PAS-TA-U: PASsword-based Threshold Authentication with Password Update

Rachit Rawat[1] and Mahabir Prasad Jhanwar[2]

[1]Ashoka University, INDIA, rachit.rawat@alumni.ashoka.edu.in
[2]Ashoka University, INDIA, mahavir.jhawar@ashoka.edu.in

Abstract

A single-sign-on (SSO) is an authentication system that allows a user to log in with a single identity and password to any of several related, yet independent, server applications. SSO solutions eliminate the need for users to repeatedly prove their identities to different applications and hold different credentials for each application. *Token-based authentication* is commonly used to enable an SSO experience on the web, and on enterprise networks. A large body of work considers distributed token generation which can protect the long-term keys against a subset of breached servers. A recent work (CCS'18) introduced the notion of Password-based Threshold Authentication (PbTA) with the goal of making password-based token generation for SSO secure against server breaches that could compromise both long-term keys and user credentials. They also introduced a generic framework called PASTA that can instantiate a PbTA system.

The existing SSO systems built on distributed token generation techniques, including the PASTA framework, do not admit password-update functionality. In this work, we address this issue by proposing a password-update functionality into the PASTA framework. We call the modified framework PAS-TA-U.

As a concrete application, we instantiate PAS-TA-U to implement in Python a distributed SSH key manager for enterprise networks (ESKM) that also admits a password-update functionality for its clients. Our experiments show that the overhead of protecting secrets and credentials against breaches in our system compared to a traditional single server setup is low (average 119 ms in a 10-out-of-10 server setting on Internet with 80 ms round trip latency).

1 Introduction

Password-based Authentication (PbA): One of the primary purposes of authentication is to facilitate access control to a resource such as local or remote access to computer accounts; access to software applications, when an access privilege is linked to a particular identity. A typical scenario is when the remote resource is a *application server*, and a user is using a *client application* to authenticate itself to the application server. Password-based techniques are a very popular choice underlying most authentication systems. In a password-based authentication (PbA) system, a user U creates its user credential record — a user identity $\mathtt{id}_\mathtt{U}$, and a hash of a secret password pwd, i.e. hash(pwd), with the application server through a one-time registration process. Later, in order to gain access (log in) to the application server, U provides $(\mathtt{id}_\mathtt{U}, \mathtt{pwd})$ to its client which then computes hash(pwd), and sends $(\mathtt{id}_\mathtt{U}, \mathtt{hash}(\mathtt{pwd}))$ to the application server. The server compares this to the stored record for the stated $\mathtt{id}_\mathtt{U}$ and gives access if it matches.

Token-based Single-sign-on (SSO): A single-sign-on is an authentication system that allows a user to log in with a single identity and password to any of several related, yet independent, server applications. SSO solutions eliminates the need for users to repeatedly prove their identities to different applications using different credentials for each application. We make a distinction by calling one of the application servers as the identity provider, i.e. IP, and the rest of the related application servers as AS. Token-based techniques are currently very common for the implementation of an SSO system. In such a system, an IP generates a

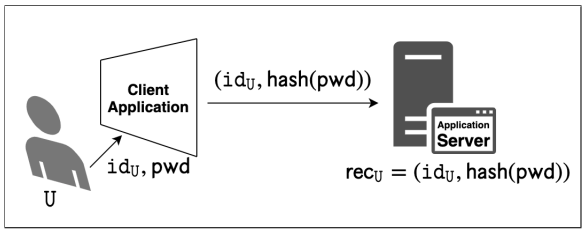

Figure 1: PbA: Login

one-time key pair of signing and verification keys $(\mathsf{sk}, \mathsf{vk})$, keeps the signing key sk with itself, and makes the verification key vk available to all $\mathtt{AS}$s. A one-time registration phase allows a user $\mathtt{U}$ to create and store its credential $(\mathtt{id}_\mathtt{U}, \mathsf{pwd})$ with the $\mathtt{IP}$. In order to gain access to any $\mathtt{AS}$, the user $\mathtt{U}$ must reach out to $\mathtt{IP}$ with its user credential $(\mathtt{id}_\mathtt{U}, \mathsf{hash}(\mathsf{pwd}))$, and a payload pld that contains the user's information/attributes, expiration time and a policy that would control the nature of access. The $\mathtt{IP}$ verifies $\mathtt{U}$'s credential by matching it against a stored record before issuing an authentication token tkn which is produced for the payload pld with the help of the secret singing key sk, i.e. $\mathsf{tkn} \leftarrow \mathsf{Sign}(\mathsf{pld}, \mathsf{sk})$. The token so obtained is stored by the user client in a cookie or the local storage, and can then be used for all future accesses to $\mathtt{AS}$'s without using the pwd, until it expires. In particular, when the user client presents the tkn to an $\mathtt{AS}$ requesting for an access, the tkn is verified by the $\mathtt{AS}$ which holds the verification key vk, i.e. $1/\perp \leftarrow \mathsf{Verify}(\mathsf{tkn}, \mathsf{vk})$. Popular token-based $\mathtt{SSO}$ systems include JSON Web Token (JWT), SAML, OAuth, and OpenID.

