Password-Hardened Encryption Revisited

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

[1] Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany
ruben.baecker@fau.de
[2] TU Wien, Vienna, Austria
{paul.gerhart, dominique.schroeder}@tuwien.ac.at

Abstract. Passwords remain the dominant form of authentication on the Internet. The rise of single sign-on (SSO) services has centralized password storage, increasing the devastating impact of potential attacks and underscoring the need for secure storage mechanisms. A decade ago, Facebook introduced a novel approach to password security, later formalized in Pythia by Everspaugh et al. (USENIX'15), which proposed the concept of password hardening. The primary motivation behind these advances is to achieve provable security against offline brute-force attacks. This work initiated significant follow-on research (CCS'16, USENIX'17), including Password-Hardened Encryption (PHE) (USENIX'18, CCS'20), which was introduced shortly thereafter. Virgil Security commercializes PHE as a software-as-a-service solution and integrates it into its messenger platform to enhance security.

In this paper, we revisit PHE and provide both negative and positive contributions. First, we identify a critical weakness in the original design (USENIX'18) and present a practical cryptographic attack that enables offline brute-force attacks – the very threat PHE was designed to mitigate. This weakness stems from a flawed security model that fails to account for real-world attack scenarios and the interaction of security properties with key rotation, a mechanism designed to enhance security by periodically updating keys. Our analysis shows how the independent treatment of security properties in the original model leaves PHE vulnerable. We demonstrate the feasibility of the attack by extracting passwords in seconds that were secured by the commercialized but open-source PHE provided by Virgil Security.

On the positive side, we propose a novel, highly efficient construction that addresses these shortcomings, resulting in the first practical PHE scheme that achieves security in a realistic setting. We introduce a refined security model that accurately captures the challenges of practical deployments, and prove that our construction meets these requirements. Finally, we provide a comprehensive evaluation of the proposed scheme, demonstrating its robustness and performance.

1 Introduction

Passwords remain popular because they are easy to use, require no additional hardware, and ensure backward compatibility with existing systems. However, the growing number of cyberattacks and data breaches has led to the introduction of single sign-on (SSO) services, which simplify user management and offload the responsibility of password management from individual servers. While SSO reduces the burden on individual systems, it centralizes password storage, increasing the risk as attackers can now focus on compromising a few critical servers. For instance, in 2023, there were a record-breaking 3,205 data compromises, representing a 72% increase over the previous high recorded in 2021. These breaches exposed approximately 353 million individual records, underscoring the escalating threat of cyberattacks targeting user databases [14].

To address this growing issue, Facebook proposed a system involving a ratelimiter – a cryptographic service that remains oblivious to the password and assists only in its verification. This idea was initially formalized in Pythia [11] and later further developed into Password-Hardening (PH) [20] and Password-Hardened Encryption (PHE) [19]. The key innovation extends beyond simply protecting passwords, introducing a password-based key derivation mechanism. This proposal was quickly commercialized by Virgil Security as part of its software-as-a-service solution, demonstrating its immediate practical relevance and applicability.

Additionally, PH and PHE incorporate the capability for performing key updates without user involvement, a feature not only required by the PCI-DSS standard [26] but also widely recommended as a best

practice by major providers. For example, Google advises regular key rotation to mitigate security risks associated with long-term key usage [13]. Similarly, industry standards such as those from the National Institute of Standards and Technology (NIST) emphasize the importance of key rotation in maintaining robust security practices [4]. Regular key updates enhance security by cryptographically erasing leaked keys and introducing fresh, independent keys.

The rapid adoption of PHE by Virgil Security underscores the urgent need in the industry for solutions that secure password storage while complying with industry standards. However, adapting novel cryptographic primitives to practice always carries the same high risk: novel cryptography is often not well understood, both in terms of the security model and the resulting security of the schemes. In most cases, cryptography only provides security within the model in which it has been proven. If the model deviates significantly from real-world scenarios, the scheme may be insecure against practical attacks. In this work, we show that this is the case for PHE. The original security model evaluates several properties independently and does not consider key rotation as part of all security definitions. We show that this omission leads to a practical cryptographic attack that allows offline brute-force attacks on passwords, the very problem that PHE was designed to prevent. To illustrate the practical implications, we implement the attack using a fork of the PHE implementation by Virgil Security [25]. This attack pushes the state of password hardening back to before the invention of PHE, leaving an important question unanswered:

> *Are there efficient Password-based Key Derivation schemes that support rate-limiting and are secure in practice?*

1.1 Our Contribution

In this work, we make both negative and positive contributions to the study of password-hardened encryption (PHE):

Negative Contributions: We identify a critical security flaw in the formal model of PHE that enables a concrete cryptographic attack on the scheme commercialized by Virgil Security. Our analysis shows that the root cause of this vulnerability lies in a definitional gap: the absence of key rotation as part of some security definitions. To demonstrate the practicality of this attack, we forked the PHE framework of Virgil Security [25] and show how an adversary can (offline) brute-force passwords. We give a comprehensive discussion of the underlying definitional issues and their impact on the concrete attack against Simple PHE [19] and thus on the real-world implementation of Virgil Security in Section 2.

Positive Contributions: We address the identified gap by introducing a novel and more realistic security model for PHE, explicitly modeling key rotations as part of the life cycle of a PHE scheme. Building upon this enhanced model, we propose a new PHE design called HildeGUARD that maintains security even with alternating corruptions of the login-server and the ratelimiter. Furthermore, we provide a comprehensive evaluation that underscores both the practicality of the identified attack and the efficiency and robustness of our proposed solution.

This dual approach of identifying flaws and presenting solutions highlights the challenges and opportunities of adapting cryptographic primitives to real-world scenarios.

Real World Considerations: Beyond theoretical implications, our work directly impacts PHE deployment. We provide concrete benchmarks and two migration strategies to enable a seamless adoption of HildeGUARD. We show, that the migration from hashed passwords and SimplePHE is efficient, requiring no user re-enrollment; our simulations show the migration of one million users in under a minute. We also describe an opt-out mechanism, ensuring service providers retain the flexibility to transition away from HildeGUARD also without user interaction.

1.2 Related Work

Everspaugh et al. [11] initiated the study of password hardening (PH) systems, following an initial proposal by Facebook [22], and introduced constructions based on partially-oblivious pseudorandom functions. Since then, the cryptographic community has proposed several more efficient PH schemes. *PO-COM*, introduced by Schneider *et al.* [24], suffers from offline attacks in a stronger security model. *Phoenix*, proposed by Lai *et al.* [20], achieves security under a static corruption model and does not support key derivation capabilities.

In parallel, the concept of password-hardened encryption (PHE) emerged as a natural extension of the PH framework, allowing data to be encrypted under a user's password while retaining resistance to server compromise. Two PHE schemes are currently known: *Simple* PHE, introduced by Lai *et al.* [19], and its threshold variant TPHE, proposed by Brost *et al.* [6]. In this paper, we demonstrate an offline brute-force attack against *Simple* PHE, revealing a critical vulnerability in its design that contradicts its intended security goals. The TPHE construction by Brost *et al.* [6] is not affected by our attack as their security model considers semi-adaptive corruptions–explicitly capturing the attack scenario that we exploit.

Nonetheless, TPHE's robustness comes at a significant cost. The protocol requires six rounds of communication and relies heavily on expensive zero-knowledge proofs, making it difficult to deploy in real-world systems where latency, simplicity, and efficiency are paramount. Although one can theoretically instantiate TPHE with a single server to yield a PHE scheme, the associated overhead–both in communication and computation–remains prohibitive. In contrast, our construction is tailored for practical deployment. It achieves strong post-compromise guarantees under a corruption model that reflects the real world, while requiring only a single round of interaction and avoiding costly cryptographic tools where possible.

Other Password-based cryptographic primitives include several related terms such as *Password-Protected Secret Sharing (PPSS)*, *Password-Authenticated Key Exchange (PAKE)*, and *Password-Based Threshold Authentication (PbTA)*.
PPSS [3] allows threshold sharing of a secret across servers, requiring a password for retrieval, but lacks key rotation and requires per-user key storage [19,7,16]. In this context, per-user key storage means that each server must store a key for each user it serves. In contrast, the ratelimiter in (T)PH(E) stores only a single key, regardless of the number of users. *Password-Authenticated Key Exchange (PAKE)* and its extensions, such as *Threshold PAKE (T-PAKE)* [3,15] and *Two-Factor Authenticated Key Exchange (TFA-KE)* [17], allow users to establish session keys with servers, but do not support key rotation and also require per-user key storage. *Distributed Password Verification* [8], in an n-out-of-n setting, relies on offline backup tapes for key rotation and also requires per-user keys. *Password-Based Threshold Authentication (PbTA)* [1,5] produces verifiable authentication tokens, but modifies the user-server interface by replacing passwords with tokens and requires direct user interaction with cryptographic servers. Finally, *Practical Password Hardening Based on TLS* [10] is a deployable scheme that uses HMAC with a server's TLS secret key, but lacks key rotation and is vulnerable to offline dictionary attacks if the server is compromised. DPaSE [9] enables key-derivation based on a password, but it does not support key rotation.

2 Technical Outline

We begin by describing the interface used for interaction within the system, as it provides a natural starting point for deriving the intuitive security properties one would expect from such a system. Through this exploration, we show that the Lai et al. security model is inadequate and fails to address specific threats and scenarios that arise in practical deployment. To illustrate the real-world impact of the model's shortcomings, we outline a high-level attack that exploits these vulnerabilities. Finally, we present our novel scheme that achieves security within this model.

System outline. The following description refers to the key-encapsulation variant of PHE as implemented by Virgil Security. PHE involves three participants (cf. Figure 1): the user, who submits a username and password to derive an encapsulated key after providing valid credentials; the login server, which stores an enrollment representing an encryption of the user's credentials; and the ratelimiter, which validates requests without accessing the user's secret credentials.

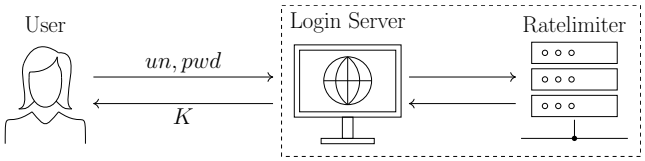

Fig. 1. Intuition of password-hardened encryption. The interfaces for the user remain the same, while the key can only be retrieved if both the server and the ratelimiter interact to run the decryption process.

