $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$ of the TPHE scheme (Part 2).

Description of $M_{\mathcal{S}}$ (continued):

Main:

// Storing //

recv $(\texttt{Store}, id, pw, m)$ **from** $\texttt{I/O}$ **to** $(0, \_, \mathcal{S})$: {*Store a message protected by a password. The ID serves as a (public) reference to that message.*

$\mathsf{storage}[id] \leftarrow \bot$ {*Clear storage.*

$r \xleftarrow{\$} \mathbb{Z}_{\mathsf{q}}^*; n_s \xleftarrow{\$} \{0,1\}^\eta$ {*Generate randomness and nonce*

if $\mathsf{T}_{\mathsf{set}} = \emptyset$: **break** {*Stop execution*

send responsively $(\texttt{GetInvolvedRatelimiters}, \texttt{Store}, id)$ **to** $\texttt{NET}$ {*Query $\mathcal{A}$ for involved ratelimiters*

wait for $(\texttt{GetInvolvedRatelimiters}, T)$ **s.t.** $T \in \mathsf{T}_{\mathsf{set}}$

if $\mathsf{nonces}[rl]$ for an $rl \in T$: **break** {*We do not have sufficient nonces to process the request*

Let n_{rl} be the first nonce in $\mathsf{nonces}[rl] \forall rl \in T$.

for all $rl \in T$ **do:** {*Prepare necessary set of nonces for encryption*

$nonces.\texttt{add}((rl, n_{rl}))$[a]; $\mathsf{nonces}[rl].\texttt{remove}(n_{rl})$. {*Remove used nonce from chache*

$nonces.\texttt{add}(n_s)$

send $nonces$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{ro}}^{\mathsf{N}} : \texttt{randomOracle})$ {*Generate n*

wait for n

send (pw, n) **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{ro}}^{2} : \texttt{randomOracle})$ {*Generate $H_2(pw, n)$*

wait for h

$[p]_2 \leftarrow r \cdot [h]_2$; [b] $msg \leftarrow \varepsilon; nonces \leftarrow \varepsilon$

for all $rl \in T$ **do:** $\mathsf{limiterResponses}_{\mathsf{Enc}}[(id, rl)] \leftarrow \bot$ {*Clear ratelimiter respsonses*

$\mathsf{reqQueue}_{\mathsf{Enc}}[id] \leftarrow (pw, m, r, n, T, p)$ {*Store current state and start to contact ratelimiters*

for all $rl \in T$ **do:** {*Prepare message to ratelimiters*

$msg.\texttt{add}[(rl, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R}), (\texttt{EncRequest}, id, [p]_2, nonces)]$

send $(\texttt{Send}, msg)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{auth}} : \texttt{auth})$

recv $(\texttt{Received}, (rl, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R}), (\texttt{FinalizeEnc}, id, \pi, u, n))$ **from** $(\_, \_, \mathcal{F}_{\mathrm{auth}} : \texttt{auth})$, **s.t.** $rl \in T$, where T from $\mathsf{reqQueue}_{\mathsf{Enc}}[id]$: {*Receive answer for encryption request from limiter rl*

Load values (pw, m, r, n, T) from $\mathsf{reqQueue}_{\mathsf{Enc}}[id]$

send (id, n) **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{ro}}^{1} : \texttt{randomOracle})$

wait for h; $[o]_t \leftarrow [h]_1 \cdot [p]_2$

send $(\texttt{Verify}, (\mathsf{g}_t, [o]_t, \mathsf{pk}_{rl}, u), \pi)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{nizk}} : \texttt{verifier})$ {*Check NIZK*

wait for $(\texttt{Verify}, b)$

if b: {*Continue with processing when NIZK was valid*

$\mathsf{nonces}[rl].\texttt{add}(n); \mathsf{limiterResponses}_{\mathsf{Enc}}[(id, rl)] \leftarrow u$ {*Store new noce and u for later processing*

$c \leftarrow \texttt{true}, u_f \leftarrow [0]_t$

Load δ as array of length n from $\mathsf{reconVectors}[T]$.

for all $rl' \in T$ **do:** {*Check whether all ratelimiters responded*

if $\mathsf{limiterResponses}_{\mathsf{Enc}}[(id, rl')] \neq \bot$:

$[u_f]_t \leftarrow [u_f]_t + \delta[rl'] \cdot [\mathsf{limiterResponses}_{\mathsf{Enc}}[(id, rl')]]_t$

else:

$c \leftarrow \texttt{false}$

if c: {*All ratelimiters from T responded*

send $((\frac{[u_f]_t}{r} + \mathsf{sk} \cdot [o]_t, pw, id, n), |m|)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{ro}}^{\mathsf{OTP}} : \texttt{randomOracle})$

wait for h_1; $c_1 \leftarrow h_1 \oplus$

send $(\frac{[u_f]_t}{r} + \mathsf{sk} \cdot [o]_t, m, pw, id, n)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{ro}}^{\mathsf{MAC}} : \texttt{randomOracle})$

wait for c_2;

$\mathsf{reqQueue}_{\mathsf{Enc}}[id] \leftarrow \bot$ {*Clean up storage*

for all $rl \in T$ **do:** $\mathsf{limiterResponses}_{\mathsf{Enc}}[(id, rl)] \leftarrow \bot$

$\mathsf{storage}[id].\texttt{add}[((c_1, c_2), n)]$

[a] We expect that $\{(rl, nonce)\}_{rl \in [1,\mathsf{n}]}$ to be an ordered set, ordered by increasing ratelimiter pid.

[b] We write $[\cdot]_i, i \in \{1, 2, t\}$ to highlight the group $\mathbb{G}_i$ we are currently considering.

Fig. 13: The real server protocol $\mathcal{P}_{\mathrm{PHE}}^{\mathcal{S}}$ of the TPHE scheme (Part 3).

Description of $M_{\mathcal{S}}$ (continued):

// Retrieving //

recv $(\texttt{Retrieve}, id, pw')$ **from** $\texttt{I/O}$ **to** $(0, \_, \mathcal{S})$: {*Try to retrieve the message stored under ID id using password pw′.*

If exits, load $((c_1, c_2), n)$ from $\mathsf{storage}[id]$. Otherwise, abort execution.

send responsively $(\texttt{GetInvolvedRatelimiters}, \texttt{Retrieve}, id)$ **to** $\texttt{NET}$ {*Query $\mathcal{A}$ for involved ratelimiters*

wait for $(\texttt{GetInvolvedRatelimiters}, T)$ **s.t.** $T \in \mathsf{T}_{\mathsf{set}}$

$\mathsf{retrieveCounter} \leftarrow \mathsf{retrieveCounter} + 1; r \xleftarrow{\$} \mathbb{Z}_{\mathsf{q}}^*$ {*Generate randomness*

send (pw', n) **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{ro}}^2 : \texttt{randomOracle})$ {*Generate $H_2(pw', n)$*

wait for h; $[p]_2 \leftarrow [h]_2 \cdot r$; $msg \leftarrow \varepsilon$

$\mathsf{reqQueue}_{\mathsf{Dec}}[\mathsf{retrieveCounter}] \leftarrow (id, pw', r, n, T, p, (\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}}))$ {*Store current state for decryption and start to contact ratelimiters*

for all $rl \in T$ **do:** $msg.\texttt{add}[(rl, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R}), (\texttt{DecRequest}, \mathsf{retrieveCounter}, id, [p]_2, n)]$ {*Prepare message to ratelimiters*

send $(\texttt{Send}, msg)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{auth}} : \texttt{auth})$

Main:

recv $(\texttt{Received}, (rl, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R}), (\texttt{FinalizeDec}, ctr, id, \pi_{\mathcal{R}}, u))$ **from** $(\_, \_, \mathcal{F}_{\mathrm{auth}} : \texttt{auth})$,
s.t. $rl \in T$, where T from $\mathsf{reqQueue}_{\mathsf{Dec}}[ctr]$: {*Receive answer for decryption request from limiter rl*

Load values $(id, pw, r, n, T, p, caller)$ from $\mathsf{reqQueue}_{\mathsf{Dec}}[ctr]$ and $((c_1, c_2), n)$ from $\mathsf{storage}[id]$.

send $(\mathsf{pid}_{\mathsf{cur}}, (id, n))$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{ro}}^1 : \texttt{randomOracle})$

wait for $(\mathsf{pid}_{\mathsf{cur}}, h)$

$[o]_t \leftarrow [h]_1 \cdot [p]_2$

send $(\texttt{Verify}, (\mathsf{g}_t, [o]_t, \mathsf{pk}_{rl}, u), \pi_{\mathcal{R}})$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{nizk}} : \texttt{verifier})$ {*Check NIZK*

wait for $(\texttt{Verify}, b)$

if b: {*Exit 1: Continue with processing when NIZK was valid*

$\mathsf{limiterResponses}_{\mathsf{Dec}}[(ctr, rl)] \leftarrow (u)$ {*Store u for later processing*

$c \leftarrow \texttt{true}, u_f \leftarrow [0]_t$

for all $rl' \in T$ **do:** {*Check whether all ratelimiters responded*

if $\mathsf{limiterResponses}_{\mathsf{Dec}}[(ctr, rl')] \neq \bot$:

$[u_f]_t \leftarrow [u_f]_t + \delta_{rl'} \cdot [\mathsf{limiterResponses}_{\mathsf{Dec}}[(id, rl')]]_t$

else:

$c \leftarrow \texttt{false}$

if c: {*Exit 2: All ratelimiters from T responded*

send $((\frac{[u_f]_t}{r} + \mathsf{sk} \cdot [o]_t, pw, id, n), |c_1|)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{ro}}^{\mathsf{OTP}} : \texttt{randomOracle})$

wait for h_1; $m \leftarrow h_1 \oplus c_1$ {*Decrypt requested ciphertext*

send $(\frac{[u_f]_t}{r} + \mathsf{sk} \cdot [o]_t, m, pw, id, n)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{ro}}^{\mathsf{MAC}} : \texttt{randomOracle})$

wait for c' {*Check that m was not manipulated in storage*

for all $rl \in T$ **do:** $\mathsf{limiterResponses}_{\mathsf{Dec}}[(ctr, rl)] \leftarrow \bot$

$\mathsf{reqQueue}_{\mathsf{Dec}}[ctr] \leftarrow \bot$ {*Clean up storage*

if $c_2 = c'$: {*"MAC" check*

send $(\texttt{Retrieve}, id, m)$ **to** *caller*.

else:

send $(\texttt{Retrieve}, id, \bot)$ **to** *caller*.

send $(\texttt{Retrieve}, id, \bot)$ **to** *caller*.

// Nonce generation//

recv $(\texttt{GetNonces}, pid)$ **from** $\texttt{NET}$: {*$\mathcal{A}$ triggers request for ne nonces*

send $(\texttt{Send}, (pid, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R}), (\texttt{GetNonces}))$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{auth}} : \texttt{auth})$

recv $(\texttt{Received}, (pid, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R}), (\texttt{GetNonces}, nonces))$ **from** $\texttt{NET}$
s.t. $pid \in [1, \mathsf{n}] \ \wedge$ *nonces* is an ordered set with elements from $\{0,1\}^{\eta}$: {*Delivery of newly generate nonces from pid.*

for all $e \in nonces$ **do:**

$\mathsf{nonces}[pid].\texttt{add}(e)$ {*In order of the ordered set*

// Model database leak//

recv $\texttt{stealDB}$ **from** $\texttt{NET}$: {*$\mathcal{A}$ may "steal" $\mathcal{S}$'s database*

reply $(\texttt{stealDB}, \mathsf{storage})$

// Dishonest key rotation with honest server //

recv $\texttt{RotationOngoing}$ **from** $(\_, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{init\&rotateKey}} : \texttt{init})$: {*Key rotation ongoing, stop other processes*

$\mathsf{rotationOngoing} \leftarrow \texttt{true}$

reply $\texttt{ack}$

Fig. 14: The real server protocol $\mathcal{P}_{\mathrm{PHE}}^{\mathcal{S}}$ of the TPHE scheme (Part 4).

Description of the protocol $\mathcal{P}^{\mathcal{R}}_{\text{PHE}} = (\mathcal{R})$:

Participating roles: $\{\mathcal{R}\}$
Corruption model: custom
Protocol parameters:
- $\mathsf{n} \in \mathbb{N}$. {*Number of ratelimiters.*
- $1 \leq \mathsf{t} \leq \mathsf{n}$. {*Threshold of ratelimiters needed for decryption.*
- $\mathsf{ng} \in \mathbb{N}$ {*The number of nonces generated in batch upon a server request*

Description of $M_{\mathcal{R}}$:

Implemented role(s): $\{\mathcal{R}\}$
Subroutines: $\mathcal{F}^{i}_{\text{ro}} : \texttt{randomOracle}(i \in \{1, 2, \mathsf{OTP}, \mathsf{MAC}, \mathsf{N}\}), \mathcal{F}_{\text{nizk}} : \texttt{nizk}, \mathcal{F}_{\text{init\&rotateKey}} : \texttt{init}, \mathcal{F}_{\text{auth}} : \texttt{auth}$
Internal state:
- $\mathsf{retrieveRate} : \mathbb{N} \to \mathbb{N}$ {*Maps an ID to the number of times the ratelimiter will support decryption of ID, initially 0 for all entities*
- $\mathsf{q} \in \mathbb{N}$ {q*: the group size of used groups* $\mathbb{G}_1, \mathbb{G}_2$, *and* $\mathbb{G}_t$, *initially set by* $\mathcal{F}_{\text{init\&rotateKey}}$
- $\mathsf{nonces} \subset \{0,1\}^{\eta}$. {*A set of nonces to be used during the protocol run*
- $\mathsf{sk}, \mathsf{pk} \in \{0,1\}^*$. {$\mathcal{R}$*'s public and secret key, initially set by* $\mathcal{F}_{\text{init\&rotateKey}}$
- $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t \subset \{0,1\}^*$, initially $\emptyset$. {*Used groups, initially set by* $\mathcal{F}_{\text{init\&rotateKey}}$
- $\mathsf{g}_t \in \{0,1\}^*$, resp. $\mathbb{G}_t$, initially ε. {*Generator of* $\mathbb{G}_t$, *initially set by* $\mathcal{F}_{\text{init\&rotateKey}}$

CheckID$(pid, sid, role)$**:**
Check that $pid \in [1, \mathsf{n}] \wedge role = \mathcal{R}$ and that we always have the same/one pid and sid.
If this check fails, output `reject`.

Initialization:
send $(\texttt{Establish}, \varepsilon)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{auth}} : \texttt{auth})$ {*Establish authenticated channel*
wait for `ack`
send `InitMe` **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{init\&rotateKey}} : \texttt{init})$
wait for $(\texttt{InitMe}, q, G_1, G_2, G_t, g_t, sk, pk, c)$
$\mathsf{q} \leftarrow q; \mathbb{G}_1 \leftarrow G_1, \mathbb{G}_2 \leftarrow G_2; \mathbb{G}_t \leftarrow G_t; \mathsf{sk} \leftarrow sk, \mathsf{pk} \leftarrow pk$
if c**:**
Add $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ to $\mathsf{CorruptionSet}$.
Leak sk, pk to `NET`.

MessagePreprocessing:
// Allow environment to check correct simulation of corruption and decorruption//

recv `CorruptionLog`? **from** `I/O`: {*Provide full chronological history of all corruption changes.*
send `CorruptionLog`? **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{init\&rotateKey}} : \texttt{init})$
wait for $(\texttt{CorruptionLog}, corrLog)$
reply $(\texttt{CorruptionLog}, corrLog)$

// Key rotation //

recv $(\texttt{RotateKey}, corr, s)$ **from** $(\_, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{init\&rotateKey}} : \texttt{init})$**:**
if $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}}) \in \mathsf{CorruptionSet}$**:**
send responsively `getState` **to** `NET` {*Get manipulated storage from* $\mathcal{A}$
wait for $(\texttt{getState}, sk, pk, nonceCtr, nonces)$ **s.t.** data formats match the specification of $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$
$\mathsf{sk} \leftarrow sk; \mathsf{pk} \leftarrow pk; \mathsf{maxNonceCtr} \leftarrow nonceCtr + 1; \mathsf{nonces} \leftarrow nonces.$
$\mathsf{sk} \leftarrow \mathsf{sk} - s$ {*Update secret key*
$\mathsf{pk} \leftarrow \mathsf{sk} \cdot [\mathsf{g}_t]_t$ {*Update public key*
if $corr = \texttt{true}$**:**
Add $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ to the set $\mathsf{CorruptionSet}$.
Leak $(\mathsf{retrieveRate}, \mathsf{nonces}, \mathsf{sk}, \mathsf{pk}, \mathsf{maxNonceCtr})$ responsively to `NET`.
else:
Remove $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ from the set $\mathsf{CorruptionSet}$.
Set $\mathsf{retrieveRate}$ and nonces to their initial (empty) state.
$n \xleftarrow{\$} \{0,1\}^{\eta}$
$\mathsf{nonces.add}(n)$ {*Generate new nonce and store it*
reply $(\texttt{RotateKey}, \mathsf{pk}, n)$

// Forwarding during corruption //

recv m **from** `NET`**:**
if $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}}) \in \mathsf{CorruptionSet}$**:** {*Forward non-modeling related calls to* `NET`
if $m = (\texttt{Fwd}, (pid, sid, role), m') \wedge role$ is a subroutine**:**
send m' **to** $(pid, sid, role)$ {*Forward message*
if the message was received via `NET`/subroutine**:**
Forward message to `NET`.

Fig. 15: The real ratelimiter protocol $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$ of the TPHE scheme (Part 1).

