Secure and Updatable Single Password Authentication

Devriş İşler[1][0000−0003−4895−8827], Hamidreza Saadi Dadmarzi[2][0009−0000−7883−816X], and Alptekin Küpçü[2][0000−0003−2099−2206]

[1] IMDEA Networks Institute, Madrid, Spain
devris.isler@gmail.com
[2] Koç University, İstanbul, Türkiye
{hdadmarzi23,akupcu}@ku.edu.tr

Abstract. Passwords remain the dominant authentication method despite weaknesses such as offline dictionary attacks and password reuse. Single Password Authentication (SPA) mitigates these risks by protecting high entropy secrets under one memorable password and distributing them across *untrusted* storage providers. However, existing SPA schemes cannot prevent preemption and overwrite attacks by storage providers, and they lack secure, efficient support for secret and password updates.
We present UpSPA, an *efficient*, *secure*, and *updatable* threshold SPA that addresses both limitations without requiring changes on the login server. UpSPA prevents preemption through a storage provider specific high entropy identifier secret, supports *secret updates* via implicit authentication, and enables *password updates* via explicit authentication using a password protected signing key. We prove security in the ideal real paradigm, including resistance to offline dictionary attacks under standard static threshold corruption assumptions. Our evaluation shows low overhead and competitive performance compared to a prior SPA scheme that does not support updates.

Keywords: Password authentication · threshold cryptography · offline dictionary attack.

1 Introduction

Problem: Passwords remain the dominant method of user authentication and are employed for more advanced techniques such as secret sharing [6] and key exchange [10]. More secure mechanisms such as hardware tokens and two-factor authentication exist, yet passwords persist because they are simple and familiar. This convenience, however, comes at a security cost: password databases are frequent attacker targets, and even large organizations struggle to safeguard them [1]. The risk is further amplified by password reuse across login servers [2]. Other alternatives such as cloud password managers do not always provide the strong security guarantees [30] implied by zero knowledge encryption claims, especially against a malicious server, due to practical attacks across the three systems, showing weaknesses in key escrow, item level vault encryption, sharing, and backward compatibility, many of which enable password recovery or full compromise.

Single Password Authentication (SPA) schemes aim to provide formal security guarantees against attacks that undermine conventional password authentication [3]. An SPA enables a user to rely on a single human-memorable password across accounts, without requiring the user to store any high-entropy secret locally and without requiring login servers to store password-related verification information. Instead, SPA derives a high-entropy secret that is *independent* of the password, registers with each login server some verification information that is derived from a *login-server-specific* high-entropy secret, and protects this secret at one or more untrusted third parties, called *storage providers* (e.g., personal mobile devices or cloud storage), encrypted under the user's password. Concretely, when a user registers with a login server, SPA generates a high entropy secret, secret-shares it, and stores the shares at storage providers under protection of the single password. The login server receives only verification information derived from its own login-server-specific high entropy secret and its domain; for example a hash output that binds the secret to the domain (domain information is included as a protection against phishing attacks trying to

steal passwords). When the user later wants to authenticate, they reconstruct the secret using the password without revealing the password to any storage provider, and then complete authentication to the login server using the reconstructed secret and the login server information. In a threshold SPA, the secret is shared across multiple storage providers, and the user can reconstruct it by contacting any threshold many providers. Among prior SPA approaches [3,33,21,18,9], only TSPA [18] and DiSPP [33] adopt a threshold approach to storage.

Alternatives: While SPAs resemble Single Sign-On (SSO) [4,7,29,24], a key distinction is that state-of-the-art SPAs can be deployed on top of traditional password authentication without requiring changes to login servers, and they never reveal the user's password to any party (i.e. there are no trusted parties). On the other hand, SSO introduces an identity provider (IdP): the user authenticates to the IdP, often using a password, to obtain a token, such as a signature, that relying parties must accept for login (and therefore must trust the IdP). In contrast, SPA has no federation token and no authentication authority. Storage providers only assist the client in recovering a high-entropy per-service secret under the password, while authentication remains an end-to-end interaction between the user and the login server. Indeed, the login server is unaware that storage providers exist. Threshold SSO schemes such as [4] and its successors distribute the IdP role across multiple servers, yet the end result remains a token based federation flow in which relying parties (e.g., login servers) must verify and accept an IdP-issued token. Indeed, implementations explicitly target standard SSO token formats, e.g., OAuth compatible tokens.

Limitations of existing SPAs: In a threshold SPA, each storage provider receives an account identifier and a share of the per-login-server high-entropy secret. The identifier is typically derived from the user identity and the login server domain, for example $\mathsf{Hash}(\mathsf{username} \parallel \mathsf{domain})$ which binds the verification value to the intended login server and reduces cross-site replay and misbinding. Storage providers cannot verify whether a requester actually knows the password (since this would result in a chicken-and-egg problem). As a result, storage providers maintain encrypted share records indexed by identifiers derived from the user identity and login server domain. This introduces weaknesses in how account state is created and updated. First, an adversary can *preempt* future registrations by injecting fraudulent identifier and random share entries into a storage provider for login servers where the user does not yet have an account, thereby preventing the user from later storing the correct shares. This is analogous to registering an account on behalf of someone else using their username or email before they do. In ordinary login systems, such abuse is typically prevented by ownership checks such as email verification. In SPA settings, where storage providers are not supposed to learn the password and may support anonymous writes for privacy reasons. Indeed, starting from initial SPA designs [3], privacy against storage providers were considered as a goal of SPA solutions. Therefore, a mechanism is needed to ensure that only the legitimate user can initialize account state for a given identifier. Second, if the storage provider's update mechanism is insufficiently protected, an adversary may *override* existing account entries with forged shares by impersonating the user, compromising authentication integrity and potentially enabling account takeover. In prior SPA constructions, such overwrite style attacks can also facilitate offline dictionary attacks in case of secret or password updates. Preventing such attacks requires update procedures for both password changes and login-server-specific secret changes that cryptographically bind updates to the original account secrets and knowledge of the password, ensuring that only the original account owner can authorize modifications. However, secret and password update have received little to no attention in existing SPA solutions.

In this work, we close these gaps by designing UpSPA, a secure and *updatable* threshold SPA that prevents preemption and unauthorized updates at storage providers while preserving compatibility with unmodified login servers.

1.1 Design Challenges and Solutions

We now discuss the threats and design challenges in building a secure and updatable single password authentication (SPA) system consisting of users, login servers, and storage providers. Our goal is to design a

secure updatable SPA protocol, called UpSPA, that remains secure under compromise of up to a threshold number of storage providers, supports both secret and password updates, and incurs minimal overhead, while addressing shortcomings of prior SPA solutions.

To make the issues concrete, consider a user who creates an account at YouTube under an SPA design such as TSPA [18]. The user registers at the login server using verification information derived from a per-service high-entropy secret $\mathcal{R}^{ls}$, for example $\mathsf{Hash}(\mathcal{R}^{ls} \parallel \mathsf{youtube.com})$ [18,33], and stores the corresponding password protected shares at storage providers (e.g., via an oblivious pseuduorandom function-derived key per storage provider). Storage providers typically index records using predictable identifiers derived from the user identity and domain, such as $\mathsf{Hash}(\mathsf{username} \parallel \mathsf{youtube.com})$ [18]. This enables *preemption*: an adversary who knows the username can inject a fake record under the predictable identifier, causing the storage provider to later reject the legitimate user's record, since providers lack out of band confirmation mechanisms.

Challenge 1: Preventing unauthorized account creation. To prevent preemption, UpSPA replaces username-derived identifiers with high entropy, storage-provider-specific identifiers derived from an *identifier secret*, denoted by $\mathcal{R}^{sp}$. The user samples $\mathcal{R}^{sp}$ once during setup and stores it, protected under the user's password across the storage providers. Whenever the user needs to communicate with storage providers (e.g., during authentication or password/secret updates), the user reconstructs $\mathcal{R}^{sp}$ by interacting with any threshold many storage providers and then derives a fresh identifier for each login-server and storage-provider pair. Because these identifiers are high-entropy, an attacker cannot guess them or reuse them on another storage provider or for another login server. Therefore, attackers cannot create or preempt the corresponding account records at storage providers. Remember that login-server side measures such as email verification should still be in place but this does not affect the storage-provider side discussion above.

Challenge 2: Secret update. A secret update changes the per-login-server verification information, and therefore corresponds to updating the high entropy secret associated with a specific account. While a login server can authorize such an update after a successful login, storage providers in SPA do not store password verifiers and thus cannot directly check whether an update request comes from a user who knows the correct password pwd or not. This creates an authentication paradox: authorizing a secret update appears to require an additional authentication mechanism at the storage provider. UpSPA resolves this using *implicit authentication*. Since only a user who knows pwd can reconstruct the random identifier $\mathcal{R}^{sp}$ and derive the correct provider-specific identifier, the ability to address the correct record at a storage provider serves as an implicit proof of password knowledge. This allows a provider to accept share updates for that record without storing any additional password related information. We further employ counters to prevent replay attacks.

Challenge 3: Password update. A password update must modify all password-protected records held at storage providers, and storage providers must ensure that only a legitimate user who knows the current password can authorize the change. While implicit authentication suffices to locate the correct records, it does not by itself bind an update to a unique user across providers or prevent malicious storage providers from attempting to substitute the user records. To obtain explicit update authorization, UpSPA uses a signature-based mechanism. The user generates a signing key pair during setup. The signing key is stored at the storage providers protected under the current password, and the verification key is made available to the providers. To update the password, the user reconstructs the signing key using the current password, signs the new update values together with freshness information (a provider challenge and a timestamp or counter), and sends the signed request to the providers. Each provider verifies the signature and then replaces its stored record. This way, the storage providers can recognize that the entity trying to update the password associated with a particular username is indeed the same entity that setup the storage initially, since only with the correct password knowledge the signing key can be decrypted back and a verifying signature can be created.

Challenge 4: Efficiency. In TSPA [18] and similar threshold SPA designs, authentication requires running an oblivious pseudo-random function (OPRF) with the contacted storage providers and decrypting the corresponding protected state. However, naively extending such designs to support updates is costly. When a user has n_{ls} login-server accounts and relies on n_{sp} storage providers, a password update requires re-encrypting

the password-protected records for all n_{ls} accounts across the n_{sp} storage providers, resulting in $\mathcal{O}(n_{sp} \cdot n_{ls})$ work. By contrast, our construction removes this dependence on the number of accounts. Both secret update and password update modify only a constant number of logical records, independent of n_{ls}. Therefore the updates in UpSPA incur only $\mathcal{O}(n_{sp})$ update cost ($\mathcal{O}(1)$ update cost per storage provider). UpSPA achieves this optimized update cost by separating per-account records from password-protected credential material. In this two-stage system that we introduced, the user maintains a compact password-protected record containing the identifier, a signing key, and a symmetric encryption key, while per-login-server secrets are stored securely under the symmetric key. As a result, a password update refreshes only this compact record, independent of the number of user accounts (n_{ls}), and thus has constant cost in the number of accounts (login servers). Secret updates affect only the records for the selected login server and do not require touching other accounts.

Challenge 5: Integrity and Record Consistency. UpSPA resists offline dictionary attacks as long as the adversary corrupts fewer than threshold-many storage providers and protects the user against malicious storage providers. To this end, UpSPA uses authenticated encryption to ensure the confidentiality and integrity of the records stored by storage providers, so that a malicious storage provider cannot tamper with encrypted records or shares without being detected. However, integrity alone does not guarantee *freshness*. A malicious provider may replay an older but valid ciphertext from user's previous records (e.g., records before an update). Due to our efficiency-oriented design, such ciphertexts may still decrypt, in which case the user may end up using outdated secrets; for example failing to reconstruct the signing key needed to authorize updates or retrieving obsolete account records for authentication. UpSPA therefore binds each stored record to an explicit version counter and requires the user to accept only the most recent records supported by a threshold number of providers. This enforces freshness and prevents fewer than threshold-many malicious providers from de-synchronizing or disrupting the records needed for user's authentication.

Challenge 6: Security and Privacy. If there are n_{sp} storage providers in total, assuming at least $n_{sp}/2$ storage providers are honest, we guarantee both security and liveness in the sense that the user would always employ up to date account information. This is done via password-protected storage of identifiers and secrets, the use of OPRF, employing the domain and the counter/timestamps. In our ideal-real paradigm security proof and simulation, we show that even when the adversary controls all login servers (except one target login server), all users (except one target user), and less than threshold many storage providers, our solution provides security for the honest user and honest login server. Other users or login servers cannot impersonate this honest user against the honest login server, nor can they obtain the long-term single password of the honest user. The best attack that they can perform is an online dictionary attack, which would be rate- and #attempts-limited at the honest login server and honest storage providers. Phishing attacks are also useless in the sense that our security definition and proof already covers the case that the honest user registers with adversarial login servers and stores values at adversarial storage providers. In our ideal world, the long term password or login-server-specific secrets are never leaked to the adversarial entities. Moreover, the identifiers are randomly picked so that they preserve privacy of the user completely. Overall, satisfying such a secure and privacy-preserving ideal world, our solution guarantees security against above-mentioned adversaries, disabling feasibility of offline dictionary attacks, impersonation, phishing, or honeypots.

Our contributions are summarized as follows:

- We design UpSPA, an *updatable* threshold SPA protocol that protects users' passwords against malicious parties and prevents offline dictionary attacks under a threshold compromise assumption while remaining compatible with unmodified login servers.
- We formally prove security of UpSPA in the ideal-real simulation paradigm, showing that any PPT adversary corrupting less than threshold storage providers and the login server has only negligible advantage in mounting offline dictionary attacks.
- We provide a numerical performance evaluation demonstrating deployability on commodity hardware, including a detailed efficiency study across multiple threshold settings and a comparison with prior work.

2 Related Work

Password authentication has evolved from traditional two party logins toward distributed architectures designed to mitigate the impact of server breaches and password reuse. We revisit three lines of work most relevant to UpSPA: (i) single password authentication (SPA); (ii) threshold token-based Single Sign On (SSO); and (iii) password hardened services (PHS) and updatable key management systems (KMS). Table 1 highlights how UpSPA differs from prior work.

Single Password Authentication (SPA): SPA solutions [16,3] enable users to rely on a single human-memorable password across accounts without storing any high-entropy secret locally. They achieve this by involving one or more storage providers, such as a cloud service or a personal mobile device, which store password-protected state so that only a user who knows the correct password can reconstruct the secret needed for authentication. Threshold SPA (TSPA) [18] extends this line by distributing the stored secret across n storage providers and allowing authentication using any t of them. This yields threshold availability without requiring changes on the login-server side, and it formalizes security via an ideal-real simulation proof. DiSPP [33] places these constructions in a broader distributed single-password framework and captures richer provider configurations and participation policies. Despite these advances, existing SPA protocols do not natively support password updates or login-server-specific secret updates. UpSPA follows the direction of TSPA [18] and extends the SPA family by integrating subprotocols for password update and secret update. At a high level, DE-PAKE [21] strengthens password authentication with the help of an auxiliary device, aiming to improve resistance to online and offline attacks while remaining deployable with conventional password-based servers. Earlier browser-based password managers [9] also explored the single-password direction on the client side, deriving site-specific passwords from a master password without requiring changes to website authentication interfaces, but they did not provide threshold availability or native support for password or secret updates in a multi-server setting.

Threshold Token-based Single Sign-on: Threshold Single Sign On (SSO) schemes resemble SPA at a high level because they also aim to let users access many services using a single password, often with threshold security. The mechanism, however, is different. In SSO, authentication is mediated by identity providers (IdPs): after the user proves knowledge of a password to the IdP side, the user obtains a token, often a signature, which is then presented to relying parties for verification. Threshold SSO distributes this token issuance by sharing the IdP master key (login-server-specific secret in SPAs) signing capability among n IdPs in a threshold manner, so that any t IdPs can jointly issue a valid token. A key design challenge in this line of work is preventing offline dictionary attacks when some IdPs are compromised. A naive threshold token-authentication design in which IdPs verify password-related material still leaves users' passwords vulnerable to offline dictionary attacks when up to $t-1$ IdPs are compromised [4]. Recent proposals address this by basing the IdP side on functions designed to resist offline attacks, such as threshold oblivious pseudorandom functions (TOPRFs), and by carefully structuring token generation and verification [4,7,31,28]. PASTA [4] formalizes password-based threshold authentication for SSO. A set of n IdPs jointly evaluates a threshold OPRF on the user's blinded password and then issue a login token. Security guarantees that any t IdPs suffice for authentication, while any coalition of at most $t-1$ IdPs cannot forge tokens or mount offline dictionary attacks. Building on PASTA, PAS-TA-U [29] adds a native password update subprotocol in which the client performs threshold OPRF evaluations under the old and new passwords and submits a threshold token generation authenticated update request that each IdP verifies before updating its local user record. Complementary efforts strengthen the same architecture in different directions. PESTO [7] adds proactive security through periodic share refresh and proactively secure threshold signatures, TSAPP [32] targets privacy with anonymous and unlinkable threshold token generation, DE-PbTA [25] hardens the password using an auxiliary device before the threshold SSO flow, and AugSSO [24] proposes an efficient password update mechanism within threshold SSO.

Password-Hardening and Key Management Services: Existing SPA works focuses on recovering an authentication secret from untrusted storage providers. In contrast, password hardening services (PHS) [5] and

updatable key management systems (KMS) [19] focus on deriving encryption keys from passwords and updating per-object ciphertext headers. PHS uses an OPRF or TOPRF, often in threshold form, to harden low entropy passwords for key derivation. They support *updatable encryption* by issuing update tokens that refresh a small per object ciphertext header, a process often called *rewrapping*, while leaving the bulk ciphertext unchanged. Many designs also include proactive share refresh. Pythia [17] initiates this line with a verifiable partially oblivious PRF and compact rotation tokens. PHE [27] formalizes a two-party hardening service with rate limiting and key rotation, and Threshold PHE [11] distributes it across multiple servers. UOKMS [20] adapts these ideas to the cloud-KMS setting by attaching a per-object header that can be refreshed via update tokens without re-encrypting the underlying data. TPH-UOKM [23] further combines threshold password hardening with an updatable oblivious KMS and supports password update, but it still remains in the cloud-storage rewrapping setting rather than the SPA setting. In contrast, UpSPA targets updatable SPA authentication and avoids per-object re-wrapping. The heavy tasks of both password and secret updates are performed on the client, while storage providers only verify and atomically replace stored shares. Furthermore, UpSPA provides a static malicious security proof, unlike TPH UOKM [23].

Table 1: Comparison of existing works. −: Not applicable.

Solution / Property	Our Work	Single-Password Authentication			SSO		PHS and KMS	
	UpSPA	SPA [3]	DE-PAKE [21]	[18],[33]	[4],[7],[32],[25]	[29],[24]	TPHE [11]	TPH-UOKM [23]
Malicious security	✔	✗	✔	✗	✗	✗	✗	✗
Threshold security (t-of-n)	✔	✗	✗	✔	✔	✔	✔	✔
Secret update supported	✔	✗	✗	✗	–	–	✔	✔
Password update supported	✔	✗	✔	✗	✗	✔	✗	✔
Prevents false account creation at storage provider	✔	✗	✗	✗	–	–	–	–

3 Preliminaries

Let $\lambda \in \mathbb{N}$ be the security parameter. A probabilistic polynomial time (PPT) algorithm A is a probabilistic algorithm taking 1^λ as an input and has running time bounded by a polynomial in λ. A function $\mathsf{neg}(\lambda)$ is negligible if for every positive polynomial $\mathsf{poly}(\lambda)$ there exists a constant $\lambda' \in N$ such that $\forall \lambda > \lambda'$ $\mathsf{neg}(\lambda) < 1/\mathsf{poly}(\lambda)$. To denote a two-party protocol between two parties P_0 and P_1, we use the notation of $P_0[\mathtt{output}_{P_0}], P_1[\mathtt{output}_{P_1}] \leftarrow \mathtt{Protocol}(P_0[\mathtt{input}_{P_0}], P_1[\mathtt{input}_{P_1}])$. We denote $[n]$ as the set $\{1, \ldots, n\}$. $||$ denotes concatenation.

Collision-resistant hash function (CRHF) A hash function H is a deterministic function from an arbitrary size input to a fixed size output, denoted $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$. H is assumed to be *collision resistant* if it is hard in PPT to find two different inputs $x \neq y$ that they hash to the same output $\mathsf{H}(x) = \mathsf{H}(y)$.