The functional requirements are as follows: The user must remain stateless and perform no cryptographic operations to ensure full compatibility. This constraint also mandates a fixed interface on the login server. To avoid offline brute-force attacks, the login server cannot independently verify the correctness of a password and must interact with the ratelimiter for validation. Besides its own secret, the ratelimiter maintains only a minimal state for counting recent decryption attempts and limiting them. This especially means that the ratelimiter does not keep individual records per user. A critical feature of the system is support for key rotation, which requires sublinear communication between the server and the ratelimiter, while ensuring the user remains uninvolved.

Security Requirements. To formulate the desired security properties for PHE, we first consider the single-server setting as the basic model. From there, we incrementally extend the model, ensuring that each extension introduces improved security guarantees. This step-by-step approach allows us to systematically express the incremental improvements in security that each modification provides.

Single Server Setting. In the single-server setting, it is obvious that no security can be provided in the event of a server compromise, as all secrets and the password database would be exposed to the attacker. While modern cryptographic techniques such as memory-hard functions [23] can significantly increase the cost of brute-force attacks, they do not fully address this vulnerability. As a result, any security model that aims for the strongest guarantees must exclude server compromise scenarios, since achieving security under such conditions is fundamentally infeasible.

Two-Server Setting. To tolerate server compromises, it is necessary to introduce a second component that keeps its own secret. This principle of *distributed trust* is also applied in PHE with the introduction of a *ratelimiter*. Following the principle of *minimal exposure and knowledge*, the ratelimiter never receives the plaintext password or any other private information; instead, it is designed only to assist in the verification process.

In the simplest implementation of this setup, the server computes an HMAC over the password and a nonce, and then sends the resulting value to the ratelimiter. The ratelimiter, in turn, computes another HMAC on this received value and returns the resulting "double HMAC" to the server, which then stores it in the database.

Since the output of the HMAC is pseudorandom, the ratelimiter does not gain any information about the password. At the same time, this design ensures that verification requires the active participation of the ratelimiter, since the server alone cannot compute the double HMAC independently. Consequently, even if the password database and the server's private key are compromised, the stored value remains pseudorandom under the ratelimiter's key.

The introduction of an additional party, the ratelimiter, inherently creates a new potential attack vector, as an adversary could target this party in addition to the server. However, unlike the single-server setting, the model can now handle corruption of either the server or the ratelimiter independently, since the remaining party's secret ensures the pseudorandomness of the stored values. The security model should therefore be designed to tolerate the compromise of *either* the server *or* the ratelimiter. However, as in the single-server setting, it is impossible for a scheme to tolerate the compromise of *both* parties simultaneously, since there would be no remaining secret to protect the records storing the password. Thus, the security of this model depends on the assumption that at least one party remains uncompromised at all times.

Two-Server Setting With Key Rotation. In practice, it is unrealistic to assume that one of the parties will remain honest at all times. To overcome the impossibility result of the previous setting, we observe that in practice, compromise is often transient: once an intrusion is detected, operators typically respond with containment measures such as credential revocation or privilege reduction, effectively cutting off the attacker's access to the compromised system. A 2024 survey conducted by Fortinet [12] found that one-third of organizations detected six or more intrusions within a year–highlighting both the frequency of compromises and the feasibility of detecting them. If the secret of the once corrupted party remains unchanged after the adversary loses access, nothing is won: when the adversary corrupts the other party, it has access to both secret states, leaving the records storing the passwords unprotected. This is the case even though the parties are never compromised *simultaneously*.

To address this, various standards and best practices advocate *periodic rotation of cryptographic keys*. However, performing *non-trivial key updates*–where data is re-secured without decrypting and re-encrypting all stored information–poses significant challenges. In our setting, this problem is further complicated by the fact that the password must, to some extent, be part of the update process. For practical reasons, involving the user in these updates is not feasible. In addition, to ensure efficiency, the communication between the server and the ratelimiter during the key update process must remain *independent* of the number of stored records.

In our example, the use of the double-HMAC approach clearly does not support key rotation, as the construction lacks the necessary flexibility to update cryptographic keys without reprocessing all stored values. It appears that achieving efficient and secure key rotation would require the use of *number theoretic properties* or more advanced cryptographic primitives.

From the perspective of the *security model*, the goal is to capture the natural lifecycle of detection, revocation, and restoration that characterizes modern incident response. In other words, we want to tolerate corruption of both the server and the ratelimiter, provided that an honest key update has been performed in the meantime. The underlying intuition is that after a key rotation, the leaked key and the other party's rotated key would be "out of sync". This desynchronization ensures that even if one key is compromised, the remaining rotated key can still protect the security of the system.

It is reasonable to assume that an adversary who compromises one party cannot automatically compromise the second. This assumption is rooted in the architectural separation in PHE, where the ratelimiter is designed in a zero-trust fashion to operate independently. It may even be managed by a different organization. As a result, the server and the ratelimiter could rely on distinct hardware, software stacks, and operational policies. This infrastructural and administrative isolation creates a natural barrier to simultaneous compromise–especially considering the narrow time window that may exist between the first compromise and the execution of containment measures, including a key rotation. Moreover, one could schedule periodic, proactive key rotations to anticipate corruption even before it is detected. Therefore, assuming independent corruption is not only theoretically meaningful but also aligned with realistic deployment scenarios.

To formalize this intuition, the security model must explicitly allow the attacker to corrupt one party, trigger an honest key rotation, and then corrupt the other party. Note that the security cannot be restored through a dishonest key rotation, since any state transition controlled or observed by the adversary preserves full knowledge of the party's internal secrets. De-corruption upon an honest key-rotation is the standard in password-based cryptography [5,6,8,11]. Even in the scenario of alternating corruption, the record storing the password must remain secure; otherwise, the concept of key rotation loses its purpose. This critical aspect is precisely where the Simple PHE model falls short, leading to the break of their scheme. Although this attack is formally outside of their defined model, it highlights the essential need to design security models that are as close to practical considerations as possible. Filling this gap is crucial to ensuring robust security against realistic attack scenarios.

Breaking Simple PHE. To describe our offline brute-force attack against Simple PHE [19], we first recall the structure of the enrollment record and then give a detailed step-by-step explanation of the attack. Let sk_S and sk_R represent the private keys of the server and the ratelimiter, respectively. The enrollment record, which encodes the password pwd and hides the encapsulated key K, is structured as $T = (t_0, t_1, n_S, n_R)$,

where n_S and n_R are nonces, and the values t_0 and t_1 are computed as

$$t_0 = H_{S,0}(pwd, n_S)^{sk_S} \cdot H_{R,0}(n_R)^{sk_R}$$

and

$$t_1 = H_{S,1}(pwd, n_S)^{sk_S} \cdot H_{R,1}(n_R)^{sk_R} \cdot K^{sk_S}.$$

We assume that the target user is already enrolled with the secure password secPW, and we denote their enrolment record as $T^{\mathsf{hon}} = (t_0^{\mathsf{hon}}, t_1^{\mathsf{hon}}, n_S^{\mathsf{hon}}, n_R^{\mathsf{hon}})$ with

$$t_0^{\mathsf{hon}} = H_{S,0}(\mathsf{secPW}, n_S^{\mathsf{hon}})^{sk_S} \cdot H_{R,0}(n_R^{\mathsf{hon}})^{sk_R}$$

and

$$t_1^{\mathsf{hon}} = H_{S,1}(\mathsf{secPW}, n_S^{\mathsf{hon}})^{sk_S} \cdot H_{R,1}(n_R^{\mathsf{hon}})^{sk_R} \cdot K^{sk_S}.$$

Step 1 – Enrolling a Malicious User: In the first step, the adversary corrupts the ratelimiter and registers a malicious user using a password malPW. During the execution of this protocol, it reuses the nonce n_R^{hon} used during the enrollment of the target user. The resulting enrollment record $T^{\mathsf{mal}} = (t_0^{\mathsf{mal}}, t_1^{\mathsf{mal}}, n_S^{\mathsf{mal}}, n_R^{\mathsf{hon}})$ is stored at the honest server. The values t_0^{mal} and t_1^{mal} are computed as

$$t_0^{\mathsf{mal}} = \mathsf{H}_{S,0}(\mathsf{malPW}, n_S^{\mathsf{mal}})^{sk_S} \cdot \mathsf{H}_{R,0}(n_R^{\mathsf{hon}})^{sk_R}$$

and

$$t_1^{\mathsf{mal}} = \mathsf{H}_{S,1}(\mathsf{malPW}, n_S^{\mathsf{mal}})^{sk_S} \cdot \mathsf{H}_{R,1}(n_R^{\mathsf{hon}})^{sk_R} \cdot \hat{K}^{sk_S}.$$

The adversary now releases the ratelimiter[3], and the system performs an honest key-rotation, turning the malicious ratelimiter into an honest party.

Step 2 – Corrupting the Server: The next step is for the adversary to corrupt the server. Note that as a result of the key rotation, the keys of both the server and the ratelimiter have been updated. We refer to the updated keys as sk'_S and sk'_R respectively. Since the attacker corrupts the server, it knows sk'_S and it also knows the password malPW, and can extract the ratelimiter part of t_0^{mal} as follows

$$\mathsf{H}_{R,0}(n_{\mathsf{R}}^{\mathsf{hon}})^{sk'_{\mathsf{R}}} = t_0^{\mathsf{mal}} / \mathsf{H}_{S,0}(n_{\mathsf{S}}^{\mathsf{mal}}, \mathsf{malPW})^{sk'_{\mathsf{S}}}.$$

Given that this value is the same in t_0^{hon}, the malicious server can extract the hash of the honest user's password from t_0^{hon}:

$$\mathsf{H}_{S,0}(n_{\mathsf{S}}^{\mathsf{hon}}, \mathsf{secPW}) = \left(t_0^{\mathsf{hon}} / \mathsf{H}_{R,0}(n_{\mathsf{R}})^{sk'_{\mathsf{R}}} \right)^{1/sk'_{\mathsf{S}}}.$$

Given the hash, it is easy to see that the adversary can simply perform an offline brute-force attack. To validate the practical impact of this vulnerability, we implemented the attack using the open-source framework maintained by Virgil Security. Full implementation details and attack traces are provided in Section 5.1.

A Novel Protocol. Our protocol shares the idea, similar to the Simple PHE protocol, that the enrolment record consists of two components–an authentication token and the encryption of the encapsulated key. Otherwise, the two protocols are fundamentally different. Below we provide an intuition of the main components, with a full description available in Section 4.

Protection against malformed ciphertexts: A critical vulnerability in the original protocol arises from the attacker's ability to reuse his own nonce to "unblind" a target user's enrollment record. To counter this, we introduce a shared nonce, derived jointly by the server and the ratelimiter as $n \leftarrow \mathsf{H}(n_S, n_R)$. This joint nonce ensures that both parties contribute to the randomness, reducing the risk of malicious reuse or manipulation.