Description of $M_{\mathcal{R}}$ (continued):

Main:

// Encryption request //

recv $(\texttt{Received}, (0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S}), (\texttt{EncRequest}, id, [p]_2, \{(rl, nonce_{rl})\}_{rl\in[1,\mathsf{n}]}{}^{a}))$
from $(\_, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{auth}} : \texttt{auth})$ **s.t.** $(\mathsf{pid}_{\mathsf{cur}}, \_) \in \{(rl, nonce_{rl})\}_{rl\in[1,\mathsf{n}]} \wedge nonce_{\mathsf{pid}_{\mathsf{cur}}} \in \mathsf{nonces}$: {*Receive encryption request*
$\mathsf{nonces}.\texttt{remove}(nonce_{\mathsf{pid}_{\mathsf{cur}}})$
send $(\{(rl, nonce)\}_{rl\in[1,\mathsf{n}]})$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{ro}}^{\mathsf{N}} : \texttt{randomOracle})$ {*Generate n*
wait for n
send (id, n) **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{ro}}^{1} : \texttt{randomOracle})$
wait for h
$[o]_t \leftarrow [h]_1 \cdot [p]_2$
$u \leftarrow \mathsf{sk} \cdot [o]_t$
send $(\texttt{Prove}, (\mathsf{g}_t, [o]_t, \mathsf{pk}, u), \mathsf{sk})$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{nizk}} : \texttt{prover})$ {*Generate NIZK*
wait for $(\texttt{Prove}, \pi_{\mathcal{R}})$
if $\pi_{\mathcal{R}} = \perp$: **break** {*Abort execution*
$n \xleftarrow{\$} \{0,1\}^{\eta}$
$\mathsf{nonces}.\texttt{add}(n)$ {*Generate new nonce and store it*
send $(\texttt{Send}, (0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S}), (\texttt{FinalizeEnc}, id, \pi_{\mathcal{R}}, u, n))$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{auth}} : \texttt{auth})$

// Decryption request //

recv $(\texttt{Received}, (0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S}), (\texttt{DecRequest}, ctr, id, [p]_2, n))$
from $(\_, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{auth}} : \texttt{auth})$ **s.t.** $id \in \mathsf{storage} \wedge \mathsf{retrieveRate}[id] > 0$: {*Server request support for decryption of id*
$\mathsf{retrieveRate}[id] \leftarrow \mathsf{retrieveRate}[id] - 1$ {*Record decryption request*
send (id, n) **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{ro}}^{1} : \texttt{randomOracle})$
wait for h
$[o]_t \leftarrow [h]_1 \cdot [p]_2$
$u \leftarrow \mathsf{sk} \cdot [o]_t$
send $(\texttt{Prove}, (\mathsf{g}_t, [o]_t, \mathsf{pk}, u), \mathsf{sk})$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{nizk}} : \texttt{prover})$ {*Generate NIZK*
wait for $(\texttt{Prove}, \pi_{\mathcal{R}})$
if $\pi_{\mathcal{R}} = \perp$:
break {*Abort execution*
send $(\texttt{Send}, (0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S}), (\texttt{FinalizeDec}, ctr, id, \pi_{\mathcal{R}}, u))$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{auth}} : \texttt{auth})$

// Nonce generation//

recv $(\texttt{Received}, (0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S}), (\texttt{GetNonces}))$ **from** NET: {*Server request a new batch of nonces*
$nonces \leftarrow \emptyset$
for $i = 0$ to ng **do**:
$n \xleftarrow{\$} \{0,1\}^{\eta}$
$\mathsf{nonces}.\texttt{add}(n)$, $nonces.\texttt{add}(n)$
send $(\texttt{Send}, (0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S}), (\texttt{GetNonces}, nonces))$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{auth}} : \texttt{auth})$

recv $(\texttt{HelpRetrieve}, id)$ **from** I/O **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R})$: {*Decryption support for id added by* $\mathcal{E}$
$\mathsf{retrieveRate}[id] \leftarrow \mathsf{retrieveRate}[id] + 1.$
reply $(\texttt{HelpRetrieve}, \texttt{OK})$

recv $(\texttt{GetRetrieveRate}, id)$ **from** NET **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R})$:
reply $(\texttt{GetRetrieveRate}, \mathsf{retrieveRate}[id])$

[a] We expect that $\{(rl, nonce_{rl})\}_{rl\in[1,\mathsf{n}]}$ to be an ordered set, orderd by increasing ratelimiter *pid*.

Fig. 16: The real server protocol $\mathcal{P}_{\text{PHE}}^{\mathcal{R}}$ of the TPHE scheme (Part 2).

Description of the protocol $\mathcal{F}_{\text{init\&rotateKey}} = (\texttt{init})$:

Participating roles: $\{\texttt{init}\}$
Corruption model: incorruptible
Protocol parameters:
- $\eta \in \mathbb{N}$. {*The security parameter.*
- $\mathsf{n} \in \mathbb{N}$. {*Number of ratelimiters.*
- $1 \leq \mathsf{t} \leq \mathsf{n}$. {*Threshold of ratelimiters needed for decryption.*

Description of $M_{\texttt{init}}$:

Implemented role(s): $\{\texttt{init}\}$
Internal state:
- $\mathsf{corrLog} = \emptyset$. {*Chronologically ordered sequence of corruption updates. Each entry is of the form* $(entity, b)$ *denoting an entity* $= (pid, sid, role)$ *whose corruption status has been changed to* $b \in \{\texttt{true}, \texttt{false}\}$.
- $\mathsf{sk}, \mathsf{sk}_{\mathcal{S}}, \mathsf{sk}_1 : \mathbb{N} \to \{0,1\}^*, \ldots, \mathsf{sk}_{\mathsf{n}} : \mathbb{N} \to \{0,1\}^*$ {*The key, the server and ratelimiter shares of the key (per round)*
- $\mathsf{pk}, \mathsf{pk}_1, \ldots, \mathsf{pk}_{\mathsf{n}} \in \mathbb{G}_t$ {*Public keys of ratelimiters*
- $\mathsf{q} \in \mathbb{N}$ {q*: the group size of used groups* $\mathbb{G}_1, \mathbb{G}_2$, *and* $\mathbb{G}_t$
- $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, \mathsf{g}_t \in \mathbb{G}_t$ {*Used groups,* g_t *is a generator of* $\mathbb{G}_t$
- $\mathsf{currentlyCorrupted} \subset [1, \mathsf{n}] \times \{0,1\}^2$, initially $\mathsf{currentlyCorrupted} = \emptyset$ {*Set of currently corrupted participants*
- $\mathsf{nonces} : [1, \mathsf{n}] \to \{0,1\}^{\eta}$. {*Buffered nonces from each ratelimiter (totally ordered), initially all entries* $\perp$

CheckID$(pid, sid, role)$**:**
Check that we always have the same/one *sid*.
If this check fails, output `reject`.

Initialization:
$(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, \mathsf{q}, g_1, g_2, e) \xleftarrow{\$} \texttt{BGGEN}(1^{\eta})$ {*Generate parameters for TPHE scheme. The usage of* e *is always written as* $[a]_1 \cdot [b]_2 = [ab]_t$ *and not mentioned explicitly*
$\mathsf{g}_t \leftarrow [g_1]_1 \cdot [g_2]_2$ {*Determine generator of* $\mathbb{G}_t$
$\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q; \mathsf{sk}_{\mathcal{S}} \xleftarrow{\$} \mathbb{Z}_q; \mathsf{sk}_{\mathcal{R}} \leftarrow \mathsf{sk} - \mathsf{sk}_{\mathcal{S}}$
$\mathsf{pk} \leftarrow \mathsf{sk} \cdot [\mathsf{g}_t]_t$
$f \xleftarrow{\$} \{g \mid g \in \mathbb{Z}_q[x], \deg(g) < \mathsf{t} \wedge g(0) = \mathsf{sk}_{\mathcal{R}}\}$ {*Choose polynomial for* (n, t)*-Shamir secret sharing*
for $i = 1$ to n **do:**
  $\mathsf{sk}_i \leftarrow f(i)$ {*Generate ratelimiter shares*
  $\mathsf{pk}_i \leftarrow \mathsf{sk}_i \cdot [\mathsf{g}_t]_t$ {*Generate public keys for ratelimiters*

send responsively $(\texttt{InitialCorruption}?, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, \mathsf{q}, \mathsf{g}_t, \mathsf{pk}, \mathsf{pk}_1, \ldots, \mathsf{pk}_{\mathsf{n}})$ **to** `NET` {*Allow adversary to choose initial corruption status of parties and leak TPHE parameters, etc.*
wait for $(\texttt{InitialCorruption}?, s, corr_{\mathcal{R}}, pk_1^c, \ldots, pk_{\mathsf{n}}^c)$ **s.t.** $corr_{\mathcal{R}} \subset \{1, \ldots, \mathsf{n}\} \wedge (|corr_{\mathcal{R}}| \leq t - 1 \vee s = \texttt{false})$.
if $s = \texttt{true}$**:**
  Add $(0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S})$ to the set $\mathsf{currentlyCorrupted}$.
  Add $((0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S}), \texttt{true})$ to $\mathsf{corrLog}$.
else:
  Add $((0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S}), \texttt{false})$ to $\mathsf{corrLog}$.
for all $i \in corr_{\mathcal{R}}$ **do:**
  $\mathsf{pk}_i \leftarrow pk_i^c$ {*Record manipulated public key*
  Add $(0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R})$ to the set $\mathsf{currentlyCorrupted}$.
  Add $((0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S}), \texttt{true})$ to $\mathsf{corrLog}$.
for all $i \in [1..\mathsf{n}] \setminus corr_{\mathcal{R}}$ **do:**
  Add $((0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S}), \texttt{false})$ to $\mathsf{corrLog}$.

Fig. 17: The initialization and key rotation functionality $\mathcal{F}_{\text{init\&rotateKey}}$ of the TPHE scheme (Part 1).

Description of the protocol $\mathcal{F}_{\text{init\&rotateKey}} = (\texttt{init})$:

Main:

recv $\texttt{InitMe}_{\mathcal{S}}$ **from** $\texttt{I/O}$ **s.t.** $(\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}}) = (0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S})$:
- **if** $(0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S})$ in currentlyCorrupted:
  - $c = \texttt{true}$
- **else:**
  - $c = \texttt{false}$
- **reply** $(\texttt{InitMe}_{\mathcal{S}}, \mathsf{q}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, \mathsf{g}_t, \mathsf{sk}_{\mathcal{S}}, \mathsf{pk}, \mathsf{pk}_1, \ldots, \mathsf{pk}_{\mathsf{n}}, c)$

recv $\texttt{InitMe}_{\mathcal{R}}$ **from** $\texttt{I/O}$ **s.t.** $(\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}}) = (i, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R}), i \in \{1, \ldots, \mathsf{n}\}$:
- **if** $(i, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R})$ in currentlyCorrupted:
  - $c = \texttt{true}$
- **else:**
  - $c = \texttt{false}$
- **reply** $(\texttt{InitMe}_{\mathcal{R}}, \mathsf{q}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, \mathsf{g}_t, \mathsf{sk}_i, \mathsf{pk}_i, c)$

recv $\texttt{getCodomain}$ **from** $(\_, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{ro}}^{i} : \texttt{randomOracle})(i \in \{1, 2, \mathsf{OTP}, \mathsf{MAC}, \mathsf{N}\})$: {*Set hashfunctions correct codomain*
- **if** $i = 1$:
  - **reply** $(\texttt{getCodomain}, \mathbb{G}_1)$
- **else if** $i = 2$:
  - **reply** $(\texttt{getCodomain}, \mathbb{G}_2)$
- **else:**
  - **reply** $(\texttt{getCodomain}, \{0,1\}^{\eta})$

Fig. 18: The initialization and key rotation functionality $\mathcal{F}_{\text{init\&rotateKey}}$ of the TPHE scheme (Part 2).

Description of $M_{\texttt{init}}$ (continued):

Main:

// Honest key rotation //

recv $(\texttt{ChangeCorruption}, corr_{\mathcal{S}}, corrIDs_{\mathcal{R}})$ **from** NET **s.t.** $corrIDs_{\mathcal{R}} \subset [1..\mathsf{n}] \wedge (|corrIDs_{\mathcal{R}}| \leq t-1 \vee s = \texttt{false})$: {*Advance round/rotate key*

  currentlyCorrupted $\leftarrow \emptyset$ {*Update* currentlyCorrupted

  if $corr_{\mathcal{S}}$: {*Record $\mathcal{S}$ to be corrupted*

    currentlyCorrupted.add$[(0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S})]$

  Add $((0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S}), corr_{\mathcal{S}})$ to corrLog.

  for all $i \in corrIDs_{\mathcal{R}}$ **do:** {*Record $\mathcal{R}$ i to be corrupted*

    currentlyCorrupted.add$[(i, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R})]$

    Add $((0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R}), \texttt{true})$ to corrLog.

  for all $i \in [1..\mathsf{n}] \setminus corrIDs_{\mathcal{R}}$ **do:** {*Record $\mathcal{R}$ i to be honest*

    Add $((0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R}), \texttt{false})$ to corrLog.

  $\alpha \xleftarrow{\$} \mathbb{Z}_q; \mathsf{sk}_{\mathcal{S}} \leftarrow \mathsf{sk}_{\mathcal{S}} + \alpha; f \xleftarrow{\$} \{g \mid g \in \mathbb{Z}_q[x], \deg(g) < \mathsf{t} \wedge g(0) = \alpha\}$ {*Choose polynomial for (n, t)-Shamir secret sharing*

  for $i = 1$ to n **do:**

    $s_i \leftarrow f(i)$ {*Generate ratelimiter shares*

  for all $i \in [1, \mathsf{n}]$ **do:**

    Let c be `true`, if $i \in corr_{\mathcal{R}}$, otherwise `false`.

    send $(\texttt{RotateKey}, c, s_i)$ **to** $(i, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R})$ {*Provide $\mathcal{R}$ data to run a key rotation*

    wait for $(\texttt{RotateKey}, pk', nonce)$ **s.t.** $pk' \in \mathbb{G}_t \wedge nonce \in \{0,1\}^\eta$ {*Resulting public key and nonce from key rotation*

    $\mathsf{pk}_i \leftarrow pk'; \mathsf{nonces}[i] \leftarrow nonce$

  send $(\texttt{RotateKey}, corr_{\mathcal{S}}, \mathsf{sk}_{\mathcal{S}}, \mathsf{pk}, \mathsf{pk}_1, \ldots, \mathsf{pk}_{\mathsf{n}}, \mathsf{nonces})$ **to** $(0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S})$

// Dishonest key rotation with malicious server //

recv $(\texttt{RotateKey}^{\mathcal{S}}_{\mathcal{A}}, rl, s)$ **from** NET **s.t.** $rl \in [1, \mathsf{n}] \wedge (0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S}) \in$ currentlyCorrupted $\wedge (rl, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R}) \notin$ currentlyCorrupted:

  send $(\texttt{RotateKey}, \texttt{false}, s)$ **to** $(rl, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R})$

  wait for $(\texttt{RotateKey}, pk', nonce)$ {*Resulting public key and nonce from key rotation*

  reply $(\texttt{RotateKey}_{\mathcal{A}}, pk', nonce)$ {*Forward result to $\mathcal{A}$*

// Dishonest key rotation with honest server //

recv $(\texttt{RotateKey}_{\mathcal{A}})$ **from** NET **s.t.** $(0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S}) \notin$ currentlyCorrupted:

  send (RotationOngoing) **to** $(0, \mathsf{pid}_{\mathsf{cur}}, \mathcal{S})$ {*Server stopps processing requests until further rotation finished*

  wait for ack

  $\alpha \xleftarrow{\$} \mathbb{Z}_q; \mathsf{sk}_{\mathcal{S}} \leftarrow \mathsf{sk}_{\mathcal{S}} + \alpha; f \xleftarrow{\$} \{g \mid g \in \mathbb{Z}_q[x], \deg(g) < \mathsf{t} \wedge g(0) = \alpha\}$ {*Choose polynomial for (n, t)-Shamir secret sharing*

  for $i = 1$ to n **do:**

    $s_i \leftarrow f(i)$ {*Generate ratelimiter shares*

  for all $i \in [1..\mathsf{n}]$ **do:** {*Clear pk_i's and* nonces *from last rotation*

    $\mathsf{pk}_i \leftarrow \varepsilon$

    $\mathsf{nonces}[i] \leftarrow \perp$

  for all $i \in \{(i, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R}) \mid (i, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R}) \in \mathsf{currentlyCorrupted}\}$ **do:**

    Let c be `true`, if $i \in corr_{\mathcal{R}}$, otherwise `false`.

    if c: {*Leak share to $\mathcal{A}$*

      send responsively $(\texttt{RotateKey}, i, s_i)$ **to** NET

      wait for ack

  send nextCmd? **to** NET $(\star)$ {*$\mathcal{A}$ may instruct further processing of $\mathcal{F}_{\text{init\&rotateKey}}$*

  wait for m **s.t.** Case I): $m = (\texttt{deliverHonestRL}, i), i \in [1..\mathsf{n}], i$ not in currentlyCorrupted$\vee$ Case II:$m = (\texttt{finishRotation}, \_)$

  if Case I: {*Trigger key rotation for an honest $\mathcal{R}$*

    send $(\texttt{RotateKey}, \texttt{false}, s_i)$ **to** $(i, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R})$ {*Provide $\mathcal{R}$ data to run a key rotation*

    wait for $(\texttt{RotateKey}, pk', nonce)$ **s.t.** $pk' \in \mathbb{G}_t \wedge nonce \in \{0,1\}^\eta$ {*Resulting public key and nonce from key rotation*

    $\mathsf{pk}_i \leftarrow pk'; \mathsf{nonces}[i] \leftarrow nonce$

    send responsively $(\texttt{Leak}, \texttt{RotateKey}, |pk'|, |nonce|)$ **to** NET {*Leak length of messages to $\mathcal{A}$*

    wait for ack

    Go to $(\star)$

  else if Case II $\wedge m = (\texttt{finishRotation}, rl^{corr}_{\text{set}}, pk, nonces)$

   s.t. rl^{corr}_{set} in currentlyCorrupted $\wedge$ pk is a map from $rl^{corr}_{\text{set}} \to \mathbb{G}_t \wedge$

   $nonces$ is a map from $rl^{corr}_{\text{set}} \to \{0,1\}^\eta$:

    for all ratelimiter i in rl^{corr}_{set} **do:**

      $\mathsf{pk}_i \leftarrow pk[i]; \mathsf{nonces}[i] \leftarrow nonces[i]$

    send $(\texttt{RotateKey}, \texttt{false}, \mathsf{sk}_{\mathcal{S}}, \mathsf{pk}, \mathsf{pk}_1, \ldots, \mathsf{pk}_{\mathsf{n}}, \mathsf{nonces})$ **to** $(0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S})$

// Allow environment to check correct simulation of corruption and decorruption//

recv CorruptionLog? **from** I/O: {*Provide full chronological history of all corruption changes.*

  reply $(\texttt{CorruptionLog}, \mathsf{corrLog})$.

Fig. 19: The initialization and key rotation functionality $\mathcal{F}_{\text{init\&rotateKey}}$ of the TPHE scheme (Part 3).