Digital Signature. A digital signature scheme [26] is a public-key cryptographic primitive specified by three PPT algorithms. The **key generation** algorithm takes as input the security parameter 1^λ and outputs a signing key ssk and a corresponding verification key svk, written as $(\mathsf{ssk}, \mathsf{svk}) \leftarrow \mathsf{SignKeyGen}(1^\lambda)$. The **signing** algorithm takes as input ssk and a message m, and outputs a signature sig as $\mathsf{sig} \leftarrow \mathsf{Sign}_{\mathsf{ssk}}(m)$. The **verification** algorithm takes as input svk, a message m, and a signature sig, and outputs a bit $b \leftarrow \mathsf{SignVerify}_{\mathsf{svk}}(\mathsf{sig}, m) \in \{0,1\}$, where $b = 1$ if and only if sig is a valid signature on the message m under the verification key svk.

Threshold Oblivious Pseudorandom Function (TOPRF). Let $\mathcal{F} = \{F_K : X \to Y\}_K$ be a PRF family. A t-out-of-n *TOPRF* [14] extends OPRFs [22] to a distributed key K held as shares $\{k_i\}_{i\in[n]}$ so that any subset of size $\geq t$ enables one evaluation while $< t$ learn nothing useful. TOPRF is composed of two phases: 1) key generation as $\{\mathsf{K}, \{\mathsf{k}_i\}_{i\in[n]}\} \leftarrow \mathsf{TOPRFKeyGen}(1^\lambda, n, t)$ and k_i is sent to each sender; and 2) PRF evaluation among a receiver (user in our case) whose input is x and t-many senders (storage providers

in our case) whose corresponding input is the key share k_i as $F_{\mathsf{K}}(x) \leftarrow \mathsf{TOPRF}(x, \{\mathsf{k}_i\}_{i\in[t]})$. As a result, senders learn *nothing* while the receiver learns *only* $F_{\mathsf{K}}(x)$.

Authenticated Symmetric Encryption (AE). An authenticated-encryption (AE) scheme is a symmetric-key primitive that simultaneously provides *confidentiality* of messages and *integrity/authenticity* of ciphertexts. Formally, an AE scheme over a message space $\mathcal{M}$ and a ciphertext space $\mathcal{C}$ is specified by three PPT algorithms. The **key generation** algorithm samples a secret key $\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\lambda)$. The **encryption** algorithm takes sk and a message $m \in \mathcal{M}$ and outputs a ciphertext $c \in \mathcal{C}$, as $c \leftarrow \mathsf{Enc}_{\mathsf{sk}}(m)$, where Enc uses internal randomness. The **decryption** algorithm takes sk and a ciphertext $c \in \mathcal{C}$ and outputs either the recovered message $m' \in \mathcal{M}$ or a distinguished failure symbol $\perp$, written $m' \leftarrow \mathsf{Dec}_{\mathsf{sk}}(c)$.

Informally, AE is secure if it achieves both message privacy and ciphertext integrity, meaning that no PPT adversary can distinguish encryptions of chosen messages or forge a new ciphertext that decrypts to a non-$\perp$ value with more than negligible probability.

4 Design and Overview

4.1 System Model

Our secure updatable single password authentication protocol, denoted UpSPA, involves three types of entities.

Users $\mathcal{U}$. There are n_u users and each user $\mathcal{U}_l$ has an identifier Uid_l (e.g., a username) and a *single* human-memorable password pwd. Using $(\mathsf{Uid}_l, \mathsf{pwd})$, $\mathcal{U}_l$ registers and authenticates to as many login servers as desired. For simplicity in our security analysis, we assume pwd is sampled uniformly at random from a password dictionary Dict. We assume that the password is not easily guessable, meaning that it would resist online dictionary attacks (limited online attempts) or social engineering attacks. Our goal is to prevent adversaries from being able to mount offline dictionary attacks on pwd, or impersonating $\mathcal{U}_l$ against honest login servers. Users do not store any persistent high entropy secret locally other than memorizing their pwd, which is not stored by any party. Certain public parameters, such as the identifiers of storage providers (e.g., their domain names), are assumed to be known to $\mathcal{U}_l$, as in prior works [18,12].

Storage Providers $\mathcal{SP}$. There are n_{sp} storage providers, denoted by $\mathcal{SP} = \{\mathcal{SP}_i\}_{i\in[n_{sp}]}$. Storage providers store password-protected user records and participate in UpSPA to support authentication and update operations. Storage providers are not trusted a priori and may be statically corrupted by the adversary, up to some threshold.

Login Servers $\mathcal{LS}$. There are n_{ls} login servers, denoted by $\mathcal{LS} = \{\mathcal{LS}_j\}_{j\in[n_{ls}]}$. Each $\mathcal{LS}_j$ is identified by a public identifier ls_j (e.g., its domain name). $\mathcal{LS}_j$ authenticates users before granting service access. Each $\mathcal{LS}_j$ stores the username Uid_l and verification information vInfo_j associated with Uid_l (essentially vInfo_j is the login-server-specific password of the user, such that the user does not need to remember). Since we assume username-password based authentication, we do not discuss any privacy or anonymity of the users against login servers.

4.2 Threat Model

A static PPT adversary $\mathcal{A}$ can corrupt a set of entities at the beginning of the protocol execution and does not adaptively change this set later. Corruption gives $\mathcal{A}$ full control over the corrupted parties, including access to their internal state. We model a threshold parameter t_{sp}. Intuitively, UpSPA should remain secure even if there is only one honest user and one honest login server as long as up to $t_{sp} - 1$ storage providers, $n_{ls} - 1$ login servers, and $n_u - 1$ users are corrupted. Naturally, we require UpSPA to satisfy the following security goals.

- **Resistance to offline dictionary attacks.** Even if $\mathcal{A}$ corrupts $t_{sp} - 1$ storage providers and any subset of other parties (i.e., $n_{ls} - 1$ login servers –all login servers except one honest login server–, all other users

in the system except one honest user), $\mathcal{A}$ should not be able to successfully mount an offline dictionary attack on an honest user's password pwd (or any intermediate high entropy secret of the user) and recover pwd with non-negligible advantage over simply guessing from a dictionary and testing it online (which would be rate- and #attempts-limited).
- **Integrity and record manipulation resistance.** $\mathcal{A}$ should not be able to (i) create or preempt an honest user's provider-specific-secret for a login server (e.g., storing records on behalf of an honest user), or (ii) trigger secret updates or password updates on behalf of an honest user without knowledge of pwd. In particular, $\mathcal{A}$ shall not cause storage providers to accept forged updates, nor cause a login server to accept updated verification information vInfo_j that was not authorized by the honest user.

We do not consider malware on the user device, such as keyloggers, nor network level attacks such as DNS spoofing. We assume that parties communicate over secure and server-authenticated channels (such as TLS) and that the storage providers required for an update are available during update operations. Furthermore, we assume a synchronous communication model with a known upper bound on message delay and sufficiently synchronized clocks. Accordingly, if threshold-many responses are not received within the prescribed time window, the user aborts the protocol execution. This synchronization does not affect security. It simply helps perform the functionality properly in a timely manner.

4.3 Overview

UpSPA naturally consists of five phases: **Setup**, **Registration**, **Authentication**, **Secret Update**, and **Password Update**.

In **Setup**, the user $\mathcal{U}$ initializes long term secrets and stores them across storage providers under the protection of pwd. The secrets include a high-entropy *identifier secret* $\mathcal{R}^{sp}$ used to derive unique storage-provider-specific identifiers, a signing key used to explicitly authorize password updates, and a symmetric encryption key used to the per-login-server authentication secret. These values are stored as a compact password-protected storage record, using a threshold mechanism, i.e., threshold OPRF, so that $\mathcal{U}$ can recover the secrets by contacting any t_{sp} storage providers.

In **Registration**, $\mathcal{U}$ wishes to register with a login server $\mathcal{LS}_j$ with its domain name ls_j. First, $\mathcal{U}$ together with $\mathcal{SP}$ reconstructs the long-term secret using its pwd by computing threshold OPRF evaluation and decryption. For a login server $\mathcal{LS}_j$, $\mathcal{U}$ derives a per-login-server high-entropy secret $\mathcal{R}^{\mathsf{ls}_j}$ and registers Uid together with verification information vInfo_j derived from $\mathcal{R}^{\mathsf{ls}_j}$ and ls_j. Then, $\mathcal{R}^{\mathsf{ls}_j}$ is encrypted under the symmetric key and stored at the storage providers, where each storage provider $\mathcal{SP}_i$ receives the ciphertext and a specific identifier $\mathsf{SUid}_{l,i,j}$ derived from the identifier secret $\mathcal{R}^{sp}$, ls_j, and $\mathcal{SP}_i$.

In **Authentication**, $\mathcal{U}$ communicates with at least t_{sp} storage providers, reconstructs the required high-entropy login-server-specific secret using pwd, and authenticates to $\mathcal{LS}_j$ using vInfo_j computed by the reconstructed per-login-server secret and ls_j, without revealing pwd (or any deterministic function of it) to any $\mathcal{SP}$ or $\mathcal{LS}_j$.

In **Secret Update**, $\mathcal{U}$ rotates the per-login-server secret for a chosen $\mathcal{LS}_j$, updates the corresponding vInfo_j at $\mathcal{LS}_j$, and updates the associated ciphertexts stored at each $\mathcal{SP}$.

In **Password Update**, $\mathcal{U}$ updates its single password from pwd to pwd_{new} and refreshes the password-protected records at $\mathcal{SP}$. To ensure correctness in the presence of malicious $\mathcal{SP}$, UpSPA uses authenticated encryption for stored records and associates provider records with an epoch or version counter so that $\mathcal{U}$ accepts only the freshest record supported by a threshold number of $\mathcal{SP}$. Note that a password update does *not* trigger secret updates for each $\mathcal{LS}_j$. This is an important feature of UpSPA regarding efficiently and securely updating passwords.

Observe that Setup and Password Update protocols involve only storage providers and the user. Essentially, these are specific to our UpSPA, without the login servers being aware. On the other hand, Registration, Authentication, and Secret Update protocols involve login servers, as the user registers with, authenticates to, or updates the login information on these login servers. In those protocols, $\mathcal{SP}$ acts as helpers. In any

case, $\mathcal{SP}$ never learns enough information about pwd. Moreover, since each $\mathcal{LS}_j$ stores a different verification information vInfo_j, they cannot impersonate the user against each other. Overall, even if there is a single honest $\mathcal{U}$ in the system who wants to authenticate against a single honest $\mathcal{LS}_j$, assuming the adversary controls fewer than threshold-many storage providers, we protect security of the user's single pwd and the user's account on the honest login server against offline dictionary attacks, impersonation, phishing, and honeypots (and even man-in-the-middle assuming TLS works as expected).

4.4 Formal Security Definition

We define the *ideal world*, denoted by $\mathcal{F}_{\mathsf{UpSPA}}$ and the *real world*, denoted by Π_{UpSPA} for an updatable single password authentication protocol in the spirit of Canetti [13].

Ideal World: The ideal world consists of a set of users $\mathcal{U}$, a set of storage providers $\mathcal{SP}$, a set of login servers $\mathcal{LS}$, and the ideal functionality $\mathcal{F}_{\mathsf{UpSPA}}$ as shown by Figure **??**. In our ideal functionality, the parties do not learn the password pwd or the per-login-server vInfo_j of the user. Thus, any protocol achieving this ideal functionality must satisfy our security goals, disabling offline attacks on the password or login server verification information. Yet, note that any interaction with the ideal functionality in the ideal world is implicitly authenticated: the ideal functionality always knows who it is talking to, and when parties receive something from the ideal functionality, they know that it is correct, originating from a legitimate request. Any protocol realizing this must ensure this implicit authentication in reality as well. In our case, we employ TLS, random identifiers, and signature verification for this purpose.

Real World: The real world consists of a user $\mathcal{U}$, a set of storage providers $\mathcal{SP}$, and a (set of) login server $\mathcal{LS}$. There is no ideal functionality $\mathcal{F}_{\mathsf{UpSPA}}$ for a real world protocol Π_{UpSPA} for the updatable SPA protocol.

Definition 1 (Secure Updatable Single Password Authentication Protocol). *Let Π_{UpSPA} be a PPT protocol for an updatable single password authentication protocol. We say that Π_{UpSPA} is secure if for every PPT real world adversary $\mathcal{A}$ attacking Π_{UpSPA}, there exists a PPT ideal world simulator $\mathcal{S}$ such that the real and ideal world interactions and outputs are computationally indistinguishable:*

$$\mathsf{IDEAL}^{\mathcal{S}(aux)}_{\mathcal{F}_{\mathsf{UpSPA}}}(\mathsf{Uid}, 1^\lambda) \equiv_c \mathsf{REAL}^{\mathcal{A}(aux)}_{\Pi_{\mathsf{UpSPA}}}(\mathsf{Uid}, 1^\lambda)$$

where $aux \in \{0,1\}^*$ denotes the auxiliary input.

Functionality $\mathcal{F}_{\mathsf{UpSPA}}$

Parameterized with a set of users $\mathcal{U} = \{\mathcal{U}_l\}_{l\in[n_u]}$, a set of login servers $\mathcal{LS} = \{\mathcal{LS}_j\}_{j\in[n_{ls}]}$, and a set of storage providers $\mathcal{SP} = \{\mathcal{SP}_i\}_{i\in[n_{sp}]}$, n_u: total number of users, n_{ls}: total number of login servers, t_{sp}: threshold value for $\mathcal{SP}$, and n_{sp}: total number of storage providers, $\mathcal{F}_{\mathsf{UpSPA}}$ works as follows:

Setup. Upon receiving a message $\langle \mathtt{Setup}, \mathsf{Uid}_l \rangle$ from $\mathcal{U}_l$,

1. If Uid_l exists in its database, return Fail to $\mathcal{U}_l$. Otherwise, send $\langle \mathtt{Setup}, \mathsf{Uid}_l \rangle$ to each storage provider $\mathcal{SP}_i \in \mathcal{SP}$.
2. Upon receiving OK from each $\mathcal{SP}_i \in \mathcal{SP}$, send OK to $\mathcal{U}_l$ and store Uid_l in its database. Otherwise (i.e., upon receiving Fail from some $\mathcal{SP}_i$), send Fail to $\mathcal{U}_l$.

Functionality $\mathcal{F}_{\mathsf{UpSPA}}$

Registration. Upon receiving a message $\langle \texttt{Register}, \mathsf{Uid}_l, \mathsf{ls}_j \rangle$ from $\mathcal{U}_l$,

1. If Uid_l does not exist in its database or a tuple including $\langle \mathsf{Uid}_l, \mathsf{ls}_j \rangle$ exists, send Fail to $\mathcal{U}_l$. Otherwise, send $\langle \texttt{Register}, \mathsf{Uid}_l \rangle$ to each $\mathcal{SP}_i \in \mathcal{SP}$.
2. Upon receiving OK from each $\mathcal{SP}_i \in \mathcal{SP}$, compute a unique identifier $\mathsf{SUid}_{l,i,j} \leftarrow \{0,1\}^\lambda$ for each $\mathcal{SP}_i$ and send $\langle \texttt{Register}, \mathsf{SUid}_{l,i,j} \rangle$ to each corresponding $\mathcal{SP}_i \in \mathcal{SP}$.
3. Upon receiving OK from each $\mathcal{SP}_i \in \mathcal{SP}$, send $\langle \texttt{Register}, \mathsf{Uid}_{l,j} \rangle$ to $\mathcal{LS}_j$ and store $\langle \mathsf{Uid}_l, \mathsf{Uid}_{l,j}, \mathsf{ls}_j, \{\mathsf{SUid}_{l,i,j}\}_{i \in [n_{sp}]} \rangle$ in the database. Otherwise (i.e., upon receiving Fail from some $\mathcal{SP}_i \in \mathcal{SP}$), send Fail to $\mathcal{U}_l$.

Authentication. Upon receiving a message $\langle \texttt{Authenticate}, \mathsf{Uid}_l, \mathsf{ls}_j \rangle$ from $\mathcal{U}_l$, $\mathcal{F}_{\mathsf{UpSPA}}$:

1. Send Fail to $\mathcal{U}_l$ if Uid_l does not exist in its database or a tuple including $\langle \mathsf{Uid}_l, \mathsf{ls}_j \rangle$ does not exist. Otherwise, send $\langle \texttt{Authenticate}, \mathsf{Uid}_l \rangle$ to each $\mathcal{SP}_i \in \mathcal{SP}$.
2. Upon receiving OK from at least t_{sp} storage providers $\{\mathcal{SP}_i\}_{i \in [t_{sp}]} \in \mathcal{SP}$, sends $\langle \texttt{Authenticate}, \mathsf{SUid}_{l,i,j} \rangle$ to each $\mathcal{SP}_i \in \mathcal{SP}$.
3. Upon receiving OK from each $\mathcal{SP}_i$ responded previously, send $\langle \texttt{Authenticate}, \mathsf{Uid}_{l,j} \rangle$ to $\mathcal{LS}_j$. Otherwise, send Fail to $\mathcal{U}_l$.
4. Upon receiving OK from $\mathcal{LS}_j$, send OK to $\mathcal{U}_l$. Otherwise, send Fail to $\mathcal{U}_l$.

Secret Update. Upon receiving a message $\langle \texttt{SecretUpdate}, \mathsf{Uid}_l, \mathsf{ls}_j \rangle$ from $\mathcal{U}_l$,

1. If $\langle \mathsf{Uid}_l, \mathsf{ls}_j \rangle$ does not exists in the database, send Fail to $\mathcal{U}_l$. Otherwise, send $\langle \texttt{SecretUpdate}, \mathsf{Uid}_l \rangle$ to each $\mathcal{SP}_i \in \mathcal{SP}$.
2. Upon receiving OK from each $\mathcal{SP}_i \in \mathcal{SP}$, send $\langle \texttt{SecretUpdate}, \mathsf{SUid}_{l,i,j} \rangle$ to $\mathcal{SP}_i \in \mathcal{SP}$.
3. Upon receiving OK from all $\mathcal{SP}_i$, send $\langle \texttt{SecretUpdate}, \mathsf{Uid}_{l,j} \rangle$ to $\mathcal{LS}_j$. Otherwise, send Fail to $\mathcal{U}_l$.
4. Upon receiving OK from $\mathcal{LS}_j$, send OK to $\mathcal{U}_l$. Otherwise, send Fail to $\mathcal{U}_l$.

Password Update. Upon receiving a message $\langle \texttt{PwdUpdate}, \mathsf{Uid}_l \rangle$ from $\mathcal{U}_l$,

1. If Uid_l does not exists in the database, send Fail to $\mathcal{U}_l$. Otherwise, send Uid_l to each $\{\mathcal{SP}_i\}_{i \in [n_{sp}]}$.
2. Upon receiving OK from each $\{\mathcal{SP}_i\}_{i \in [n_{sp}]}$, send OK to $\mathcal{U}$. Otherwise, send Fail to $\mathcal{U}_l$ and each $\mathcal{SP}_i$.

5 Updatable SPA (UpSPA) Construction

We describe the detailed construction of UpSPA. For ease of exposition, we describe our protocol where we assume a single user $\mathcal{U}_l$ with its username Uid_l and a single login server $\mathcal{LS}_j$ with its domain name ls_j. It is applicable to multiple users and login servers.