[3] As described above, this is not part of a deliberate tactic of the adversary but rather the consequence of effective incident response.

Protection against database leakage: To prevent information leakage from a compromised database, the server encrypts all elements of the enrolment record using a CCA-secure encryption scheme with a symmetric private key, sk_S. This encryption ensures that even if the database is leaked to a malicious ratelimiter, no information about the cryptographic key or salted password hash is exposed. In addition, the protocol supports key rotation: when the server updates its private key, it locally decrypts and re-encrypts all records. This guarantees that even if the ratelimiter was malicious during the attack, the records remain secure as long as the server's private key is uncompromised.

Secure Decryption Protocol: The decryption process has been fundamentally redesigned to ensure that components of the enrollment record and exchanged messages cannot be reused across multiple instances. This is achieved by incorporating ephemeral keys into the decryption protocol. The ephemeral key introduces session-specific randomness, effectively defeating replay attacks and ensuring that all cryptographic operations remain unique and secure in each instance.

3 Password-Hardened Encryption

This section defines the interfaces, explains the security model intuitively, and formally describes the security games.

3.1 Interfaces

We largely follow the work of [19], but change the encryption protocol slightly. From the user's point of view, the encryption protocol as defined in [19] takes a password and a message as input and stores a ciphertext on the server. In practice, this message is most likely a key for a private key encryption scheme used to encrypt the actual payload. This is how Virgil Security uses PHE [25]. As was done for Password-Protected Secret Sharing [15], we change the interface to reflect this practical use in the theoretical definitions. Instead of taking a message as input to be encrypted, the encryption functionality takes only the user's password and returns a fresh key and a ciphertext in which the key is stored. We define a password-hardened encryption scheme as follows:

Definition 1 (Password-Hardened Encryption). *A* password-hardened encryption (PHE) *scheme, consists of four efficient algorithms* $(\mathsf{Setup}, \mathsf{SKeyGen}, \mathsf{RKeyGen}, \mathsf{Update})$ *and three efficient protocols* $(\langle \mathsf{S}, \mathsf{R} \rangle_{\mathsf{enc}}, \langle \mathsf{S}, \mathsf{R} \rangle_{\mathsf{dec}}, \langle \mathsf{S}, \mathsf{R} \rangle_{\mathsf{rot}})$*, that we define as follows:*

$pp \leftarrow\!\$\ \mathsf{Setup}(1^\lambda)$*: The setup algorithm takes the security parameter* 1^λ *as input and returns public parameters* pp*.*

$(pk_\mathsf{S}, sk_\mathsf{S}) \leftarrow\!\$\ \mathsf{SKeyGen}(pp)$*: The server key generation algorithm takes the public parameter* pp *as input and returns a server key pair* $(pk_\mathsf{S}, sk_\mathsf{S})$*.*

$(pk_\mathsf{R}, sk_\mathsf{R}) \leftarrow\!\$\ \mathsf{RKeyGen}(pp)$*: The ratelimiter key generation algorithm takes the public parameter* pp *as input and returns a ratelimiter key pair* $(pk_\mathsf{R}, sk_\mathsf{R})$*. For brevity, we assume that all algorithms and in protocols all parties take* pp, pk_S*, and* pk_R *as inputs.*

$((C, K), \epsilon) \leftarrow\!\$\ \langle \mathsf{S}(sk_\mathsf{S}, pwd), \mathsf{R}(sk_\mathsf{R}) \rangle_{\mathsf{enc}}$*: The encryption protocol is run between the server and the ratelimiter. The server takes its secret key* sk_S *and the password* pwd *as inputs. The ratelimiter takes its secret key* sk_R *as input. The server returns a ciphertext* C *and a key* K*, while the ratelimiter returns the empty string* ϵ*.*

$(K, \epsilon) \leftarrow \langle \mathsf{S}(sk_\mathsf{S}, pwd, C), \mathsf{R}(sk_\mathsf{R}) \rangle_{\mathsf{dec}}$*: The decryption protocol is run between the server and the ratelimiter. The server takes its secret key* sk_S*, a password* pwd*, and the ciphertext* C *as inputs. The ratelimiter takes its secret key* sk_R *as input. The server returns a key* K *and the ratelimiter returns the empty string* ϵ*.*

$((sk_\mathsf{S}', pk_\mathsf{R}', \tau), (sk_\mathsf{R}', pk_\mathsf{S}')) \leftarrow\!\$\ \langle \mathsf{S}(sk_\mathsf{S}), \mathsf{R}(sk_\mathsf{R}) \rangle_{\mathsf{rot}}$*: The key rotation protocol is run between the server and the ratelimiter. The server and the ratelimiter take their respective secret key* sk_S *or* sk_R *as input. The server returns its new secret key* sk_S'*, the ratelimiter's updated public key* pk_R'*, and an update token* τ*. The ratelimiter returns its new secret key* sk_R' *and the server's updated public key* pk_S'*.*

$C' \leftarrow \mathsf{Update}(sk_\mathsf{S}, \tau, C)$*: The update algorithm takes the server's secret key* sk_S*, the update token* τ*, and the ciphertext* C *as inputs. It returns an updated ciphertext* C'*.*

3.2 Security Model

In this section, we close the gap in the existing security definition by Lai *et al.* [19] by proposing enhanced security definitions for PHE. We follow the approach by Brost *et al.* [6] by using the two security definitions, where each definition fully covers one of the security dimensions. Our hiding definition extends the domain of password-hardened encryption to a semi-adaptive corruption model.

Hiding. The main limitation of recent security definitions for PHE (cf. [19]) is that they handle different attack vectors in separate games. Partial obliviousness only considers the corruption of the ratelimiter while ignoring the possibility of a corrupt server. Hiding only covers security against a corrupt server while ignoring corrupt ratelimiters. Formally speaking, Lai *et al.* restrict their definition to a static corruption model which implies that an adversary can only ever corrupt a single party but never both, one after the other.

In conclusion, Lai *et al.* assume that all adversaries restrict themselves to exploiting only one attack vector even if a combination of attack vectors would lead to a successful attack. This assumption is unrealistic, as real-world adversaries will exploit all available attack vectors concurrently to accomplish their objectives. As shown in Section 2, this theoretical gap can lead to real-world attacks, like the one we have found against SimplePHE [19]. To close this gap, we follow the approach by Brost *et al.* [6] and merge the security notions of partial obliviousness, hiding, and forward security into a single, comprehensive definition called *hiding*, which allows semi-adaptive corruptions of the server and the ratelimiter. In addition, our definition of hiding offers an important improvement by enforcing rate-limiting, a feature not present in the definition proposed by Brost *et al.*

It is worth noting that we cannot simply remove the threshold-related parts of the TPHE definitions of [6] to obtain definitions for PHE as we have modified the encryption protocol to fit the encapsulation setting. Consequently, we no longer have a distinguishing game where the adversary has to decide which of two messages is encrypted in a ciphertext, but rather a search game where the adversary has to find the uniformly random key encrypted in a ciphertext. This change in comparison to [6] allows for more efficient instantiations while still providing meaningful guarantees for real-world deployments, as in practice, the encrypted message is a uniformly random key anyway.

Corruption Model. We require that an honestly executed key rotation must reset the corruption. Therefore, we split the execution of a PH scheme into epochs separated by honest key rotations. During an honest key rotation, all parties are honest. The update token is used only within this honest time frame and safely deleted before any party is corrupted. If the adversary were able to obtain the update token, it could recompute the private state transition of the honest key-rotation, making it a dishonest key rotation that does not reset corruption.

In every epoch, either the server or the ratelimiter can be corrupted. It is important to note that achieving security against a simultaneous corruption of the ratelimiter and the server is generally impossible. Following [6], we focus on semi-adaptive corruption, where the desired corruption has to be announced *before* the next key-rotation, instead of fully adaptive corruption, where corruption can happen at any time. In our security proof, we inject different challenges according to the upcoming corruption during the key rotation. To achieve security under fully adaptive corruption, we would have to guess the corruption beforehand to inject the correct challenge. This reduces the tightness of the proof by a factor of $(1/2)^{Q_{\mathsf{rot}}}$, where Q_{rot} is the number of key rotations.

A Corrupted Server. When the server is corrupted, it trivially learns all passwords and keys for newly generated ciphertexts from the encryption request. Similarly, it learns all passwords used to decrypt ciphertexts and the key if the decryption is successful. However, all remaining passwords should not be brute-forceable by a corrupted server and the keys should remain hidden from it.

A Corrupted Ratelimiter. When the ratelimiter is corrupted, it can refuse to answer requests, thus preventing the scheme from working. Nevertheless, it should not learn any password or key used during encryption or decryption and should not be able to brute-force passwords without the server, even if the records are leaked.

Furthermore, a corrupt ratelimiter should not be able to make a decryption attempt successful if an incorrect password is used.

Enforced Rate-limiting. Existing security definitions for PHE (and PH) impose an upper limit on the number of decryption attempts for the challenge ciphertext (resp. the challenge enrollment record), even if the ratelimiter (or server) would continue to respond to these requests. This scenario creates a situation where PHE/PH schemes can be proven secure even without any rate-limiting measures implemented within the scheme, because the security game already handles the rate-limiting. In addition, all existing PHE and PH schemes rely solely on the ratelimiter to limit decryption attempts. As a result, a malicious ratelimiter can use unrestricted server interactions to brute-force a password while disregarding its own quota bounds. To address this limitation in current definitions, we allow the adversary to query the decryption oracle arbitrarily often, forcing the scheme to implement rate-limiting measures on both the server and the ratelimiter side.

The Hiding Game. We address the above issues in the security game of hiding (Figure 2). The security game samples keys for the server and ratelimiter and draws a random challenge password pwd^* from a fixed password space $\mathcal{PW}$. The challenge password pwd^* is used in an encryption protocol to generate a ciphertext C^* and a key K^*. The adversary wins the security game if it correctly guesses which key is encrypted in the challenge ciphertext C^*. Furthermore, the adversary can interact with honest parties through multiple oracles which we intuitively describe in the following. We formalize the oracles in Figure 3.
Encryption & Decryption Oracle: The oracle runs the corresponding protocol with the adversary, simulating the non-corrupted party. It returns the ciphertext and key or the decrypted key, respectively.
Challenge Oracles: If the server is not corrupted during the current epoch, the challenge encryption/decryption oracle calls the corresponding "standard" encryption/decryption oracle on the challenge password. The challenge encryption oracle can only be called once and does not return the challenge key.
Rotation Oracles: The honest rotation oracle performs an honest key rotation and resets the corruption state. It returns a secret key according to the adversary's corruption choice Corr' for the next epoch. The dishonest rotation oracle performs a key rotation with the adversary, simulating the non-corrupted party.
Update Oracle: The update oracle uses the current update token τ to update a ciphertext C into an updated ciphertext C' and returns it.