Description of the ideal authenticated channel functionality $\mathcal{F}_{\text{auth}} = (\texttt{auth})$:

Participating roles: $\{\texttt{auth}\}$
Corruption model: custom

Description of $M_{\texttt{auth}}$:

Implemented role(s): $\{\texttt{auth}\}$
Internal state:
- $\mathsf{queue} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ {*Queue of messages from entity e_1 to entity e_2, initially $\perp$ for all entries*
- $\mathsf{RegStatus} : (\{0,1\}^*)^3 \to \{\texttt{inactive}, \texttt{active}, \texttt{established}\}$ {*The status of $(pid, sid, role)$, initially* `inactive` *for all entries*
- $\mathsf{corrStatus} \in \{0,1\}$ {*The corruption status of the $\mathcal{F}_{\text{auth}}$*

CheckID$(pid, sid, role)$**:**
Accept all messages for the same sid.
Main:

recv `Corrupt` **from** `NET`: {*$\mathcal{A}$ (tries to) corrupts current instance of $\mathcal{F}_{\text{auth}}$*
  if $\forall$ entries in $\mathsf{RegStatus}$ are not equal to `established`: {*This models static corruption*
    $\mathsf{corrStatus} \leftarrow \texttt{true}$
    reply $(\texttt{Corrupt}, \texttt{ack})$
  else:
    reply $(\texttt{Corrupt}, \texttt{nack})$

recv `CorruptionStatus`? **from** `I/O`: {*Allows environment to check correct simulation of corrupted parties.*
  if $\mathsf{corrStatus} = \texttt{true}$:
    reply $(\texttt{CorruptionStatus}?, \texttt{true})$
  else:
    reply $(\texttt{CorruptionStatus}?, \texttt{false})$

recv $(\texttt{Establish}, m)$ **from** `I/O` *s.t.* $\mathsf{sid}_{\mathsf{call}} = \mathsf{sid}_{\mathsf{cur}}, \mathsf{RegStatus}[(\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})] = \texttt{inactive}$: {*Establish session*
  $\mathsf{RegStatus}[(\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})] \leftarrow \texttt{active}$
  send responsively $(\texttt{Establish}, m, (\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}}))$ **to** `NET`
  wait for `ack`
  reply `ack`

recv $(\texttt{Establish}, (pid, \mathsf{sid}_{\mathsf{cur}}, role))$ **from** `NET` *s.t.* $\mathsf{RegStatus}[(pid, \mathsf{sid}_{\mathsf{cur}}, role)] = \texttt{active}$: {*Establish session*
  $\mathsf{RegStatus}[(pid, \mathsf{sid}_{\mathsf{cur}}, role)] \leftarrow \texttt{established}$
  send $(\texttt{Establish}, m, (pid, \mathsf{sid}_{\mathsf{cur}}, role))$ **to** $(pid, \mathsf{sid}_{\mathsf{cur}}, role)$

recv $(\texttt{Send}, ([(pid_1, \mathsf{sid}_{\mathsf{cur}}, role_1), m_1], \ldots, [(pid_n, \mathsf{sid}_{\mathsf{cur}}, role_n), m_n]))$ **from** `I/O`
  s.t. *s.t.* $\mathsf{RegStatus}[(\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})] = \texttt{established} \wedge pid \neq \mathsf{pid}_{\mathsf{call}}$: {*Send message via authenticated channel*
  if $\mathsf{corrStatus} = \texttt{false}$:
    for $i = 1$ to n **do:**
      $\mathsf{queue}[(\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}}), (pid_i, \mathsf{sid}_{\mathsf{cur}}, role_i)].\texttt{add}(m_i)$ {*Add m_i to the queue of pid_i*
    send $(\texttt{Send}, (\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}}), ([(pid_1, \mathsf{sid}_{\mathsf{cur}}, role_1), m_1], \ldots, [(pid_n, \mathsf{sid}_{\mathsf{cur}}, role_n), m_n])$ **to** `NET` {*Leak communication*

recv $(\texttt{Deliver}, (pid_1, \mathsf{sid}_{\mathsf{cur}}, role_1), (pid_2, \mathsf{sid}_{\mathsf{cur}}, role_2), m)$ **from** `NET` {*$\mathcal{A}$ triggers message delivery via authenticated channel*
  s.t. $\mathsf{RegStatus}[(pid_2, \mathsf{sid}_{\mathsf{cur}}, role_2)] = \texttt{established} \wedge pid_1 \neq pid_2$:
  if $\mathsf{corrStatus} = \texttt{false}$:
    if $\mathsf{queue}[(pid_1, \mathsf{sid}_{\mathsf{cur}}, role_1), (pid_2, \mathsf{sid}_{\mathsf{cur}}, role_2)] = \perp$: {*There are no queged messages*
      reply $(\texttt{Deliver}, \perp)$ {*Return error*
    else:
      remove the first message from $\mathsf{queue}[(pid_1, \mathsf{sid}_{\mathsf{cur}}, role_1), (pid_2, \mathsf{sid}_{\mathsf{cur}}, role_2)$, let m' be this message
      send $(\texttt{Received}, (pid_1, \mathsf{sid}_{\mathsf{cur}}, role_1), m')$ **to** $(pid_2, \mathsf{sid}_{\mathsf{cur}}, role_2)$ {*Deliver first message from* queue
  else:
    send $(\texttt{Received}, (pid_1, \mathsf{sid}_{\mathsf{cur}}, role_1), m)$ **to** $(pid_2, \mathsf{sid}_{\mathsf{cur}}, role_2)$ {*Deliver message from adversary if corrupted*

recv $(\texttt{Drop}, (pid_1, \mathsf{sid}_{\mathsf{cur}}, role_1), (pid_2, \mathsf{sid}_{\mathsf{cur}}, role_2))$ **from** `NET`:
  if $\mathsf{queue}[(pid_1, \mathsf{sid}_{\mathsf{cur}}, role_1), (pid_2, \mathsf{sid}_{\mathsf{cur}}, role_2)] \neq \perp$:
    remove the first message from $\mathsf{queue}[(pid_1, \mathsf{sid}_{\mathsf{cur}}, role_1), (pid_2, \mathsf{sid}_{\mathsf{cur}}, role_2)]$. {*Drop message*
    reply $(\texttt{Drop}, \texttt{ack})$

Fig. 20: The ideal authenticated channel functionality $\mathcal{F}_{\text{auth}}$ (cf. [40]).

Description of the protocol $\mathcal{F}_{\text{ro}}^{i} = (\texttt{randomOracle})$:

Participating roles: $\{\texttt{randomOracle}\}$
Corruption model: *incorruptible*

Description of $M_{\texttt{randomOracle}}$:

Implemented role(s): $\{\texttt{randomOracle}\}$
Subroutines:
$\mathcal{F}_{\text{init\&rotateKey}} : \texttt{init}$
Internal state:
– $\mathsf{codomain} \subset \{0,1\}^*$, initially $\mathsf{hashHistory} = \emptyset$ {*The codomain of the random oracle*
– $\mathsf{hashHistory} \subseteq \{0,1\}^* \times \mathsf{codomain}$, initially $\mathsf{hashHistory} = \emptyset$ {*The set of recorded value/hash pairs*
CheckID$(pid, sid, role)$**:**
Accept all messages with the same *sid*.
Initialization:
send `getCodomain` **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{init\&rotateKey}} : \texttt{init})$ {*Get codomain of the random oracle*
wait for $(\texttt{getCodomain}, C)$
$\mathsf{codomain} \leftarrow C$
Main:
recv x **from** `I/O` or `NET`: {*Requesting the* $\mathcal{F}_{\text{ro}}^{i}$ *for "hashes"*
if $\exists h \in \mathsf{codomain}$ *s.t.* $(x,h) \in \mathsf{hashHistory}$: {*Extract existing value from* hashHistory
reply h
else:
$h \xleftarrow{\$} \mathsf{codomain}$ {*Generate "hash value" uniformly at random*
$\mathsf{hashHistory} \leftarrow \mathsf{hashHistory}.\texttt{add}((x,h))$ {*Store generated key value pair in* hashHistory
reply h

Fig. 21: The random oracle $\mathcal{F}_{\text{ro}}^{i}$ with variable codomain

Description of the protocol $\mathcal{F}_{\text{ro}}^{\mathsf{OTP}} = (\texttt{randomOracle})$:

Participating roles: $\{\texttt{randomOracle}\}$
Corruption model: *incorruptible*

Description of $M_{\texttt{randomOracle}}$:

Implemented role(s): $\{\texttt{randomOracle}\}$
Internal state:
– $\mathsf{hashHistory} \subseteq \{0,1\}^* \times \mathbb{N} \times \{0,1\}^{\eta}$, initially $\mathsf{hashHistory} = \emptyset$ {*The set of recorded value/hash pairs including the desired hash length*
CheckID$(pid, sid, role)$**:**
Accept all messages with the same *sid*.
Main:
recv (x,l) **from** `I/O` or `NET`: {*Requesting the* $\mathcal{F}_{\text{ro}}^{\mathsf{OTP}}$ *for "hashes"*
if $\exists h \in \{0,1\}^{l}$ *s.t.* $(x,l,h) \in \mathsf{hashHistory}$: {*Extract existing value from* hashHistory
reply h
else:
$h \xleftarrow{\$} \{0,1\}^{l}$ {*Generate "hash value" uniformly at random*
$\mathsf{hashHistory} \leftarrow \mathsf{hashHistory}.\texttt{add}((x,l,h))$ {*Store generated values in* hashHistory
reply h

Fig. 22: The random oracle $\mathcal{F}_{\text{ro}}^{\mathsf{OTP}}$ that produces hashes of desired length.

Description of the protocol $\mathcal{F}_{\text{nizk}} = (\texttt{prover}, \texttt{verifier})$:

Participating roles: $\{\texttt{prover}, \texttt{verifier}\}$
Corruption model: *incorruptible*
Protocol parameters:
– $\mathsf{R} \subset \{0,1\}^* \times \{0,1\}^*$ {*A relation for a language* L

Description of $M_{\texttt{nizk}}$:

Implemented role(s): $\{\texttt{prover}, \texttt{verifier}\}$
Internal state:
– $\mathsf{nizkHistory} \subseteq \mathsf{R} \times \{0,1\}^*$, initially $\mathsf{nizkHistory} = \emptyset$ {*The set of recorded NIZK proofs*
CheckID$(pid, sid, role)$**:**
Accept all messages with the same sid.

Main:

recv $(\texttt{Prove}, x, w)$ **from** $\texttt{I/O}$ **to** $(\_, \_, \texttt{prover})$: {*Provide prove for* x
  if $(x, w) \in \mathsf{R}$: {*Witness* w *matches statement* x
    send responsively $(\texttt{Prove}, x)$ **to** $\texttt{NET}$ {*Query* $\mathcal{A}$ *for NIZK proof*
    wait for $(\texttt{Prove}, \pi)$
    $\mathsf{nizkHistory} \leftarrow \mathsf{nizkHistory} \cup (x, w, \pi)$ {*Store NIZK proof*
    reply $(\texttt{Prove}, \pi)$ {*Return* π *to prover*
  else:
    reply $(\texttt{Prove}, \bot)$ {*Return proof was not accepted*

recv $(\texttt{Verify}, x, \pi)$ **from** $\texttt{I/O}$ or $\texttt{NET}$: {*Check NIZK proof* π *for* x
  if $(x, \_, \pi) \in \mathsf{nizkHistory}$: {*If* π *is recorded as proof for* x *verification succeeds*
    reply $(\texttt{Verify}, \texttt{true})$
  else:
    send responsively $(\texttt{Verify}, x, \pi)$ **to** $\texttt{NET}$ {*Ask* $\mathcal{A}$ *to verify the proof*
    wait for $(\texttt{Verify}, w)$ {$\mathcal{A}$ *provides witness for* x
    if $(x, w) \in \mathsf{R}$: {*If* $\mathcal{A}$ *provides a valid witness* w *for* x*...*
      $\mathsf{nizkHistory} \leftarrow \mathsf{nizkHistory} \cup (x, w, \pi)$ {*Store NIZK proof*
      reply $(\texttt{Verify}, \texttt{true})$ {*Inform requestor that verification succeeds*
    else:
      reply $(\texttt{Verify}, \texttt{false})$ {*Inform requestor that verification fails*

Fig. 23: The NIZK proof functionality $\mathcal{F}_{\text{nizk}}$ (cf. [26, 27])

C Security Analysis of UCPY

In this section, we provide the formal security analysis of UCPY. As part of the proof of Theorem 1 (see Section 4.3), we first define a responsive simulator Sim such that the real world running the protocol $\mathcal{P}_{\text{PHE}}$ is indistinguishable from the ideal world running $\{\mathsf{Sim}, \mathcal{F}_{\text{PHE}}\}$ for every ppt environment $\mathcal{E}$.

The simulator Sim is a single machine that is connected to $\mathcal{F}_{\text{PHE}}$ and the environment $\mathcal{E}$ via their network interfaces. In a run, there is only a single instance of the machine Sim that accepts and processes all incoming messages. The simulator Sim internally simulates the realization $\mathcal{P}_{\text{PHE}}$, including its behavior on the network interface connected to the environment, and uses this simulation to compute responses to incoming messages.

Intuitively, Sim internally simulates the real-world protocol $\mathcal{P}_{\text{TPHE}}$ and relays messages between the environment $\mathcal{E}$ and the ideal functionality $\mathcal{F}_{\text{TPHE}}$. The simulator receives messages from the ideal functionality and, with the information given, constructs messages for the environment that appear to come from the real parties. It outputs a dummy database when the server is compromised, responds appropriately to adversarial messages, and consistently answers queries to the random oracle. Nevertheless, we only consider the behavior of honest parties as messages addressed to corrupt parties are forwarded to the environment. For brevity, we write $y \leftarrow \mathsf{H}_i(x), i \in \{1, 2, \mathsf{OTP}, \mathsf{MAC}, \mathsf{N}\}$ instead of explicitly calling $\mathcal{F}_{\text{ro}}^i$ with input x and receiving response y. We depict the simulator strategy Sim in Figures 24 to 26.

In general, Sim simulates the real protocol, where it is possible with the data leaked by the ideal protocol. In particular, the computations of an honest ratelimiter are identical to a real ratelimiter because all messages to the ratelimiter containing a password are blinded by the server with a random factor and are, therefore, independent of any secret data. On the other hand, Sim hast to mimic the honest server's behavior without knowing the password or the message.

We briefly describe the main challenges we faced when constructing the simulator and explain our tactics for solving them:

- **(4) and (10) Storing/Retrieving Initialization:** The simulator only learns the length of an encrypted message $|m|$ (from the ideal functionality/its message storage) in contrast to the password pw and the message m itself. Hence, it cannot compute the hash of the password $[h]_2 \leftarrow \mathcal{F}_{\text{ro}}^2(pw, n)$ as in the real protocol and blind it with a random factor $r \xleftarrow{\$} \mathbb{Z}_q^*$ to form $[p]_2 \leftarrow r \cdot [h]_2$. Instead, it samples $a \xleftarrow{\$} \mathbb{Z}_q^*$ and sets $[p]_2 \leftarrow a \cdot [1]_2$, which is identical in the view of a ratelimiter.
  If, during storing, the freshly generated nonce n already appears in any $\mathsf{hashHistory}_i, i \in \{1, 2, \mathsf{OTP}, \mathsf{MAC}\}$, the simulator aborts to ensure that the environment does not already know responses of the random oracles to inputs containing the nonce.
- **(6) Storing Finalization:** The simulator cannot compute c_1, c_2 according to the UCPY protocol, as they depend on the password and the message (which are hidden from Sim when the server is honest). Therefore, it samples c_1 from $\{0,1\}^{|m|}$ and c_2 from $\{0,1\}^{\eta}$.
- **(8) Honest Key Rotations:** The main challenge here is to verify database entries received from the environment when the server was corrupted before the key rotation. This can be done for entry (c_1, c_2, n) by checking if there was a query to $\mathcal{F}_{\text{ro}}^{\mathsf{MAC}}$ consistent with c_2, n. Hereby, the simulator learns message m and can check if there was also a query to $\mathcal{F}_{\text{ro}}^{\mathsf{OTP}}$ consistent with c_1, n, m. Finally, it checks the correctness of $\mathsf{sk} \cdot \mathsf{H}_1(id, n) \cdot \mathsf{H}_2(pw, n) = [u]_t$. If all checks passed, it marks the entry by storing the tuple $(id, n, [u]_t, m, pw)$ in $\mathsf{injectedMessages}$. Furthermore, it informs the ideal functionality of queries to the ratelimiters not used in (12). That enables the simulator to use the unused quota of the ratelimiters in future epochs for guessing passwords in (13).
- **(12) Retrieving Finalization:** For this phase, we distinguish between three cases:
  1. If the simulator can find an entry for id in $\mathsf{injectedMessages}$, it takes the password and the message from that entry and sends $(\texttt{FinishRetrieve}, ctr, T, \texttt{success}, (\texttt{false}, 0, pw, m))$ to the ideal functionality. The ideal functionality will send m to the party that initiated the retrieval if the password matches the entered password.
  2. If Sim cannot find such an entry in $\mathsf{injectedMessages}$, it looks for a matching entry in its $\mathsf{storageHistory}$. If it finds one, it sends $(\texttt{FinishRetrieve}, ctr, T, \texttt{success}, (\texttt{true}, i, \epsilon, \epsilon))$ to the ideal functionality. The ideal functionality will check whether it has an entry (pw, m) for id in its $\mathsf{storage}$. If the password from the entry matches the entered password, it sends m to the party that initiated the retrieval.
  3. If the simulator cannot find an entry for either $\mathsf{injectedMessages}$ or $\mathsf{storageHistory}$, it indicates failure by sending $(\texttt{FinishRetrieve}, ctr, T, \texttt{failed}, (\texttt{true}, 0, \epsilon, \epsilon))$ to the ideal functionality.
- **(13) Special Queries to the Random Oracles OTP and MAC:** To ensure that the retrieval from dummy records generated in (6) is successful for correct passwords, the simulator must answer queries to $\mathcal{F}_{\text{ro}}^{\mathsf{OTP}}$ and $\mathcal{F}_{\text{ro}}^{\mathsf{MAC}}$ in a way that allows this.
  Upon receiving a query $(([u]_t, pw, id, n), l)$ to $\mathcal{F}_{\text{ro}}^{\mathsf{OTP}}$ that is potentially consistent with a dummy record $\mathsf{storageHistory}[id, i] = (c_1, c_2, n)$, it has to check whether the password matches the one entered to the ideal functionality on creating the corresponding dummy record. This can be done by sending $(\texttt{PwGuessStart}, id, i, RLset)$ and $(\texttt{PwGuessFinish}, id, i, pw)$ to the ideal functionality. If the password is the same, the ideal functionality responds with the corresponding message m'', and the simulator stores $h \leftarrow c_1 \oplus m''$ in $\mathsf{hashHistory}_{\mathsf{OTP}}$ for $([u]_t, pw, id, n), |c_1|)$ and c_2 in $\mathsf{hashHistory}_{\mathsf{MAC}}$ for $([u]_t, m'', pw, id, n)$. If the length of the returned message $|m''|$ is not equivalent to the entered length l, it samples a fresh value as described in (3). Note that for using the $\texttt{PwGuessFinish}$ functionality, we must provide a set of $t - n_c$ honest ratelimiters with n_c being the number of currently corrupted ratelimiters. Furthermore, for every ratelimiter in the set, it must hold that $\mathsf{retrieveRate}[rl] > 0$. We find such ratelimiters with the help of $\mathsf{retrieveRequests}$ that keeps track of queries to the ratelimiters not used in (12).