Setup phase Π_{Setup}, as defined by Protocol 1, is a one-time phase when a user $\mathcal{U}_l$ joins the system, in order to initialize long-term high-entropy secrets using its username Uid_l and single password pwd. It is only executed between the user and the storage providers, without the login servers being included. First, $\mathcal{U}_l$ generates a random identifier as $\mathcal{R}^{sp} \leftarrow \{0,1\}^\lambda$. $\mathcal{R}^{sp}$ is employed to derive unique storage-provider-specific identifiers in the registration phase. Later, to ensure threshold security, $\mathcal{U}_l$ runs a TOPRF key generation to derive a PRF key K and a corresponding PRF key share k_i of K for each $\mathcal{SP}_i$ as $\{\mathsf{K}, \{\mathsf{k}_i\}_{i \in [n_{sp}]}\} \leftarrow \mathsf{TOPRFKeyGen}(1^\lambda, n_{sp}, t_{sp})$. Then, $\mathcal{U}_l$ assigns an encryption key $\mathsf{K_0} = \mathsf{KeyGen}(1^\lambda)$. $\mathsf{K_0}$ will be used to encrypt the authentication credentials generated during the registration and secret update phases; hence, we do not re-encrypt every login server secret with the new password during password update. This enables UpSPA to be very efficient during the password update. $\mathcal{U}_l$ generates signing and verification keys by running the digital signature key generation algorithm as $\mathsf{ssk}, \mathsf{svk} \leftarrow \mathsf{SignKeyGen}(1^\lambda)$. The purpose of signature key generation is to create an explicit yet private authentication mechanism for $\mathcal{SP}_i$ when $\mathcal{U}_l$ wishes to update

its single password. The verification key svk enables $\mathcal{SP}_i$ to verify that an update request originates from the legitimate $\mathcal{U}_l$, while the corresponding signing key ssk serves as the secret signing material, allowing only $\mathcal{U}_l$, upon successful reconstruction with the correct pwd, to generate a valid signature. $\mathcal{U}_l$ stores Uid_l, svk, and k_i in plaintext, while storing $\mathcal{R}^{sp}$, ssk, and K_0 in an encrypted, password-protected record. Furthermore, UpSPA requires a key-binding symmetric encryption scheme so that decryption fails whenever an incorrect key is used (e.g., if k_i is incorrect) or the ciphertext is malformed. To achieve this, UpSPA adopts an authenticated encryption scheme and encrypts $\mathcal{R}^{sp}$, ssk, and K_0 under $F_{\mathsf{K}}(\mathsf{pwd})$ as $\mathsf{cid} \leftarrow \mathsf{Enc}_{F_{\mathsf{K}}(\mathsf{pwd})}(\mathsf{ssk} \parallel \mathcal{R}^{sp} \parallel \mathsf{K}_0)$. $\mathcal{U}_l$ then sends $\langle \mathsf{Uid}_l, \{\mathsf{svk}, \mathsf{cid}, \mathsf{k}_i\}\rangle$ to each $\mathcal{SP}_i$, and each $\mathcal{SP}_i$ looks up its storage $\mathsf{DB_i}$ if $\mathsf{DB_i}.\mathsf{Search}(\mathsf{Uid}_l) = \perp$ stores the corresponding tuple in $\mathsf{DB_i}$ and sends OK to $\mathcal{U}_l$; else, sends FAIL to $\mathcal{U}_l$. After this step completes successfully, $\mathcal{U}_l$ may discard all computed values and only needs to remember Uid_l and pwd.

Protocol 1: Setup Π_{Setup}

1. $\mathcal{U}_l:\ \mathcal{R}^{sp} \leftarrow \{0,1\}^{\lambda}$
2. $\mathcal{U}_l: \{\mathsf{K}, \{\mathsf{k}_i\}_{i\in[n_{sp}]}\} \leftarrow \mathsf{TOPRFKeyGen}(1^{\lambda}, n_{sp}, t_{sp})$
3. $\mathcal{U}_l:\ \langle \mathsf{ssk}, \mathsf{svk}\rangle \leftarrow \mathsf{SignKeyGen}(1^{\lambda})$;
4. $\mathcal{U}_l: \mathsf{K}_0 \leftarrow \mathsf{KeyGen}(1^{\lambda})$
5. $\mathcal{U}_l:\ \mathsf{cid} \leftarrow \mathsf{Enc}_{F_{\mathsf{K}}(\mathsf{pwd})}(\mathsf{ssk} \parallel \mathcal{R}^{sp} \parallel \mathsf{K}_0)$
6. $\mathcal{U}_l \rightarrow \mathcal{SP}_i:\ \langle \mathsf{Uid}_l, \mathsf{svk}, \mathsf{cid}, \mathsf{k}_i\rangle$
7. $\mathcal{SP}_i:$ **IF** $\mathsf{DB}_i.\mathsf{SEARCH}(\mathsf{Uid}_l) = \perp : \mathsf{DB}_i.\mathsf{STORE}(\mathsf{Uid}_l, \mathsf{svk}, \mathsf{cid}, \mathsf{k}_i)$ and send OK to $\mathcal{U}$; **ELSE** return FAIL

Protocol 2: Registration Π_{Register}

1. $\mathcal{U}_l \rightarrow \{\mathcal{SP}_i\}_{i\in[n_{sp}]}:\ \mathsf{Uid}_l$
2. $\mathcal{SP}_i$: **IF** $\mathsf{DB}_i.\mathsf{SEARCH}(\mathsf{Uid}_l) = \perp:$ return Fail; **ELSE** return OK
3. $\mathcal{U}_l$: **IF** $|\{\mathsf{OK}\}| = n_{sp}$: **COMPUTE** $\mathcal{U}[F_{\mathsf{K}}(\mathsf{pwd})] \leftarrow \mathsf{TOPRF}(\mathcal{U}[\mathsf{pwd}], \{\mathcal{SP}_i[\mathsf{k}_i]\}_{i\in[t_{sp}]})$; **ELSE** ABORT
4. $\mathcal{SP}_i \rightarrow \mathcal{U}_l:\ \mathsf{cid}$
5. $\mathcal{U}_l$: **IF** $(\mathsf{ssk} \parallel \mathcal{R}^{sp} \parallel \mathsf{K}_0) \leftarrow \mathsf{Dec}_{F_{\mathsf{K}}(\mathsf{pwd})}(\mathsf{cid})$; **ELSE** ABORT
6. $\mathcal{U}_l:\ \mathsf{SUid}_{l,i,j} \leftarrow \mathsf{H}(\mathcal{R}^{sp} \parallel \mathsf{ls}_j \parallel i) \quad \forall i \in [n_{sp}]$;
7. $\mathcal{U}_l:\ \mathcal{R}^{\mathsf{ls}_j} \leftarrow \{0,1\}^{\lambda}$; $\mathsf{ctr} \leftarrow 0; \mathsf{c}_j \leftarrow \mathsf{Enc}_{\mathsf{K}_0}(\mathcal{R}^{\mathsf{ls}_j} \parallel \mathsf{ctr})$
8. $\mathcal{U}_l:\ \mathsf{vInfo}_j \leftarrow \mathsf{H}(\mathcal{R}^{\mathsf{ls}_j} \parallel \mathsf{ls}_j)$
9. $\mathcal{U}_l \rightarrow \mathcal{SP}_i:\ \langle \mathsf{SUid}_{l,i,j}, \mathsf{c}_j\rangle$
10. $\mathcal{SP}_i$: **IF** $\mathsf{DB}_i.\mathsf{SEARCH}(\mathsf{SUid}_{l,i,j}) = \perp : \mathsf{DB}_i.\mathsf{STORE}(\mathsf{SUid}_{l,i,j}, \mathsf{c}_j)$ and return OK; **ELSE** return Fail
11. $\mathcal{U}_l \rightarrow \mathcal{LS}_j:$ **IF** $|\{\mathsf{OK}\}| = n_{sp}$: send $(\mathsf{Uid}_l, \mathsf{vInfo}_j)$; **ELSE** ABORT
12. $\mathcal{LS}_j:$ **IF** $\mathsf{DB}_j.\mathsf{SEARCH}(\mathsf{Uid}_l) = \perp : \mathsf{DB}_j.\mathsf{STORE}(\mathsf{Uid}_l, \mathsf{vInfo}_j)$; **ELSE** return FAIL

In the **Registration** phase Π_{Register}, defined in Protocol 2, $\mathcal{U}_l$, with username Uid_l and single password pwd, wishes to create an account on a login server $\mathcal{LS}_j$ with domain name ls_j. Note that in our approach, Uid_l is used for both storage providers and login servers for simplicity. However, without loss of generality, $\mathcal{U}_l$ may choose a username with the login servers different from the one with the storage providers. Indeed, as in the current case, as long as the user can remember, it can pick different usernames with each login server as well. We note that the usernames are assumed to be known/guessed by the adversary, and hence they cannot be employed as secret information. To do so, $\mathcal{U}_l$ must first reconstruct $\mathcal{R}^{sp}$ and K_0. $\mathcal{U}_l$ begins by sending its username Uid_l to all storage providers. Once all n_{sp} storage providers approve the request, $\mathcal{U}_l$ selects, without loss of generality, t_{sp} storage providers (or proceed with n_{sp} storage providers) and jointly executes the threshold oblivious pseudorandom function (TOPRF) protocol with $\{\mathcal{SP}_i\}_{i\in[t_{sp}]}$. During the TOPRF

execution, each storage provider $\mathcal{SP}_i$ acts as a sender with input k_i, while $\mathcal{U}_l$ acts as the receiver with input pwd. At the end of the protocol, $\mathcal{U}_l$ obtains the TOPRF output $F_{\mathsf{K}}(\mathsf{pwd})$, while the storage providers learn nothing about pwd or the output.

After the TOPRF execution, each $\mathcal{SP}_i$ sends cid associated with Uid_l from its database DB_i to $\mathcal{U}_l$. Thus, $\mathcal{U}_l$ receives t_{sp} ciphertexts and checks whether they are consistent. Note that an adversarial storage provider may return an arbitrary ciphertext $\tilde{\mathsf{cid}}$. However, as long as there is at least one honest $\mathcal{SP}_i$, $\mathcal{U}_l$ can recover the correct cid since the arbitrary ciphertext(s) won't be decrypted properly using authenticated encryption. Using $F_{\mathsf{K}}(\mathsf{pwd})$ as the decryption key, $\mathcal{U}_l$ decrypts cid to reconstruct ssk, $\mathcal{R}^{sp}$, and K_0 as $\mathsf{ssk} \parallel \mathcal{R}^{sp} \parallel \mathsf{K}_0 = \mathsf{Dec}_{F_{\mathsf{K}}(\mathsf{pwd})}(\mathsf{cid})$. The reconstructed $\mathcal{R}^{sp}$ are then used to generate storage provider and login server specific identifiers. Specifically, $\mathcal{U}_l$ computes an identifier $\mathsf{SUid}_{l,i,j}$ for each storage provider $\mathcal{SP}_i \in \{\mathcal{SP}_i\}_{i\in[n_{sp}]}$ using a collision-resistant hash function: $\mathsf{SUid}_{l,i,j} \leftarrow \mathsf{H}(\mathcal{R}^{sp} \parallel \mathsf{ls}_j \parallel i)$. Only the legitimate $\mathcal{U}_l$ can later reconstruct $\mathsf{SUid}_{l,i,j}$, and this identifier serves as a means of implicit authentication during future updates. Observe that the identifier is different for each storage provider $\mathcal{SP}_i$ due to the input i to the hash function, so they cannot use this information to impersonate the user against each other. Next, $\mathcal{U}_l$ generates a high-entropy secret $\mathcal{R}^{\mathsf{ls}_j}$ for login server $\mathcal{LS}_j$: $\mathcal{R}^{\mathsf{ls}_j} \leftarrow \{0,1\}^{\lambda}$. This value is used to derive login-server-specific verification information. To store $\mathcal{R}^{\mathsf{ls}_j}$ securely at the storage providers, $\mathcal{U}_l$ protects it using K_0. In particular, $\mathcal{U}_l$ encrypts $\mathcal{R}^{\mathsf{ls}_j}$ together with a counter ctr, initialized to 0, as $\mathsf{cid} \leftarrow \mathsf{Enc}_{F_{\mathsf{K}_0}}(\mathcal{R}^{\mathsf{ls}_j} \parallel \mathsf{ctr})$. The counter ctr acts as a version number and ensures that the most up-to-date $\mathcal{R}^{\mathsf{ls}_j}$ is used, thereby preventing a malicious $\mathcal{SP}_i$ from replying with an outdated ciphertext. The user can detect this via the response of the honest storage providers, as long as the adversary controls fewer than threshold many $\mathcal{SP}$. $\mathcal{U}_l$ later computes the verification information vInfo_j for $\mathcal{LS}_j$ using a collision-resistant hash function: $\mathsf{vInfo}_j \leftarrow \mathsf{H}(\mathcal{R}^{\mathsf{ls}_j} \parallel \mathsf{ls}_j)$. Including ls_j in the hash binds the verification information to the intended login server and protects $\mathcal{U}_l$ against phishing and man-in-the-middle attacks, as discussed in [33,18,21]. vInfo_j can be thought as a site-specific password normally an ideal user would employ (a different complex password per login server). It can be safely re-generated only by the user who knows the correct password. After computing all required values, $\mathcal{U}_l$ sends $\langle \mathsf{SUid}_{l,i,j}, \mathsf{c}_j \rangle$ to each storage provider $\mathcal{SP}_i$ for storage. If any $\mathcal{SP}_i$ finds an existing tuple (i.e., account) in its database associated with $\mathsf{SUid}_{l,i,j}$, it responds with FAIL to $\mathcal{U}_l$. Otherwise, all storage providers return an OK message, indicating successful registration and storage.After receiving an OK message from each $\mathcal{SP}_i$, $\mathcal{U}_l$ sends $\langle \mathsf{Uid}_l, \mathsf{vInfo}_j \rangle$ to $\mathcal{LS}_j$. Upon receiving $\langle \mathsf{Uid}_l, \mathsf{vInfo}_j \rangle$, $\mathcal{LS}_j$ stores vInfo_j, provided that no account is already associated with Uid_l; otherwise, it sends FAIL to $\mathcal{U}_l$. Therefore, $\mathcal{U}_l$ sends a $\mathsf{Discard}$ command to all storage providers. Furthermore, if $\mathcal{SP}_i$ store the tuple, then, due to the uniqueness of $\mathsf{SUid}_{l,i,j}$, $\mathcal{U}_l$ cannot create an account with that specific ls_j. This part is the current standard username-password authentication; indeed UpSPA does not necessitate any changes on $\mathcal{LS}$. Once setup, it can be immediately used on all login servers employing username-password based authentication. As in traditional password authentication, $\mathcal{LS}_j$ should ideally store a salted and hashed form of vInfo_j. Yet, since vInfo_j is different for each $\mathcal{LS}_j$, even when a login server is attacked, the user's accounts on other login servers would be protected in our system. Finally, $\mathcal{U}_l$ can discard all generated secret values that are cumbersome to remember, the human remembering only the necessary long-term credentials (Uid_l and pwd) as in the setup phase.

Authentication phase $\Pi_{\mathsf{Authenticate}}$ is defined by Protocol 3 and is executed when $\mathcal{U}_l$ wishes to log in to its account on $\mathcal{LS}_j$. To authenticate, $\mathcal{U}_l$ must first reconstruct $\mathcal{R}^{sp}$ in order to derive the provider-specific identifiers $\mathsf{SUid}_{l,i,j}$. Thus, $\mathcal{U}_l$ follows Steps 1–6 of Π_{Register} to reconstruct $\mathcal{R}^{sp}$ and K_0, and then computes $\mathsf{SUid}_{l,i,j} \leftarrow \mathsf{H}(\mathcal{R}^{sp} \parallel \mathsf{ls}_j \parallel i)$ for each corresponding $\mathcal{SP}_i$. $\mathcal{U}_l$ sends each identifier $\mathsf{SUid}_{l,i,j}$ to its corresponding storage provider $\mathcal{SP}_i$, who then respond with c_j. Next, using the reconstructed K_0, $\mathcal{U}_l$ decrypts the recovered c_j values to obtain $\mathcal{R}^{\mathsf{ls}_j}$ and the associated counter ctr. If multiple candidate ciphertexts are received, $\mathcal{U}_l$ selects the freshest value according to the largest counter. If there are mismatches, $\mathcal{U}_l$ can find the correct one due to the use of authenticated encryption. Then, $\mathcal{U}_l$ computes the verification information for $\mathcal{LS}_j$ as $\mathsf{vInfo}'_j \leftarrow \mathsf{H}(\mathcal{R}^{\mathsf{ls}_j} \parallel \mathsf{ls}_j)$ and sends $\langle \mathsf{Uid}_l, \mathsf{vInfo}'_j \rangle$ to $\mathcal{LS}_j$. Upon receiving $\langle \mathsf{Uid}_l, \mathsf{vInfo}'_j \rangle$, $\mathcal{LS}_j$ retrieves the record associated with Uid_l and verifies whether the received value matches the stored verifier.

Protocol 3: Authentication $\Pi_{\mathsf{Authenticate}}$

1. Follow the Steps 1-6 in Π_{Register} with t_{sp}-many $\mathcal{SP}$
2. $\mathcal{U}_l \rightarrow \mathcal{SP}_i$: $\mathsf{SUid}_{l,i,j}$
3. $\mathcal{SP}_i$: **IF** $\mathsf{DB}_i.\mathsf{SEARCH}(\mathsf{SUid}_{l,i,j}) \neq \perp$ sends c_j to $\mathcal{U}_l$; **ELSE** sends FAIL to $\mathcal{U}_l$
4. $\mathcal{U}_l$: $\mathcal{R}^{\mathsf{ls}_j} \parallel \mathsf{ctr} \leftarrow \mathsf{Dec}_{\mathsf{K}_0}(\mathsf{c}_j); \mathcal{R}^{\mathsf{ls}_j} \leftarrow \max_{\mathsf{ctr}}(\mathcal{R}^{\mathsf{ls}_j})$
5. $\mathcal{U}_l$: $\mathsf{vInfo}'_j \leftarrow \mathsf{H}(\mathcal{R}^{\mathsf{ls}_j} \parallel \mathsf{ls}_j)$
6. $\mathcal{U}_l \rightarrow \mathcal{LS}_j$: $\langle \mathsf{Uid}_l, \mathsf{vInfo}'_j \rangle$
7. $\mathcal{LS}_j$: **IF** $\mathsf{H}(\mathsf{vInfo}_j) \equiv \mathsf{H}(\mathsf{vInfo}'_j)$: return OK; **ELSE** return FAIL

Secret Update phase ($\Pi_{\mathsf{SecretUpdate}}$). In $\Pi_{\mathsf{SecretUpdate}}$, $\mathcal{U}_l$ wishes to update the verification information associated with its account on a chosen login server $\mathcal{LS}_j$ with domain name ls_j, as defined in Protocol 4. To perform the secret update, $\mathcal{U}_l$ must authenticate itself to both the storage providers and $\mathcal{LS}_j$. Authentication to $\mathcal{LS}_j$ requires proving knowledge of the current verifier by sending the correct vInfo_j after which the new verification information $\mathsf{vInfo}_{j,new}$ can be set in the server database. In other words, this step amounts to a successful authentication by $\mathcal{U}_l$ and then a regular password update on a login server. Authentication to each $\mathcal{SP}_i$ is implicit: sending the correct provider-specific identifier $\mathsf{SUid}_{l,i,j}$ is sufficient for $\mathcal{SP}_i$ to locate and update the corresponding stored record.

Protocol 4: Secret Update $\Pi_{\mathsf{SecretUpdate}}$

1. Follow the Steps 1-5 in $\Pi_{\mathsf{Authenticate}}$ with n_{sp}-many $\mathcal{SP}$
2. $\mathcal{U}_l$: $\mathsf{vInfo}'_j \leftarrow \mathsf{H}(\mathcal{R}^{\mathsf{ls}_j} \parallel \mathsf{ls}_j)$; $\mathcal{R}^{\mathsf{ls}_j}_{new} \leftarrow \{0,1\}^{\lambda}$; $\mathsf{vInfo}_{j,new} \leftarrow \mathsf{H}(\mathcal{R}^{\mathsf{ls}_j}_{new} \parallel \mathsf{ls}_j)$; $\mathsf{ctr} \leftarrow \mathsf{ctr}+1$; $\mathsf{c}_{j,new} \leftarrow \mathsf{Enc}_{\mathsf{K}_0}(\mathcal{R}^{\mathsf{ls}_j}_{new} \parallel \mathsf{ctr})$
3. $\mathcal{U}_l \rightarrow \mathcal{SP}_i : \langle \mathsf{SUid}_{l,i,j}, \mathsf{c}_{j,new} \rangle$
4. $\mathcal{SP}_i$: IF $\mathsf{DB}_i.\mathsf{SEARCH}(\mathsf{SUid}_{l,i,j}) \neq \perp$ then $\mathsf{DB}_i.\mathsf{UPDATE}$ $(\mathsf{SUid}_{l,i,j}, \mathsf{c}_{j,new})$ and return OK to $\mathcal{U}_l$ **ELSE** sends FAIL to $\mathcal{U}_l$
5. $\mathcal{U}_l \rightarrow \mathcal{LS}_j$: **IF** $|\{\mathsf{OK}\}| = n_{sp} : \langle \mathsf{Uid}_l, \mathsf{vInfo}'_j, \mathsf{vInfo}_{j,new} \rangle$; **ELSE** ABORT
6. $\mathcal{LS}_j$: **IF** $\mathsf{H}(\mathsf{vInfo}_j) = \mathsf{H}(\mathsf{vInfo}'_j)$: $\mathsf{DB}_j.\mathsf{UPDATE}(\mathsf{Uid}_l, \mathsf{vInfo}_{j,new})$, and return OK; **ELSE** return FAIL

To proceed with this phase, $\mathcal{U}_l$ contacts all storage providers and performs Steps 1-5 of $\Pi_{\mathsf{Authenticate}}$. In doing so, $\mathcal{U}_l$ reconstructs $\mathcal{R}^{sp}$ and K_0 from cid, reconstructs $\mathcal{R}^{\mathsf{ls}_j}$ from c_j, and derives $\mathsf{SUid}_{l,i,j}$ using $\mathcal{R}^{sp}$ Next, $\mathcal{U}_l$ computes the current verification information as $\mathsf{vInfo}'_j \leftarrow \mathsf{H}(\mathcal{R}^{\mathsf{ls}_j} \parallel \mathsf{ls}_j)$, and generates a fresh high-entropy login-server-specific secret as $\mathcal{R}^{\mathsf{ls}_j}_{new} \leftarrow \{0,1\}^{\lambda}$. Using this fresh value, $\mathcal{U}_l$ computes the new verification information as $\mathsf{vInfo}_{j,new} \leftarrow \mathsf{H}(\mathcal{R}^{\mathsf{ls}_j}_{new} \parallel \mathsf{ls}_j)$. To update the storage providers, $\mathcal{U}_l$ encrypts the fresh secret together with an incremented counter $\mathsf{ctr} \leftarrow \mathsf{ctr}+1$ as $\mathsf{c}_{j,new} \leftarrow \mathsf{Enc}_{\mathsf{K}_0}(\mathcal{R}^{\mathsf{ls}_j}_{new} \parallel \mathsf{ctr})$. $\mathcal{U}_l$ then sends $\langle \mathsf{SUid}_{l,i,j}, \mathsf{c}_{j,new} \rangle$ to each storage provider $\mathcal{SP}_i$. If $\mathsf{SUid}_{l,i,j}$ matches an existing record, this confirms that $\mathcal{U}_l$ is the legitimate user for that entry, and $\mathcal{SP}_i$ updates its database accordingly before returning an OK message to $\mathcal{U}_l$. Once $\mathcal{U}_l$ receives OK from all storage providers, it sends $\langle \mathsf{Uid}_l, \mathsf{vInfo}'_j, \mathsf{vInfo}_{j,new} \rangle$ to $\mathcal{LS}_j$. The login server retrieves the current verifier associated with Uid_l and accepts the update if and only if vInfo'_j matches the existing value in its database. If so, $\mathcal{LS}_j$ replaces the old verification information with $\mathsf{vInfo}_{j,new}$ and returns OK; otherwise, it returns FAIL to $\mathcal{U}_l$.Therefore, $\mathcal{U}_l$ sends a $\mathsf{Discard}$ command to all storage providers. Furthermore, if $\mathcal{SP}_i$ stores the tuple, then, due to the uniqueness of $\mathsf{SUid}_{l,i,j}$, $\mathcal{U}_l$ cannot create an account with that specific ls_j.