Definition 2 (Hiding). *A password-hardened encryption scheme* PHE *is semi-adaptively* hiding *if, for any* PPT *adversary* $\mathcal{A}$*,* $Q_{\mathtt{val}} \geq 0$*, and any password space* $\mathcal{PW}$[4]*, there exists a negligible function* negl*, such that for all* $\lambda \in \mathbb{N}$

$$\Pr[\mathsf{Hid}_{\mathsf{PHE},\mathcal{A},Q_{\mathsf{dec}},\mathcal{PW}}(\lambda) = 1] \leq \frac{Q_{\mathsf{dec}}}{|\mathcal{PW}|} + \mathsf{negl}(\lambda),$$

where Q_{dec} *is the rate-limiting threshold of the* PHE *scheme. The randomness is taken over the random coins of all randomized algorithms. The game* $\mathsf{Hid}_{\mathsf{PHE},\mathcal{A},Q_{\mathsf{dec}},\mathcal{PW}}(\lambda)$ *is defined in Figure 2.*

Binding. Intuitively, binding ensures that a corrupt ratelimiter cannot influence en-/decryption in a way that an incorrect key is obtained. In other words, if decryption returns a key, it is always the same one as returned during encryption, even if the ratelimiter is corrupt. Because the derived key is then used outside of PHE to en-/decrypt data, the importance of always correct keys becomes clearer when looking at the impact there. Assume, for example, that the ratelimiter is temporarily corrupt and deviates from the protocol such that an incorrect key is returned to the user. The user then uses this incorrect key to encrypt data. Now, even if the ratelimiter is honest again, the user cannot obtain the same key to decrypt the data. Consequently, this data is lost forever.

[4] For simplicity, we assume a uniform password distribution. Extending the definition to arbitrary password distributions is straightforward and only requires replacing $\frac{Q_{\mathsf{dec}}}{|\mathcal{PW}|}$ in the upper bound to a min-entropy term.

$\mathsf{Hid}_{\mathsf{PHE},\mathcal{A},Q_{\mathsf{dec}},\mathcal{PW}}(\lambda)$

$pp \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda)$
$(pk_\mathsf{S}, sk_\mathsf{S}) \leftarrow\!\!\$\ \mathsf{SKeyGen}(pp), (pk_\mathsf{R}, sk_\mathsf{R}) \leftarrow\!\!\$\ \mathsf{RKeyGen}(pp)$
$\mathsf{Corr} := \bot, \tau := \epsilon, K^* := \epsilon, pwd^* \leftarrow\!\!\$\ \mathcal{PW}$
$\mathbb{O} := \{\mathcal{O}_{\mathsf{enc}}, \mathcal{O}_{\mathsf{dec}}, \mathcal{O}_{\mathsf{encCh}}, \mathcal{O}_{\mathsf{decCh}},$
$\quad \mathcal{O}_{\mathsf{HonRot}}, \mathcal{O}_{\mathsf{DishonRot}}, \mathcal{O}_{\mathsf{Update}}\}$
$K' \leftarrow \mathcal{A}^{\mathbb{O}}(1^\lambda, pk_\mathsf{S}, pk_\mathsf{R})$
return $K' = K^*$

Fig. 2. The security game of hiding for PHE.

Lai *et al.* [19] and Brost *et al.* [6] even go beyond that and define a stronger notion that they call soundness. Intuitively, soundness requires that a corrupt ratelimiter cannot make the server output an incorrect key for the correct password and vice versa. We argue that this is unnecessarily strong because binding together with hiding already covers the same scenarios as their soundness definitions. We show that all scenarios are covered as intended:

- **Correct password:**
  - **Correct key:** This is the intended behaviour of the protocol.
  - **Incorrect key:** This is not possible as binding ensures that if a key is returned, it is the correct one.
  - **No key:** This is trivially achievable by a corrupt ratelimiter by refusing to answer or by sending an incorrect proof. This scenario is also not covered by the other soundness definitions.
- **Incorrect password:**
  - **Correct key:** This is not possible as otherwise, the corrupt ratelimiter could break the hiding property of the PHE scheme.
  - **Incorrect key:** This is not possible as binding ensures that if a key is returned, it is the correct one.
  - **No key:** This is the intended behaviour of the protocol.

Therefore, we capture the above requirement in a novel definition that we call binding. In the binding game, the adversary is given access to an encryption and a decryption oracle. In both games, the adversary provides all inputs to the server, including the server's secret key and the used randomness. The adversary wins if it can interact with the honest server in two protocol runs using the same server secret key, ratelimiter public key, and ciphertext but still make the server output two different keys. We formally define binding in Definition 3.

Definition 3 (Binding). *A password-hardened encryption scheme* PHE *is* binding *if, for any* PPT *adversary* $\mathcal{A}$ *there exists a negligible function* negl*, such that for all* $\lambda \in \mathbb{N}$

$$\Pr[\mathsf{Bind}_{\mathsf{PHE},\mathcal{A}}(\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where the randomness is taken over the random coins of all randomized algorithms. The game $\mathsf{Bind}_{\mathsf{PH},\mathcal{A}}(\lambda)$ *is defined in Figure 4.*

4 HildeGUARD

In this section, we propose HildeGUARD, the first round optimal password hardened encryption scheme secure in a semi-adaptive corruption model. One of our primary design goals in developing HildeGUARD was to rely exclusively on well-established cryptographic primitives–specifically, elliptic curves, hash functions, and

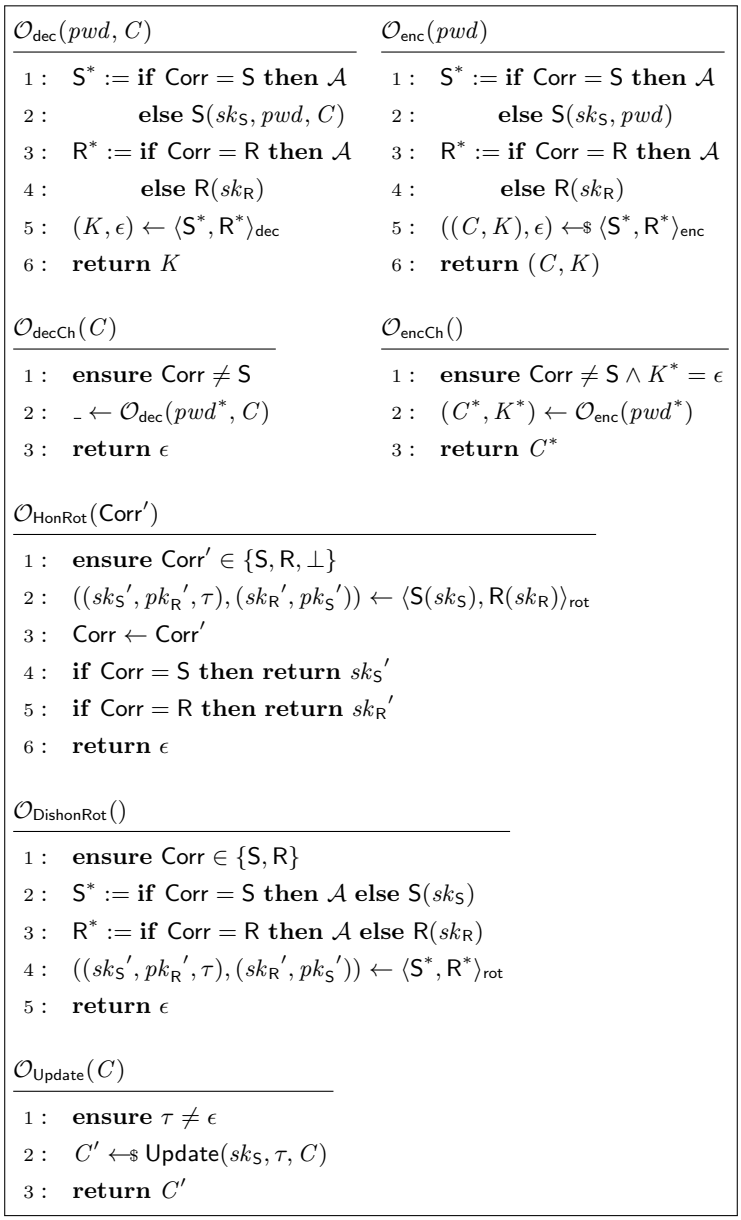

Fig. 3. Oracles for the semi-adaptive PHE hiding definition.

AES encryption. By avoiding more complex or experimental cryptographic constructions, such as pairing-friendly curves, we reduce the risk of subtle implementation flaws and side-channel vulnerabilities that often accompany advanced primitives. This approach allows for a minimalistic codebase that is easier to audit and maintain.

In the following, let $\mathbb{G}$ be a multiplicative finite cyclic group of order $q = q(\lambda)$ and $\mathsf{H}_{\{\mathsf{S},\mathsf{R},\mathsf{B},\mathsf{F}\}} : \{0,1\}^* \to \mathbb{G}$ are hash functions. Let Π_{Enc} be a CCA-secure private-key encryption scheme. If an assertion fails during the execution of an algorithm, it terminates and returns the special symbol $\perp$ to indicate an error.