(1) Parameter Generation
Upon the first activation of the simulator, it runs the initialization defined by the real protocol in $\mathcal{F}_{\text{init\&rotateKey}}$ to obtain group descriptions of $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t$ and the key pair $(\mathsf{sk}, \mathsf{pk})$ that is shared among the server and the ratelimiters. Furthermore, it initializes the ideal authenticated channel functionality $\mathcal{F}_{\text{auth}}$. In addition to the internal states of the real protocol, the following data structures are initialized:

- $\mathsf{storageHistory} : \{0,1\}^* \times \mathbb{N} \to (\{0,1\}^*)^4 \cup \{\bot\}$
- $\mathsf{injectedMessages} \subseteq (\{0,1\}^*)^6$
- $\mathsf{retrieveRequests} \subseteq (\{0,1\}^*)^4 \times \mathbb{G}_t$
- $\mathsf{registeredGuesses} \subset \{0,1\}^* \times \mathbb{N}$
- $\mathsf{hashHistory}_1 \subseteq \{0,1\}^* \times \mathbb{Z}_q^* \times \mathbb{G}_1$
- $\mathsf{hashHistory}_2 \subseteq \{0,1\}^* \times \mathbb{Z}_q^* \times \mathbb{G}_2$
- $\mathsf{hashHistory}_{\mathsf{OTP}} \subseteq \{0,1\}^* \times \mathbb{N} \times \{0,1\}^\eta$
- $\mathsf{hashHistory}_{\mathsf{MAC}} \subseteq \{0,1\}^* \times \{0,1\}^\eta$
- $\mathsf{hashHistory}_{\mathsf{N}} \subseteq \{0,1\}^* \times \{0,1\}^\eta$

(2) Forwarding during Corruption
Upon receiving a message from `I/O` addressed to the corrupted server, the simulator forwards the message to the environment if it is of the form $(\texttt{Store}, \_, \_, \_), (\texttt{Retrieve}, \_, \_)$, or $\texttt{Received}$. Upon receiving a message $m = (\texttt{Fwd}, (pid, sid, role), (\texttt{Retrieve}, id, m))$ from the environment addressed to the corrupted server, the simulator sends $(\texttt{CorruptedRetrieve}, id, m, (pid, sid, role))$ to the ideal functionality. Upon receiving a message $m = (\texttt{Fwd}, (pid, sid, role), m')$ from the environment addressed to the corrupted server, the simulator checks whether *role* is a subroutine of $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$. If the checks pass, it forwards the message to $(pid, sid, role)$. Upon receiving a message $m = (\texttt{Fwd}, (pid, sid, role), m')$ from the environment to a corrupted ratelimiter, the simulator forwards m' to $(pid, sid, role)$. All other messages addressed to a corrupted ratelimiter are forwarded to the environment. This is a perfect simulation of the real protocol $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$.

(3) Answering Queries to the Random Oracles
Upon receiving a query to one of the random oracles, the simulator checks whether an entry for the specific input exists in the corresponding $\mathsf{hashHistory}$. If not, and the query was for $\mathcal{F}^1_{\text{ro}}$ or $\mathcal{F}^2_{\text{ro}}$, it samples $a \xleftarrow{\$} \mathbb{Z}_q^*$ and stores a and $[a]_{\{1,2\}}$ with the input in $\mathsf{hashHistory}_{\{1,2\}}$. Otherwise, it samples a fresh value from the codomain and stores it with the input in $\mathsf{hashHistory}$. It answers the query with the value stored in $\mathsf{hashHistory}$. Some exceptions to the above sampling are specified in (13).

(4) Storing Initialization (server)
Upon receiving $(\mathsf{store}, id, |m|)$ from $\mathcal{F}_{\mathsf{TPHE}} : \mathcal{S}$, the simulator follows the protocol as described in $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$ except for steps that depend on the password or the message as they are unknown to the simulator. Consequently, it samples $a \xleftarrow{\$} \mathbb{Z}_q^*$ and sets $[p]_2 \leftarrow a \cdot [1]_2$ (instead of computing $[h]_2$ with $\mathcal{F}^2_{\text{ro}}$ and blinding it with a random factor $r \xleftarrow{\$} \mathbb{Z}_q^*$). This is equivalent to the behavior of $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$, as there exists a blinding factor r such that $\mathsf{H}_2(pw, n)^r = [p]_2$ for every password unless $\mathcal{F}^2_{\text{ro}}$ outputs the identity element $[1]_2$. We show later on that this happens with negligible probability. It stores $(\epsilon, \epsilon, \epsilon, \epsilon)$ in $\mathsf{storageHistory}[id, i]$ with $i \in \mathbb{N}$ minimal such that $\mathsf{storageHistory}[id, i] = \bot$ and adds $(i, |m|, a, n, T, p)$ instead of (pw, m, r, n, T, p) in $\mathsf{reqQueue}_{\mathsf{Enc}}[id]$. If the freshly generated nonce n already appears in any $\mathsf{hashHistory}_i, i \in [\mathsf{OTP}, \mathsf{MAC}]$, the simulator aborts, and we call this event $\mathsf{FAIL1}$.

(5) Storing Response (ratelimiter)
Upon receiving $(\texttt{Received}, (0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S}), (\texttt{EncRequest}, id, [p]_2, \{(rl, nonce_{rl})\}_{rl \in [1,\mathsf{n}]}))$ from the simulated authenticated channel $(\_, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{auth}} : \texttt{auth})$, the simulator strictly follows the behaviour defined in $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$.

(6) Storing Finalization (server)
Upon receiving $(\texttt{Received}, (rl, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R}), (\texttt{FinalizeEnc}, id, \pi, u, n))$ from the simulated authenticated channel $(\_, \_, \mathcal{F}_{\text{auth}} : \texttt{auth})$, the simulator follows the behaviour defined in $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$, until all ratelimiters $rl \in T$ responded with a valid proof. Instead of computing c_1, c_2 according to the protocol with the random oracles $\mathcal{F}^{\mathsf{OTP}}_{\text{ro}}$ and $\mathcal{F}^{\mathsf{MAC}}_{\text{ro}}$, it samples c_1, c_2 uniformly at random from $\{0,1\}^{|m|}$ and $\{0,1\}^\eta$ respectively. The simulator continues with the protocol and stores the resulting values (c_1, c_2, n) in $\mathsf{storage}$ and $\mathsf{storageHistory}[id, i]$ with i coming from $\mathsf{reqQueue}_{\mathsf{Enc}}[id]$.

(7) Steal Database
Upon receiving `stealDB` from the environment, the simulator responds with $\mathsf{storage}$. Note that the leaking of the storage leads to the environment learning simulated records. To ensure that future protocol interactions with the simulated records work correctly, we implement additional steps in (13).

Fig. 24: Description of the UC simulator Sim for TPHE (Part 1).

(8) Honest Key Rotation
Upon receiving $(\texttt{ChangeCorruption}, corr_{\mathcal{S}}, corrIDs_{\mathcal{R}})$ from the environment, the simulator checks whether the server has been corrupted in the previous epoch. If it was, the simulator checks whether $t - n_c$ entries for any (id, n) exist in $\mathsf{retrieveRequests}$ with matching $[o]_t$ values and distinct ratelimiters which form $RLset$ (n_c is the number of corrupted ratelimiters). If so, it checks whether an $i \in \mathbb{N}$ exists such that $\mathsf{storageHistory}[id, i] = (\_, \_, n)$. If so, it removes the entries from $\mathsf{retrieveRequests}$ and sends $(\texttt{PwGuessStart}, id, i, RLset)$ to the ideal functionality to ensure that the valid set of ratelimiters can be used in future epochs for password guesses. Once it cannot find such entries anymore, it removes all remaining entries from $\mathsf{retrieveRequests}$.
Now that potential password guesses are registered, the simulator follows the protocol defined in $\mathcal{F}_{\text{init\&rotateKey}}$ and informs the ideal functionality with $(\texttt{ChangeCorruption}, (corr))$ about the new corruption status of each party. The following invocation of each simulated ratelimiter with $(\texttt{RotateKey}, corr, s)$ leads to the simulator executing the protocol as specified in $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$. The following invocation of the simulated server with $(\texttt{RotateKey}, corr_{\mathcal{S}}, sk, pk, pk_1, \ldots, pk_{\mathsf{n}}, nonces)$ also leads to the simulator executing the protocol as specified in $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$. Note that in case the server gets corrupted, the leaking of the storage leads to the environment learning simulated records. To ensure that future protocol interactions with the simulated records work correctly, we implement additional steps in (13).
Furthermore, if the server has been corrupted in the last epoch, the environment provides the simulator with the server's state, consisting of the $\mathsf{storage}$. To ensure correct retrieval in future protocol runs, the simulation checks the validity of the given records. It checks the entry (c_1, c_2, n) by checking whether some $[u]_t, m, pw$ exists such that $(([u]_t, m, pw, id, n), c_2) \in \mathsf{hashHistory}_{\mathsf{MAC}}$. If more than one entry exists, the simulator aborts, and we denote this event as $\mathsf{FAIL3}$. If it finds an entry, it computes $h \leftarrow c_1 \oplus m$ and also checks whether $(([u]_t, pw, id, n), |m|, h) \in \mathsf{hashHistory}_{\mathsf{OTP}}$. Finally, it checks whether $[u]_t$ matches $\mathsf{sk} \cdot \mathsf{H}_1(id, n) \cdot \mathsf{H}_2(pw, n)$. If all the above checks pass, it adds $(id, n, [u]_t, m, pw)$ to $\mathsf{injectedMessages}$. We denote the event that the environment guessed c_2 correctly without querying $\mathcal{F}^{\mathsf{MAC}}_{\text{ro}}$ as $\mathsf{FAIL4}$.

(9) Dishonest Key Rotation
Upon receiving $(\texttt{RotateKey}^{\mathcal{S}}_{\mathcal{A}}, rl, s)$ or $(\texttt{RotateKey}_{\mathcal{A}})$ addressed to $\mathcal{F}_{\text{init\&rotateKey}}$, the simulator follows the protocol exactly as described in $\mathcal{F}_{\text{init\&rotateKey}}$. Note that because the rotation is dishonest, no corruption status is changed; thus, the ideal functionality is not informed about the rotation.

(10) Retrieving Initialization (server)
Upon receiving $(\texttt{Retrieve}, \mathsf{retrieveCounter}, id)$ from the ideal functionality, the simulator follows the protocol specification as described in $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$, except that instead of computing $[p]_2$ as the product of a random scalar and $[h]_2$ received upon sending pw', n to $\mathcal{F}^2_{\text{ro}}$, it samples $a \xleftarrow{\$} \mathbb{Z}^*_q$ and sets $[p]_2 \leftarrow a \cdot [1]_2$.

(11) Retrieving Response (ratelimiter)
Upon receiving $(\texttt{Received}, (0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S}), (\texttt{DecRequest}, ctr, id, [p]_2, n))$ from the simulated authenticated channel $(\_, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{auth}} : \texttt{auth})$, the simulator strictly follows the behaviour defined in $\mathcal{P}^{\mathcal{R}}_{\text{PHE}}$. Finally, it adds $(rl, ctr, id, n, [o]_t)$ to $\mathsf{retrieveRequests}$.

(12) Retrieving Finalization (server)
Upon receiving $(\texttt{Received}, (rl, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R}), (\texttt{FinalizeDec}, ctr, id, \pi, [u]_t))$ from the simulated authenticated channel $(\_, \_, \mathcal{F}_{\text{auth}} : \texttt{auth})$, the simulator follows the behaviour defined in $\mathcal{P}^{\mathcal{S}}_{\text{PHE}}$, until all ratelimiters $rl \in T$ responded with a valid proof. We now differentiate between three cases, where n is taken from $\mathsf{storage}[id]$:

- $\exists\ ([u]_t, m, pw)\ s.t.\ (id, n, [u]_t, m, pw) \in \mathsf{injectedMessages}$: The simulator responds to the ideal functionality with $(\texttt{FinishRetrieve}, ctr, T, \texttt{success}, (\texttt{false}, 0, pw, m))$.
- $\nexists\ ([u]_t, m, pw)\ s.t.\ (id, n, [u]_t, m, pw) \in \mathsf{injectedMessages} \wedge \exists i \in \mathbb{N}\ s.t.\ \mathsf{storageHistory}[id, i] = (c_1, c_2, n)$: The simulator responds to the ideal functionality with $(\texttt{FinishRetrieve}, ctr, T, \texttt{success}, (\texttt{true}, i, \epsilon, \epsilon))$.
- $\nexists\ ([u]_t, m, pw)\ s.t.\ (id, n, [u]_t, m, pw) \in \mathsf{injectedMessages} \wedge \nexists i \in \mathbb{N}\ s.t.\ \mathsf{storageHistory}[id, i] = (c_1, c_2, n)$: The simulator responds to the ideal functionality with $(\texttt{FinishRetrieve}, ctr, T, \texttt{failed}, (\texttt{true}, 0, \epsilon, \epsilon))$.

Finally, the simulator removes $(rl, ctr, id, n, [o]_t)$ from $\mathsf{retrieveRequests}$ for all $rl \in T$ with $[o]_t \leftarrow \mathsf{H}_1(id, n) \cdot [p]_2$ and $[p]_2$ taken from $\mathsf{reqQueue}_{\mathsf{Dec}}$.

Fig. 25: Description of the UC simulator Sim for TPHE (Part 2).

(13) Special Queries to the Random Oracles OTP and MAC
As described in (3), the simulator samples fresh values from the corresponding codomain if the query has not been seen yet. For the following cases, the simulator does not simply sample a fresh value but works as described in the following:

- Upon receiving $(([u]_t, pw, id, n), l)$ to $\mathcal{F}_{\text{ro}}^{\mathsf{OTP}}$, the simulator checks whether $\mathsf{sk} \cdot \mathsf{H}_1(id, n) \cdot \mathsf{H}_2(pw, n) = [u]_t$. If so, it checks whether $\exists i \in \mathbb{N}, c_1, c_2 \ s.t.\ \mathsf{storageHistory}[id, i] = (c_1, c_2, n) \wedge |c_1| = l$ and sends $(\texttt{PwGuessStart}, id, i, RLset)$ to the ideal functionality. The set of ratelimiters $RLset$ is obtained by looking for $t - n_c$ ratelimiters rl such that $(rl_,, id, n, [o]_t) \in \mathsf{retrieveRequests}$ with n_c being the number of currently corrupted ratelimiters and $[o]_t \leftarrow \mathsf{H}(id, n) \cdot \mathsf{H}(pw, n)$. The entries are then removed from $\mathsf{retrieveRequests}$. If no such set of ratelimiters exists, it checks whether (id, i) exists in $\mathsf{registeredGuesses}$ and removes it. We call the event that no such entry exists $\mathsf{FAIL2}$, and the simulator aborts. The simulator sends $(\texttt{PwGuessFinish}, id, i, pw)$ to the ideal functionality and checks, upon receiving $(\texttt{PwGuessFinish}, m')$ from the ideal functionality, whether $m' = (\texttt{Correct}, m'')$ and stores $h \leftarrow c_1 \oplus m''$ in $\mathsf{hashHistory}_{\mathsf{OTP}}$ for $([u]_t, pw, id, n), |c_1|$ and c_2 in $\mathsf{hashHistory}_{\mathsf{MAC}}$ for $([u]_t, m'', pw, id, n)$. If $l \neq |m''|$, it continues as described in (3). If one of the checks above fails, it continues as described in (3).
- Upon receiving $([u]_t, m, pw, id, n)$ to $\mathcal{F}_{\text{ro}}^{\mathsf{MAC}}$, the simulator checks whether $\mathsf{sk} \cdot \mathsf{H}_1(id, n) \cdot \mathsf{H}_2(pw, n) = [u]_t$. If so, it checks whether $\exists i \in \mathbb{N}, c_1, c_2 \ s.t.\ \mathsf{storageHistory}[id, i] = (c_1, c_2, n) \wedge |c_1| = |m|$ and sends $(\texttt{PwGuessStart}, id, i, RLset)$ to the ideal functionality as described above. The simulator sends $(\texttt{PwGuessFinish}, id, i, pw)$ to the ideal functionality and checks, upon receiving $(\texttt{PwGuessFinish}, m')$ from the ideal functionality, whether $m' = (\texttt{Correct}, m'')$ and stores $h \leftarrow c_1 \oplus m''$ in $\mathsf{hashHistory}_{\mathsf{OTP}}$ for $([u]_t, pw, id, n), |c_1|$ and c_2 in $\mathsf{hashHistory}_{\mathsf{MAC}}$ for $([u]_t, m'', pw, id, n)$. If $m \neq m''$, it continues as described in (3). If one of the checks above fails, it continues as described in (3).

Fig. 26: Description of the UC simulator Sim for TPHE (Part 3).

If we cannot find a set of $t - n_c$ ratelimiters, we check whether we already registered a password guess for id, i in (8). The simulation aborts if no guess has been registered.
Upon receiving a query $([u]_t, m, pw, id, n)$ to $\mathcal{F}_{\text{ro}}^{\mathsf{MAC}}$ that is potentially consistent with a dummy record $\mathsf{storageHistory}[id, i] = (c_1, c_2, n)$, it sends a password guess as described for $\mathcal{F}_{\text{ro}}^{\mathsf{OTP}}$ to the ideal functionality. If it gets a message m'' back, it stores $h \leftarrow c_1 \oplus m''$ in $\mathsf{hashHistory}_{\mathsf{OTP}}$ for $([u]_t, pw, id, n), |c_1|)$ and c_2 in $\mathsf{hashHistory}_{\mathsf{MAC}}$ for $([u]_t, m'', pw, id, n)$. If the returned message m'' is not the entered m, it samples a fresh value as described in (3).

We note that it is easy to see that (i) $\{\mathsf{Sim}, \mathcal{F}_{\text{PHE}}\}$ is environmentally bounded [11] and (ii) Sim is a responsive simulator for $\mathcal{F}_{\text{PHE}}$, *i.e.*, restricting messages from $\mathcal{F}_{\text{PHE}}$ are answered immediately as long as $\{\mathsf{Sim}, \mathcal{F}_{\text{PHE}}\}$ runs with a responsive environment. We now argue that $\mathcal{R}$ and $\{\mathsf{Sim}, \mathcal{F}_{\text{PHE}}\}$ are indeed indistinguishable for any (responsive) environment $\mathcal{E} \in \mathsf{Env}(\mathcal{R})$.