Password Update phase ($\Pi_{\mathsf{PwdUpdate}}$). In $\Pi_{\mathsf{PwdUpdate}}$, as defined in Protocol 5, $\mathcal{U}_l$ wishes to replace its single password pwd with a new password pwd_{new} of its choice. The password update phase does not involve the login server, since the login-server-side verification information remains unchanged. The goal

of $\Pi_{\mathsf{PwdUpdate}}$ is only to refresh the encryption of identification secrets (i.e., ssk, $\mathcal{R}^{sp}$) and update TOPRF keys for improved efficiency, thereby avoiding re-running the TOPRF interaction with $\mathcal{SP}$-side to update the password-protected record stored at the storage providers. To initiate this phase, $\mathcal{U}_l$ contacts all storage providers and executes Steps 1–5 of Π_{Register}. Through this process, $\mathcal{U}_l$ reconstructs ssk, $\mathcal{R}^{sp}$ and K_0 from cid, reconstructs $\mathcal{R}^{\mathsf{ls}_j}$ from c_j, and derives $\mathsf{SUid}_{l,i,j}$ using $\mathcal{R}^{sp}$.

Protocol 5: Password Update $\Pi_{\mathsf{PwdUpdate}}$

1. Follow the Steps 1-5 in Π_{Register} with n_{sp}-many $\mathcal{SP}$
2. $\mathcal{U}_l$: $\{\mathsf{K}_{new}, \{\mathsf{k}_{i,new}\}_{i\in[n_{sp}]}\} \leftarrow \mathsf{TOPRFKeyGen}(1^\lambda, n_{sp}, t_{sp})$
3. $\mathcal{U}_l$: $\mathsf{cid}_{\mathsf{new}} \leftarrow \mathsf{Enc}_{F_{\mathsf{K}_{new}}(\mathsf{pwd}_{new})}(\mathsf{ssk} \parallel \mathcal{R}^{sp} \parallel \mathsf{K}_0)$
4. $\mathcal{U}_l$: $\mathsf{sig}_i \leftarrow \mathsf{Sign}_{\mathsf{ssk}}(\mathsf{cid}_{\mathsf{new}} \parallel \mathsf{k}_{i,new} \parallel \mathsf{time} \parallel i)$
5. $\mathcal{U}_l \rightarrow \mathcal{SP}_i$: $\langle \mathsf{sig}_i, \mathsf{cid}_{\mathsf{new}}, \mathsf{k}_{i,new} \rangle$
6. $\mathcal{SP}_i$: **IF** $\mathsf{SignVerify}_{\mathsf{svk}}(\mathsf{sig}_i, \mathsf{cid}_{\mathsf{new}} \parallel \mathsf{k}_{i,new} \parallel \mathsf{time} \parallel i) = 1$: $\mathsf{DB}_i.\mathsf{UPDATE}(\mathsf{Uid}_l, \mathsf{cid}_{\mathsf{new}}, \mathsf{k}_{i,new})$; **ELSE** return FAIL

After reconstructing these values, $\mathcal{U}_l$ generates a fresh TOPRF key set as $\{\mathsf{K}_{new}, \{\mathsf{k}_{i,new}\}_{i\in[n_{sp}]}\} \leftarrow \mathsf{TOPRFKeyGen}(1^\lambda, n_{sp}, t_{sp})$. Then, using the new password pwd_{new}, $\mathcal{U}_l$ then re-encrypts the recovered secret state under the derived key $F_{\mathsf{K}_{new}}(\mathsf{pwd}_{new})$ as $\mathsf{cid}_{\mathsf{new}} \leftarrow \mathsf{Enc}_{F_{\mathsf{K}_{new}}(\mathsf{pwd}_{new})}(\mathsf{ssk} \parallel \mathcal{R}^{sp} \parallel \mathsf{K}_0)$. Thus, the underlying secret values remain unchanged, while their confidentiality is now protected under the new password pwd_{new}. Moreover, the complexity of our password update protocol does not depend on the number of accounts associated with $\mathcal{U}_l$, since the login-server-specific secret $\mathcal{R}^{\mathsf{ls}_j}$ is encrypted under K_0, which is independent of pwd.

To convince each $\mathcal{SP}_i$ that the password update request and the new records originate from the legitimate $\mathcal{U}_l$ (the same user who performed the setup for this Uid_l), $\mathcal{U}_l$ authenticates the update using the signing key ssk. Concretely, for each $\mathcal{SP}_i$, $\mathcal{U}_l$ computes a signature sig_i on the tuple $\mathsf{cid}_{\mathsf{new}} \parallel \mathsf{k}_{i,new} \parallel \mathsf{time} \parallel i$ as $\mathsf{sig}_i \leftarrow \mathsf{Sign}_{\mathsf{ssk}}(\mathsf{cid}_{\mathsf{new}} \parallel \mathsf{k}_{i,new} \parallel \mathsf{time} \parallel i)$. The use of the timestamp and the index i prevents replay attacks. $\mathcal{U}_l$ then sends $\langle \mathsf{sig}_i, \mathsf{cid}_{\mathsf{new}}, \mathsf{k}_{i,new} \rangle$ to each $\mathcal{SP}_i$. Upon receiving this tuple, each $\mathcal{SP}_i$ verifies the signature and its freshness using the stored verification key svk. If $\mathsf{SignVerify}_{\mathsf{svk}}(\mathsf{sig}_i, \mathsf{cid}_{\mathsf{new}} \parallel \mathsf{k}_{i,new} \parallel \mathsf{time} \parallel i) = 1$, then $\mathcal{SP}_i$ updates its database entry associated with Uid_l by replacing the old values with $(\mathsf{cid}_{\mathsf{new}}, \mathsf{k}_{i,new})$ and returns OK. Otherwise, it returns FAIL. After a successful execution of $\Pi_{\mathsf{PwdUpdate}}$, the password-protected state stored at the storage providers is updated consistently with pwd_{new}, while the login-server-side account state remains unchanged.

Efficient Design. The key K_0 is reused to encrypt multiple ciphertexts c_j, each containing a distinct high-entropy login-server-specific secret $\mathcal{R}^{\mathsf{ls}_j}$. Since each $\mathcal{R}^{\mathsf{ls}_j}$ is independently random, this reuse does not introduce any security risk that would arise from repeatedly protecting related low-entropy values. Notice that alternatively, if K_0 was to be updated, then all ciphertexts protected under K_0 must be refreshed accordingly. Concretely, $\mathcal{U}_l$ must have reconstructed each stored $\mathcal{R}^{\mathsf{ls}_j}$, derived a fresh key K_0', re-encrypted every corresponding record under K_0', and updated all resulting c_j values at the storage providers. Hence, an K_0 refresh requires a global update across all account-specific records, and its complexity is linear in the number of accounts. Our design choice securely eliminates this complexity, achieving constant cost update per storage server, independent of the number of login servers or the number of accounts of the user.

6 Security

6.1 Preliminaries

In this section, we formally define the game definitions of cryptographic primitives in Section 3 and include the definition of public bluetin board needed for our unavailable reconstruction.

Definition 2. *Receiver-side pseudorandomness (PRF security) This game captures indistinguishability of* TOPRF *outputs from random against a malicious receiver colluding with $< t$ senders.*

Let $C \subseteq \mathsf{S} = \{S_1, S_2, ..., S_n\}$ with $|C| \leq t-1$. Let $R : X \to Y$ be a random function and $b \xleftarrow{\$} \{0,1\}$.

$\mathsf{Exp}^{\mathsf{prf}}_{\mathsf{TOPRF}}(\mathcal{A})$*:*

1. *Sample $(K, \{k_i\}_{i\in[n]}) \leftarrow \mathsf{TOPRF.KeyGen}(1^\lambda, n, t)$; give $\{k_i\}_{i\in C}$ to $\mathcal{A}$.*
2. *$\mathcal{A}$ adaptively issues evaluation queries (x, U) with $|U| \geq t$.*
3. *For each query, the challenger runs the evaluation protocol with $\mathcal{A}$, simulating honest $i \in U \setminus C$ and relaying for corrupted $i \in U \cap C$.*
4. *Let y_{real} be the receiver's output (or $\perp$ if abort).*
5. *Return $y := y_{\text{real}}$ if $b = 0$, and $y := R(x)$ if $b = 1$ (if $y_{\text{real}} = \perp$, return $\perp$).*
6. *$\mathcal{A}$ outputs b'; output 1 iff $b' = b$.*

The advantage $\mathsf{Adv}^{\mathsf{prf}}_{\mathsf{TOPRF}}(\mathcal{A}) = \left| \Pr[\mathsf{Exp}^{\mathsf{prf}}_{\mathsf{TOPRF}}(\mathcal{A}) = 1] - \frac{1}{2} \right|$ *is negligible.*
Usage: *protects derived secrets even under adaptive queries/collusion ($< t$).*

Definition 3. *IND-CPA Security (semantic security [8]) No PPT adversary with adaptive chosen-plaintext (encryption-oracle) access can distinguish encryptions of two equal-length messages except with negligible advantage.*

$\mathsf{Exp}^{\mathsf{ind\text{-}cpa}}_{\mathsf{AE}}(\mathcal{A})$*:*

1. *Sample $\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\lambda)$; give $\mathcal{A}$ oracle access to $\mathcal{O}_{\mathsf{Enc}_{\mathsf{sk}}}(\cdot)$, which returns $\mathsf{Enc}_{\mathsf{sk}}(m)$ on query m.*
2. *$\mathcal{A}$ may make any number of (adaptive) encryption queries.*
3. *$\mathcal{A}$ outputs (m_0, m_1) with $|m_0| = |m_1|$. Sample $b \xleftarrow{\$} \{0,1\}$ and compute $c^\star \leftarrow \mathsf{Enc}_{\mathsf{sk}}(m_b)$; return $c^\star$ to $\mathcal{A}$.*
4. *$\mathcal{A}$ continues to query $\mathcal{O}_{\mathsf{Enc}_{\mathsf{sk}}}(\cdot)$ adaptively (no decryption oracle is available).*
5. *$\mathcal{A}$ outputs b'. Output 1 iff $b' = b$.*

The advantage is

$$\mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{AE}}(\mathcal{A}) = \left| \Pr\left[\mathsf{Exp}^{\mathsf{ind\text{-}cpa}}_{\mathsf{AE}}(\mathcal{A}) = 1\right] - \frac{1}{2} \right|.$$

We say AE *is* IND-CPA secure *if* $\mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{AE}}(\mathcal{A}) \leq \mathsf{neg}(\lambda)$ *for all PPT $\mathcal{A}$.*

Together, these properties imply IND-CCA security and model protection against a static malicious adversary controlling the communication channel.

6.2 Security Proof

The security proof adopts the ideal/real-world methodology. In the *ideal world*, a simulator Sim interacts with the functionality $\mathcal{F}_{\mathsf{UpSPA}}$ and in the *real world*, it interacts with the actual protocol Π and an adversary $\mathcal{A}$ corrupting $(t_{sp} - 1)$ storage providers, $n_{ls} - 1$ many login servers and $n_u - 1$ many users. The goal is to show that, for every PPT adversary, the view produced in the real world is computationally indistinguishable from the view produced in the ideal world. The argument proceeds via a short sequence of hybrid games (G). Without loss of generality, the simulator Sim simulates honest parties which are one of the users $\mathcal{U}$ with Uid_l, $n_{sp} - t_{sp} + 1$-many $\mathcal{SP}_i$, and one of the login servers with ls_j in the real world whereas in the ideal world, Sim simulates $(t_{sp} - 1)$ storage providers, $n_{ls} - 1$ many login servers and $n_u - 1$ many users. Sim behaves as follows in each phase and stores all the generated and received information in its local database.

Setup Phase: Sim:

1. Upon receiving $\langle \mathtt{Setup}, \mathsf{Uid}^o_l \rangle$ from $\mathcal{F}_{\mathsf{UpSPA}}$ in the ideal world,

(a) generates: 1) a PRF key $\tilde{\mathsf{K}}$ and n_{sp} key shares of $\tilde{\mathsf{K}}$ as $\{\tilde{\mathsf{k}}_i\}_{i\in[n_{sp}]}$ by running TOPRF key generation function as $\{\tilde{\mathsf{K}}, \{\tilde{\mathsf{k}}_i\}_{i\in[n_{sp}]}\} \leftarrow \mathsf{TOPRFKeyGen}\ (1^\lambda, n_{sp}, t_{sp})$.; and 2) a signing key and a verification key $\tilde{\mathsf{ssk}}, \tilde{\mathsf{svk}} \leftarrow \mathsf{SignKeyGen}(1^\lambda)$.
(b) assigns $\tilde{\sigma} = \tilde{\mathsf{svk}}$.
(c) samples an encryption key $\tilde{\mathsf{sk}}$ to compute: $\tilde{\mathsf{cid}} \leftarrow \mathsf{Enc}_{\tilde{\mathsf{sk}}}(\{0,1\}^{\lambda+|\tilde{\mathsf{ssk}}|})$.
(d) sends $t_{sp}-1$ many $\langle \mathsf{Uid}_l^o, (\tilde{\sigma}, \tilde{\mathsf{cid}}, \tilde{\mathsf{k}}_i)\rangle$ for each corrupted $\mathcal{SP}_i$ to $\mathcal{A}$ and forwards the response (OK / Fail) of $\mathcal{A}$ to $\mathcal{F}_{\mathsf{UpSPA}}$.
(e) For each corrupted $\mathcal{U}$ with Uid_l^c, receives $\langle \mathsf{Uid}_l^c, (\sigma_l^c, \mathsf{cid}^c, \mathsf{k}_i^c)\rangle$ for each honest $\mathcal{SP}$ from $\mathcal{A}$.
(f) Sends $\langle \mathtt{Setup}, \mathsf{Uid}_l^c\rangle$ to $\mathcal{F}_{\mathsf{UpSPA}}$.

2. If Uid_l^c exists, $\mathcal{F}_{\mathsf{UpSPA}}$ sends Fail to Sim. Otherwise, $\mathcal{F}_{\mathsf{UpSPA}}$ sends OK.
3. If received OK from $\mathcal{F}_{\mathsf{UpSPA}}$, stores $\langle \mathsf{Uid}_l^c, \sigma_l^c, \mathsf{cid}^c, \{\mathsf{k}_i^c\}_{i\in[n_{sp}+t_{sp}-1]}\rangle$ and sends OK to $\mathcal{A}$. Otherwise, sends FAIL to $\mathcal{A}$.

Registration phase:

1. Sim **Behavior for** Uid_l^o. The registration request for the honest user Uid_l^o is carried out with the honest $\mathcal{LS}$ or any one of the malicious login servers corrupted by $\mathcal{A}$. The only difference in the simulation would be whether to send a simulated $\tilde{\mathsf{vInfo}}_j$ to $\mathcal{A}$. Sim receives $\langle \mathtt{Register}, \mathsf{Uid}_l^o\rangle$ from $\mathcal{F}_{\mathsf{UpSPA}}$ and sends Uid_l^o to $\mathcal{A}$. Then Sim:
   (a) If $\mathcal{A}$ sends FAIL for any one of the corrupted $\mathcal{SP}_i$, responses FAIL to $\mathcal{F}_{\mathsf{UpSPA}}$. Otherwise, Sim computes TOPRF with $\mathcal{A}$ using a random value as a simulated password $\tilde{\mathsf{pwd}}$ as $\mathsf{Sim}[F_{\mathsf{K}^c}(\tilde{\mathsf{pwd}})] \leftarrow \mathsf{TOPRF}\big(\mathsf{Sim}[\tilde{\mathsf{pwd}}]\ , \{\mathcal{SP}_i[\mathsf{k}_i^c]\}_{i\in[n_{sp}]}\big)$.
   (b) Receives cid^c from $\mathcal{A}$ for each corrupted $\mathcal{SP}_i$.
   (c) Performs two checks: 1) if $F_{\mathsf{K}^c}(\tilde{\mathsf{pwd}}) = F_{\tilde{\mathsf{K}}}(\tilde{\mathsf{pwd}})$ since Sim can easily compute $F_{\tilde{\mathsf{K}}}(\tilde{\mathsf{pwd}})$ locally; and 2) Received cid^c from $\mathcal{A}$ are the same with the associated $\tilde{\mathsf{cid}}$ in its database. If any of these checks fail, Sim aborts by sending FAIL to $\mathcal{A}$ and $\mathcal{F}_{\mathsf{UpSPA}}$.
   (d) Otherwise, Sim sends OK to $\mathcal{F}_{\mathsf{UpSPA}}$ and receives $\langle \mathtt{Register}, \mathsf{SUid}_{l,i,j}\rangle$ for each $\mathcal{SP}_i$ that it simulates in the ideal world. Then, Sim computes $\tilde{\mathsf{vInfo}}_j$ based on the simulated value $\tilde{\mathcal{R}}^{\mathsf{ls}_j}$ as $\tilde{\mathsf{vInfo}}_j = \mathsf{H}(\tilde{\mathcal{R}}^{\mathsf{ls}_j}||\mathsf{ls}_j^c)$ for the corrupted $\mathcal{LS}$ and $\tilde{\mathsf{SUid}}_{l,i,j} \leftarrow \{0,1\}^\lambda$ for each corrupted $\mathcal{SP}_i$. **Remark:** Note that alternatively Sim could have created the mix of $\mathsf{SUid}_{l,i,j}$ received from $\mathcal{F}_{\mathsf{UpSPA}}$ and its simulated $\tilde{\mathsf{SUid}}_{l,i,j}$ for $\mathcal{A}$.
   (e) Computes $\tilde{\mathsf{c}}_j \leftarrow \mathsf{Enc}_{\tilde{\mathsf{sk}}}(\{0,1\}^\lambda)$ and sends $(\tilde{\mathsf{SUid}}_{l,i,j}, \tilde{\mathsf{c}_j})$ to $\mathcal{A}$ for each corrupted $\mathcal{SP}_i$.
   (f) If receives Fail from $\mathcal{A}$, sends FAIL to $\mathcal{F}_{\mathsf{UpSPA}}$ and aborts the registration in the real world; else, sends OK to $\mathcal{F}_{\mathsf{UpSPA}}$.
   (g) If receives OK and the registration request is for any malicious $\mathcal{LS}$, Sim sends $\langle \mathsf{Uid}_{l,j}, \tilde{\mathsf{vInfo}}_j\rangle$ to $\mathcal{A}$. If $\mathcal{A}$ replies with OK, Sim sends OK to $\mathcal{F}_{\mathsf{UpSPA}}$. Otherwise, aborts in both worlds.
2. Sim **Behavior for** Uid_l^c. $\mathcal{A}$ as a malicious user with Uid_l^c requests to register either for an honest $\mathcal{LS}$ with ls_j^o simulated by Sim or for a malicious $\mathcal{LS}$ with ls_j^c corrupted by $\mathcal{A}$. Sim receives Uid_l^c from $\mathcal{A}$. If Uid_l^c does not exist, Sim aborts. Otherwise, for each corrupted login server ls_j^c, Sim sends $\langle \mathtt{Register}, \mathsf{Uid}_l, \mathsf{ls}_j^c\rangle$ to $\mathcal{F}_{\mathsf{UpSPA}}$. If receives FAIL, Sim sends FAIL to $\mathcal{F}_{\mathsf{UpSPA}}$ and aborts registration in the real world. Otherwise, Sim sends OK to $\mathcal{F}_{\mathsf{UpSPA}}$ and $\mathcal{A}$ and receives $\langle \mathtt{Register}, \mathsf{SUid}_{l,i,j}\rangle$ for each $\mathcal{SP}_i$ it simulates from $\mathcal{F}_{\mathsf{UpSPA}}$.
   (a) Sim executes TOPRF with $\mathcal{A}$ using the k_i^c sent by $\mathcal{A}$ in the setup phase as $\mathcal{A}[F_{\mathsf{K}^c}(\mathsf{pwd}^c)] \leftarrow \mathsf{TOPRF}\big(\mathcal{A}[\mathsf{pwd}^c], \{\mathsf{Sim}[\mathsf{k}_i^c]\}_{i\in[n_{sp}-t_{sp}+1]} \cup \mathcal{A}[\mathsf{k}_i^c]\}_{i\in[t_{sp}-1]}\big)$ and sends cid^c to $\mathcal{A}$.
   (b) Sim receives $\langle \{\mathsf{SUid}_{i,j}^c, \mathsf{c}_j^c\}_{i\in[n_{sp}-t_{sp}+1]}\rangle$ from $\mathcal{A}$ and sends OK to $\mathcal{A}$.
   (c) If $\mathcal{A}$ has requested to register for the honest login server ls_j^o, Sim also receives $\mathsf{Uid}_{l,j}^c, \mathsf{vInfo}_j^c$ from $\mathcal{A}$ and $\mathtt{Register}, \mathsf{Uid}_{l,j}$ from $\mathcal{F}_{\mathsf{UpSPA}}$, Sim sends OK to both $\mathcal{A}$ and $\mathcal{F}_{\mathsf{UpSPA}}$.