In HildeGUARD, we use a single nonce $n \leftarrow \mathsf{H}_{\mathsf{N}}(n_{\mathsf{S}}, n_{\mathsf{R}})$ that combines the randomness contributed by the server and the ratelimiter for rate-limiting. To ensure that decryption attempts are bound to this rate-limit,

$\mathsf{Bind}_{\mathsf{PHE},\mathcal{A}}(\lambda)$

1 : $\mathsf{Queries} := \emptyset$
2 : $(i, j) \leftarrow\!\$\ \mathcal{A}^{\mathcal{O}_{\mathsf{enc}},\mathcal{O}_{\mathsf{dec}}}(1^\lambda)$
3 : $(sk_{\mathsf{S}}, pk_{\mathsf{R}}, C, K) := \mathsf{Queries}[i]$
4 : $(sk_{\mathsf{S}}', pk_{\mathsf{R}}', C', K') := \mathsf{Queries}[j]$
5 : $b_0 \leftarrow (sk_{\mathsf{S}}, pk_{\mathsf{R}}, C) = (sk_{\mathsf{S}}', pk_{\mathsf{R}}', C')$
6 : $b_1 \leftarrow (K \neq \bot) \wedge (K' \neq \bot)$
7 : $b_2 \leftarrow K \neq K'$
8 : **return** $b_0 \wedge b_1 \wedge b_2$

$\mathcal{O}_{\mathsf{enc}}(sk_{\mathsf{S}}, pk_{\mathsf{R}}, pwd, r)$

1 : $\mathcal{S} =: \mathsf{S}(sk_{\mathsf{S}}, pwd^*, pk_{\mathsf{R}}; r)$
2 : $((C, K), \epsilon) \leftarrow\!\$\ \langle \mathcal{S}, \mathcal{A} \rangle_{\mathsf{enc}}$
3 : $\mathsf{Queries} \stackrel{\cup}{=} \{(sk_{\mathsf{S}}, pk_{\mathsf{R}}, C, K)\}$
4 : **return** ϵ

$\mathcal{O}_{\mathsf{dec}}(sk_{\mathsf{S}}, pk_{\mathsf{R}}, pwd, C, r)$

1 : $\mathcal{S} =: \mathsf{S}(sk_{\mathsf{S}}, pwd^*, C, pk_{\mathsf{R}}; r)$
2 : $(K, \epsilon) \leftarrow\!\$\ \langle \mathcal{S}, \mathcal{A} \rangle_{\mathsf{dec}}$
3 : $\mathsf{Queries} \stackrel{\cup}{=} \{(sk_{\mathsf{S}}, pk_{\mathsf{R}}, C, K)\}$
4 : **return** ϵ

Fig. 4. PHE binding experiment.

the nonce is used in the computations of the server and the ratelimiter. Besides the nonce, an enrollment record consists of two components t_0 and t_1 that are computed as

$$t_0 \leftarrow \mathsf{H}_{\mathsf{R}}(n, 0)^{sk_{\mathsf{R}}} \cdot \mathsf{H}_{\mathsf{S}}(pwd, n)$$
$$t_1 \leftarrow \mathsf{H}_{\mathsf{R}}(n, 1)^{sk_{\mathsf{R}}} \cdot K.$$

The ratelimiter parts of these computations $h_{R,0} = \mathsf{H}_{\mathsf{R}}(n, 0)^{sk_{\mathsf{R}}}$ and $h_{R,1} = \mathsf{H}_{\mathsf{R}}(n, 1)^{sk_{\mathsf{R}}}$ can be thought of as an PRF evaluation on the nonce $\mathsf{PRF}_{sk_{\mathsf{R}}}^{\{0,1\}}(n)$. Consequently, t_0 is–on an intuitive level–the first ratelimiter PRF multiplied by the salted hash of the password, and t_1 the second ratelimiter PRF multiplied by the encapsulated key. From the server's perspective, both values are uniformly random because of the pseudorandom values computed by the ratelimiter. To ensure that the record does not leak anything to the ratelimiter, the whole record has to be encrypted by the server, because the ratelimiter knows the corresponding PRF key and consequently also the PRF values.

The decryption protocol can be thought of as a password-authenticated key exchange protocol (PAKE), where the key exchange only succeeds if both parties enter the same password. In contrast to standard PAKE, the server and the ratelimiter enter the first ratelimiter PRF value $h_{R,0}$ instead of the password. While the ratelimiter can directly compute $h_{R,0}$, the server has to use the correct password to extract the PRF value from the stored record $h'_{R,0} \leftarrow t_0 / \mathsf{H}_{\mathsf{S}}(pwd', n)$. The ratelimiter uses the resulting key to encrypt the second ratelimiter PRF value $h_{R,1}$ before sending it to the server. If the entered password is correct, the server obtains the correct decryption key and can decrypt $h_{R,1}$ to deblind t_1, yielding K.

4.1 Description of HildeGUARD

Encryption Protocol. To generate a new cryptographic key K protected by the password pwd, the server begins by sampling a random nonce n_{S} and sending it to the ratelimiter. The ratelimiter also samples a random nonce n_{R} and derives a combined nonce $n \leftarrow \mathsf{H}_{\mathsf{N}}(n_{\mathsf{S}}, n_{\mathsf{R}})$. This unified nonce is one of the countermeasures mitigating malformed ciphertexts when using a semi-adaptive corruption model. To proceed, the ratelimiter evaluates two PRFs on n, using domain separation via a single bit: it computes $h_{\mathsf{R},0} \leftarrow \mathsf{H}_{\mathsf{R}}(n, 0)^{sk_{\mathsf{R}}}$ and $h_{\mathsf{R},1} \leftarrow \mathsf{H}_{\mathsf{R}}(n, 1)^{sk_{\mathsf{R}}}$. Finally, it constructs a Schnorr proof of well-formedness of $h_{\mathsf{R},0}$ and $h_{\mathsf{R},1}$ and sends the PRF values alongside the nonce-share n_{R}, and the proof π to the server. Receiving these values, the server validates the proof π. As described above, $h_{\mathsf{R},0}$ is used to blind the salted hash of the password, while $h_{\mathsf{R},1}$ is used to blind the cryptographic key K. Consequently, the values t_0 and t_1 are computed as

$$t_0 \leftarrow \mathsf{H}_{\mathsf{R}}(n, 0)^{sk_{\mathsf{R}}} \cdot \mathsf{H}_{\mathsf{S}}(pwd, n)$$
$$t_1 \leftarrow \mathsf{H}_{\mathsf{R}}(n, 1)^{sk_{\mathsf{R}}} \cdot K.$$

$\mathsf{Setup}(1^\lambda)$	$\mathsf{SKeyGen}(pp)$	$\mathsf{RKeyGen}(pp)$
$g \leftarrow\!\!\$\ \mathbb{G}$	$sk_\mathsf{S} \leftarrow\!\!\$\ \Pi_{\mathsf{Enc}}.\mathsf{Gen}(1^\lambda)$	$sk_\mathsf{R} \leftarrow\!\!\$\ \mathbb{Z}_q^*$
$crs \leftarrow\!\!\$\ \Pi_{\mathsf{NIZK}}(1^\lambda)$	$pk_\mathsf{S} \leftarrow \epsilon$	$pk_\mathsf{R} \leftarrow g^{sk_\mathsf{R}}$
$\mathsf{H}_{\{\mathsf{S},\mathsf{R},\mathsf{B},\mathsf{F}\}} \leftarrow\!\!\$\ \{\mathsf{H} : \{0,1\}^* \to \mathbb{G}\}$	**return** $(pk_\mathsf{S}, sk_\mathsf{S})$	**return** $(pk_\mathsf{R}, sk_\mathsf{R})$
$\mathsf{H}_\mathsf{N} \leftarrow\!\!\$\ \{\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda\}$		
return $pp \leftarrow (g, crs, \mathsf{H}_{\{\mathsf{S},\mathsf{R},\mathsf{B},\mathsf{F},\mathsf{N}\}})$		

Fig. 5. The setup algorithms of HildeGUARD.

Finally, the server encrypts the record (t_0, t_1, n) using its secret key sk_S and stores the ciphertext. It is important to note that the encapsulated key is a group element. Before using it outside of HildeGUARD, it should be processed by a key derivation function to ensure that the key is a uniformly random bit string. We formally describe the encryption protocol in Figure 6.

Encryption Protocol		
Server(sk_S, pwd)		**Ratelimiter**(sk_R)
$K \leftarrow\!\!\$\ \mathbb{G}, n_\mathsf{S} \leftarrow\!\!\$\ \{0,1\}^\lambda$	$\xrightarrow{n_\mathsf{S}}$	$n_\mathsf{R} \leftarrow\!\!\$\ \{0,1\}^\lambda, n \leftarrow \mathsf{H}_\mathsf{N}(n_\mathsf{S}, n_\mathsf{R})$
		$h_{\mathsf{R},0} \leftarrow \mathsf{H}_\mathsf{R}(n,0)^{sk_\mathsf{R}}$
		$h_{\mathsf{R},1} \leftarrow \mathsf{H}_\mathsf{R}(n,1)^{sk_\mathsf{R}}$
		$stmt \leftarrow (\mathsf{H}_\mathsf{R}(n,0), h_{\mathsf{R},0}, \mathsf{H}_\mathsf{R}(n,1), h_{\mathsf{R},1}, g, pk_\mathsf{R})$
$n \leftarrow \mathsf{H}_\mathsf{N}(n_\mathsf{S}, n_\mathsf{R})$	$\xleftarrow{(h_{\mathsf{R},0}, h_{\mathsf{R},1}, n_\mathsf{R}, \pi)}$	$\pi \leftarrow\!\!\$\ \mathsf{Prove}(crs, stmt, sk_\mathsf{R})$
$stmt \leftarrow (\mathsf{H}_\mathsf{R}(n,0), h_{\mathsf{R},0}, \mathsf{H}_\mathsf{R}(n,1), h_{\mathsf{R},1}, g, pk_\mathsf{R})$		
ensure $\mathsf{Verify}(crs, stmt, \pi) = 1$		
$h_\mathsf{S} \leftarrow \mathsf{H}_\mathsf{S}(pwd, n)$		
$t_0 \leftarrow h_{\mathsf{R},0} \cdot h_\mathsf{S}$		
$/\!/\ t_0 = \mathsf{H}_\mathsf{R}(n,0)^{sk_\mathsf{R}} \cdot \mathsf{H}_\mathsf{S}(pwd, n)$		
$t_1 \leftarrow h_{\mathsf{R},1} \cdot K$		
$/\!/\ t_1 = \mathsf{H}_\mathsf{R}(n,1)^{sk_\mathsf{R}} \cdot K$		
$C \leftarrow \mathsf{Enc}_{sk_\mathsf{S}}(t_0, t_1, n)$		
return (C, K)		

Fig. 6. The encryption protocol of HildeGUARD.

Decryption Protocol. The decryption protocol serves two primary purposes: it authenticates the user by verifying the candidate password and, upon successful authentication, provides the cryptographic key K to the user. To defend against online brute-force attacks, both the server and the ratelimiter enforce rate-limiting via a **ratelimit** function that tracks the number of decryption attempts per user. This function is initialized with a fixed quota $Q_{\texttt{val}}$ and aborts the protocol if the limit is exceeded, thereby blocking excessive login attempts.

The protocol begins with the server decrypting the ciphertext to recover the tags t_0 and t_1. It then divides t_0 by the hash of the candidate password $\mathsf{H_S}(pwd', n)$, yielding $h'_{\mathsf{R},0} = \mathsf{H_R}(n, 0)^{sk_\mathsf{R}}$ if and only if $pwd' = pwd$. Next, the server derives a group generator h'_B from $h'_{\mathsf{R},0}$ using a hash function $h'_\mathsf{B} \leftarrow \mathsf{H_B}(h'_{\mathsf{R},0})$. This generator serves as the base in a Diffie-Hellman-like key exchange. The server begins as in a standard Diffie-Hellman protocol by sampling an ephemeral secret key r_S and sending the public key $x \leftarrow h'^{r_\mathsf{S}}_\mathsf{B}$ to the ratelimiter.