Now, let $\mathcal{E} \in \mathsf{Env}(\mathcal{P}_{\text{PHE}})$[12] be an arbitrary but fixed environment. In the following, we will argue by induction that all interactions with $\mathcal{P}_{\text{PHE}}$ and $\{\mathsf{Sim}, \mathcal{F}_{\text{PHE}}\}$ result in identical behavior towards $\mathcal{E}$, *i.e.*, both systems are indistinguishable. At the start of a run, there were no interactions on the network, resp. I/O, interface yet. Thus, the induction base case holds true. In the following, assume that all network, resp. I/O, interactions so far have resulted in the same behavior visible towards the environment in both the real and ideal world.

To show the indistinguishability between the real and the ideal world, we will prove by induction that if the real and the ideal world are *in sync*, they respond to a message received on either the I/O or the NET interface indistinguishably and are still in sync afterward. We begin by introducing the notion of *synchronized states*. The real and the ideal world are *in sync* or *synchronized* if the invariants defined in Figure 27 hold and the internal variables not covered in the invariants are identically distributed. Note that the invariants must only hold if the server is honest. We use $\sim$ to denote an identical distribution and $\mathsf{H}_i(x), i \in \{1, 2, \mathsf{OTP}, \mathsf{MAC}, \mathsf{N}\}$ to denote the entry in $\mathsf{hashHistory}_i$ for input x.

Now, we show for each message received from the environment that the real and the ideal world respond indistinguishably and are in sync afterward if they were in sync before. We do so by depicting the actions of the real protocol next to the actions of the simulator and the ideal functionality in the ideal world. To improve readability, we only list commands relevant to synchronization and output and completely omit procedures where the simulator follows exactly the protocol description. In this part of the proof, we assume that no $\mathsf{FAIL1}$ event occurs and prove later that this is the case with overwhelming probability. An additional event that we assume not to happen is that the random oracle $\mathcal{F}_{\text{ro}}^2$ outputs the neutral element. We denote this even by $\mathsf{FAIL5}$.

[11] As all algorithms are in polynomial time and parameters ensure that the execution of non-a-priori fixed code finishes in polynomial time.
[12] For some system $\mathcal{Q}$, we denote by $\mathsf{Env}(\mathcal{Q})$ the set of all environments $\mathcal{E}$ that can be connected to $\mathcal{Q}$.

1. $\forall id \in \mathsf{storage}^{\mathsf{Real}} : (c_1^{\mathsf{Real}}, c_2^{\mathsf{Real}}, n^{\mathsf{Real}}) \sim (c_1^{\mathsf{Sim}}, c_2^{\mathsf{Sim}}, n^{\mathsf{Sim}}) \wedge [$
   $[\exists i \in \mathbb{N}\ s.t.\ (id \in \mathsf{correctMessageIDs}^{\mathsf{Ideal}} \implies \mathsf{storageHistory}^{\mathsf{Ideal}}[id, i+1] = \bot)$
   $\wedge\, \mathsf{H}^{\mathsf{Real}}_{\mathsf{OTP}}(\mathsf{sk}^{\mathsf{Real}} \cdot \mathsf{H}^{\mathsf{Real}}_1(pw^{\mathsf{Ideal}}, n^{\mathsf{Real}}) \cdot \mathsf{H}^{\mathsf{Real}}_2(id, n^{\mathsf{Real}}), pw^{\mathsf{Ideal}}, id, n^{\mathsf{Real}}) = c_1^{\mathsf{Real}} \oplus m^{\mathsf{Ideal}}$
   $\wedge\, \mathsf{H}^{\mathsf{Real}}_{\mathsf{MAC}}(\mathsf{sk}^{\mathsf{Real}} \cdot \mathsf{H}^{\mathsf{Real}}_1(pw^{\mathsf{Ideal}}, n^{\mathsf{Real}}) \cdot \mathsf{H}^{\mathsf{Real}}_2(id, n^{\mathsf{Real}}), m^{\mathsf{Ideal}}, pw^{\mathsf{Ideal}}, id, n^{\mathsf{Real}}) = c_2^{\mathsf{Real}}$
   $\wedge\, \mathsf{H}^{\mathsf{Sim}}_{\mathsf{OTP}}(\mathsf{sk}^{\mathsf{Sim}} \cdot \mathsf{H}^{\mathsf{Sim}}_1(pw^{\mathsf{Ideal}}, n^{\mathsf{Sim}}) \cdot \mathsf{H}^{\mathsf{Sim}}_2(id, n^{\mathsf{Sim}}), pw^{\mathsf{Ideal}}, id, n^{\mathsf{Sim}}) \in \{c_1^{\mathsf{Sim}} \oplus m^{\mathsf{Ideal}}, \bot\}$
   $\wedge\, \mathsf{H}^{\mathsf{Sim}}_{\mathsf{MAC}}(\mathsf{sk}^{\mathsf{Sim}} \cdot \mathsf{H}^{\mathsf{Sim}}_1(pw^{\mathsf{Ideal}}, n^{\mathsf{Sim}}) \cdot \mathsf{H}^{\mathsf{Sim}}_2(id, n^{\mathsf{Sim}}), m^{\mathsf{Ideal}}, pw^{\mathsf{Ideal}}, id, n^{\mathsf{Sim}}) \in \{c_2^{\mathsf{Sim}}, \bot\}]$
   $\vee[\exists (id^{\mathsf{Sim}}, n^{\mathsf{Sim}}, u, m^{\mathsf{Sim}}, pw^{\mathsf{Sim}}) \in \mathsf{injectedMessages}^{\mathsf{Sim}}]\ s.t.\ id = id^{\mathsf{Sim}}$
   $\wedge\, c_1^{\mathsf{Real}} = \mathsf{H}^{\mathsf{Real}}_{\mathsf{OTP}}(\mathsf{sk}^{\mathsf{Real}} \cdot \mathsf{H}^{\mathsf{Real}}_1(pw^{\mathsf{Sim}}, n^{\mathsf{Real}}) \cdot \mathsf{H}^{\mathsf{Real}}_2(id, n^{\mathsf{Real}}), pw^{\mathsf{Sim}}, id, n^{\mathsf{Real}}) \oplus m^{\mathsf{Sim}}$
   $\wedge\, c_2^{\mathsf{Real}} = \mathsf{H}^{\mathsf{Real}}_{\mathsf{MAC}}(\mathsf{sk}^{\mathsf{Real}} \cdot \mathsf{H}^{\mathsf{Real}}_1(pw^{\mathsf{Sim}}, n^{\mathsf{Real}}) \cdot \mathsf{H}^{\mathsf{Real}}_2(id, n^{\mathsf{Real}}), m^{\mathsf{Sim}}, pw^{\mathsf{Sim}}, id, n^{\mathsf{Real}})$
   $\wedge\, c_1^{\mathsf{Sim}} = \mathsf{H}^{\mathsf{Sim}}_{\mathsf{OTP}}(\mathsf{sk}^{\mathsf{Sim}} \cdot \mathsf{H}^{\mathsf{Sim}}_1(pw^{\mathsf{Sim}}, n^{\mathsf{Sim}}) \cdot \mathsf{H}^{\mathsf{Sim}}_2(id, n^{\mathsf{Sim}}), pw^{\mathsf{Sim}}, id, n^{\mathsf{Sim}}) \oplus m^{\mathsf{Sim}}$
   $\wedge\, c_2^{\mathsf{Sim}} = \mathsf{H}^{\mathsf{Sim}}_{\mathsf{MAC}}(\mathsf{sk}^{\mathsf{Sim}} \cdot \mathsf{H}^{\mathsf{Sim}}_1(pw^{\mathsf{Sim}}, n^{\mathsf{Sim}}) \cdot \mathsf{H}^{\mathsf{Sim}}_2(id, n^{\mathsf{Sim}}), m^{\mathsf{Sim}}, pw^{\mathsf{Sim}}, id, n^{\mathsf{Sim}})]$
   $\vee[\nexists (pw, m)\ s.t.$
   $c_1^{\mathsf{Real}} = \mathsf{H}^{\mathsf{Real}}_{\mathsf{OTP}}(\mathsf{sk}^{\mathsf{Real}} \cdot \mathsf{H}^{\mathsf{Real}}_1(pw, n^{\mathsf{Real}}) \cdot \mathsf{H}^{\mathsf{Real}}_2(id, n^{\mathsf{Real}}), pw, id, n^{\mathsf{Real}}) \oplus m$
   $\wedge\, c_2^{\mathsf{Real}} = \mathsf{H}^{\mathsf{Real}}_{\mathsf{MAC}}(\mathsf{sk}^{\mathsf{Real}} \cdot \mathsf{H}^{\mathsf{Real}}_1(pw, n^{\mathsf{Real}}) \cdot \mathsf{H}^{\mathsf{Real}}_2(id, n^{\mathsf{Real}}), m, pw, id, n^{\mathsf{Real}})$
   $\wedge\, c_1^{\mathsf{Sim}} = \mathsf{H}^{\mathsf{Sim}}_{\mathsf{OTP}}(\mathsf{sk}^{\mathsf{Sim}} \cdot \mathsf{H}^{\mathsf{Sim}}_1(pw, n^{\mathsf{Sim}}) \cdot \mathsf{H}^{\mathsf{Sim}}_2(id, n^{\mathsf{Sim}}), pw, id, n^{\mathsf{Sim}}) \oplus m$
   $\wedge\, c_2^{\mathsf{Sim}} = \mathsf{H}^{\mathsf{Sim}}_{\mathsf{MAC}}(\mathsf{sk}^{\mathsf{Sim}} \cdot \mathsf{H}^{\mathsf{Sim}}_1(pw, n^{\mathsf{Sim}}) \cdot \mathsf{H}^{\mathsf{Sim}}_2(id, n^{\mathsf{Sim}}), m, pw, id, n^{\mathsf{Sim}})]]$
   with $\mathsf{storage}^{\mathsf{Real}}[id] = (c_1^{\mathsf{Real}}, c_2^{\mathsf{Real}}, n^{\mathsf{Real}})$,
   $\mathsf{storage}^{\mathsf{Sim}}[id] = (c_1^{\mathsf{Sim}}, c_2^{\mathsf{Sim}}, n^{\mathsf{Sim}})$,
   $\mathsf{storageHistory}^{\mathsf{Ideal}}[id, i] = (pw^{\mathsf{Ideal}}, m^{\mathsf{Ideal}})$
2. $\forall id \in \mathsf{reqQueue}^{\mathsf{Real}}_{\mathsf{Enc}} : (pw^{\mathsf{Real}}, m^{\mathsf{Real}}, T^{\mathsf{Real}}) = (pw^{\mathsf{Ideal}}, m^{\mathsf{Ideal}}, T^{\mathsf{Sim}}) \wedge |m^{\mathsf{Real}}| = |m|^{\mathsf{Sim}} = |m^{\mathsf{Ideal}}|$
   $\wedge i^{\mathsf{Sim}} = i^{\mathsf{Ideal}} \wedge n^{\mathsf{Real}} \sim n^{\mathsf{Sim}}$
   with $\mathsf{reqQueue}^{\mathsf{Real}}_{\mathsf{Enc}}[id] = (pw^{\mathsf{Real}}, m^{\mathsf{Real}}, r, n^{\mathsf{Real}}, T^{\mathsf{Real}})$,
   $\mathsf{reqQueue}^{\mathsf{Sim}}_{\mathsf{Enc}}[id] = (i^{\mathsf{Sim}}, |m|^{\mathsf{Sim}}, n^{\mathsf{Real}}, T^{\mathsf{Real}}, p)$
   $\mathsf{storageHistory}^{\mathsf{Ideal}}[id, i^{\mathsf{Ideal}}] = (pw^{\mathsf{Ideal}}, m^{\mathsf{Ideal}})$ with i^{Ideal} max
3. $\forall \mathsf{retrieveCounter} \in \mathsf{reqQueue}^{\mathsf{Real}}_{\mathsf{Dec}} : \exists\ \mathsf{retrieveCounter}' \in \mathsf{reqQueue}^{\mathsf{Ideal}}_{\mathsf{Dec}}\ s.t.\ (pw'^{\mathsf{Real}}, T^{\mathsf{Real}}) = (pw'^{\mathsf{Ideal}}, T^{\mathsf{Sim}})$
   $\wedge (id^{\mathsf{Real}}, caller^{\mathsf{Real}}) = (id^{\mathsf{Sim}}, caller^{\mathsf{Sim}}) = (id^{\mathsf{Ideal}}, caller^{\mathsf{Ideal}}) \wedge (n^{\mathsf{Real}}, p^{\mathsf{Real}}) \sim (n^{\mathsf{Sim}}, p^{\mathsf{Sim}})$
   with $\mathsf{reqQueue}^{\mathsf{Real}}_{\mathsf{Dec}}[\mathsf{retrieveCounter}] = (id^{\mathsf{Real}}, pw^{\mathsf{Real}}, r, n^{\mathsf{Real}}, T^{\mathsf{Real}}, p^{\mathsf{Real}}, caller^{\mathsf{Real}})$,
   $\mathsf{reqQueue}^{\mathsf{Sim}}_{\mathsf{Dec}}[\mathsf{retrieveCounter}] = (id^{\mathsf{Sim}}, n^{\mathsf{Sim}}, T^{\mathsf{Sim}}, p^{\mathsf{Sim}}, caller^{\mathsf{Sim}})$
   $\mathsf{reqQueue}^{\mathsf{Ideal}}_{\mathsf{Dec}}[\mathsf{retrieveCounter}'] = (id^{\mathsf{Ideal}}, \mathsf{storageHistory}[id, i]^{\mathsf{Ideal}}, pw'^{\mathsf{Ideal}}, caller^{\mathsf{Ideal}})$
4. $\forall id \in \mathsf{retrieveRate}^{\mathsf{Real}}, rl \notin \mathsf{currentlyCorrupted} : \mathsf{retrieveRate}^{\mathsf{Real}}_{rl}[id] \leq$
   $\leq \mathsf{retrieveRate}^{\mathsf{Ideal}}_{rl}[id] - |\{(rl, \_, id, \_, \_)\} \subseteq \mathsf{retrieveRequests}^{\mathsf{Sim}}|$
5. $\forall id, i \in \mathsf{storageHistory}^{\mathsf{Sim}} : \mathsf{H}^{\mathsf{Sim}}_{\mathsf{OTP}}(\mathsf{sk}^{\mathsf{Sim}} \cdot \mathsf{H}^{\mathsf{Sim}}_1(pw^{\mathsf{Ideal}}, n^{\mathsf{Sim}}) \cdot \mathsf{H}^{\mathsf{Sim}}_2(id, n^{\mathsf{Sim}}), pw^{\mathsf{Ideal}}, id, n^{\mathsf{Sim}}) \in \{c_1^{\mathsf{Sim}} \oplus m^{\mathsf{Ideal}}, \bot\}$
   $\wedge \mathsf{H}^{\mathsf{Sim}}_{\mathsf{MAC}}(\mathsf{sk}^{\mathsf{Sim}} \cdot \mathsf{H}^{\mathsf{Sim}}_1(pw^{\mathsf{Ideal}}, n^{\mathsf{Sim}}) \cdot \mathsf{H}^{\mathsf{Sim}}_2(id, n^{\mathsf{Sim}}), m^{\mathsf{Ideal}}, pw^{\mathsf{Ideal}}, id, n^{\mathsf{Sim}}) \in \{c_2^{\mathsf{Sim}}, \bot\}]$
   with $\mathsf{storageHistory}^{\mathsf{Sim}}[id, i] = (c_1^{\mathsf{Sim}}, c_2^{\mathsf{Sim}}, n^{\mathsf{Sim}})$,
   $\mathsf{storageHistory}^{\mathsf{Ideal}}[id, i] = (pw^{\mathsf{Ideal}}, m^{\mathsf{Ideal}})$

Fig. 27: Invariants for the notion of synchronization between the ideal and real world

Server

– $(\texttt{Store}, id, pw, m)$

Real

$\mathsf{reqQueue}^{\mathsf{Real}}_{\mathsf{Enc}}[id] \leftarrow (pw, m, r, n, T, p)$

Ideal

$\mathsf{storageHistory}^{\mathsf{Ideal}}[id, i] \leftarrow (pw, m)$
  with i max $s.t.$ $\mathsf{storageHistory}^{\mathsf{Ideal}}[id, i] = \bot$
$\mathsf{reqQueue}^{\mathsf{Sim}}_{\mathsf{Enc}}[id] \leftarrow (i, |m|, n, T, p)$

These actions preserve the synchronization of the $\mathsf{reqQueue}_{\mathsf{Enc}}$ as described in invariant 2.

– $(\texttt{FinalizeEnc}, id, \pi, u, n)$

Real

$(pw, m, r, n, T) \leftarrow \mathsf{reqQueue}^{\mathsf{Real}}_{\mathsf{Enc}}[id]$
if verify $(([1]_t, [o]_t, \mathsf{pk}, [u]_t), \pi) = 0$: **return** $\bot$
$c_1 \leftarrow \mathsf{H}_{\mathsf{OTP}}(u, pw, id, n) \oplus m$
$c_2 \leftarrow \mathsf{H}_{\mathsf{MAC}}(u, m, pw, id, n)$
$\mathsf{storage}[id] \leftarrow (c_1, c_2, n)$

Ideal

$(i, |m|, n, T, p) \leftarrow \mathsf{reqQueue}^{\mathsf{Sim}}_{\mathsf{Enc}}[id]$
if verify $(([1]_t, [o]_t, \mathsf{pk}, [u]_t), \pi) = 0$: **return** $\bot$
$c_1 \xleftarrow{\$} \{0,1\}^{|m|}$
$c_2 \xleftarrow{\$} \{0,1\}^{\eta}$
$\mathsf{storage}[id] \leftarrow (c_1, c_2, n)$
$\mathsf{storageHistory}^{\mathsf{Sim}}[id, i] \leftarrow (c_1, c_2, n)$

We know from invariant 2 that there is already an entry $\mathsf{storageHistory}^{\mathsf{Ideal}}[id, i] = (pw, m)$ and because of the NIZK, we know that $[u]_t$ is well formed. Therefore, case 1 of invariant 1 is fulfilled, and hence the synchronization of the $\mathsf{storage}$ is preserved. Furthermore, the persistence of the $\mathsf{storageHistory}$ as described in invariant 5 is also fulfilled. Note that if, for some reason, encryption was started but did not end with a valid entry in $\mathsf{storage}^{\mathsf{Real}}$ and $\mathsf{storage}^{\mathsf{Sim}}$, there will still be a valid entry in $\mathsf{storageHistory}^{\mathsf{Ideal}}$. However, there is also an entry in $\mathsf{storageHistory}^{\mathsf{Sim}}$ (guaranteed by invariant 2) that consists only of empty strings, which leads to a failed decryption, just like in the real world.