Authentication phase:

1. Sim **Behavior for** Uid_l^o. The authentication request for the honest user Uid_l^o could be to the honest $\mathcal{LS}$ or any one of the malicious login servers corrupted by $\mathcal{A}$. The only difference between these two sub-cases in the simulation would be whether to send the simulated $\tilde{\mathsf{vInfo}}_j$ to $\mathcal{A}$ since in the later one it will send $\tilde{\mathsf{vInfo}}_j$ to $\mathcal{A}$. Sim receives $\langle \mathtt{Authenticate}, \mathsf{Uid}_l^o \rangle$ from $\mathcal{F}_{\mathsf{UpSPA}}$ and sends Uid_l^o to $\mathcal{A}$. Then, Sim:
   (a) If $\mathcal{A}$ sends FAIL for any one of the corrupted $\mathcal{SP}_i$, responses FAIL to $\mathcal{F}_{\mathsf{UpSPA}}$. Otherwise, Sim computes TOPRF with $\mathcal{A}$ using the simulated password $\tilde{\mathsf{pwd}}$ as:
   $$\mathsf{Sim}[F_{\mathsf{K}^c}(\tilde{\mathsf{pwd}})] \leftarrow \mathsf{TOPRF}\big(\mathsf{Sim}[\tilde{\mathsf{pwd}}], \{\mathcal{SP}_i[\mathsf{k}_i^c]\}_{i\in[n_{sp}]}\big)$$
   (b) Receives cid^c from $\mathcal{A}$ for each corrupted $\mathcal{SP}_i$.
   (c) Sim performs two checks: 1) if $F_{\mathsf{K}^c}(\tilde{\mathsf{pwd}}) = F_{\tilde{\mathsf{K}}}(\tilde{\mathsf{pwd}})$ since Sim can easily compute $F_{\tilde{\mathsf{K}}}(\tilde{\mathsf{pwd}})$ locally; and 2) Received cid^c from $\mathcal{A}$ are the same with the associated $\tilde{\mathsf{cid}}$ in its database. If any of these checks fail, Sim aborts by sending FAIL to $\mathcal{A}$ and $\mathcal{F}_{\mathsf{UpSPA}}$.
   (d) Otherwise, sends OK to $\mathcal{F}_{\mathsf{UpSPA}}$ and receives $\langle \mathtt{Authenticate}, \mathsf{SUid}_{l,i,j} \rangle$ for each $\mathcal{SP}_i$ it simulates in the ideal world.
   (e) Sends $\tilde{\mathsf{SUid}}_{i,j}^c$ for each corresponding corrupted $\mathcal{SP}_i$ to $\mathcal{A}$ and receives $\tilde{\mathsf{c}}_j$ from $\mathcal{A}$.
   (f) Checks if received $\tilde{\mathsf{c}}_j$ are consistent with each other and with the ones stored in its database.
   (g) For the honest ls_j^o, Sim does not do anything except sending OK to $\mathcal{F}_{\mathsf{UpSPA}}$. For any of the malicious login servers ls_j^c,Sim receives
   $\langle \mathtt{Authenticate}, \mathsf{Uid}_{l,j} \rangle$ from $\mathcal{F}_{\mathsf{UpSPA}}$ and sends $\mathsf{Uid}_{l,j}^o, \tilde{\mathsf{vInfo}}_j$ to $\mathcal{A}$. If $\mathcal{A}$ sends OK for ls_j^c, Sim responses OK to $\mathcal{F}_{\mathsf{UpSPA}}$. Otherwise, Sim sends FAIL.
2. Sim **Behavior for** Uid_l^c. $\mathcal{A}$ as a malicious user with Uid_l^c requests to authenticate with either an honest $\mathcal{LS}$ with ls_j^o simulated by Sim or a malicious $\mathcal{LS}$ with ls_j^c corrupted by $\mathcal{A}$. Sim receives Uid_l^c from $\mathcal{A}$. If Uid_l^c does not exist, Sim aborts. Otherwise, for each corrupted login server ls_j^c, Sim sends $\langle \mathtt{Register}, \mathsf{Uid}_l, \mathsf{ls}_j^c \rangle$ to $\mathcal{F}_{\mathsf{UpSPA}}$. If receives FAIL, Sim sends FAIL to $\mathcal{F}_{\mathsf{UpSPA}}$ and aborts registration in the real world. Otherwise, Sim sends OK to $\mathcal{F}_{\mathsf{UpSPA}}$ and $\mathcal{A}$ and receives $\langle \mathtt{Authenticate}, \mathsf{SUid}_{l,i,j} \rangle$ for each $\mathcal{SP}_i$ it simulates from $\mathcal{F}_{\mathsf{UpSPA}}$.
   (a) Sim computes TOPRF with $\mathcal{A}$ using k_i^c sent by $\mathcal{A}$ in the set up phase as:
   $$\mathcal{A}[F_{\mathsf{K}^c}(\mathsf{pwd}^c)] \leftarrow \mathsf{TOPRF}\big(\mathcal{A}[\mathsf{pwd}^c], \{\mathsf{Sim}[\mathsf{k}_i^c]\}_{i\in[n_{sp}t_{sp}+1]} \cup \mathcal{A}[\mathsf{k}_i^c]\}_{i\in[t_{sp}-1]}\big)$$
   and sends cid^c to $\mathcal{A}$.
   (b) Sim receives $\langle \{\mathsf{SUid}_{i,j}^c, \mathsf{c}_j^c\}_{i\in[n_{sp}-t_{sp}+1]} \rangle$ from $\mathcal{A}$ and sends OK to $\mathcal{A}$.
   (c) If $\mathcal{A}$ has requested to register for the honest login server ls_j^o, Sim also receives $\mathsf{Uid}_{l,j}^c, \mathsf{vInfo}_j^c$ from $\mathcal{A}$ and $\mathtt{Authenticate}, \mathsf{Uid}_{l,j}$ from $\mathcal{F}_{\mathsf{UpSPA}}$, Sim sends OK to both $\mathcal{A}$ and $\mathcal{F}_{\mathsf{UpSPA}}$.

Secret update phase:

1. Sim **Behavior for** Uid_l^o. The secret update request for the honest user Uid_l^o could be to the honest login server ls_j^o or any one of the malicious login servers corrupted ls_j^c by $\mathcal{A}$. Sim receives $\langle \mathtt{SecretUpdate}, \mathsf{Uid}_l^o \rangle$ from $\mathcal{F}_{\mathsf{UpSPA}}$ and sends Uid_l^o to $\mathcal{A}$. Then Sim:
   (a) If receives Fail from $\mathcal{A}$ or Uid_l is not stored in its storage, sends Fail to $\mathcal{F}_{\mathsf{UpSPA}}$ and aborts in the real world. Otherwise, Sim sends OK to $\mathcal{F}_{\mathsf{UpSPA}}$ and receives $\langle \mathtt{SecretUpdate}, \mathsf{SUid}_{l,i,j} \rangle$ from $\mathcal{F}_{\mathsf{UpSPA}}$.
   (b) Then, computes TOPRF with $\mathcal{A}$ using another simulated random (password) $\tilde{\mathsf{pwd}}$' as:
   $$\mathsf{Sim}[F_{\mathsf{K}^c}(\tilde{\mathsf{pwd}}')] \leftarrow \mathsf{TOPRF}\big(\mathsf{Sim}[\tilde{\mathsf{pwd}}'], \{\mathcal{SP}_i[\mathsf{k}_i^c]\}_{i\in[n_{sp}]}\big).$$
   (c) Receives cid^c from $\mathcal{A}$ for each corrupted $\mathcal{SP}_i$.
   (d) Performs two checks: 1) if $F_{\mathsf{K}^c}(\tilde{\mathsf{pwd}}') = F_{\tilde{\mathsf{K}}}(\tilde{\mathsf{pwd}}')$ since Sim can easily compute $F_{\tilde{\mathsf{K}}}(\tilde{\mathsf{pwd}}')$ locally; and 2) Received cid^c from $\mathcal{A}$ are the same with the associated $\tilde{\mathsf{cid}}$ in its database. If any of these checks fail, Sim aborts by sending FAIL to $\mathcal{A}$ and $\mathcal{F}_{\mathsf{UpSPA}}$.

(e) Otherwise, Sim sends $\tilde{\mathsf{SUid}}^c_{i,j}$ to $\mathcal{A}$ and receives $\tilde{\mathsf{c}}_j$.
(f) Checks if received $\tilde{\mathsf{c}}_j$ are consistent with each other and with the ones stored in its database.
(g) Sends $\mathsf{SUid}_{l,i,j}$ to $\mathcal{A}$; if receives Fail, sends Fail to $\mathcal{F}_{\mathsf{UpSPA}}$ and aborts computes $\mathsf{c}_{j,\tilde{new}} \leftarrow \mathsf{Enc}_{\tilde{\mathsf{sk}}}(1^{\lambda-1}||0)$ and sends $\langle \tilde{\mathsf{SUid}}_{l,i,j}, \mathsf{c}_{j,\tilde{new}} \rangle$ to all $\mathcal{SP}_i$s. sends OK to $\mathcal{F}_{\mathsf{UpSPA}}$.
(h) For the honest ls^o_j, Sim does not do anything except sending OK to $\mathcal{F}_{\mathsf{UpSPA}}$. For any of the malicious login servers ls^c_j, Sim receives
$\langle \mathtt{SecretUpdate}, \mathsf{Uid}_{l,j} \rangle$ from $\mathcal{F}_{\mathsf{UpSPA}}$ and sends $\mathsf{Uid}^o_{l,j}, \tilde{\mathsf{vInfo}}_j$ to $\mathcal{A}$. If $\mathcal{A}$ sends OK for ls^c_j, Sim responses OK to $\mathcal{F}_{\mathsf{UpSPA}}$. Otherwise, Sim sends FAIL.

2. Sim **Behavior for** Uid^c_l. $\mathcal{A}$ as a malicious user with Uid^c_l the secret update request with either an honest $\mathcal{LS}$ with ls^o_j simulated by Sim or a malicious $\mathcal{LS}$ with ls^c_j corrupted by $\mathcal{A}$. Sim receives Uid^c_l from $\mathcal{A}$. If Uid^c_l does not exist, Sim aborts. Otherwise, for each corrupted login server ls^c_j, Sim sends $\langle \mathtt{SecretUpdate}, \mathsf{Uid}_l, \mathsf{ls}^c_j \rangle$ to $\mathcal{F}_{\mathsf{UpSPA}}$. If receives FAIL, Sim sends FAIL to $\mathcal{F}_{\mathsf{UpSPA}}$ and aborts secret update in the real world. Otherwise, Sim sends OK to $\mathcal{A}$ and receives $\langle \mathtt{SecretUpdate}, \mathsf{SUid}_{l,i,j} \rangle$ from $\mathcal{F}_{\mathsf{UpSPA}}$ for each $\mathcal{SP}_i$ it simulates.
(a) Sim computes TOPRF with $\mathcal{A}$ using k^c_i sent by $\mathcal{A}$ in the set up phase as:

$$\mathcal{A}[F_{\mathsf{K}^c}(\mathsf{pwd}^c)] \leftarrow \mathsf{TOPRF}\big(\mathcal{A}[\mathsf{pwd}^c], \{\mathsf{Sim}[\mathsf{k}^c_i]\}_{i\in[n_{sp}-t_{sp}+1]} \cup \mathcal{A}[\mathsf{k}^c_i]\}_{i\in[t_{sp}-1]}\big)$$

and sends the corresponding cid^c to $\mathcal{A}$.
(b) Sim receives $\mathsf{SUid}^c_{i,j}$ for each honest $\mathcal{SP}_i$ and sends c^c_j to $\mathcal{A}$ if $\mathsf{SUid}^c_{i,j}$ matches its database. **Remark:** Note that Sim checks each $\mathsf{SUid}^c_{i,j}$ to its corresponding $\mathcal{SP}_i$ that Sim is simulating and decides whether to send c^c_j or ABORT (if $\mathsf{SUid}^c_{i,j}$ does not match with its database).
(c) Sim receives $\mathsf{c}^c_{j,new}$ from $\mathcal{A}$ and sends OK to $\mathcal{A}$ and $\mathcal{F}_{\mathsf{UpSPA}}$. Furthermore, for the honest login server ls^o_j, Sim also receives $\langle \mathsf{Uid}^c_{l,j}, \mathsf{vInfo}^c_j, \mathsf{vInfo}^c_{j,new} \rangle$ from $\mathcal{A}$. Sim checks if vInfo^c_j associated with its login information of $\mathsf{Uid}^c_{l,j}$. If it matches, Sim sends OK to $\mathcal{F}_{\mathsf{UpSPA}}$ and $\mathcal{A}$ for the honest storage providers Sim simulates. Otherwise, Sim sends FAIL. Sim also stores the new values sent by $\mathcal{A}$ with historical data.

Password update phase: Sim

1. Sim **Behavior for** Uid^o_l. The password update request is for the honest user Uid^o_l. Sim receives $\langle \mathtt{PwdUpdate}, \mathsf{Uid}_l \rangle$ from $\mathcal{F}_{\mathsf{UpSPA}}$ and sends Uid^o_l to $\mathcal{A}$. Then, Sim:
(a) If $\mathcal{A}$ replies with FAIL, Sim aborts and sends FAIL to $\mathcal{F}_{\mathsf{UpSPA}}$. Otherwise, Sim If Sim receives Fail from $\mathcal{A}$ or Uid_l is not stored in its storage, sends Fail to $\mathcal{F}_{\mathsf{UpSPA}}$ and aborts in the real world. Otherwise, Sim sends OK to $\mathcal{F}_{\mathsf{UpSPA}}$ and receives $\langle \mathtt{SecretUpdate}, \mathsf{SUid}_{l,i,j} \rangle$ from $\mathcal{F}_{\mathsf{UpSPA}}$.
(b) Sim computes TOPRF with $\mathcal{A}$ another simulated random (password) $\tilde{\mathsf{pwd}}''$ as $\mathsf{Sim}[F_{\mathsf{K}^c}(\tilde{\mathsf{pwd}}'')] \leftarrow \mathsf{TOPRF}\big(\mathsf{Sim}[\tilde{\mathsf{pwd}}''], \{\mathsf{Sim}[\tilde{\mathsf{k}}_i]\}_{i\in[n_{sp}-t_{sp}+1]} \cup \mathcal{A}[\mathsf{k}^c_i]_{i\in[t_{sp}-1]}\}\big)$.
(c) Receives cid^c from $\mathcal{A}$ for each corrupted $\mathcal{SP}_i$.
(d) Sim performs two checks: 1) if $F_{\mathsf{K}^c}(\tilde{\mathsf{pwd}}'') = F_{\tilde{\mathsf{K}}}(\tilde{\mathsf{pwd}}'')$ since Sim can easily compute $F_{\tilde{\mathsf{K}}}(\tilde{\mathsf{pwd}}'')$ locally; and 2) Received cid^c from $\mathcal{A}$ are the same with the associated $\tilde{\mathsf{cid}}$ in its database. If any of these checks fail, Sim aborts by sending FAIL to $\mathcal{A}$ and $\mathcal{F}_{\mathsf{UpSPA}}$.
(e) Generates a set of TOPRF keys $\tilde{\mathsf{k}}_{i,new}$ and $\tilde{\mathsf{K}}_{new}$ as $\tilde{\mathsf{K}}_{new}, \{\tilde{\mathsf{k}}_{i,new}\}_{i\in[n_{sp}]} \leftarrow \mathsf{TOPRFKeyGen}(1^\lambda, n_{sp}, t_{sp})$ and a new encryption key $\tilde{\mathsf{sk}}_{new}$ $\tilde{\mathsf{sk}}_{new} \leftarrow \mathsf{KeyGen}(1^\lambda)$.
(f) Generates $\tilde{\mathsf{cid}}_{new}$ as $\tilde{\mathsf{cid}}_{new} \leftarrow \mathsf{Enc}_{\tilde{\mathsf{sk}}_{new}}(\{0,1\}^{\lambda+|\tilde{\mathsf{ssk}}|})$
(g) Computes a signature using $\tilde{\mathsf{ssk}}$ stored in its database as $\tilde{\mathsf{sig}}_i \leftarrow \mathsf{Sign}(\tilde{\mathsf{ssk}}, \tilde{\mathsf{cid}}_{new}||\tilde{\mathsf{k}}_{i,new}||\mathsf{timestamp})$ for each corrupted $\mathcal{SP}_i$ and sends each corresponding tuple of $\langle \tilde{\mathsf{sig}}_i, \tilde{\mathsf{cid}}_{new}, \tilde{\mathsf{k}}_{i,new} \rangle$ to $\mathcal{A}$.
(h) Behalf of the simulated $\mathcal{SP}_i$ in the ideal world, sends OK to $\mathcal{F}_{\mathsf{UpSPA}}$ if $\mathcal{A}$ responses OK. Otherwise, sends FAIL to $\mathcal{F}_{\mathsf{UpSPA}}$.