The ratelimiter computes $h_{\mathsf{R},0} \leftarrow \mathsf{H_R}(n, 0)^{sk_\mathsf{R}}$ and also derives a group generator $h_\mathsf{B} \leftarrow \mathsf{H_B}(h_{\mathsf{R},0})$. If the password is correct, the server and the ratelimiter will derive the same generator $h'_\mathsf{B} = h_\mathsf{B}$. The ratelimiter then samples an ephemeral secret key r_R and computes what would be the final key in a Diffie-Hellman key-exchange $y_0 \leftarrow x^{r_\mathsf{R}}$. It uses the public key associated with its ephemeral secret key $h^{r_\mathsf{R}}_\mathsf{B}$ as the final key. To protect the second PRF output, the ratelimiter uses a hash function to derive a one-time pad from the final key: $\mathsf{H_F}(h^{r_\mathsf{R}}_\mathsf{B})$. It uses this pad to encrypt $h_{\mathsf{R},1} \leftarrow \mathsf{H_R}(n, 1)^{sk_\mathsf{R}}$, resulting in $y_1 \leftarrow \mathsf{H_F}(h^{r_\mathsf{R}}_\mathsf{B}) \cdot h_{\mathsf{R},1}$. Finally, the ratelimiter sends y_0, y_1, and a Schnorr proof π–proving the well-formedness of $h_{\mathsf{R},1}$–to the server.

Upon receiving the ratelimiter's response, the server "removes" its ephemeral secret key r_S from y_0 to obtain the final key $y_0^{r_\mathsf{S}^{-1}} = h'^{r_\mathsf{R}}_\mathsf{B}$. It derives the one-time pad $\mathsf{H_F}(y_0^{r_\mathsf{S}^{-1}})$ and uses it to decrypt y_1: $h'_{\mathsf{R},1} \leftarrow y_1 / \mathsf{H_F}(y_0^{r_\mathsf{S}^{-1}})$. After validating the proof π, the server recovers K by computing $K \leftarrow t_1 / h'_{\mathsf{R},1}$. Figure 7 shows the decryption protocol of HildeGUARD.

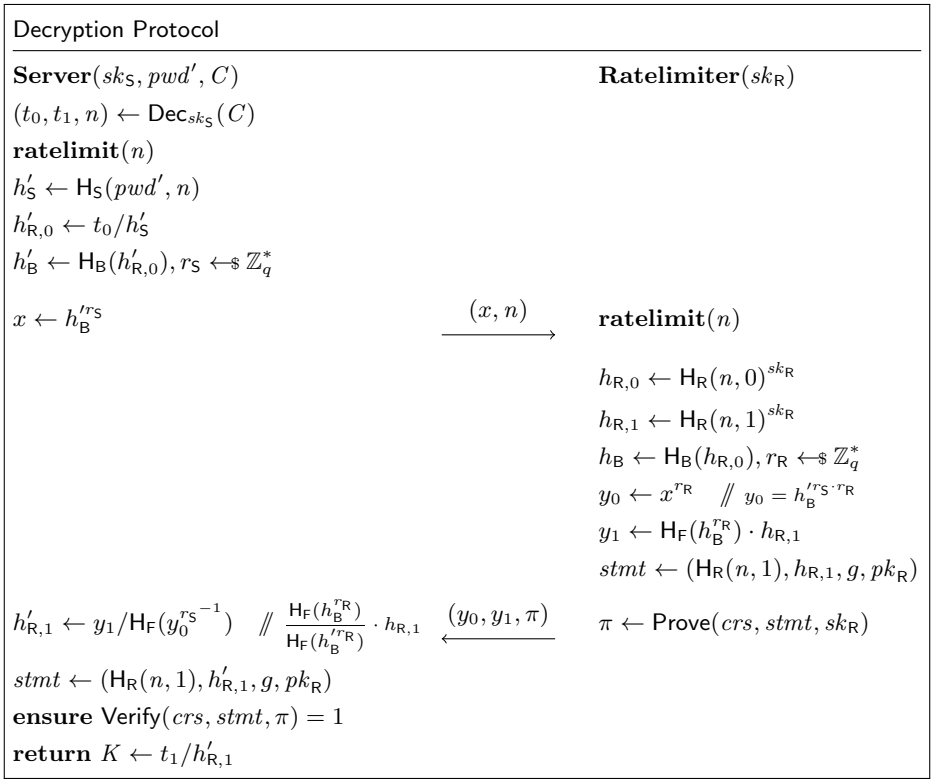

Fig. 7. The decryption protocol of HildeGUARD.

Key Rotation. Key rotation and updates follow a standard approach, where the ratelimiter samples a random α and updates its secret key by $sk_\mathsf{R}' \leftarrow sk_\mathsf{R} + \alpha$. The server updates the public key accordingly and samples a fresh symmetric encryption key. To update the stored records, the server exploits the key-homomorphism of the ratelimiter's PRF by multiplying $\mathsf{H_R}(n, 0)^\alpha$ (respectively $\mathsf{H_R}(n, 1)^\alpha$) to t_0 (respectively t_1), resulting in

PRF values consistent with the new ratelimiter key $sk_{\mathsf{R}}' = sk_{\mathsf{R}} + \alpha$. It re-encrypts the resulting values t_0', t_1', n with its new encryption key. We formalize the key-rotation protocol and update algorithm in Figure 8.

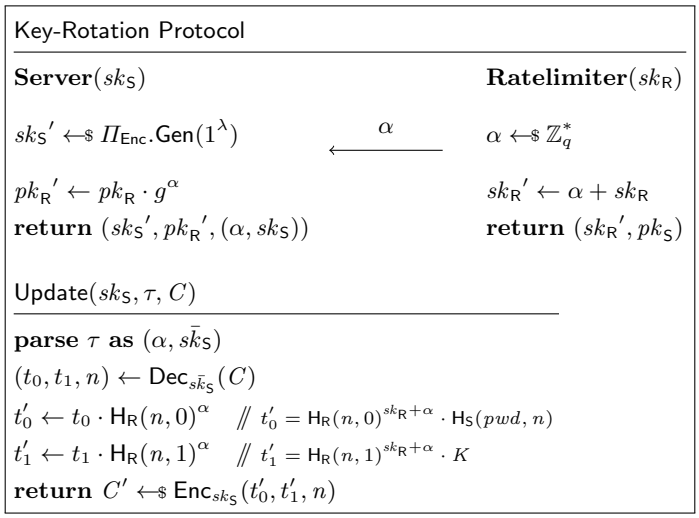

Fig. 8. The key rotation protocol of HildeGUARD.

4.2 Security of HildeGUARD

We give an intuition on why HildeGUARD is secure and defer the formal proofs to the full version of the paper.

Hiding. We break down the lifetime of a HildeGUARD instance into epochs, where the server's and ratelimiter's keys of an epoch are independent of the keys of other epochs. We distinguish two cases based on the corruption.

If the server is corrupt it learns nothing from t_0 and t_1 because of the pseudorandomness of $h_{\mathsf{R},0}$ and $h_{\mathsf{R},1}$. Furthermore, we show that the adversary can validate only a single password guess per decryption protocol run with the honest ratelimiter. If it could compute $h_{\mathsf{B}}^{r_{\mathsf{R}}}$ for multiple h_{B} to test multiple password guesses, it breaks the OM-gapDH assumption.

If the ratelimiter is corrupt, it learns nothing from the ciphertext because of the CCA-security of the used symmetric-key encryption scheme. From interactions with the honest server in the decryption protocol, it only learns if an entered password was correct from the output of the server. Beyond that, it learns nothing because the value x is uniformly random because of the blinding with r_{S}.

The **ratelimit** function restricts the number of decryption protocol runs with honest parties, such that the adversary can test at most Q_{limit} passwords.

Binding. Proving that HildeGUARD is binding is relatively straightforward. If the adversary interacts with an honest server in two protocol runs involving the same ciphertext but different keys, we can extract two NIZKs that have contradicting statements. Because the same ciphertext is used, we know that also the same $t_1 = h_{\mathsf{R},1} \cdot K$ and n is used by the server. The two different keys K and K' lead to two different ratelimiter PRF values $h_{\mathsf{R},1} \leftarrow t_1/K$ and $h_{\mathsf{R},1}' \leftarrow t_1/K'$. The NIZKs provided by the adversary prove that both $h_{\mathsf{R},1} \neq h_{\mathsf{R},1}'$ are well-formed which cannot be true. These NIZKs break the soundness of the NIZK scheme.

5 Impact

We support our work with implementations and benchmarks, demonstrating both the feasibility of our attack and the efficiency of our proposed protocol, HildeGUARD. We begin by demonstrating and benchmarking our attack against simplePHE and Virgil Security's implementation. Then, we implement HildeGUARD and benchmark its efficiency, comparing it to SimplePHE.

5.1 Breaking SimplePHE

To demonstrate the feasibility of our attack while adhering to ethical standards, we cloned the latest version of Virgil Security's PHE implementation [25] and conducted the attack on a local machine. The implementation, written in Golang [2], was executed using Go version 1.23.4 on a MacBook Pro equipped with an M3 Pro chip and 36GB of unified memory, running macOS Sonoma 14.6.1.

The attack involves a malicious ratelimiter reusing a nonce from an honest user's enrollment record during the login of a user with a known password. After performing a key rotation and compromising the server, the attacker uses the malicious password and login record to extract the ratelimiter's PRF. Using the extracted PRF, the attacker computes the salted hash of the honest user's password and initiates a brute-force attack to recover it. This attack has been implemented and validated on the described system configuration. We refer the reader to Section 2, a full description of which is given in the paragraph *Breaking Simple PHE*.

Benchmark of our attack. For testing the feasibility of our attack, we used a password list from SecLists containing 10 million passwords [21] and selected the secPW uniformly out of it. The brute-force attack successfully recovered the honest user's password in all test cases. We evaluated the attack using different levels of parallelism, varying the number of concurrent workers. The results are summarized in Figure 9.