– $(\texttt{Retrieve}, id, pw')$

Real

$\mathsf{reqQueue}^{\mathsf{Real}}_{\mathsf{Dec}}[\mathsf{retrieveCounter}] \leftarrow (id, pw', r, n, T, p, caller)$

Ideal

$\mathsf{reqQueue}^{\mathsf{Ideal}}_{\mathsf{Dec}}[\mathsf{retrieveCounter}] \leftarrow (id, pw', caller)$
$\mathsf{reqQueue}^{\mathsf{Sim}}_{\mathsf{Dec}}[\mathsf{retrieveCounter}] \leftarrow (id, n, T, p, caller)$

These actions preserve the synchronization of the $\mathsf{reqQueue}_{\mathsf{Enc}}$ as described in invariant 3.

– $(\texttt{FinalizeDec}, ctr, id, \pi, u)$

Real

$(id, pw', r, n, T, p, caller) \leftarrow \mathsf{reqQueue}^{\mathsf{Real}}_{\mathsf{Dec}}[ctr]$
Exit 1: $b = \texttt{false}$
$\mathsf{limiterResponses}^{\mathsf{Real}}_{\mathsf{Dec}}[(ctr, rl)] \leftarrow u$
Exit 2: $c = \texttt{false}$

  send $(\texttt{Ret}, id, \bot)$ **to** $caller$
$[u_f]_t \leftarrow \mathsf{comb}(\mathsf{limiterResponses}^{\mathsf{Real}}_{\mathsf{Dec}}[(id, rl')])$
$m^{\mathsf{Real}} \leftarrow \mathsf{H}_{\mathsf{OTP}}((\frac{\mathsf{sk}_{\mathcal{S}} \cdot [u_f]_t}{r}, pw', id, n), |c_1|) \oplus c_1$
$c' \leftarrow \mathsf{H}_{\mathsf{MAC}}(\frac{\mathsf{sk}_{\mathcal{S}} \cdot [u_f]_t}{r}, m^{\mathsf{Real}}, pw', id, n)$
if $c_2 = c'$: **send** $(\texttt{Ret}, id, m^{\mathsf{Real}})$ **to** $caller$
else : **send** $(\texttt{Ret}, id, \bot)$ **to** $caller$

Ideal

$(id, n, T, p, caller) \leftarrow \mathsf{reqQueue}^{\mathsf{Sim}}_{\mathsf{Dec}}[ctr]$
Exit 1: $b = \texttt{false}$
$\mathsf{limiterResponses}^{\mathsf{Sim}}_{\mathsf{Dec}}[(ctr, rl)] \leftarrow u$
Exit 2: $c = \texttt{false}$
  send $(\texttt{FinishRetrieve}, ctr, T, \texttt{failed}, (\texttt{true}, 0, \epsilon, \epsilon))$ **to** $\mathcal{F}_{\mathrm{TPHE}}$
  send $(\texttt{Ret}, id, \bot)$ **to** $caller$
if $\exists\ ([u]_t, m^{\mathsf{Sim}}, pw^{\mathsf{Sim}})$ $s.t.$ $(id, n^{\mathsf{Sim}}, [u]_t, m^{\mathsf{Sim}}, pw^{\mathsf{Sim}})$
  $\in \mathsf{injectedMessages}^{\mathsf{Sim}}$:
  send $(\texttt{FinishRetrieve}, ctr, T, \texttt{success},$
    $(\texttt{false}, 0, pw^{\mathsf{Sim}}, m^{\mathsf{Sim}}))$ **to** $\mathcal{F}_{\mathrm{TPHE}}$
  if $pw^{\mathsf{Sim}} = pw'$: **send** $(\texttt{Ret}, id, m^{\mathsf{Sim}})$ **to** $caller$
  else : **send** $(\texttt{Ret}, id, \bot)$ **to** $caller$
elseif $\exists i \in \mathbb{N}$ $s.t.$ $\mathsf{storageHistory}^{\mathsf{Sim}}[id, i] = (c_1, c_2, n)$:
  send $(\texttt{FinishRetrieve}, ctr, T, \texttt{success}, (\texttt{true}, i, \epsilon, \epsilon))$ **to** $\mathcal{F}_{\mathrm{TPHE}}$
  $(pw^{\mathsf{Ideal}} m^{\mathsf{Ideal}}) \leftarrow \mathsf{storageHistory}^{\mathsf{Ideal}}[id, i]$
  if $pw^{\mathsf{Ideal}} = pw'$: **send** $(\texttt{Ret}, id, m^{\mathsf{Ideal}})$ **to** $caller$
  else : **send** $(\texttt{Ret}, id, \bot)$ **to** $caller$
else :
  send $(\texttt{FinishRetrieve}, ctr, T, \texttt{failed}, (\texttt{true}, 0, \epsilon, \epsilon))$ **to** $\mathcal{F}_{\mathrm{TPHE}}$
  send $(\texttt{Ret}, id, \bot)$ **to** $caller$

remove $(rl, ctr, id, n, [o]_t)$ from $\mathsf{retrieveRequests}^{\mathsf{Sim}}$
$i \in T$: $\mathsf{retrieveRate}^{\mathsf{Ideal}}[i, id] - -$

Server (continued)

– $(\texttt{FinalizeDec}, ctr, id, \pi, u)$ (continued)
Up to exit 2, it is clear that the states remain in sync as the computations are essentially identical. Note that we know because of the NIZKs that $[u]_t$ is well-formed. For the remaining, we differentiate between five cases:

1. $\exists\ ([u]_t, m^{\mathsf{Sim}}, pw^{\mathsf{Sim}})\ s.t.\ (id, n^{\mathsf{Sim}}, [u]_t, m^{\mathsf{Sim}}, pw^{\mathsf{Sim}}) \in \mathsf{injectedMessages}^{\mathsf{Sim}} \wedge pw^{\mathsf{Sim}} = pw'$
Case 2 of invariant 1 shows that the real protocol has a well-formed entry for $m^{\mathsf{Sim}}, pw^{\mathsf{Sim}}$ at index id. Hence, $c_2 = c'$ because $pw^{\mathsf{Sim}} = pw'$ and, therefore, both send $(\texttt{Ret}, id, m^{\mathsf{Sim}})$ with $m^{\mathsf{Sim}} = m^{\mathsf{Real}}$.
2. $\exists\ ([u]_t, m^{\mathsf{Sim}}, pw^{\mathsf{Sim}})\ s.t.\ (id, n^{\mathsf{Sim}}, [u]_t, m^{\mathsf{Sim}}, pw^{\mathsf{Sim}}) \in \mathsf{injectedMessages}^{\mathsf{Sim}} \wedge pw^{\mathsf{Sim}} \neq pw'$
Case 2 of invariant 1 shows that the real protocol has a well-formed entry for $m^{\mathsf{Sim}}, pw^{\mathsf{Sim}}$ at index id. Hence, $c_2 \neq c'$ because $pw^{\mathsf{Sim}} \neq pw'$ and, therefore, both send $(\texttt{Ret}, id, \bot)$, except a collision occurs in a hash function which happens with the same probability in both worlds as they are computed in the same way.
3. $\nexists\ ([u]_t, m^{\mathsf{Sim}}, pw^{\mathsf{Sim}})\ s.t.\ (id, n^{\mathsf{Sim}}, [u]_t, m^{\mathsf{Sim}}, pw^{\mathsf{Sim}}) \in \mathsf{injectedMessages}^{\mathsf{Sim}} \wedge \exists i \in \mathbb{N}\ s.t.\ \mathsf{storageHistory}^{\mathsf{Sim}}[id, i] = (c_1^{\mathsf{Sim}}, c_2^{\mathsf{Sim}}, n^{\mathsf{Sim}}) \wedge pw^{\mathsf{Ideal}} = pw'$
Case 1 of invariant 1 shows that the real protocol has a well-formed entry for $m^{\mathsf{Ideal}}, pw^{\mathsf{Ideal}}$ at index id. Hence, $c_2 = c'$ because $pw^{\mathsf{Ideal}} = pw'$ and, therefore, both send $(\texttt{Ret}, id, m^{\mathsf{Ideal}})$ with $m^{\mathsf{Ideal}} = m^{\mathsf{Real}}$.
4. $\nexists\ ([u]_t, m^{\mathsf{Sim}}, pw^{\mathsf{Sim}})\ s.t.\ (id, n^{\mathsf{Sim}}, [u]_t, m^{\mathsf{Sim}}, pw^{\mathsf{Sim}}) \in \mathsf{injectedMessages}^{\mathsf{Sim}} \wedge \exists i \in \mathbb{N}\ s.t.\ \mathsf{storageHistory}^{\mathsf{Sim}}[id, i] = (c_1^{\mathsf{Sim}}, c_2^{\mathsf{Sim}}, n^{\mathsf{Sim}}) \wedge pw^{\mathsf{Ideal}} \neq pw'$
Case 1 of invariant 1 shows that the real protocol has a well-formed entry for $m^{\mathsf{Ideal}}, pw^{\mathsf{Ideal}}$ at index id. Hence, $c_2 \neq c'$ because $pw^{\mathsf{Ideal}} \neq pw'$ and, therefore, both send $(\texttt{Ret}, id, \bot)$, except a collision occurs in a hash function which happens with the same probability in both worlds as they are computed in the same way.
5. $\nexists\ ([u]_t, m^{\mathsf{Sim}}, pw^{\mathsf{Sim}})\ s.t.\ (id, n^{\mathsf{Sim}}, [u]_t, m^{\mathsf{Sim}}, pw^{\mathsf{Sim}}) \in \mathsf{injectedMessages}^{\mathsf{Sim}} \wedge\ \nexists i \in \mathbb{N}\ s.t.\ \mathsf{storageHistory}^{\mathsf{Sim}}[id, i] = (c_1^{\mathsf{Sim}}, c_2^{\mathsf{Sim}}, n^{\mathsf{Sim}})$
Case 3 of invariant 1 shows that the real protocol has no well-formed entry at index id. Hence, $c_2 \neq c'$ and, therefore, both send $(\texttt{Ret}, id, \bot)$.

Because of invariant 3, we know that the recipients *caller* of the messages sent in the real and ideal world are identical. The simulator finishes by removing all entries from $\mathsf{retrieveRequests}$ used during the protocol run, and the ideal functionality decrements $\mathsf{retrieveRate}$ for all ratelimiters involved in the protocol run, which maintains the synchronization of the $\mathsf{retrieveRate}$ as described in invariant 4.

Ratelimiter

– $(\texttt{DecRequest}, ctr, id, [p]_2, n)$

Real	Ideal
$\mathsf{retrieveRate}^{\mathsf{Real}}[id] - -$	$\mathsf{retrieveRate}^{\mathsf{Sim}}[id] - -$
	$\mathsf{retrieveRequests.add}(rl, ctr, id, n, [o]_t)$

These actions preserve the synchronization of the $\mathsf{retrieveRate}$ (invariant 4).

Init & Rotate

– $(\texttt{ChangeCorruption}, corr_{\mathcal{S}}, corrIDs_{\mathcal{R}})$

Real	Ideal
	if $\mathcal{S} \in \mathsf{currentlyCorrupted}$: **forall** id, n, i :
	if $\mathsf{storageHistory}[id, i] = (\_, \_, \_, n)$:
	while $\left\lvert\{(rl, \_, id, \_, \_)\} \subseteq \mathsf{retrieveRequests}^{\mathsf{Sim}}\right\rvert \geq t - n_c$:
	remove $t - n_c$ elements $(rl, \_, id, \_, \_)$ **from** $\mathsf{retrieveRequests}^{\mathsf{Sim}}$
	send $(\texttt{PwGuessStart}, id, i, RLset)$ **to** $\mathcal{F}_{\text{TPHE}}$
	$\mathsf{retrieveRequests} \leftarrow \emptyset$
forall rl : **send** $(\texttt{RotateKey}, c, s_i)$ **to** $(i, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R})$	**forall** rl :
if c : **send** $(\mathsf{retrieveRate}, \mathsf{nonces}, \mathsf{sk}, \mathsf{pk}, \mathsf{maxNonceCtr})$ **to** NET	**if** c : **send** $(\mathsf{retrieveRate}, \mathsf{nonces}, \mathsf{sk}, \mathsf{pk}, \mathsf{maxNonceCtr})$ **to** NET
else : $\mathsf{retrieveRate} \leftarrow 0$	**else** : $\mathsf{retrieveRate} \leftarrow 0$
send $(\texttt{RotateKey}, corr_{\mathcal{S}}, \mathsf{sk}_{\mathcal{S}}, \mathsf{pk}, \mathsf{pk}_1, \ldots, \mathsf{pk}_n, \mathsf{nonces})$ **to** $(0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S})$	
$\mathsf{limiterResponses}_{\mathsf{Enc}}[id, rl] \leftarrow \bot \quad \forall (id, rl)$	$\mathsf{limiterResponses}_{\mathsf{Enc}}[id, rl] \leftarrow \bot \quad \forall (id, rl)$
$\mathsf{reqQueue}_{\mathsf{Enc}}[c] \leftarrow \bot \quad \forall c$	$\mathsf{reqQueue}_{\mathsf{Enc}}[c] \leftarrow \bot \quad \forall c$
$\mathsf{reqQueue}_{\mathsf{Dec}}[c] \leftarrow \bot \quad \forall c$	$\mathsf{reqQueue}_{\mathsf{Dec}}[c] \leftarrow \bot \quad \forall c$
$\mathsf{limiterResponses}_{\mathsf{Dec}}[id, rl] \leftarrow \bot \quad \forall (id, rl)$	$\mathsf{limiterResponses}_{\mathsf{Dec}}[id, rl] \leftarrow \bot \quad \forall (id, rl)$

Init & Rotate (continued)

– (`ChangeCorruption`, $corr_{\mathcal{S}}$, $corrIDs_{\mathcal{R}}$) (continued)

Real

if $\mathcal{S} \notin$ currentlyCorrupted :
  send $\mathsf{pk}, \mathsf{pk}_1, \ldots, \mathsf{pk}_n, \mathsf{T}_{\mathsf{set}}$ **to** `NET`
else :
  send `getState` **to** `NET`
  receive (`getState`, *storage*) **from** `NET`
  storage $\leftarrow$ *storage*
  send $(\mathsf{storage}, \mathsf{sk}, \mathsf{pk}, \mathsf{pk}_1, \ldots, \mathsf{pk}_n, \mathsf{nonces}, \mathsf{T}_{\mathsf{set}},$
    $\mathsf{limiterResponses}_{\mathsf{Enc}}, \mathsf{reqQueue}_{\mathsf{Enc}}, \mathsf{reqQueue}_{\mathsf{Dec}},$
    $\mathsf{limiterResponses}_{\mathsf{Dec}})$ **to** `NET`

Ideal

if $\mathcal{S} \notin$ currentlyCorrupted :
  send $\mathsf{pk}, \mathsf{pk}_1, \ldots, \mathsf{pk}_n, \mathsf{T}_{\mathsf{set}}$ **to** `NET`
else :
  send `getState` **to** `NET`
  receive (`getState`, *storage*) **from** `NET`
  storage $\leftarrow$ *storage*
  send $(\mathsf{storage}, \mathsf{sk}, \mathsf{pk}, \mathsf{pk}_1, \ldots, \mathsf{pk}_n, \mathsf{nonces}, \mathsf{T}_{\mathsf{set}},$
    $\mathsf{limiterResponses}_{\mathsf{Enc}}, \mathsf{reqQueue}_{\mathsf{Enc}}, \mathsf{reqQueue}_{\mathsf{Dec}},$
    $\mathsf{limiterResponses}_{\mathsf{Dec}})$ **to** `NET`
  forall $(c_1, c_2, n) \in \mathsf{storage}^{\mathsf{Sim}}$:
    if $[u]_t, m, pw$ **exist** $s.t.$
      $(([u]_t, m, pw, id, n), c_2) \in \mathsf{hashHistory}_{\mathsf{MAC}}$:
      $h \leftarrow c_1 \oplus m$
      if $(([u]_t, pw, id, n), |m|, h) \in \mathsf{hashHistory}_{\mathsf{OTP}} \wedge$
        $[u]_t = \mathsf{sk} \cdot \mathcal{F}_{\mathrm{ro}}^1(id, n) \cdot \mathcal{F}_{\mathrm{ro}}^2(pw, n)$:
        add $(id, n, [u]_t, m, pw)$ **to** injectedMessages

Invariants 2 and 3 are preserved because both reqQueue are reset to their initial state. Invariant 4 is also preserved as all retrieveRate of honest ratelimiters are reset to 0, and retrieveRequests is emptied. Invariant 1 is clearly preserved if the server was honest in the forgoing epoch, as the records are left unchanged. If the server was corrupted in the forgoing epoch, we distinguish between three cases for each record c_1, c_2, n of the storage given by the environment:

1. The record was generated by an honest server. Consequently, there exists an entry $(c_1^{\mathsf{Sim}}, c_2^{\mathsf{Sim}}, n^{\mathsf{Sim}})$ in $\mathsf{storageHistory}^{\mathsf{Sim}}$. Invariant 5 shows that there also exists a corresponding entry in $\mathsf{storageHistory}^{\mathsf{Ideal}}$, and hence, option 1 of invariant 1 holds.
2. The record was generated by a corrupted server according to the protocol. Consequently, entries exist in $\mathsf{hashHistory}_{\mathsf{OTP}}$ and $\mathsf{hashHistory}_{\mathsf{MAC}}$ for the corresponding inputs. The simulator finds those entries and adds $(id, n, [u]_t, m, pw)$ to injectedMessages. This fulfills option 2 of invariant 1.
3. The record was generated by a corrupted server not according to the protocol. Consequently, at least one entry in $\mathsf{hashHistory}_{\mathsf{OTP}}$ and $\mathsf{hashHistory}_{\mathsf{MAC}}$ for the corresponding inputs is missing or inconsistent with the tuple (c_1, c_2). Hence, option 3 of invariant 1 holds.