2. Sim **Behavior for** Uid_l^c. $\mathcal{A}$ as a malicious user with Uid_l^c requests the password update. Sim receives Uid_l^c from $\mathcal{A}$. If Uid_l^c does not exist, Sim aborts. Otherwise, Sim sends $\langle \texttt{PwdUpdate}, \mathsf{Uid}_l^c \rangle$ to $\mathcal{F}_{\mathsf{UpSPA}}$ as a user it simulates:
   (a) Sim receives $\langle \texttt{PwdUpdate}, \mathsf{Uid}_l^c \rangle$ for each corrupted $\mathcal{SP}_i$.
   (b) Sim computes TOPRF with $\mathcal{A}$ using the k_i^c associated with k_i^c as:

$$\mathcal{A}[F_{\mathsf{K}^c}(\mathsf{pwd}^c)] \leftarrow \mathsf{TOPRF}\big(\mathcal{A}[\mathsf{pwd}^c], \{\mathsf{Sim}[\mathsf{k}_i^c]\}_{i\in[n_{sp}-t_{sp}+1]} \cup \mathcal{A}[\mathsf{k}_i^c]\}_{i\in[t_{sp}-1]}\big)$$

   and sends the corresponding cid^c to $\mathcal{A}$.
   (c) Sim receives $\langle \mathsf{sig}_i^c, \mathsf{cid}_{new}^c, \mathsf{k}_{i,new}^c \rangle$.
   (d) Sim checks if sig_i^c is a valid signature by running $\mathsf{SignVerify}(\mathsf{svk}_l^c, \mathsf{sig}_i^c, \mathsf{cid}_{new}^c \ ||\mathsf{k}_{i,new}^c||\mathsf{timestamp})$ where $\mathsf{svk}_l^c = \sigma_l^c$. If the verification is successful, Sim sends OK to $\mathcal{A}$ and $\mathcal{F}_{\mathsf{UpSPA}}$ and receives OK from $\mathcal{F}_{\mathsf{UpSPA}}$ (as a user simulating Uid_l^c in the ideal world). Otherwise, Sim sends FAIL to both $\mathcal{A}$ and $\mathcal{F}_{\mathsf{UpSPA}}$ and receives FAIL from $\mathcal{F}_{\mathsf{UpSPA}}$.

Sim stores the new updated values as well as the previous versions in its database.

Claim. The view of $\mathcal{A}$ in its interaction with Sim is indistinguishable from the view in its interaction with the real-world honest parties.

For each game G_i, let $p_i := \Pr[\mathcal{A}$ outputs 1 in G For each adjacent game (G_i, G_{i+1}), define $\Delta_i := |p_i - p_{i+1}|$. We upper-bound each Δ_i by the advantage of an adversary $\mathcal{B}$ in the corresponding underlying security experiment, and then invoke the assumed security (negligible advantage).

Game G_0 (Real execution). G_0 is the standard real-world execution of UpSPA with $\mathcal{A}$ controlling the corrupted users, corrupted login servers, corrupted storage providers, and the network scheduling/delivery.

Game G_1: Replace TOPRF outputs by a random function
In G_1, we modify the execution as follows. Let $R : X \to Y$ be a truly random function sampled once at the beginning of the experiment. For every TOPRF evaluation session in $\mathcal{T}$ where $\mathsf{Uid}^\star$ is the receiver with input $\mathsf{pwd}^\star$, instead of using the receiver's real output $F_{\mathsf{K}}(\mathsf{pwd}^\star)$, we define the session output (i.e., the value used as $F_{\mathsf{K}}(\mathsf{pwd}^\star)$ in the protocol) to be $R(\mathsf{pwd}^\star)$ (and output $\perp$ if the real protocol would output $\perp$). In particular, whenever the honest user uses $F_{\mathsf{K}}(\mathsf{pwd}^\star)$ as the symmetric key for Enc in the protocol (namely, in the creation of cid in Setup and in later decryptions of cid), we use the replaced value $R(\mathsf{pwd}^\star)$ consistently. The transcript of TOPRF messages sent/received by corrupted storage providers is generated exactly as in G_0.

Lemma 1. *TOPRF outputs are computationally indistinguishable from random to any PPT adversary colluding with fewer than t_{sp} senders. Formally, letting p_0 and p_1 denote the probabilities that $\mathcal{A}$ outputs 1 when given a real TOPRF output and a uniformly random value, respectively, we have $\Delta_0 = |p_0 - p_1| \leq \mathsf{neg}(\lambda)$.*

Proof. We build a reduction $\mathcal{B}_{\mathsf{prf}}$ to $\mathsf{Exp}^{\mathsf{prf}}_{\mathsf{TOPRF}}$ (Definition 2.3). The reduction $\mathcal{B}_{\mathsf{prf}}$ internally runs $\mathcal{A}$ and simulates the full protocol execution for all parties, with the following embedding:

1. $\mathcal{B}_{\mathsf{prf}}$ forwards the set of corrupted senders C (the $t_{sp}-1$ corrupted storage providers) to the TOPRF challenger and receives the corresponding key shares $\{k_i\}_{i\in C}$. It uses these shares as the internal stored values of the corrupted $\mathcal{SP}_i$ that are controlled by $\mathcal{A}$ in the simulated execution. For each honest $\mathcal{SP}_i \notin C$, $\mathcal{B}_{\mathsf{prf}}$ simulates the honest sender behavior locally as prescribed by the experiment, while for each corrupted $\mathcal{SP}_i \in C$ it relays $\mathcal{A}$'s sender-side messages.

2. Whenever the honest user $\mathsf{Uid}^\star$ needs a TOPRF evaluation on input $\mathsf{pwd}^\star$ (Registration/Authentication/Secret Update/Password Update), $\mathcal{B}_{\mathsf{prf}}$ defines the receiver's query set U to be the set of participating storage providers in that protocol invocation (so $|U| \geq t_{sp}$ and U contains the corrupted senders in C that participate in that session), and submits the evaluation query $(x = \mathsf{pwd}^\star, U)$ to the challenger. During this evaluation, $\mathcal{B}_{\mathsf{prf}}$ runs the TOPRF evaluation protocol between the (simulated) honest receiver $\mathsf{Uid}^\star$ and the senders in U as follows: it forwards the *receiver-side TOPRF messages* (i.e., the protocol messages computed by the receiver on input $\mathsf{pwd}^\star$, which can be viewed as the *blinded* $\mathsf{pwd}^\star$ messages) to the corrupted senders controlled by $\mathcal{A}$, and it forwards the *sender-side replies / partial evaluations* produced by the corrupted senders back to the challenger, while simulating the honest senders in $U \setminus C$ locally. At the end of the evaluation, the challenger samples a random bit $b \leftarrow \{0,1\}$ and returns an output y to $\mathcal{B}_{\mathsf{prf}}$, where y is distributed as either $y_{\text{real}} = F_K(\mathsf{pwd}^\star)$ (if $b = 0$) or $y_{\text{rand}} = R(\mathsf{pwd}^\star)$ (if $b = 1$).
3. $\mathcal{B}_{\mathsf{prf}}$ uses the received value y as the derived key $F_{\mathsf{K}}(\mathsf{pwd}^\star)$ within the protocol execution at that point (in particular, as the symmetric key for $\mathsf{Enc}/\mathsf{Dec}$ on cid whenever the protocol uses $F_{\mathsf{K}}(\mathsf{pwd}^\star)$), and continues the simulation exactly according to the protocol specification. Consequently, $\mathcal{A}$ observes only the induced protocol view (ciphertexts, accept/reject outcomes, and any messages delivered to corrupted parties). Finally, $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$ based on this view, and $\mathcal{B}_{\mathsf{prf}}$ outputs the same bit b' to the challenger (i.e., $\mathcal{B}_{\mathsf{prf}}$ wins iff $b' = b$ in the experiment).

If the challenger samples $b = 0$, then $\mathcal{B}_{\mathsf{prf}}$ produces a view distributed exactly as G_0; if $b = 1$, it produces a view distributed exactly as G_1. Therefore, any distinguisher between G_0 and G_1 yields a distinguisher for Definition 2 with the same advantage, implying $\Delta_0 \leq \mathsf{Adv}^{\mathsf{prf}}_{\mathsf{TOPRF}}(\mathcal{B}_{\mathsf{prf}}) \leq \mathsf{neg}(\lambda)$. □

consistent). **Game G_2: Replace honest encryptions of type cid by encryptions of random strings** In G_2, we modify every honest encryption call $E^{\mathsf{id}}_r = (\mathsf{sk}_r, m^{\mathsf{id}}_r) \in \mathcal{E}_{\mathsf{id}}$ as follows: instead of encrypting m^{id}_r, the honest party encrypts a uniformly random string $\widetilde{m}^{\mathsf{id}}_r \leftarrow \{0,1\}^{|m^{\mathsf{id}}_r|}$ of the same length under the same key. Concretely, each ciphertext $\mathsf{cid}_r \leftarrow \mathsf{Enc}_{\mathsf{sk}_r}(m^{\mathsf{id}}_r)$ is replaced by $\widetilde{\mathsf{cid}}_r \leftarrow \mathsf{Enc}_{\mathsf{sk}_r}(\widetilde{m}^{\mathsf{id}}_r)$. The locally recorded values are updated accordingly: the honest party keeps the same locally recorded plaintext value m^{id}_r for subsequent computations, and only records that the new ciphertext $\widetilde{\mathsf{cid}}_r$ is associated with that locally recorded plaintext value. Note that the plaintext length $|m^{\mathsf{id}}_r|$ is determined by the corresponding protocol instance producing a ciphertext of type cid.

Lemma 2. *Assuming* AE *is IND-CPA secure, the modification in Game G_2 changes the adversary's view only negligibly. Formally, letting p_1 and p_2 denote the probabilities that $\mathcal{A}$ outputs 1 in Games G_1 and G_2, respectively, we have $\Delta_1 = |p_1 - p_2| \leq \mathsf{neg}(\lambda)$, for all encryption calls in $\mathcal{E}_{\mathsf{id}}$ in which the encrypted plaintext is independent of the corresponding encryption key.*

Proof. We apply a standard hybrid argument over the ordered list of encryption calls $\mathcal{E}_{\mathsf{id}}$. For $0 \leq i \leq q_{\mathsf{id}}$, define hybrid $H_{2,i}$ as the execution in which the first i encryption calls $E^{\mathsf{id}}_1, \ldots, E^{\mathsf{id}}_i$ encrypt random strings and the remaining calls $E^{\mathsf{id}}_{i+1}, \ldots, E^{\mathsf{id}}_{q_{\mathsf{id}}}$ encrypt the real protocol plaintexts (as in G_1). Then $H_{2,0} = G_1$ and $H_{2,q_{\mathsf{id}}} = G_2$. Suppose $\mathcal{A}$ distinguishes $H_{2,0}$ and $H_{2,q_{\mathsf{id}}}$ with advantage ϵ. By averaging, there exists an index $i^\star$ such that $\mathcal{A}$ distinguishes $H_{2,i^\star}$ and $H_{2,i^\star+1}$ with advantage at least ϵ/q_{id}. We construct an IND-CPA adversary $\mathcal{B}_{\mathsf{cpa}}$ that breaks AE.

Reduction $\mathcal{B}_{\mathsf{cpa}}$. $\mathcal{B}_{\mathsf{cpa}}$ guesses $i^\star$ uniformly in $[0, q_{\mathsf{id}} - 1]$ and simulates the entire execution for $\mathcal{A}$. Let $E^{\mathsf{id}}_{i^\star+1} = (\mathsf{sk}^\star, m^\star)$ be the $(i^\star + 1)$-st encryption call. $\mathcal{B}_{\mathsf{cpa}}$ embeds the IND-CPA challenger as the encryption oracle under key $\mathsf{sk}^\star$:

1. $\mathcal{B}_{\mathsf{cpa}}$ internally runs $\mathcal{A}$ and simulates the full execution for $\mathcal{A}$. It guesses uniformly at random an index $i^\star \in \{0, \ldots, q_{\mathsf{id}} - 1\}$, where $i^\star$ is intended to be the hybrid index for which $H_{2,i^\star}$ and $H_{2,i^\star+1}$ are distinguished. Let $E^{\mathsf{id}}_{i^\star+1} = (\mathsf{sk}^\star, m^\star)$ be the $(i^\star + 1)$-st encryption call in $\mathcal{E}_{\mathsf{id}}$.

2. $\mathcal{B}_{\mathsf{cpa}}$ embeds the IND-CPA challenger as an encryption oracle under the key $\mathsf{sk}^\star$. Concretely, for every encryption call $E_r^{\mathsf{id}} = (\mathsf{sk}_r, m_r^{\mathsf{id}}) \in \mathcal{E}_{\mathsf{id}}$ with $\mathsf{sk}_r \neq \mathsf{sk}^\star$, $\mathcal{B}_{\mathsf{cpa}}$ samples sk_r honestly and computes the ciphertext locally according to the corresponding hybrid. For every encryption call with $\mathsf{sk}_r = \mathsf{sk}^\star$, $\mathcal{B}_{\mathsf{cpa}}$ answers it using its oracle access to $\mathcal{O}_{\mathsf{Enc}_{\mathsf{sk}^\star}}(\cdot)$, except for the single challenge call $E_{i^\star+1}^{\mathsf{id}}$ described below.
3. For each encryption call $E_r^{\mathsf{id}} = (\mathsf{sk}_r, m_r^{\mathsf{id}})$ with $r \leq i^\star$, the simulated execution should follow $H_{2,i^\star+1}$ on that call, that is, encrypt a uniformly random string of the same length. Accordingly, if $\mathsf{sk}_r = \mathsf{sk}^\star$ then $\mathcal{B}_{\mathsf{cpa}}$ samples $\widetilde{m}_r^{\mathsf{id}} \xleftarrow{\$} \{0,1\}^{|m_r^{\mathsf{id}}|}$ and queries $\mathcal{O}_{\mathsf{Enc}_{\mathsf{sk}^\star}}(\widetilde{m}_r^{\mathsf{id}})$ to obtain the ciphertext used in the simulation; if $\mathsf{sk}_r \neq \mathsf{sk}^\star$ it computes $\mathsf{Enc}_{\mathsf{sk}_r}(\widetilde{m}_r^{\mathsf{id}})$ locally. For each encryption call $E_r^{\mathsf{id}} = (\mathsf{sk}_r, m_r^{\mathsf{id}})$ with $r \geq i^\star + 2$, the simulated execution should follow $H_{2,i^\star}$ on that call, that is, encrypt the real plaintext m_r^{id}. Accordingly, if $\mathsf{sk}_r = \mathsf{sk}^\star$ then $\mathcal{B}_{\mathsf{cpa}}$ queries $\mathcal{O}_{\mathsf{Enc}_{\mathsf{sk}^\star}}(m_r^{\mathsf{id}})$; otherwise it computes $\mathsf{Enc}_{\mathsf{sk}_r}(m_r^{\mathsf{id}})$ locally.
4. For the single challenge encryption call $E_{i^\star+1}^{\mathsf{id}} = (\mathsf{sk}^\star, m^\star)$, $\mathcal{B}_{\mathsf{cpa}}$ submits the IND-CPA challenge pair $m_0 := m^\star$ and $m_1 \xleftarrow{\$} \{0,1\}^{|m^\star|}$ to the challenger and receives the challenge ciphertext $c^\star = \mathsf{Enc}_{\mathsf{sk}^\star}(m_b)$ for a hidden bit $b \in \{0,1\}$. $\mathcal{B}_{\mathsf{cpa}}$ uses $c^\star$ as the ciphertext output for $E_{i^\star+1}^{\mathsf{id}}$ in the simulated execution.
5. Finally, $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$ based on its view of the simulated execution, and $\mathcal{B}_{\mathsf{cpa}}$ outputs the same bit b' to the challenger.

If $b = 0$ then the above simulation is distributed exactly as $H_{2,i^\star}$, and if $b = 1$ it is distributed exactly as $H_{2,i^\star+1}$. Therefore, $\Delta_1 \leq q_{\mathsf{id}} \cdot \mathsf{Adv}_{\mathsf{AE}}^{\mathsf{ind\text{-}cpa}}(\mathcal{B}_{\mathsf{cpa}}) \leq q_{\mathsf{id}} \cdot \mathsf{neg}(\lambda) \leq \mathsf{neg}(\lambda)$. □

Game G_3: Replace honest encryptions of type c_j by encryptions of random strings
In G_3, we apply the same modification as in G_2 to the honest encryptions of type c_j. That is, for each encryption call $E_r^j = (\mathsf{sk}_r, m_r^j) \in \mathcal{E}_j$, the honest party replaces the encryption of m_r^j by the encryption of a uniformly random string $\widetilde{m}_r^j \leftarrow \{0,1\}^{|m_r^j|}$ of the same length under the same key, while keeping m_r^j locally recorded for subsequent computations. The only difference from G_2 is that we now consider the ciphertext family c_j, whose plaintext lengths and number of encryption instances differ from those of cid.

Lemma 3. *Assuming* AE *is IND-CPA secure, the modification in Game G_3 changes the adversary's view only negligibly. Formally, if p_2 and p_3 denote the probabilities that $\mathcal{A}$ outputs 1 in Games G_2 and G_3, respectively, then $\Delta_2 = |p_2 - p_3| \leq \mathsf{neg}(\lambda)$.*

Proof. The proof is identical to that of Lemma 2, except that the ordered list of encryption calls $\mathcal{E}_{\mathsf{id}}$ is replaced by $\mathcal{E}_j$, the number of hybrid steps q_{id} is replaced by q_j, and each challenge pair consists of two messages of length $|m_r^j|$ rather than $|m_r^{\mathsf{id}}|$. Thus, by the same IND-CPA hybrid argument, $\Delta_2 \leq q_j \cdot \mathsf{Adv}_{\mathsf{AE}}^{\mathsf{ind\text{-}cpa}}(\mathcal{B}_{\mathsf{cpa}}) \leq q_{\mathsf{j}} \cdot \mathsf{neg}(\lambda) \leq \mathsf{neg}(\lambda)$. □

At this point, all information remaining in G_3 is exactly the information already available from the ideal functionality together with local bookkeeping, including the number of protocol instances and the corresponding plaintext lengths. Therefore, no further hybrid is needed: the simulator interacting with $\mathcal{F}_{\mathsf{UpSPA}}$ can already reproduce the view in G_3. Finally, Combining Lemmas 1 to 3 using the triangle inequality, we obtain:

$|p_0 - p_3| \leq \Sigma_{i=0}^{i=2} \Delta_i \leq \mathsf{Adv}_{\mathsf{TOPRF}}^{\mathsf{prf}}(\mathcal{B}_{\mathsf{prf}}) + (q_{\mathsf{enc}} + q_j) \cdot \mathsf{Adv}_{\mathsf{AE}}^{\mathsf{ind\text{-}cpa}}(\mathcal{B}_{\mathsf{cpa}}) \leq \mathsf{neg}(\lambda)$. □

7 Performance Evaluation

Setup. All of our experimental results based on our proof of concept implementation are produced on a standard laptop machine with 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00 GHz, 16 GB RAM, NVIDIA GeForce RTX 2060, running a 64-bit Windows operating system on an x64-based processor and implemented in Rust. Our code is publicly available on GitHub [15]. Our implementation uses the `ristretto255` group instantiated via `Curve25519`[3] and `BLAKE3`[4] as a collision resistant hash function.
We evaluate the client-side performance of UpSPA and compare its core costs with TSPA.We report median (p50) runtimes in ms for the asymptotic (n_{sp}, t_{sp}) settings, with $n_{sp} \in \{20, 40, 60, 80, 100\}$ and t_{sp} set to 20%, 40%, 60%, 80%, and 100% of n_{sp}. We also report the low-threshold (n_{sp}, t_{sp}) settings, with $n_{sp} \in \{3, 5, 7, 10\}$ and t_{sp} set to 35%, 70%, and 100% of n_{sp}, respectively, rounded to the nearest integer. For each parameter point, we collect 2000 measurements after discarding 300 warm-up runs and report the median (p50) as the primary statistic due to its robustness to noise. While we also compute p95, mean, and standard deviation, we omit them from the plots since p50 already captures the comparison clearly.

7.1 Client-side asymptotical settings:

Results:
Setup phase [1,2] remains lightweight, ranging from 0.047 ms to 0.640 ms. Registration phase [2,3] spans from 0.135 ms to 2.000 ms for UpSPA, compared to 0.463 ms to 3.067 ms for TSPA. Authentication phase [3,3] ranges from 0.139 ms to 2.013 ms for UpSPA and from 0.120 ms to 2.249 ms for TSPA. Secret update phase [2,4] varies from 0.141 ms to 2.193 ms. Password update phase [2,5] is the most expensive phase, ranging from 0.393 ms to 4.478 ms.

Phase	Setup					Secret Update					Password Update				
n_{sp} \ t_{sp}%	20	40	60	80	100	20	40	60	80	100	20	40	60	80	100
20	0.047	0.054	0.061	0.068	0.075	0.141	0.217	0.305	0.385	0.468	0.393	0.469	0.562	0.654	0.738
40	0.080	0.095	0.112	0.130	0.151	0.226	0.389	0.547	0.717	0.879	0.724	0.922	1.111	1.319	1.515
60	0.115	0.143	0.177	0.215	0.258	0.346	0.613	0.801	1.043	1.287	1.223	1.566	1.795	2.081	2.386
80	0.168	0.217	0.275	0.341	0.416	0.398	0.720	1.066	1.374	1.702	1.462	1.943	2.437	2.885	3.368
100	0.235	0.312	0.405	0.514	0.640	0.488	0.885	1.421	1.724	2.193	1.882	2.495	3.144	3.867	4.478

Table 2: Client-side time for Setup, Secret Update, and Password Update.