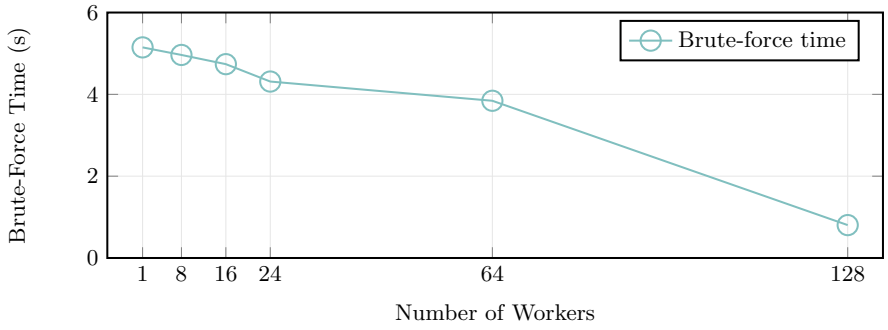

Fig. 9. Brute-force attack time with varying worker counts

Our results show that using 128 workers provides a significant speedup, completing the brute-force attack in approximately 804 ms. However, even with a single worker, the attack remains practical, recovering the password in just over 5 seconds. We publish the implementation of our attack for review purposes. For an ethical discussion on this matter, we refer to Section 7.

5.2 Benchmark of HildeGUARD

We benchmark the performance of HildeGUARD against the existing practical PHE scheme Simple PHE [19]. We do not compare the performance compared to TPHE [6], since this work uses three rounds which makes it highly unpractical. In addition, the spare latency induced by the number of rounds makes TPHE meaningles as comparisson.

Our implementation is written in Rust and compiled using `rustc` 1.84.0. It utilizes `curve25519-dalek` 4.1.3 for elliptic curve operations and AES-GCM for encryption.

Cryptographic Benchmark Results. To evaluate the efficiency of our implementation, we benchmark HildeGUARD against SimplePHE across the cryptographic operations. Our tests were conducted on a MacBook Pro with an M3 Pro chip and 36GB of unified memory, running macOS Sonoma 14.6.1. The benchmarking framework is powered by Criterion, and multi-threading is managed using `tokio` with `rt-multi-thread` enabled. Table 1 and Figure 10 present the execution times (in microseconds) for HildeGUARD and SimplePHE across different cryptographic steps.

Table 1. Cryptographic Benchmark Comparison: HildeGUARD vs. SimplePHE. Time is measured in µs.

Step	HildeGUARD	SimplePHE
R.encrypt	132.7	130.42
S.encrypt_init	5.74	-
S.encrypt_finish	166.83	248.21
S.decrypt_init	43.57	38.98
R.decrypt	168.02	127.58
S.decrypt_finish	142.78	213.24

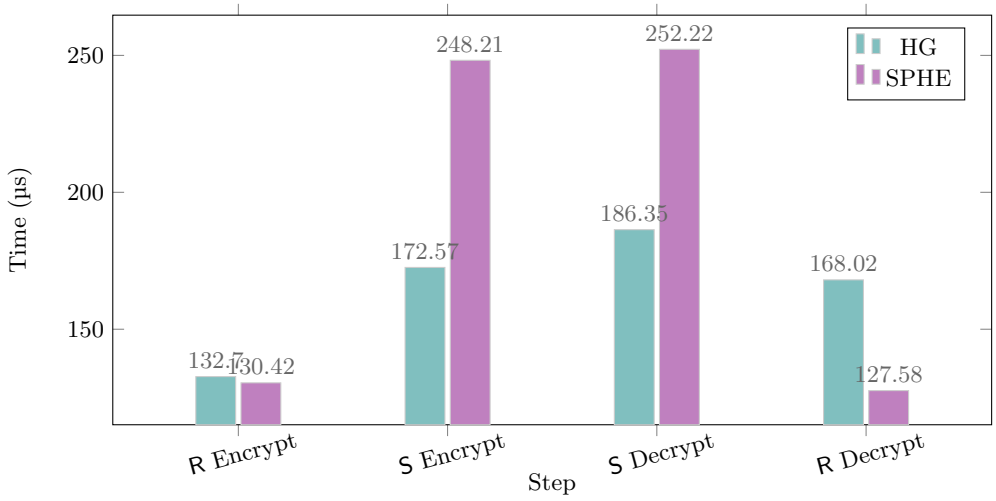

Fig. 10. Filtered Cryptographic Benchmark: HildeGUARD vs. SimplePHE

Encryption results show that R.encrypt performs similarly in both schemes (132.7 µs for HildeGUARD vs. 130.42 µs for SimplePHE). However,
S.encrypt_finish in HildeGUARD is faster (166.83 µs vs. 248.21 µs), reducing the total encryption time. For decryption, S.decrypt_init is identical for both schemes (43.57 µs vs. 38.98 µs), while S.decrypt_finish is faster (142.78 µs vs. 213.24 µs). R.decrypt is slower in HildeGUARD (168.02 µs vs. 127.58 µs), leading to a small overhead at the ratelimiter. Overall, HildeGUARD reduces the computational cost of server-side encryption and decryption but introduces some additional cost at the ratelimiter. However, these differences do not impact authentication latency as we explore in the online performance benchmark.

Online Performance. To evaluate encryption and decryption latencies in a real-world setting, we deployed our prototype on AWS EC2 `c5.2xlarge` instances, equipped with an Intel(R) Xeon(R) Platinum 8000 series processor featuring eight virtual CPUs and 16 GiB of memory. The servers run an Actix web server

(version 4) to handle requests. Performance evaluations were conducted under varying network latencies. We measure latencies using Apache Benchmark 2.3 and report averages over 1000 interactions in Table 2. The validation time for HildeGUARD is 50.3 ms, while SimplePHE achieves a slightly lower value of 49.668 ms. The observed differences are beyond the variance introduced by network latency, which aligns with our analysis of the cryptographic core.

To isolate the cryptographic overhead, we also benchmarked a "dummy implementation" where the web server simply responds to requests without performing any cryptographic operations. This dummy benchmark includes one round of communication between the server and the ratelimiter. Interestingly, some validation times for SimplePHE and HildeGUARD are slightly lower than the dummy's 50.429 ms response time. This confirms that the cryptographic operations introduce negligible overhead compared to the overall request handling and network communication. Both schemes share the same round complexity, meaning that differences in execution time primarily stem from cryptographic operations rather than additional protocol steps.

Table 2. Online Performance Benchmark: HildeGUARD vs. SimplePHE. Time is measured in milliseconds.

Scheme	**Enroll (ms)**	**Validate (ms)**
HildeGUARD	49.789	50.3
SimplePHE	50.758	49.668
Dummy	50.429	

5.3 Deployment

We have demonstrated an attack against SimplePHE and introduced HildeGUARD as a more secure alternative. For systems where SimplePHE is already deployed, we provide a seamless migration strategy to transition ciphertexts from SimplePHE to HildeGUARD without requiring password re-enrollment or user interaction. Additionally, we describe how HildeGUARD can be integrated into systems that currently use salted hashes for password security. This ensures compatibility with existing authentication frameworks while enhancing security. Finally, we address a critical limitation of PHE schemes: their reliance on a specific ratelimiter, which undermines robustness. If the ratelimiter's service becomes unavailable–whether due to termination, service disruptions, or infrastructure changes–all ciphertexts stored on the server become unusable. To enhance system robustness and resilience, we propose an opt-out mechanism that allows a seamless transition from HildeGUARD back to traditional salted and hashed passwords. This approach ensures that deployments can switch authentication methods or replace the ratelimiter without compromising the usability of existing records.

Migration from Salted Hash. We demonstrate how to migrate an existing authentication system from hashed passwords to HildeGUARD while preserving compatibility with legacy storage. Inspired by Facebook's *Password Onion* [22], we use the stored salted and hashed password as the input password for our PHE scheme. Besides that, the HildeGUARD encryption protocol stays as is.

Our experiments validate the practicality of adopting HildeGUARD without user involvement. To measure the computational overhead of this migration, we benchmarked the enrollment and verification process of one million hashed passwords. The results, summarized in Table 3, show that encryption–mapping hashed passwords to HildeGUARD-protected ciphertexts–completed in 48.18 seconds, averaging 0.0482 ms per user.

Importantly, this initial migration can be performed locally by the server without any interaction with the future ratelimiter. During the process, the server generates the ratelimiter's secret key and performs the rate limiter's part of the encryption process locally. Once the migration is complete, the server securely transfers the secret key to the ratelimiter and erases it from local storage. This temporary exposure of the ratelimiter's secret key to the server does not introduce any additional security risks. At this stage, a malicious server

could already copy the unprotected database and perform a dictionary attack, as the data has not yet been protected by the ratelimiter.

Table 3. Migrating one million users from salted hash to HildeGUARD

Operation	Total (s)	Average per User (ms)
Encryption	48.181	0.0482
Verification	53.172	0.0532

Opt-Out. The primary advantage of PHE is its enhanced security for users. However, in practice, service providers require flexibility to adapt their authentication mechanisms over time. Locking in on a single cryptographic scheme can hinder long-term maintainability. To address this, we introduce opt-out for PHE, a method for transitioning from PHE-protected records back to traditional salted hashes. This follows a concept that Jia *et al.* introduced for PH [18].

Opt-out enables providers to migrate to a different authentication scheme or replace the ratelimiter without requiring user interaction. Opt-out takes place when the ratelimiter's service should be discarded. Therefore, we assume a service provider knows both the server's key and the ratelimiter's key. Even though this might seem like a severe security risk that renders the whole scheme useless, we argue why it is reasonable to do so. First of all, the sole purpose of the opt-out functionality is to release the ratelimiter from its duty, enabling the server to verify passwords on its own. Therefore, we no longer need the security that the secrecy of the ratelimiter's secret key provides. Furthermore, the opt-out protocol is executed only after the server and the ratelimiter agree to proceed through an out-of-band process.

$$\begin{aligned} \mathsf{H}_{\mathsf{S}}(pwd, n) &\leftarrow t_0 / h_{\mathsf{R},0} \\ K &\leftarrow t_1 / h_{\mathsf{R},1}. \end{aligned}$$

The resulting salted hash of the password is used to compute two new tokens. One is solely for verifying an entered password, while the other one stores the key. To achieve domain separation, h_{S} is hashed with a single bit as a domain separator. The first token consists only of this hash $t'_0 \leftarrow \mathsf{H}(h_{\mathsf{S}}, 0)$, while the second token is the key blinded by the second hash $t'_1 \leftarrow \mathsf{H}(n_{\mathsf{S}}, 1) \cdot K$. The formal definition of the opt-out algorithm can be seen in Figure 11.

It is essential that the opt-out procedure does not expose passwords directly. Furthermore, the resulting enrollment record must offer at least the same level of protection against dictionary attacks as a standard salted password hash.