– ($\texttt{RotateKey}_{\mathcal{A}}$)

Real

send `RotationOngoing` **to** $(0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S})$
rotationOngoing $\leftarrow$ `true`
As often as $\mathcal{E}$ wants and with given inputs (asynchronous) :
  send (`RotateKey`, `false`, s_i) **to** $(i, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R})$
  retrieveRate $\leftarrow 0$
send (`RotateKey`, `false`, $\mathsf{sk}_{\mathcal{S}}$, pk, $\mathsf{pk}_1, \ldots, \mathsf{pk}_n$, nonces) **to** $(0, \mathsf{sid}_{\mathsf{cur}}, \mathcal{S})$
$\mathsf{limiterResponses}_{\mathsf{Enc}}[id, rl] \leftarrow \bot \quad \forall (id, rl)$
$\mathsf{reqQueue}_{\mathsf{Enc}}[c] \leftarrow \bot \quad \forall c$
$\mathsf{reqQueue}_{\mathsf{Dec}}[c] \leftarrow \bot \quad \forall c$
$\mathsf{limiterResponses}_{\mathsf{Dec}}[id, rl] \leftarrow \bot \quad \forall (id, rl)$
send $\mathsf{pk}, \mathsf{pk}_1, \ldots, \mathsf{pk}_n, \mathsf{T}_{\mathsf{set}}$ **to** `NET`
rotationOngoing $\leftarrow$ `false`

Ideal

rotationOngoing $\leftarrow$ `true`
As often as $\mathcal{E}$ wants and with given inputs (asynchronous) :
  retrieveRate $\leftarrow 0$
$\mathsf{limiterResponses}_{\mathsf{Enc}}[id, rl] \leftarrow \bot \quad \forall (id, rl)$
$\mathsf{reqQueue}_{\mathsf{Enc}}[c] \leftarrow \bot \quad \forall c$
$\mathsf{reqQueue}_{\mathsf{Dec}}[c] \leftarrow \bot \quad \forall c$
$\mathsf{limiterResponses}_{\mathsf{Dec}}[id, rl] \leftarrow \bot \quad \forall (id, rl)$
send $\mathsf{pk}, \mathsf{pk}_1, \ldots, \mathsf{pk}_n, \mathsf{T}_{\mathsf{set}}$ **to** `NET`
rotationOngoing $\leftarrow$ `false`

forall id, n, i :
  if $\mathsf{storageHistory}[id, i] = (\_, \_, \_, n)$:
    while $\left|\{(rl, \_, id, \_, \_)\} \subseteq \mathsf{retrieveRequests}^{\mathsf{Sim}}\right| \geq t - n_c$:
      remove $t - n_c$ elements $(rl, \_, id, \_, \_)$ **from** $\mathsf{retrieveRequests}^{\mathsf{Sim}}$
      send (`PwGuessStart`, $id, i, RLset$) **to** $\mathcal{F}_{\mathrm{TPHE}}$
retrieveRequests $\leftarrow \emptyset$

Note that the environment can send `RotateKey` as often as it wants to honest ratelimiters asynchronous. That means that in between those messages, it can send other messages to any other machine. Nevertheless, the server ignores all incoming messages, from receiving `RotationOngoing` until receiving `RotateKey`. Therefore, the only relevant message that is actually processed is `HelpRetrieve` sent to a ratelimiter that leads to an incrementation of $\mathsf{retrieveRate}[id]$ which maintains all invariants. Invariants 2 and 3 are preserved because both reqQueue are reset to their initial state. Invariant 4 is also preserved as $\mathsf{retrieveRate}^{\mathsf{Real}}$ and $\mathsf{retrieveRate}^{\mathsf{Sim}}$ are reset to 0 for the same ratelimiters, and retrieveRequests is emptied. Invariant 1 is clearly preserved as the records are left unchanged.

Init & Rotate (continued)

– $(\texttt{RotateKey}^{\mathcal{S}}_{\mathcal{A}}, rl, s)$

Real	Ideal
send $(\texttt{RotateKey}, \texttt{false}, s)$ **to** $(rl, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R})$	**send** $(\texttt{RotateKey}, \texttt{false}, s)$ **to** $(rl, \mathsf{sid}_{\mathsf{cur}}, \mathcal{R})$
$\mathsf{retrieveRate} \leftarrow 0$	$\mathsf{retrieveRate} \leftarrow 0$

Invariant 4 shows that $\mathsf{retrieveRate}^{\mathsf{Real}}[id] = \mathsf{retrieveRate}^{\mathsf{Sim}}[id] \leq \mathsf{retrieveRate}^{\mathsf{Ideal}}[id] - \left|\{(rl, \_, id, \_, \_)\} \subseteq \mathsf{retrieveRequests}^{\mathsf{Sim}}\right|$ which is preserved as both $\mathsf{retrieveRate}^{\mathsf{Real}}$ and $\mathsf{retrieveRate}^{\mathsf{Sim}}$ are reset to 0.

RO-OTP

– $\mathcal{F}^{\mathsf{OTP}}_{\mathrm{ro}}(x, l)$

Real	Ideal
if $(x, l, \_) \notin \mathsf{hashHistory}_{\mathsf{OTP}}$:	**if** $(x, l, \_) \notin \mathsf{hashHistory}_{\mathsf{OTP}}$:
	if x can be parsed as $([u]_t, pw, id, n) \wedge \mathsf{sk} \cdot [h_1]_1 \cdot [h_2]_2 = [u]_t$ $\wedge\, \exists i \ s.t.\ \mathsf{storageHistory}[id, i] = (c_1, c_2, n) \wedge |c_1| = l$:
	send $(\texttt{PwGuessFinish}, id, i, pw)$ **to** $\mathcal{F}_{\mathrm{TPHE}}$
	receive $(\texttt{PwGuessFinish}, m')$ **from** $\mathcal{F}_{\mathrm{TPHE}}$
	if $m' = (\texttt{Correct}, m'')$:
	add $(([u]_t, pw, id, n), |m''|, c_1 \oplus m'')$ **to** $\mathsf{hashHistory}_{\mathsf{OTP}}$
	add $(([u]_t, m'', pw, id, n), c_2)$ **to** $\mathsf{hashHistory}_{\mathsf{MAC}}$
	if $|m''| = l$
	return $c_1 \oplus m''$
$h \xleftarrow{\$} \{0,1\}^l$	$h \xleftarrow{\$} \{0,1\}^l$
add (x, l, h) **to** $\mathsf{hashHistory}_{\mathsf{OTP}}$	**add** (x, l, h) **to** $\mathsf{hashHistory}_{\mathsf{OTP}}$
return $h \ s.t.\ (x, l, h) \in \mathsf{hashHistory}_{\mathsf{OTP}}$	**return** $h \ s.t.\ (x, l, h) \in \mathsf{hashHistory}_{\mathsf{OTP}}$

The behavior of the real and the ideal world is identical except if the input is consistent with a dummy record. In that case, entries for $\mathsf{hashHistory}_{\mathsf{OTP}}$ and $\mathsf{hashHistory}_{\mathsf{MAC}}$ that are consistent with the dummy record and, hence, preserve option 1 of invariant 1 are generated and stored.

RO-MAC

– $\mathcal{F}^{\mathsf{MAC}}_{\mathrm{ro}}(x)$

Real	Ideal
if $(x, \_) \notin \mathsf{hashHistory}_{\mathsf{MAC}}$:	**if** $(x, \_) \notin \mathsf{hashHistory}_{\mathsf{MAC}}$:
	if x can be parsed as $([u]_t, m, pw, id, n)$ $\wedge\, \mathsf{sk} \cdot [h_1]_1 \cdot [h_2]_2 = [u]_t$ $\wedge\, \exists i \ s.t.\ \mathsf{storageHistory}[id, i] = (c_1, c_2, n) \wedge |c_1| = |m|$:
	send $(\texttt{PwGuessFinish}, id, i, pw)$ **to** $\mathcal{F}_{\mathrm{TPHE}}$
	receive $(\texttt{PwGuessFinish}, m')$ **from** $\mathcal{F}_{\mathrm{TPHE}}$
	if $m' = (\texttt{Correct}, m'')$:
	add $(([u]_t, pw, id, n), |m''|, c_1 \oplus m'')$ **to** $\mathsf{hashHistory}_{\mathsf{OTP}}$
	add $(([u]_t, m'', pw, id, n), c_2)$ **to** $\mathsf{hashHistory}_{\mathsf{MAC}}$
	if $m'' = m$
	return c_2
$h \xleftarrow{\$} \{0,1\}^\eta$	$h \xleftarrow{\$} \{0,1\}^\eta$
add (x, h) **to** $\mathsf{hashHistory}_{\mathsf{MAC}}$	**add** (x, h) **to** $\mathsf{hashHistory}_{\mathsf{MAC}}$
return $h \ s.t.\ (x, h) \in \mathsf{hashHistory}_{\mathsf{MAC}}$	**return** $h \ s.t.\ (x, h) \in \mathsf{hashHistory}_{\mathsf{MAC}}$

The analysis of $\mathcal{F}^{\mathsf{MAC}}_{\mathrm{ro}}$ works in accordance with that of $\mathcal{F}^{\mathsf{OTP}}_{\mathrm{ro}}$.

Authenticated Channel, NIZK, RO1, RO2, and RO-N

These functionalities are perfectly simulated by the simulator and, therefore, are indistinguishable in their outputs and preserve all invariants.

According to the argumentation above, the simulation can only be distinguished from the real protocol execution if it aborts (in (4), (8), or (13)) or if event FAIL4 happens. We introduce a hybrid simulator Sim', functionally equivalent to simulator Sim, that injects a Gap-OM-BCDH challenge to show that if Sim aborts in (13), we can break the Gap-OM-BCDH assumption. This step of the reduction follows the proof technique for proving Partially Oblivious Pseudo-Random Functions secure introduced by [4].

Before we show how Sim' injects the challenge, we introduce a function $\texttt{Interpolate}(i, T, \{(j, \mathsf{sk}_j \cdot [x]_t)\}_{j \in T})$ that takes an index i, an index set T of size t, and a set of tuples $\{(j, \mathsf{sk}_j \cdot [x]_t)\}_{j \in T}$. Each tuple consists of an index from T and a group element $[x]_t$ multiplied by the j-th share of the secret key. The function uses polynomial interpolation with Lagrange factors to compute $\mathsf{sk}_i \cdot [x]_t$. This function can compute every $\mathsf{sk}_i \cdot [x]_t$ as long as t distinct tuples $(j, \mathsf{sk}_j \cdot [x]_t)$ are available. Note that $\mathsf{sk}_0 = \mathsf{sk}$.

The hybrid simulator Sim' is equivalent to Sim except for the following changes:

- **Guessing** (id^*, n^*) The simulator guesses for which (id, n) tuple the break happens. If this guess is not correct, the simulator aborts. We call this event FAIL6.
- **(1) Parameter Generation** Instead of sampling a key pair $(\mathsf{sk}, \mathsf{pk})$ and sharing it between the server and the ratelimiters, it uses the public key given by the Gap-OM-BCDH game. If the server is honest, it samples a random combined ratelimiter key $\mathsf{sk}_{\mathcal{R}}$ and shares it for the ratelimiters. If the server is dishonest, it samples random key shares sk_i for each corrupted party $i \in \mathsf{CorruptionSet}$ and computes the public key as $\mathsf{pk}_i \leftarrow [sk_i]_t$. If less than $t-1$ ratelimiters are corrupt, it samples more random key shares until it has $t-1$ ratelimit keys in total and computes the corresponding public keys. The remaining secret keys are unknown to the simulator, but it can compute the public keys with the `Interpolate` function. It inputs the $t-1$ ratelimiter public keys of the known key shares and the overall public key $\mathsf{pk}_{\mathcal{R}} = \mathsf{pk}_0 \cdot [\mathsf{sk}_{\mathcal{S}}]_t^{-1}$. Furthermore, it has additional data structures:
  - $\mathsf{challenges}_1 \subseteq \mathbb{N} \times \mathbb{G}_1$ to keep track of the challenges $[x_i]_1$ and their indices i returned by Targ_1.
  - $\mathsf{challenges}_2 \subseteq \mathbb{N} \times \mathbb{G}_2$ to keep track of the challenges $[y_j]_2$ and their indices j returned by Targ_2.
  - $\mathsf{solutions} \subseteq \mathbb{N}^2 \times \mathbb{G}_t$ to keep track of valid solutions consisting of the indices i, j of two challenges $[x_i]_1, [y_j]_2$ and the solution $\sigma = \mathsf{sk} \cdot [x_i]_1 \cdot [y_j]_2$.
- **(3) Answering Queries to the Random Oracles** For $\mathcal{F}_{\text{ro}}^1$, the simulator checks whether the input is (id^*, n^*). If it is, it uses the oracle Targ_1 to obtain a random group element and stores it in $\mathsf{challenges}_1$. For inputs that are not (id^*, n^*), the simulator programms $\mathcal{F}_{\text{ro}}^1$ as before. Instead of injecting trapdoors into $\mathcal{F}_{\text{ro}}^2$, it uses the oracle Targ_2 to obtain random group elements and stores them in $\mathsf{challenges}_2$.
- **(5) and (11) Storing/Retrieving Responses** To compute $[u]_t$, it differentiates between two cases if the server is corrupt (otherwise it knows all ratelimiter keys anyway):
  - If the corresponding key share is known, it computes $[u]_t \leftarrow \mathsf{sk}_i \cdot [h_1]_1 \cdot [p]_2$ and the NIZK as before.
  - If the corresponding key share is unknown, it computes $\sigma = \mathsf{sk} \cdot [h_1]_1 \cdot [p]_2$ and uses the `Interpolate` function with $(0, \sigma - \mathsf{sk}_{\mathcal{S}} \cdot [h_1]_1 \cdot [p]_2)$ and $(i, \mathsf{sk}_i \cdot [h_1]_1 \cdot [p]_2)$ for every known key share to compute $[u]_t$. The simulator differentiates between three cases to compute $\sigma = \mathsf{sk} \cdot [h_1]_1 \cdot [p]_2$:
    - If the query comes from an honest client, it takes a from $\mathsf{reqQueue}_{\{\mathsf{Enc/Dec}\}}[id]$ to compute $\mathsf{sk}_i \cdot [h_1]_1 \cdot [p]_2 \leftarrow a \cdot [\mathsf{pk}_i]_1 \cdot [p]_2$.
    - If $(id, n) = (id^*, n^*)$, it uses the Help oracle to compute the value.
    - If $(id, n) \neq (id^*, n^*)$, it takes a from $\mathsf{hashHistory}_1$ to compute $\mathsf{sk}_i \cdot [h_1]_1 \cdot [p]_2 \leftarrow a \cdot [\mathsf{pk}_i]_1 \cdot [p]_2$.

  Because the simulator does not know sk_i, it simulates the `Prove` function of $\mathcal{F}_{\text{nizk}}$ without a witness w. Since $[u]_t$ computed with the Help oracle is surely well-formed, it skips the check of whether $(x, w) \in \mathsf{R}$ and directly adds (x, ϵ, π) to nizkHistory.
- **(8) Honest Key Rotation** Instead of sampling a polynomial, it reruns the procedure as described for the initial key generation.

  Instead of checking the correctness of $[u]_t$ with the computation $\mathsf{sk} \cdot [h_1]_1 \cdot [h_2]_2 = [u]_t$, it queries the DDH oracle on the inputs $([\mathsf{sk}]_t, [h_1]_1 \cdot [h_2]_2, [1]_t, [u]_t)$ which returns 1 if the equation above holds.
- **(9) Dishonest Key Rotation** Upon receiving $(\texttt{RotateKey}_{\mathcal{A}}^{\mathcal{S}}, rl, s)$, the ratelimiter stores s and computes $[u]_t \leftarrow \mathsf{sk}_i \cdot [o]_t$ in (5) and (11) as described but adds $s \cdot [o]_t$.
- **(13) Special Queries to the Random Oracles** OTP **and** MAC Instead of checking the correctness of $[u]_t$ with the computation $\mathsf{sk} \cdot [h_1]_1 \cdot [h_2]_2 = [u]_t$, it queries the DDH oracle on the inputs $([\mathsf{sk}]_t, [h_1]_1 \cdot [h_2]_2, [1]_t, [u]_t)$ which returns 1 if the equation above holds. If it finds a new solution, it adds $(i, j, [u]_t)$ to solutions with i, j taken from the two challenges sets corresponding to the outputs of $\mathcal{F}_{\text{ro}}^1$ and $\mathcal{F}_{\text{ro}}^2$ such that $[u]_t = \mathsf{sk} \cdot [x_i]_1 \cdot [y_j]_2$.

The simulator only aborts in (13) if the environment can compute a valid $[u]_t$ with at most $t - n_c - 1$ interactions with honest ratelimiters, with n_c being the number of currently corrupted ratelimiters. If the simulator can answer these $t - n_c - 1$ queries to honest ratelimiters without a query to the Help oracle, it obtains "one-more" solution (i, j, σ) then it queried the Help oracle. In that case, it can give the Q tuples from solutions in addition to the newly obtained $Q+1$-st solution to the Gap-OM-BCDH game, thus winning the Gap-OM-BCDH game. To answer the queries without a Help oracle query, the simulator has to guess the ratelimiters those queries will be addressed to at the beginning of an epoch to assign the known key shares to those ratelimiters. The probability of guessing right is

$$\frac{1}{\binom{t-n_c-1}{n-n_c}}.$$

It is important to see that this probability is non-negligible as long as $\binom{t-n_c-1}{n-n_c}$ is polynomial in the security parameter. Furthermore, n and t are usually small in practice, resulting in a reasonably small loss. Note that guessing right for an epoch is only relevant in the single epoch that the simulator aborts in.