[3] `https://doc.dalek.rs/curve25519_dalek/ristretto/`
[4] `https://docs.rs/blake3/latest/blake3/`

Phase	Registration										Authentication									
Scheme	UpSPA					TSPA					UpSPA					TSPA				
n_{sp} \ t_{sp}%	20	40	60	80	100	20	40	60	80	100	20	40	60	80	100	20	40	60	80	100
20	0.135	0.214	0.286	0.364	0.452	0.463	0.468	0.474	0.498	0.492	0.139	0.210	0.287	0.368	0.448	0.120	0.202	0.286	0.378	0.465
40	0.214	0.371	0.521	0.685	0.841	0.942	0.982	1.004	1.016	1.062	0.214	0.368	0.520	0.676	0.852	0.201	0.375	0.556	0.737	0.991
60	0.309	0.589	0.788	0.990	1.223	1.624	1.569	1.640	1.609	1.665	0.320	0.580	0.782	0.991	1.219	0.323	0.567	0.831	1.093	1.359
80	0.377	0.748	0.994	1.305	1.786	1.969	2.071	2.151	2.239	2.595	0.371	0.705	0.993	1.304	1.779	0.386	0.740	1.096	1.540	1.993
100	0.485	0.841	1.321	1.613	2.000	2.473	2.646	2.769	3.050	3.067	0.460	0.833	1.339	1.630	2.013	0.468	1.022	1.358	1.808	2.249

Table 3: TSPA/UpSPA Client-side time for Registration and Authentication.

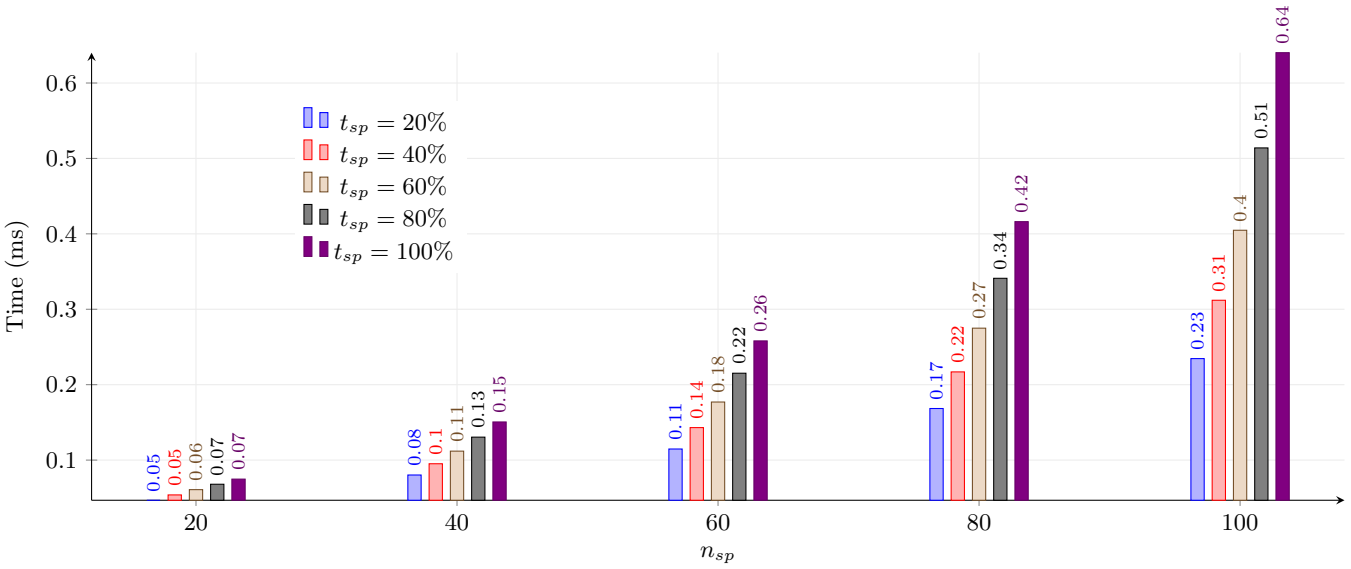

Fig. 1: Client-side setup time for asymptotical settings.

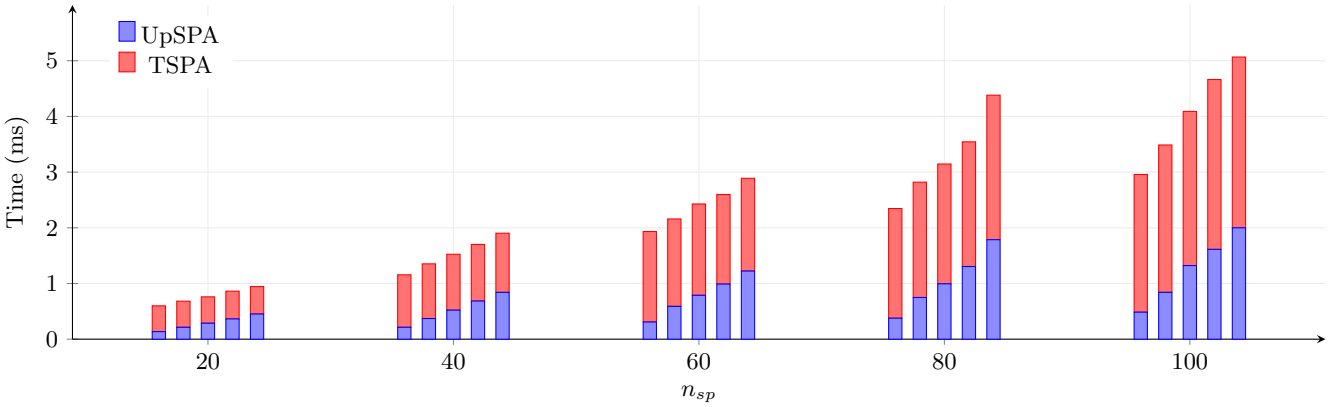

Fig. 2: Client-side registration time comparison between UpSPA and TSPA for asymptotical settings.

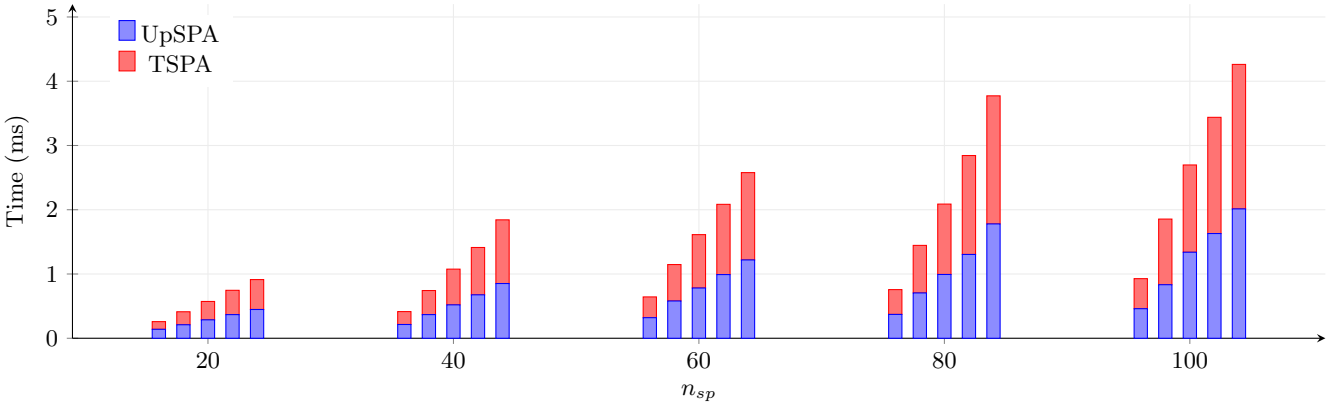

Fig. 3: Client-side authentication time comparison between UpSPA and TSPA for asymptotical settings.

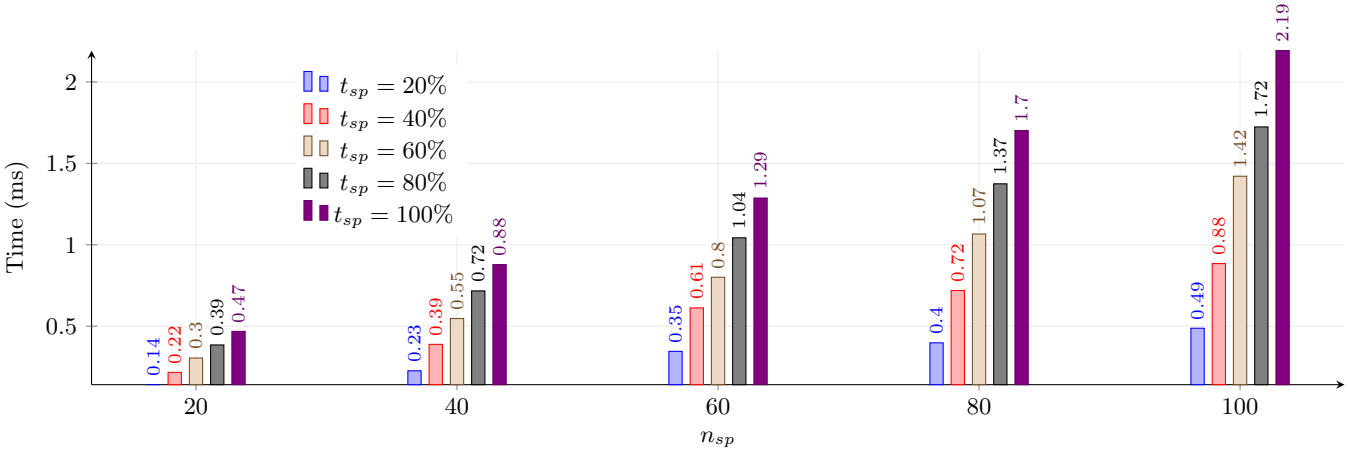

Fig. 4: Client-side secret update time for asymptotical settings.

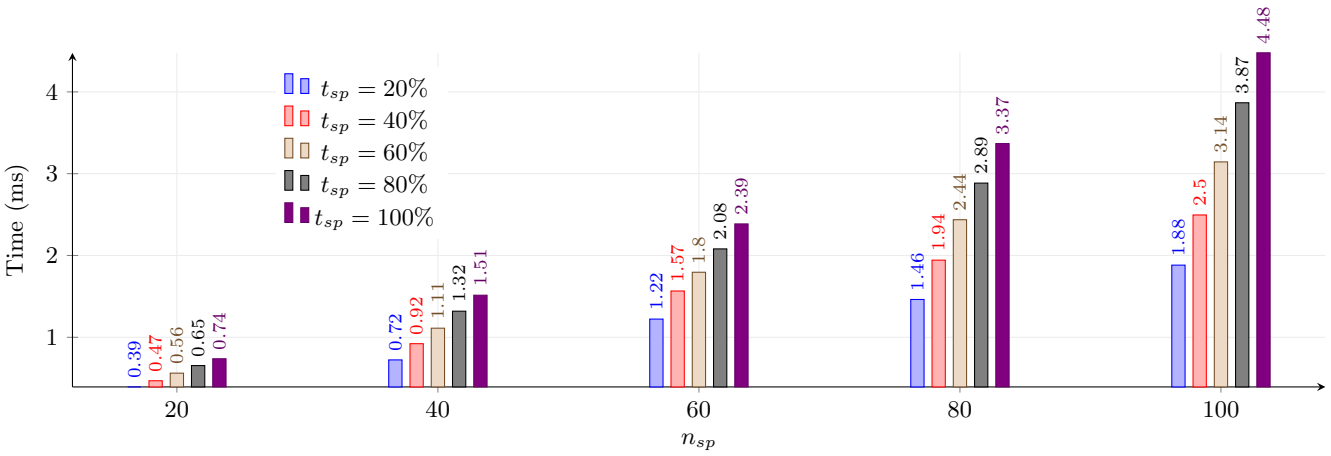

Fig. 5: Client-side password update time for asymptotical settings.

Comparison:

UpSPA **vs.** TSPA. In registration, UpSPA consistently outperforms TSPA across all measured configurations. UpSPA ranges from 0.135 ms to 2.000 ms, whereas TSPA ranges from 0.463 ms to 3.067 ms. This is consistent with the protocol structures: in TSPA, each registration generates a fresh random secret, creates threshold shares, derives one OPRF key per storage provider, and encrypts each share separately, while in UpSPA these long-term materials are established in the one-time setup phase and registration only derives and stores the per-login-server secret under the recovered long-term state. By contrast, authentication is much closer in the two schemes. UpSPA ranges from 0.139 ms to 2.013 ms, while TSPA ranges from 0.120 ms to 2.249 ms. TSPA is slightly faster at a few low-threshold points, but the gap remains small overall. This is expected, since both schemes are dominated by essentially the same online task: interacting with threshold-

many storage providers to recover the high-entropy secret needed to recompute the login-server verification information. Overall, UpSPA preserves authentication efficiency at roughly the same level as TSPA, while substantially improving registration performance.

Secret update vs. password update in UpSPA. The main difference between secret update and password update is their scaling behavior. The secret update is applied per login-server credential, so refreshing all accounts requires repeating it once for each of the n_{ls} login servers, and thus scales linearly with n_{ls} in addition to the threshold interaction cost. The password update, in contrast, refreshes the password-protected state only once, independent of the number of login-server accounts. For example, when $n_{ls} = 100$, $n_{sp} = 20$, and $t_{sp} = 20\%$, we obtain $\frac{T_{\Pi_{\mathsf{PwdUpdate}}}}{(n_{ls})\cdot(T_{\Pi_{\mathsf{SecretUpdate}}})} = \frac{0.393}{14.1} \approx 2.8 \times 10^{-2}$ which is already very negligible. Hence, the password update quickly becomes a more efficient operation as the number of login-server accounts grows, whereas the secret update remains natural when only a single login-server credential must be rotated.

7.2 Client-side low-threshold settings:

Results: Setup phase [4,6] remains lightweight, ranging from 0.028 ms to 0.040 ms. Registration phase [5,7] spans from 0.068 ms to 0.197 ms for UpSPA, improving over TSPA in the high range (up to 0.219 ms). Authentication phase [5,8] ranges from 0.067 ms to 0.210 ms for UpSPA and from 0.041 ms to 0.219 ms for TSPA. Secret update phase [4,9] varies from 0.068 ms to 0.252 ms. Password update phase [4,10] remains the most expensive phase, ranging from 0.123 ms to 0.397 ms.

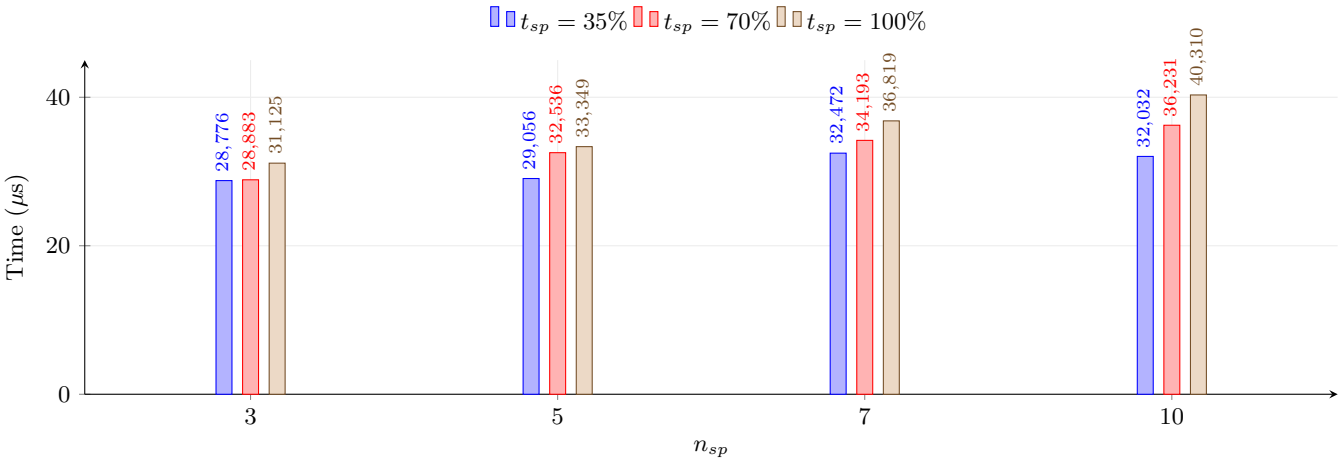

Fig. 6: Client-side setup time (low-scale settings).

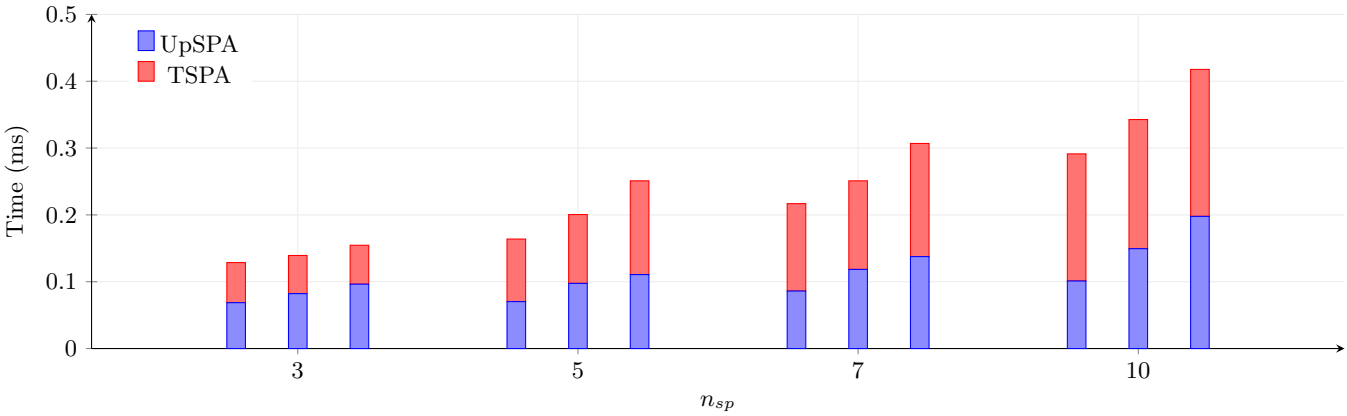

Fig. 7: Client-side registration time comparison between UpSPA and TSPA (low-scale settings).

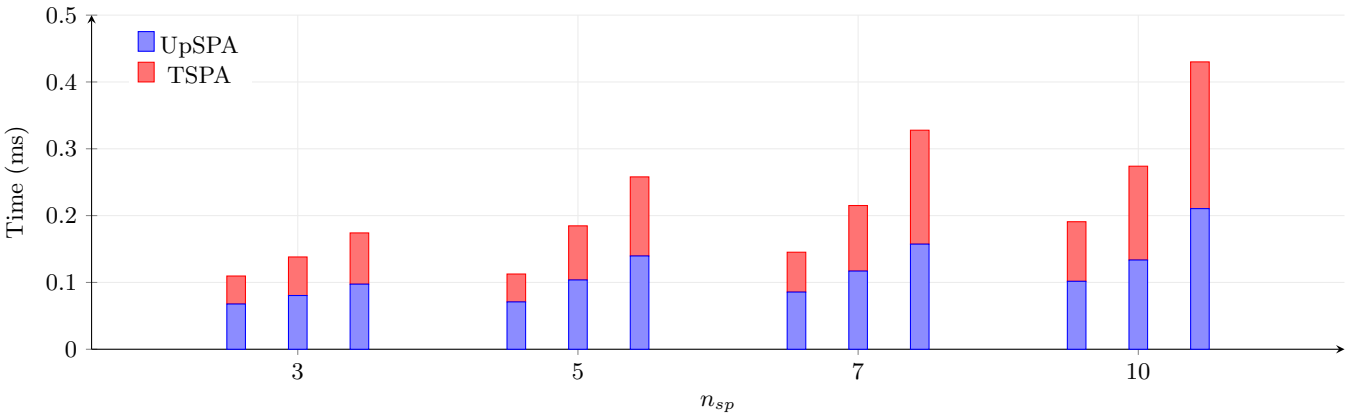

Fig. 8: Client-side authentication time comparison between UpSPA and TSPA (low-scale settings).