Given a stored HildeGUARD record, the opt-out process proceeds as follows. The service provider first reconstructs the ratelimiter's contribution to the ciphertext by computing the PRF outputs:

$$\begin{aligned} t'_0 &\leftarrow t_0 / h_{\mathsf{R},0} = \mathsf{H}_{\mathsf{S}}(pwd, n) \\ t'_1 &\leftarrow t_1 / h_{\mathsf{R},1} = K. \end{aligned}$$

These values are used to obtain the hashed password and the encapsulated key from the ciphertext:

$$\begin{aligned} \mathsf{H}_{\mathsf{S}}(pwd, n) &\leftarrow t_0 / h_{\mathsf{R},0} \\ K &\leftarrow t_1 / h_{\mathsf{R},1}. \end{aligned}$$

The resulting salted password hash $\mathsf{H}_{\mathsf{S}}(pwd, n)$ is used to derive two separate tokens: one for password verification and another for storing the cryptographic key. Domain separation is achieved by hashing h_{S} with a distinguishing bit. Specifically, the first token is $t'_0 \leftarrow \mathsf{H}(h_{\mathsf{S}}, 0)$, and the second token is $t'_1 \leftarrow \mathsf{H}(h_{\mathsf{S}}, 1) \cdot K$, which conceals the key. The formal description of the opt-out algorithm is provided in Figure 11. We benchmarked this opt-out procedure for HildeGUARD and the opt-out completes in $63.92\mu s$ per ciphertext on average.

$\mathsf{Opt\text{-}Out}(sk_{\mathsf{S}}, sk_{\mathsf{R}}, C)$

$(t_0, t_1, n) \leftarrow \mathsf{Dec}_{sk_{\mathsf{S}}}(C)$

$h_{\mathsf{S}} \leftarrow t_0/\mathsf{H}_{\mathsf{R}}(n,0)^{sk_{\mathsf{R}}}, K \leftarrow t_1/\mathsf{H}_{\mathsf{R}}(n,1)^{sk_{\mathsf{R}}}$

$t'_0 \leftarrow \mathsf{H}(h_{\mathsf{S}}, 0), t'_1 \leftarrow \mathsf{H}(h_{\mathsf{S}}, 1) \cdot K$

$C' \leftarrow \mathsf{Enc}_{sk_{\mathsf{S}}}(t'_0, t'_1, n)$

return C'

Fig. 11. The opt-out algorithm of HildeGUARD.

Migration from SimplePHE. The opt-out mechanism not only allows migration from HildeGUARD to traditional salted hashes but also enables a seamless transition from existing PHE schemes, such as SimplePHE, to HildeGUARD. This is crucial for providers seeking to upgrade their security model without requiring user action.

To migrate from SimplePHE, the provider first applies the opt-out procedure to extract the salted hash of the password. This process follows the same steps as transitioning from HildeGUARD to salted hashes: computing the ratelimiter's PRF values, stripping the ratelimiter's contribution, and recovering the hashed password. Once obtained, this salted hash is used as the password input for HildeGUARD, ensuring compatibility with existing credentials. Re-enrollment into HildeGUARD follows the standard encryption workflow, using the extracted salted hash in place of a plaintext password.

6 Conclusion

In this work, we revisited the security of Password-Hardened Encryption (PHE) and identified a critical vulnerability in its formal model. We demonstrated a practical attack that allows offline brute-force recovery of passwords, highlighting the necessity of a more comprehensive security framework. Our findings emphasize that security definitions must align with real-world deployment scenarios, particularly in the presence of key rotations.

To address these shortcomings, we introduced HildeGUARD, a novel PHE scheme that remains secure even under adaptive corruptions. We formalized a refined security model, ensuring robustness against the identified attack while maintaining efficiency. Our implementation and benchmarks demonstrate that HildeGUARD not only mitigates the vulnerabilities of SimplePHE but also achieves comparable performance.

Beyond theoretical contributions, we provide a migration strategy for transitioning existing PHE deployments to HildeGUARD and offer integration pathways for systems relying on traditional salted hashes. Our work underscores the importance of continuous scrutiny in cryptographic protocol design, ensuring that practical deployments uphold security guarantees against evolving threats.

Open Science

To facilitate transparency, reproducibility, and further research, we publicly release our attack and prototype implementations. The full codebase is available at:

`https://github.com/pGerhart/Attack-SimplePHE`
`https://github.com/pGerhart/HildeGUARD`
`https://github.com/pGerhart/SimplePHE`

7 Ethics Considerations

This research adheres to established ethical principles in cybersecurity research. We do not exploit vulnerabilities in deployed systems or compromise real user data. Instead, our work identifies a design flaw in Virgil Security's PHE protocol and demonstrates its feasibility in a controlled environment.

Responsible Disclosure. Following ethical best practices, we engaged in responsible disclosure with Virgil Security prior to publication. We provided a detailed report outlining the attack, its security implications, and possible mitigations. This ensured that the vendor had an opportunity to address the issue before public disclosure, aligning with the principles of coordinated vulnerability disclosure.

Compliance with the Menlo Report. Our study aligns with the ethical principles set forth in the Menlo Report, which extends traditional research ethics to the field of cybersecurity:

- **Respect for Persons**: Our research does not target real users or deployed systems. All experiments were conducted on a locally controlled instance of the protocol.
- **Beneficence**: The goal of our work is to strengthen security, not to facilitate attacks. By responsibly disclosing our findings, we help prevent potential misuse.
- **Justice**: The identified vulnerability has broad implications for users relying on PHE-based authentication. Our research ensures that security risks are addressed equitably.
- **Respect for Law and Public Interest**: We operate within legal and ethical boundaries, following responsible disclosure procedures and ensuring that no unauthorized access occurs.

Post-Publication Plans. We publicly release our implementation for verification and transparency. However, we do not provide a direct exploit, preventing misuse while enabling security researchers and vendors to evaluate and improve their implementations. By adhering to these ethical guidelines, we ensure that our work contributes to the security of cryptographic protocols without introducing harm or facilitating real-world attacks.

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 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.

References

1. Shashank Agrawal, Peihan Miao, Payman Mohassel, and Pratyay Mukherjee. PASTA: PASsword-based threshold authentication. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 2042–2059. ACM Press, October 2018.
2. The Go Authors. The go programming language. `https://go.dev`, 2025.
3. Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011*, pages 433–444. ACM Press, October 2011.
4. Elaine Barker. Nist special publication 800-57 part 1, revision 5. `https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf`. [Online; accessed 21-January-2025].
5. Carsten Baum, Tore Kasper Frederiksen, Julia Hesse, Anja Lehmann, and Avishay Yanai. PESTO: Proactively secure distributed single sign-on, or how to trust a hacked server. In *2020 IEEE European Symposium on Security and Privacy*, pages 587–606. IEEE Computer Society Press, September 2020.
6. Julian Brost, Christoph Egger, Russell W. F. Lai, Fritz Schmid, Dominique Schröder, and Markus Zoppelt. Threshold password-hardened encryption services. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 409–424. ACM Press, November 2020.

7. Jan Camenisch, Anja Lehmann, Anna Lysyanskaya, and Gregory Neven. Memento: How to reconstruct your secrets from a single password in a hostile environment. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 256–275. Springer, Berlin, Heidelberg, August 2014.
8. Jan Camenisch, Anja Lehmann, and Gregory Neven. Optimal distributed password verification. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 182–194. ACM Press, October 2015.
9. Poulami Das, Julia Hesse, and Anja Lehmann. DPaSE: Distributed password-authenticated symmetric-key encryption, or how to get many keys from one password. In Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako, editors, *ASIACCS 22*, pages 682–696. ACM Press, May / June 2022.
10. Constantinos Diomedous and Elias Athanasopoulos. Practical password hardening based on tls. In Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 441–460, Cham, 2019. Springer International Publishing.
11. Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The pythia PRF service. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 547–562. USENIX Association, August 2015.
12. Fortinet. 2024 state of operational technology and cybersecurity report. Technical report, Fortinet, June 2024. Accessed: 2025-05-07.
13. Google. Google cloud kms documentation: Key rotation. `https://cloud.google.com/kms/docs/key-rotation`. [Online; accessed 21-January-2025].
14. Identity Theft Resource Center. 2023 annual data breach report reveals record number of compromises, 72 percent increase over previous high. 2023. Accessed: 2025-01-21.
15. Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 233–253. Springer, Berlin, Heidelberg, December 2014.
16. Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17International Conference on Applied Cryptography and Network Security*, volume 10355 of *LNCS*, pages 39–58. Springer, Cham, July 2017.
17. Stanislaw Jarecki, Hugo Krawczyk, Maliheh Shirvanian, and Nitesh Saxena. Two-factor authentication with end-to-end password security. In Michel Abdalla and Ricardo Dahab, editors, *PKC 2018, Part II*, volume 10770 of *LNCS*, pages 431–461. Springer, Cham, March 2018.
18. C. Jia, S. Wu, and D. Wang. Reliable password hardening service with opt-out. In *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*, pages 250–261, Los Alamitos, CA, USA, sep 2022. IEEE Computer Society.
19. Russell W. F. Lai, Christoph Egger, Manuel Reinert, Sherman S. M. Chow, Matteo Maffei, and Dominique Schröder. Simple password-hardened encryption services. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 1405–1421. USENIX Association, August 2018.
20. Russell W. F. Lai, Christoph Egger, Dominique Schröder, and Sherman S. M. Chow. Phoenix: Rebirth of a cryptographic password-hardening service. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017*, pages 899–916. USENIX Association, August 2017.
21. Daniel Miessler and contributors. Seclists: Common-credentials 10 million password list (top 1,000,000). `https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10-million-password-list-top-1000000.txt`, 2025. Accessed: 2025-01-22.
22. Alec Muffett. Facebook password hashing & authentication. `https://www.youtube.com/watch?v=7dPRFoKteIU`. [Online; accessed 21-February-2023].
23. Colin Percival. Stronger key derivation via sequential memory-hard functions. In *BSDCan 2009*, 2009.
24. Jonas Schneider, Nils Fleischhacker, Dominique Schröder, and Michael Backes. Efficient cryptographic password hardening services from partially oblivious commitments. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1192–1203. ACM Press, October 2016.
25. Virgil Security. Virgil phe go. `https://github.com/VirgilSecurity/virgil-phe-go`, 2025. Commit 7acc0de, available at `https://github.com/VirgilSecurity/virgil-phe-go/tree/7acc0de`.
26. Security Standards Council. Pci dss v4.0.1 (section 15, 3.7.4 & 3.7.5). `https://docs-prv.pcisecuritystandards.org/PCI%20DSS/Standard/PCI-DSS-v4_0_1.pdf`. [Online; accessed 21-January-2025].