The additional valid $[u]_t$ obtained from the oracle query is only a valid solution to Gap-OM-BCDH if $[o]_t$ can be expressed as the pairing of two challenge elements obtained from $[x_1]_1 \xleftarrow{\$} \mathsf{Targ}_1$ and $[y_j]_2 \xleftarrow{\$} \mathsf{Targ}_2$ such that $[o]_t = [x_i]_1 \cdot [y_j]_2$. This is only the case if the simulator guessed correctly for which tuple (id, n) the break happens, resulting in $H_1(id^*, n^*) = [x_i]_1$ (in other words, the event FAIL6 did not occur). The probability of guessing right is

$$\frac{1}{q_{\text{ro1}}}$$

with q_{ro1} being the number of queries to $\mathcal{F}_{\text{ro}}^2$. Therefore, it holds that

$$\mathsf{Pr}[\mathsf{FAIL2}] \leq \binom{t-n_c-1}{n-n_c} \cdot q_{\text{ro1}} \cdot \mathsf{Adv}_{\eta,\mathsf{BG}}^{\mathsf{Gap\text{-}OM\text{-}BCDH}},$$

with n_c being the number of corrupted ratelimiters in the epoch where the simulator aborts in (13). The probability that the simulation aborts in (4) is bounded by the probability of a collision. A collision for the nonce occurs when the same nonce is sampled twice by the random oracle $\mathcal{F}_{\text{ro}}^{\mathsf{N}}$ for different inputs or when the same individual nonces are input to $\mathcal{F}_{\text{ro}}^{\mathsf{N}}$. Hence we get

$$\mathsf{Pr}[\mathsf{FAIL1}] \leq \mathsf{Pr}[\mathsf{coll}_{\text{ro-N}}] + \mathsf{Pr}[\mathsf{coll}_{\text{inputs}}]$$

The probability of a collision in the random oracle $\mathcal{F}_{\text{ro}}^{\mathsf{N}}$ is bound by

$$\mathsf{Pr}[\mathsf{coll}_{\text{ro-N}}] \leq \frac{q_{\text{ro-N}}^2}{2^\eta}$$

based on the birthday paradox with $q_{\text{ro-N}}$ being the number of queries to $\mathcal{F}_{\text{ro}}^{\mathsf{N}}$. Because at least one ratelimiter or the server is honest and samples its nonce honestly and uniformly at random, the probability that the honest party samples a nonce that was input to $\mathcal{F}_{\text{ro}}^{\mathsf{N}}$ before is bound by

$$\mathsf{Pr}[\mathsf{coll}_{\text{inputs}}] \leq \frac{q_{\text{store}} \cdot q_{\text{ro-N}}}{2^\eta}$$

with q_{store} being the number of invocations of the store protocol and $q_{\text{ro-N}}$ being the number of queries to $\mathcal{F}_{\text{ro}}^{\mathsf{N}}$. This is because the probability that a randomly chosen nonce was part of the input of a previous query to $\mathcal{F}_{\text{ro}}^{\mathsf{N}}$ is bound by $\frac{q_{\text{ro-N}}}{2^\eta}$, and there are q_{store} samplings where this can happen. Combining both equations above yields

$$\mathsf{Pr}[\mathsf{FAIL1}] \leq \frac{q_{\text{store}} \cdot q_{\text{ro-N}} + q_{\text{ro-N}}^2}{2^\eta}.$$

The probability that the simulator aborts in (8) is equivalent to the probability that a collision in the random oracle $\mathcal{F}_{\text{ro}}^{\mathsf{MAC}}$ occurs. This probability is bound by

$$\mathsf{Pr}[\mathsf{FAIL3}] = \mathsf{Pr}[\mathsf{coll}_{\text{ro-MAC}}] \leq \frac{q_{\text{ro-MAC}}^2}{2^\eta}.$$

The probability that the event FAIL4 happens is the probability that the environment guessed a c_2 correct without querying $\mathcal{F}_{\text{ro}}^{\mathsf{MAC}}$. This probability is bound by

$$\mathsf{Pr}[\mathsf{FAIL4}] = \mathsf{Pr}[\text{guess}_{c_2}] \leq \frac{q_{\text{rec}}}{2^\eta},$$

with q_{rec} being the number of injected records by a corrupted server. The event FAIL5 only occurs when the random oracle $\mathcal{F}_{\text{ro}}^2$ outputs the neutral element $[1]_2 \in \mathbb{G}_2$. The probability of FAIL5 happening is

$$\mathsf{Pr}[\mathsf{FAIL5}] = \frac{q_{\text{ro2}}}{2^\eta}.$$

The event FAIL6 only occurs when the simulator guessed (id^*, n^*) wrong

As stated above, the advantage of the environment in distinguishing the real world from the ideal world is equivalent to the probability of FAIL1 events happening. Hence we get

$$\begin{aligned}\mathsf{Adv}_{\mathsf{Real}}^{\mathsf{Ideal}} &\leq \mathsf{Pr}[\mathsf{FAIL1}] + \mathsf{Pr}[\mathsf{FAIL2}] \\ &+ \mathsf{Pr}[\mathsf{FAIL3}] + \mathsf{Pr}[\mathsf{FAIL4}] + \mathsf{Pr}[\mathsf{FAIL5}] \\ &\leq \binom{t-n_c-1}{n-n_c} \cdot q_{\text{ro1}} \cdot \mathsf{Adv}_{\eta,\mathsf{BG}}^{\mathsf{Gap\text{-}OM\text{-}BCDH}}\end{aligned}$$

$$+\frac{q_{\text{store}} \cdot q_{\text{ro-N}} + q_{\text{rec}} + q_{\text{ro2}} + q_{\text{ro-MAC}}^2 + q_{\text{ro-N}}^2}{2^\eta}.$$

The store function is only invoked by the environment; hence, q_{store} is bound by the runtime of the environment. Records are only injected by the environment; hence, q_{rec} is bound by the runtime of the environment. The random oracles are only directly invoked or indirectly through another function by the environment; hence, q_{ro1}, q_{ro2}, $q_{\text{ro-MAC}}$, and $q_{\text{ro-N}}$ are bound by the runtime of the environment. Because the runtime of the environment is bound by a polynomial, q_{store}, q_{rec}, q_{ro1}, q_{ro2}, $q_{\text{ro-MAC}}$, and $q_{\text{ro-N}}$ are also bound by that polynomial.

The first part of the sum consists of three factors: the first one is independent of the security parameter, the second is polynomial in the security parameter, and the third one is assumed to be negligible in the security parameter, which makes the whole product negligible. The second part of the sum is negligible because the numerator is a sum of polynomially bounded values, and the denominator is exponential in the security parameter, and we, therefore, conclude our proof.

D A Brief Introduction to the iUC Framework

This section provides a brief introduction to the iUC framework, which underlies all results in this paper. The iUC framework [13] is a highly expressive and user friendly model for universal composability. It allows for the modular analysis of different types of protocols in various security settings.

The iUC framework uses interactive Turing machines as its underlying computational model. Such interactive Turing machines can be connected to each other to be able to exchange messages. A set of machines $\mathcal{Q} = \{M_1, \ldots, M_k\}$ is called a *system*. In a run of $\mathcal{Q}$, there can be one or more instances (copies) of each machine in $\mathcal{Q}$. One instance can send messages to another instance. At any point in a run, only a single instance is active, namely, the one to receive the last message; all other instances wait for input. The active instance becomes inactive once it has sent a message; then the instance that receives the message becomes active instead and can perform arbitrary computations. The first machine to run is the so-called *master*. The master is also triggered if the last active machine did not output a message. In iUC, the environment (see next) takes the role of the master. In the iUC framework a special user-specified **CheckID** algorithm is used to determine which instance of a protocol machine receives a message and whether a new instance is to be created (see below).

To define the universal composability security experiment (cf. [13]), one distinguishes between three types of systems: protocols, environments, and adversaries. As is standard in universal composability models, all of these types of systems have to meet a polynomial runtime notion. Intuitively, the security experiment in any universal composability model compares a protocol $\mathcal{P}$ with another protocol $\mathcal{F}$, where $\mathcal{F}$ is typically an ideal specification of some task, called *ideal protocol* or *ideal functionality*. The idea is that if one cannot distinguish $\mathcal{P}$ from $\mathcal{F}$, then $\mathcal{P}$ must be "as good as" $\mathcal{F}$. More specifically, the protocol $\mathcal{P}$ is considered secure (written $\mathcal{P} \leq \mathcal{F}$) if for all adversaries $\mathcal{A}$ controlling the network of $\mathcal{P}$ there exists an (ideal) adversary $\mathcal{S}$, called *simulator*, controlling the network of $\mathcal{F}$ such that $\{\mathcal{A}, \mathcal{P}\}$ and $\{\mathcal{S}, \mathcal{F}\}$ are indistinguishable for all environments $\mathcal{E}$. Indistinguishability means that the probability of the environment outputting 1 in runs of the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}\}$ is negligibly close to the probability of outputting 1 in runs of the system $\{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$ (written $\{\mathcal{E}, \mathcal{A}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$). The environment can also subsume the role of the network attacker $\mathcal{A}$, which yields an equivalent definition in the iUC framework. We usually show this equivalent but simpler statement in our proofs, i.e., that there exists a simulator $\mathcal{S}$ such that $\{\mathcal{E}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$ for all environments.

A protocol $\mathcal{P}$ in the iUC framework is specified via a system of machines $\{M_1, \ldots, M_l\}$; the framework offers a convenient template for the specification of such systems. Each machine M_i implements one or more roles of the protocol, where a role describes a piece of code that performs a specific task. For example, a (real) protocol $\mathcal{P}_{\mathsf{Sig}}$ for digital signatures might contain a $\mathcal{S}$ role for signing messages and a $\mathcal{V}$ role for verifying signatures. In a run of a protocol, there can be several instances of every machine, interacting with each other (and the environment) via I/O interfaces and interacting with the adversary (and possibly the environment subsuming a network attacker) via network interfaces. An instance of a machine M_i manages one or more so-called *entities*. An entity is identified by a tuple $(pid, sid, role)$ and describes a specific party with party ID (PID) *pid* running in a session with session ID (SID) *sid* and executing some code defined by the role *role* where this role has to be (one of) the role(s) of M_i according to the specification of M_i. Entities can send messages to and receive messages from other entities and the adversary using the I/O and network interfaces of their respective machine instances. More specifically, the I/O interfaces of both machines need to be connected to each other (because one machine specifies the other as a subroutine) to enable communication between entities of those machines.

Roles of a protocol can be either public or private. The I/O interfaces of private roles are only accessible by other (entities belonging to) roles of the same protocol, whereas I/O interfaces of public roles can also be accessed by other (potentially unknown) protocols/the environment. Hence, a private role models some internal subroutine that is protected from access outside of the protocol, whereas a public role models some publicly accessible operation that can be used by other protocols. One uses the syntax "($\texttt{pubrole}_1, \ldots, \texttt{pubrole}_n \mid \texttt{privrole}_1, \ldots, \texttt{privrole}_n$)" to uniquely determine public and private roles of a protocol. Two protocols $\mathcal{P}$ and $\mathcal{Q}$ can be combined to form a new more complex protocol as long as their I/O interfaces connect only via their public roles. In the context of the new combined protocol, previously private roles remain private while previously public roles may either remain public or be considered private, as determined by the protocol designer. The set of all possible combinations of $\mathcal{P}$ and $\mathcal{Q}$, which differ only in the set of public roles, is denoted by $\mathsf{Comb}(\mathcal{Q}, \mathcal{P})$.

An entity in a protocol might become corrupted by the adversary, in which case it acts as a pure message forwarder between the adversary and any connected higher-level protocols as well as subroutines. In addition, an entity might also consider itself (implicitly) corrupted while still following its own protocol because, e.g., a subroutine has been corrupted. Corruption of entities in the iUC framework is highly customizable; one can, for example, prevent corruption of certain entities during a protected setup phase.

As explained, the iUC framework offers a convenient template for specifying protocols (which can then also be combined with each other). This template includes many optional parts with sensible defaults such that protocol designers can customize exactly those parts that they need. The specifications using the iUC template that we give in this paper are mostly self explanatory, except for a few aspects:

- The **CheckID** algorithm is used to determine which machine instance is responsible for and hence manages which entities. Whenever a new message is sent to some entity e whose role is implemented by a machine M, the **CheckID** algorithm is run with input e by each instance of M (in order of their creation) to determine whether e is managed by the current instance. The first instance that accepts e then gets to process the incoming message. By default, **CheckID** accepts entities of a single party in a single session, which captures a traditional formulation of a real protocol. Other common definitions include accepting all entities from the same session, which captures a traditional formulation of an ideal functionality.
- The special variable $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ refers to the currently active entity of the current machine instance (that was previously accepted by **CheckID**). If the current activation is due to a message received from another entity, then $(\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})$ refers to that entity.
- The special macro $\mathbf{corr}(pid_{sub}, sid_{sub}, role_{sub})$ can be used to obtain the current corruption status (i.e., whether this entity is still honest or considers itself to be implicitly/explicitly corrupted) of an entity belonging to a subroutine.
- The iUC framework supports so-called responsive environments and responsive adversaries [15]. Such environments and adversaries can be forced to respond to certain messages on the network, called *restricting messages*, immediately and without first activating the protocol in any other way. This is a useful mechanism for modeling purposes, e.g., to leak some information to the attacker or to let the attacker decide upon the corruption status of a new entity but without disrupting the intended execution of the protocol. Such network messages are marked by writing "send responsively" instead of just "send".

The iUC framework supports the modular analysis of protocols via a so-called composition theorem:

Corollary 1 (Concurrent composition in iUC; informal). *Let $\mathcal{P}$ and $\mathcal{F}$ be two protocols such that $\mathcal{P} \leq \mathcal{F}$. Let $\mathcal{Q}$ be another protocol such that $\mathcal{Q}$ and $\mathcal{F}$ can be connected. Let $\mathcal{R} \in \mathsf{Comb}(\mathcal{Q}, \mathcal{P})$ and let $\mathcal{I} \in \mathsf{Comb}(\mathcal{Q}, \mathcal{F})$ such that $\mathcal{R}$ and $\mathcal{I}$ agree on their public roles. Then $\mathcal{R} \leq \mathcal{I}$.*

By this theorem, one can first analyze and prove the security of a subroutine $\mathcal{P}$ independently of how it is used later on in the context of a more complex protocol. Once we have shown that $\mathcal{P} \leq \mathcal{F}$ (for some other, typically ideal protocol $\mathcal{F}$), we can then analyze the security of a higher-level protocol $\mathcal{Q}$ based on $\mathcal{F}$. Note that this is simpler than analyzing $\mathcal{Q}$ based on $\mathcal{P}$ directly as ideal protocols provide absolute security guarantees while typically also being less complex, reducing the potential for errors in proofs. Once we have shown that the combined protocol, say, $(\mathcal{Q} \mid \mathcal{F})$ realizes some other protocol, say, $\mathcal{F}'$, the composition theorem and transitivity of the $\leq$ relation then directly implies that this also holds true if we run $\mathcal{Q}$ with an implementation $\mathcal{P}$ of $\mathcal{F}$. That is, $(\mathcal{Q} \mid \mathcal{P})$ is also a secure realization of $\mathcal{F}'$. Please note that the composition theorem does not impose any restrictions on how the protocols $\mathcal{P}$, $\mathcal{F}$, and $\mathcal{Q}$ look like internally. For example, they might have disjoint sessions, but they could also freely share some state between sessions, or they might be a mixture of both. They can also freely share some of their subroutines with the environment, modeling so-called globally available state. This is unlike most other models for universal composability, such as the UC model, which impose several conditions on the structure of protocols for their composition theorem.

E iUC Notation for Pseudo Code Specifications

Formal ITMs in our paper are specified using pseudo code notation that mostly follows the notation introduced by Camenisch*et al.* [13]. To ease readability of our figures, we provide a brief overview over the used notation here.

The description in the main part of the ITMs consists of blocks of the form **recv** ⟨*msg*⟩ **from** ⟨*sender*⟩ **to** ⟨*receiver*⟩ **s.t.** ⟨condition⟩:⟨code⟩ where ⟨*msg*⟩ is an input pattern, ⟨*sender*⟩ is either the receiving interface (`I/O` for higher-level protocols, `NET` for the network, `SUB` for subroutines) or a dedicated sender connected via `I/O` or `SUB`, ⟨*receiver*⟩ is the dedicated receiver of the message in this ITM and ⟨condition⟩ is a condition on the input. ⟨code⟩ is the (pseudo) code of this block. The block is executed if an incoming message matches the pattern, the message sender and intended receiver match those specified by the block, and the condition is satisfied. More specifically, ⟨*msg*⟩ defines the format of the message m that invokes this code block. Messages contain local variables, state variables, strings, and maybe special characters. To compare a message m to a message pattern *msg*, the values of all global and local variables (if defined) are inserted into the pattern. The resulting pattern p is then compared to m, where uninitialized local variables match with arbitrary parts of the message. If the message matches the pattern p, the actual sender/receiver match those in the block, and ⟨condition⟩ of that block is met, then uninitialized local variables are initialized

with the part of the message that they matched to and ⟨code⟩ is executed in the context of ⟨*receiver*⟩; no other blocks are executed in this case. If m does not match p, the sender is incorrect, or ⟨condition⟩ is not met, then m is compared with the next block.

Usually the ⟨code⟩ in a **recv from** block ends with a **send from to** clause of form **send** ⟨$\overline{msg}$⟩ **from** ⟨$\overline{sender}$⟩ **to** ⟨$\overline{receiver}$⟩ where $\overline{msg}$ is a message that is sent out in the name of ⟨$\overline{sender}$⟩, denoting a dedicated sender in this ITM, to ⟨$\overline{receiver}$⟩, either denoting a dedicated receiver in a higher-level protocol/subroutine or being `NET` denoting the adversary connected to the network interface. In cases where sender/receiver do not matter, one can omit those parts of **recv from to** and **send from to** . For the special case of **send from to** where a message is returned in the name of the currently active party to the sender who activated it, one can instead write **reply** ⟨$\overline{msg}$⟩.

If an ITM invokes another ITM, *e.g.*, as a subroutine, ITMs may expect an immediate response. In this case, in a **recv from** block, a **send to** statement is directly followed by a **wait for** statement. We write **wait for** ⟨$\overline{msg}$⟩ **from** ⟨$\overline{sender}$⟩ **s.t.** ⟨*condition*⟩ to denote that the ITM stays in its current state and discards all incoming messages until it receives a message m matching the pattern $\overline{msg}$ from the specified sender and fulfilling the **wait for** condition. Then the ITM continues the run where it left of, including all values of local variables.

The iUC framework supports a feature called responsive adversaries [15], where protocols can choose to send a network message to the attacker in such a way that the attacker is forced to return an immediate response to the sender, i.e., without interacting with other parts of the protocol and without altering the state of any other party. This is typically used to leak information or let the adversary decide on parameters without giving up the control flow of the protocol, which in turn simplies the reasoning in the security proof. If we want to make use of this feature, we write **send responsively** ⟨$\overline{msg}$⟩ **to** `NET` followed by a **wait for** statement.

To clarify the presentation and distinguish different types of variables, constants, strings, etc., we follow the naming conventions of Camenisch*et al.* [13]:

1. (Internal) state variables are denoted by sans-serif fonts.
2. Local (i.e., ephemeral) variables are denoted in *italic font*.
3. Keywords are written in **bold font** (*e.g.*, for operations such as **sending** or **receiving**).
4. Commands, procedure, function names, strings and constants are written in `teletype`.

We use the following additional nomenclature from [13]:

- $\mathsf{entity}_{\mathsf{cur}} := (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ denotes the currently active entity, i.e., the one that received a message. $\mathsf{entity}_{\mathsf{call}} := (\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})$ denotes the entity which called the currently active ITM by sending a message if the caller/sender was a higher-level protocol or a subroutine. In cases where the sender is not necessarily a protocol party but might also be the network adversary who does not have, e.g., a sender party ID $\mathsf{pid}_{\mathsf{call}}$, we still write $\mathsf{entity}_{\mathsf{call}}$ (but not, e.g., $\mathsf{pid}_{\mathsf{call}}$) to denote whoever the sender is.
- The macro **corr**$(pid, sid, role)$ queries the ITM of the entity $(pid, sid, role)$ to obtain the current corruption status of that entity. Internally, it sends a special `CorruptionStatus`? message and waits for the response.
- The macro **init**$(pid, sid, role)$ triggers the initialization of $(pid, sid, role)$ and then returns the activation to the calling ITM.
- Each machine implicitly keeps and updates two state variables CorruptionSet and transcript. CorruptionSet stores all entities (of that machine) that are currently considered corrupted. transcript is a transcript of all messages received and sent so far.