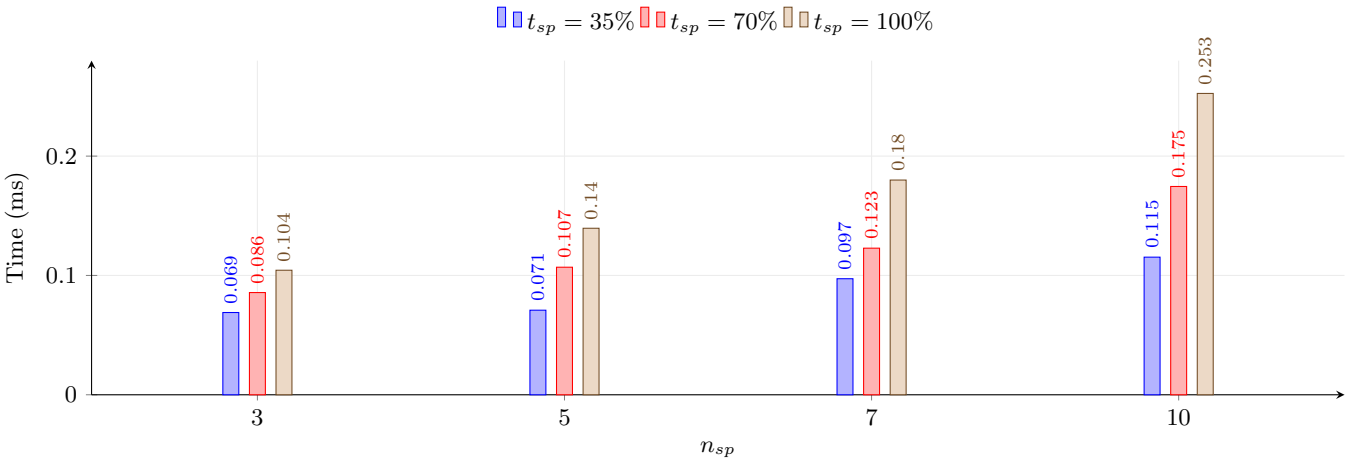

Fig. 9: Client-side secret update time (low-scale settings).

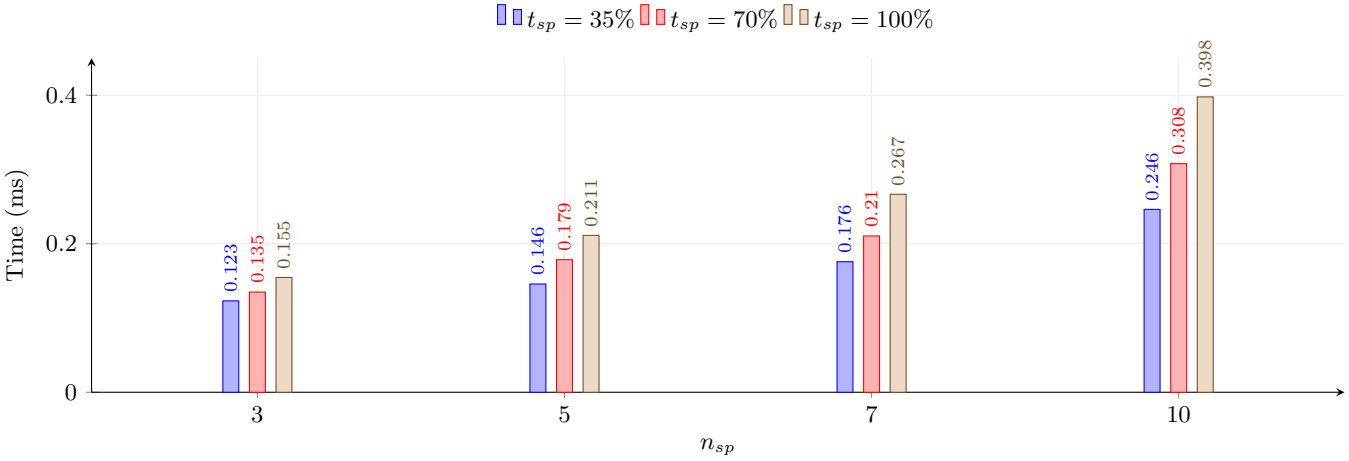

Fig. 10: Client-side password update time (low-scale settings).

Phase	Setup			Secret Update			Password Update		
n_{sp} \ t_{sp}%	35	70	100	35	70	100	35	70	100
3	0.028776	0.028883	0.031125	0.068934	0.085653	0.104361	0.123088	0.134915	0.154630
5	0.029056	0.032536	0.033349	0.070877	0.106895	0.139541	0.145817	0.178527	0.211323
7	0.032472	0.034193	0.036819	0.097248	0.122857	0.179939	0.175776	0.210485	0.266670
10	0.032032	0.036231	0.040310	0.115327	0.174555	0.252518	0.246292	0.307879	0.397834

Table 4: Client-side time for Setup, Secret Update, and Password Update (low-scale settings).

Phase	Registration						Authentication					
Scheme	UpSPA			TSPA			UpSPA			TSPA		
n_{sp} \ t_{sp}%	35	70	100	35	70	100	35	70	100	35	70	100
3	0.068623	0.082169	0.096331	0.059989	0.057101	0.058235	0.067689	0.080421	0.097391	0.041911	0.057687	0.076702
5	0.070259	0.097534	0.110664	0.093611	0.102976	0.140310	0.070847	0.103695	0.139585	0.041742	0.080984	0.118366
7	0.086051	0.118408	0.137557	0.130775	0.132560	0.169361	0.085560	0.117006	0.157301	0.059654	0.098094	0.170494
10	0.101095	0.149429	0.197797	0.190210	0.193245	0.219907	0.101685	0.133529	0.210451	0.089159	0.140407	0.219572

Table 5: Client-side time for Registration and Authentication (low-scale settings).

Scheme	UpSPA			TSPA			TSPA			Ratio to UpSPA TOPRF					
Primitive	TOPRF			Reg. OPRF			Auth. OPRF			Reg. OPRF/UpSPA			Auth. OPRF/UpSPA		
n_{sp} \ t_{sp}%	35	70	100	35	70	100	35	70	100	35	70	100	35	70	100
3	0.055	0.070	0.086	0.053	0.053	0.053	0.018	0.035	0.053	0.963	0.749	0.616	0.321	0.500	0.616
5	0.056	0.089	0.121	0.088	0.088	0.088	0.018	0.053	0.088	1.558	0.981	0.722	0.312	0.589	0.722
7	0.076	0.110	0.159	0.133	0.133	0.133	0.038	0.076	0.133	1.757	1.216	0.838	0.502	0.695	0.838
10	0.088	0.135	0.197	0.173	0.173	0.177	0.052	0.104	0.177	1.963	1.282	0.902	0.589	0.769	0.902

Table 6: Client-side TOPRF/OPRF costs for UpSPA and TSPA(low-scale setting).

Comparison:
UpSPA vs. TSPA at low scale. The client-side measurements in Table 5, together with the primitive TOPRF/OPRF ratios in Table 6, show that the asymptotic advantage of UpSPA does not necessarily translate into a uniform advantage at very small parameter sizes. In registration, TSPA is faster than UpSPA for $n_{sp} = 3$: TSPA registration ranges from 0.057 ms to 0.060 ms, whereas UpSPA registration ranges from

0.069 ms to 0.096 ms The ratio

$$\frac{\text{TSPA registration all-OPRF cost}}{\text{UpSPA TOPRF cost}}$$

is below one for all three $n_{sp} = 3$ threshold settings, namely 0.963, 0.749, and 0.616. Thus, at this scale, the total cost of the few TSPA OPRF evaluations is still smaller than, or comparable to, the single TOPRF reconstruction in UpSPA. As n_{sp} increases, this effect reverses. For the low-threshold settings, the same ratio becomes $1.558 \quad (n_{sp} = 5), \qquad 1.757 \quad (n_{sp} = 7), \qquad 1.963 \quad (n_{sp} = 10)$.

This means that the TSPA registration OPRF block alone becomes more expensive than the entire TOPRF block of UpSPA. The client-side measurements reflect this crossover: for $n_{sp} = 5, 7, 10$, UpSPA registration is usually faster than TSPA registration. For example, at $n_{sp} = 10$, UpSPA registration costs 0.101, 0.149, and 0.198 ms, whereas TSPA registration costs 0.190, 0.193, and 0.220 ms. Hence, registration is the phase in which the advantage of replacing n_{sp} independent OPRF evaluations by one TOPRF reconstruction becomes visible even at low scale.

Authentication has a later crossover. The primitive table shows that the ratio

$$\frac{\text{TSPA authentication all-OPRF cost}}{\text{UpSPA TOPRF cost}}$$

is below one in all reported low-scale settings, ranging from 0.312 to 0.902. This indicates that, for small thresholds, the OPRF component of TSPA authentication can be cheaper than the TOPRF reconstruction in UpSPA. Consequently, TSPA authentication is faster in several small-parameter cases. For example, at $n_{sp} = 5$, TSPA authentication costs 0.0417, 0.0810, and 0.1184 ms, whereas UpSPA authentication costs 0.0708, 0.1037, and 0.1396 ms. This agrees with the primitive breakdown: the TSPA authentication OPRF block is only a fraction of the UpSPA TOPRF cost at these parameter sizes.

As t_{sp} grows, however, TSPA authentication pays more repeated per-provider work, while the number of UpSPA state decryptions decreases. In particular, when $t_{sp} = n_{sp}$, we have $\lceil n_{sp}/t_{sp} \rceil + 1 = 2$, so UpSPA decrypts only two AEAD ciphertexts: one `cipherid` and one `ciphersp`. This explains why UpSPA becomes competitive or faster in the higher-threshold settings. For instance, at $n_{sp} = 7$ and $t_{sp} = 100\%$, UpSPA authentication costs 0.1573 ms, while TSPA costs 0.1705 ms. Similarly, for $n_{sp} = 10$, UpSPA is slightly faster at the 70% and 100% threshold settings.

The reason for this behavior is that the two schemes have different primitive cost profiles. We use the following notation to separate the cost of one full TSPA OPRF evaluation from the cost of a threshold OPRF reconstruction: $T_{\mathsf{OPRF_full}} = T_{\mathsf{MulP}} + T_{\mathsf{OPRF_finalize}}$. Here, $T_{\mathsf{OPRF_full}}$, denotes the cost of one point-scalar multiplication followed by the OPRF finalization step. This corresponds to the TSPA implementation path, $Y \leftarrow P \cdot k_{i,i} \leftarrow \mathsf{OPRF_Finalize}(\mathsf{pwd}, Y)$.

In contrast, the client-side TOPRF reconstruction used by UpSPA is closer to

$$T_{\mathsf{TOPRF}}(t_{sp}) \approx (t_{sp} + 2)T_{\mathsf{MulP}} + t_{sp}T_{\mathsf{AddP}} + T_{\mathsf{InvS}} + T_{\mathsf{OPRF_finalize}}.$$

The term $(t_{sp} + 2)T_{\mathsf{MulP}}$ accounts for the client blinding multiplication, the t_{sp} scalar multiplications used in the Lagrange combination of partial evaluations, and the final unblinding multiplication. The TOPRF computation therefore performs only one final OPRF finalization, but it performs a threshold reconstruction over t_{sp} partial points. Thus, T_{TOPRF} should not be interpreted as t_{sp} independent OPRF evaluations. For registration, the implementation-level costs can be written as

$$T_{\mathsf{reg}}^{\mathsf{TSPA}}(n_{sp}, t_{sp}) = n_{sp} \cdot T_{\mathsf{OPRF_full}} + n_{sp} \cdot T_{\mathsf{Enc}} + 2 \cdot T_{\mathsf{Hash}} + n_{sp} \cdot (t_{sp} - 1) \cdot T_{\mathsf{FieldOp}},$$

whereas UpSPA has

$$T_{\mathsf{reg}}^{\mathsf{UpSPA}}(n_{sp}, t_{sp}) = T_{\mathsf{TOPRF}}(t_{sp}) + T_{\mathsf{AE\text{-}Dec}} + (n_{sp} + 1)T_{\mathsf{Hash}} + T_{\mathsf{AE\text{-}Enc}}.$$

The main difference is that TSPA registration performs one full OPRF evaluation per storage provider, while UpSPA performs one TOPRF reconstruction. Hence the dominant OPRF-related term in TSPA registration scales as $n_{sp}T_{\mathsf{OPRF_full}}$, whereas the dominant TOPRF term in UpSPA scales mainly with t_{sp}. Authentication exhibits a different scaling behavior. The TSPA authentication cost can be written as

$$T_{\mathsf{auth}}^{\mathsf{TSPA}}(n_{sp}, t_{sp}) = t_{sp} \cdot T_{\mathsf{OPRF_full}} + t_{sp} \cdot T_{\mathsf{Dec}} + 2 \cdot T_{\mathsf{Hash}} + T_{\mathsf{MulP}} + T_{\mathsf{InvS}} + t_{sp} \cdot T_{\mathsf{FieldOp}}.$$

Here, $t_{sp} \cdot T_{\mathsf{OPRF_full}}$ accounts for the t_{sp} contacted providers, while the additional T_{MulP} term corresponds to the client blinding step. In UpSPA, authentication has the form

$$T_{\mathsf{auth}}^{\mathsf{UpSPA}}(n_{sp}, t_{sp}) = T_{\mathsf{TOPRF}}(t_{sp}) + (\lceil n_{sp}/t_{sp} \rceil + 1) \cdot T_{\mathsf{AE\text{-}Dec}} + (n_{sp} + 1) \cdot T_{\mathsf{Hash}}.$$

Thus, TSPA authentication grows linearly with the number of contacted providers through t_{sp} OPRF evaluations, t_{sp} decryptions, and t_{sp} field combination steps. In contrast, UpSPA performs one TOPRF reconstruction and decrypts only $\lceil n_{sp}/t_{sp} \rceil + 1$ encrypted state records. These results do not contradict the asymptotic comparison. Rather, they identify the small-parameter crossover region. At very small n_{sp} and t_{sp}, TSPA can benefit from cheaper constants: it uses AES-CTR over 32-byte shares and performs only a small number of OPRF evaluations. By contrast, UpSPA pays the fixed cost of TOPRF reconstruction and uses authenticated encryption through XChaCha20-Poly1305, which has a larger constant cost than AES-CTR in this benchmark. However, as n_{sp} and t_{sp} increase, the repeated OPRF and per-provider costs in TSPA dominate. Registration scales with $n_{sp} \cdot T_{\mathsf{OPRF_full}}$ and $n_{sp} \cdot (t_{sp} - 1) \cdot T_{\mathsf{FieldOp}}$, while authentication scales with $t_{sp} \cdot T_{\mathsf{OPRF_full}}$ and $t_{sp} \cdot T_{\mathsf{Dec}}$. In UpSPA, the corresponding online cost is centered around one TOPRF reconstruction, together with a linear number of hashes and only $\lceil n_{sp}/t_{sp} \rceil + 1$ AEAD decryptions in authentication. This explains why TSPA is faster in some very small cases, while UpSPA scales more favorably and dominates in the larger asymptotic experiments.

8 Conclusion

We presented UpSPA, a secure and efficient updatable threshold SPA that addresses key limitations of prior works. UpSPA prevents preemption and unauthorized updates at storage providers while supporting both secret and password updates without requiring changes to login servers. We proved its security in the ideal real paradigm under standard threshold corruption assumptions and showed that it incurs low practical overhead. We demonstrate that strong security and password and secret updates can be achieved in SPA without sacrificing deployability or efficiency.

References

1. 16 billion passwords stolen from 320 million+ computers leaked online (2025), https://gbhackers.com/16-billion-passwords-stolen-from-320-million-computers-leaked-online/
2. The Hacker News (2026), https://thehackernews.com/2026/01/password-reuse-in-disguise-often-missed.html
3. Acar, T., Belenkiy, M., Küpçü, A.: Single password authentication. Elsevier Computer Networks (2013)
4. Agrawal, S., Miao, P., Mohassel, P., Mukherjee, P.: Pasta: password-based threshold authentication. ACM CCS (2018)
5. Baecker, R., Gerhart, P., Schröder, D.: Password-hardened encryption revisited. In: ASIACRYPT (2025)
6. Bagherzandi, A., Jarecki, S., Saxena, N., Lu, Y.: Password-protected secret sharing. In: ACM CCS (2011)
7. Baum, C., Frederiksen, T., Hesse, J., Lehmann, A., Yanai, A.: Pesto: proactively secure distributed single sign-on, or how to trust a hacked server. In: EuroS&P (2020)
8. Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. Journal of Cryptology (2008)
9. Bicakci, K., Atalay, N.B., Yuceel, M., van Oorschot, P.C.: Exploration and field study of a browser-based password manager using icon-based passwords. In: Workshop on Real-Life Cryptographic Protocols and Standardization (2011)

10. Boyko, V., MacKenzie, P., Patel, S.: Provably secure password-authenticated key exchange using diffie-hellman. In: EUROCRYPT (2000)
11. Brost, J., Egger, C., Lai, R.W.F., Schmid, F., Schröder, D., Zoppelt, M.: Threshold password-hardened encryption services. In: ACM CCS (2020)
12. Camenisch, J., Lehmann, A., Lysyanskaya, A., Neven, G.: Memento: How to reconstruct your secrets from a single password in a hostile environment. CRYPTO (2014)
13. Canetti, R.: Security and composition of multiparty cryptographic protocols. Journal of CRYPTOLOGY (2000)
14. Casacuberta, S., Hesse, J., Lehmann, A.: Sok: Oblivious pseudorandom functions. Cryptology ePrint Archive (2022)
15. CRYPTO KU Research Group: Benchmark for Secure and updatable single password authentication. `https://github.com/rezasaadi/UpSpa_benchmark` (2026)
16. Doğanay, C., Küpçü, A.: Comparative survey on single password authentication techniques. ISCTURKEY (2020)
17. Everspaugh, A., Chaterjee, R., Scott, S., Juels, A., Ristenpart, T.: The pythia PRF service. In: USENIX Security Symposium (2015)
18. İşler, D., Küpçü, A.: Threshold single password authentication. In: ESORICS DPM (2017)
19. Jarecki, S., Krawczyk, H., Resch, J.: Threshold partially-oblivious prfs with applications to key management. Cryptology ePrint Archive, Report 2018/733 (2018)
20. Jarecki, S., Krawczyk, H., Resch, J.: Updatable oblivious key management for storage systems. In: ACM CCS (2019)
21. Jarecki, S., Krawczyk, H., Shirvanian, M., Saxena, N.: Device-enhanced password protocols with optimal online-offline protection. In: ASIA CCS (2016)
22. Jarecki, S., Liu, X.: Efficient oblivious pseudorandom function with applications to adaptive ot and secure computation of set intersection. In: TCC (2009)
23. Jiang, C., Xu, C., Liu, Z., Dong, X., Zhang, W.: Threshold password-hardening updatable oblivious key management. T-IFS (2025)
24. Jiang, C., Xu, C., Yang, G.: Augsso: Secure threshold single-sign-on authentication with popular password collection. IEEE Transactions on Mobile Computing (2025)
25. Jiang, C., Xu, C., Yang, G., Zhang, Z., Chen, J.: Device-enhanced password-based threshold single-sign-on authentication. T-IFS (2025)
26. Johnson, D., Menezes, A., Vanstone, S.: The elliptic curve digital signature algorithm (ecdsa). International journal of information security (2001)
27. Lai, R.W.F., Egger, C., Reinert, M., Chow, S.S.M., Maffei, M., Schröder, D.: Simple Password-Hardened encryption services. In: USENIX Security Symposium (2018)
28. Lueks, W., Hampiholi, B., Alpár, G., Troncoso, C.: Tandem: Securing keys by using a central server while preserving privacy. PoPETs (2020)
29. Rawat, R., Jhanwar, M.P.: Pas-ta-u: Password-based threshold authentication with password update. In: International Conference on Security, Privacy, and Applied Cryptography Engineering (2020)
30. Scarlata, M., Torrisi, G., Backendal, M., Paterson, K.G.: Zero knowledge (about) encryption: A comparative security analysis of three cloud-based password managers. In: USENIX Security Symposium (2026)
31. Zhang, Y., Xu, C., Li, H., Yang, K., Cheng, N., Shen, X.: Protect: efficient password-based threshold single-sign-on authentication for mobile users against perpetual leakage. IEEE Transactions on Mobile Computing (2020)
32. Zhang, Z., Xu, C., Jiang, C., Chen, K.: Tsapp: Threshold single-sign-on authentication preserving privacy. IEEE Trans. Dependable Secur. Comput. (2024)
33. İşler, D., Küpçü, A.: Distributed single password protocol framework. Cryptology ePrint Archive, Paper 2018/976 (2018)