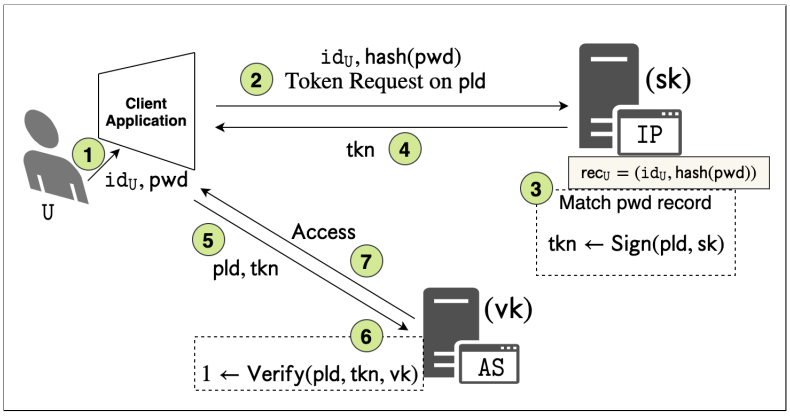

Figure 2: SSO : Token-based

Password-based Threshold Authentication ($\mathtt{PbTA}$): The basic $\mathtt{PbA}$ systems, and the more advanced token-based $\mathtt{SSO}$ systems place greater responsibility on an identity provider $\mathtt{IP}$ of maintaining the confidentiality of a large number of users credentials. Such an $\mathtt{IP}$ is a single point of failure that if breached, enables an attacker to (1) recover the long term signing key sk and forge arbitrary tokens that enable access to all application servers and (2) obtain hashed passwords of users to use as part of an offline dictionary attack to recover their credentials. A large body of work considers distributed token generation through threshold digital signatures and threshold message authentication codes which can protect the long term signing key sk against a subset of breached servers [1, 2, 3, 4, 5, 6]. A separate line of work on threshold password-authenticated key exchange (T-PAKE) aims to prevent offline dictionary attacks in standard password-authenticated key exchange (PAKE) by employing multiple servers [7, 8, 9, 10].

A very recent work of Agrawal et al. [11] introduces the notion of Password-based Threshold Authentication ($\mathtt{PbTA}$) for token-based $\mathtt{SSO}$ where they propose to distribute the role of identity provider $\mathtt{IP}$ among n identity servers $\{\mathtt{S}_1, \ldots, \mathtt{S}_n\}$ such that at any point a subset of these servers, any t of them, collectively verify users' passwords and generate authentication token for them. A $\mathtt{PbTA}$ system proposes a very strong *unforgeability* and *password-safety* properties with the goal of making password-based token generation secure against identity server breaches (up to $t-1$ at a time) that could compromise both long-term keys and

user credentials. This is enabled by requiring that any attacker who compromises at most $t-1$ identity servers cannot forge valid tokens or mount offline dictionary attacks. In [11] a generic framework called PASTA which can instantiate a secure PbTA system was also proposed. The PASTA framework uses as building blocks any threshold oblivious pseudorandom function (TOPRF) and any threshold token generation (TTG) scheme, i.e., a threshold digital signature.

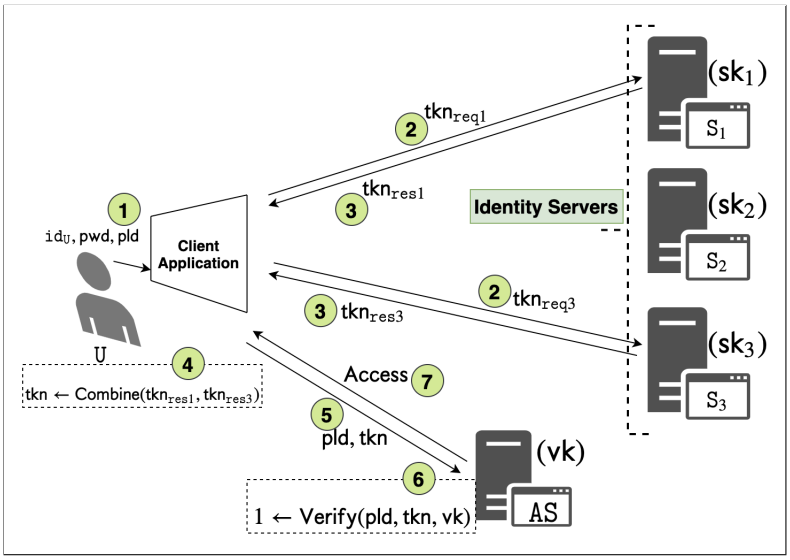

Figure 3: PbTA : 2-out-of-3 Token Generation

1.1 Our Contribution

The existing distributed token generation based SSO systems, including the PASTA framework, have so far not addressed the issue of password update functionality in their designs. The password update mechanism in a basic PbA is easy to implement and typically requires a user client to submit the hashes of both its current and a newly selected password. The current password hash is used for authentication by the identity provider before it replaces the current credentials with the updated credentials. Implementing password update mechanism in the setting of distributed token generation is not easy for the obvious reason that the identity provider is now split into several identity servers. This is particularly more problematic in the case of PASTA framework where no identity server holds the hashed user-passwords. In this work we propose to address this issue. The details are summarised as follows:

- We propose a password-update functionality to PASTA framework. The modified framework is now referred to as PAS-TA-U.
- We show how PASTA framework can be used to implement a distributed SSH key manager for enterprise networks (ESKM). The concept of an ESKM was first introduced by Harchol et al. in [12] and was instantiated therein using threshold cryptographic primitives. ESKM is a secure and fault-tolerant logically-centralised SSH key manager. The logically central key server of [12] was implemented in a distributed fashion using n key servers (also know as *control cluster* (CC) nodes) such that users in the enterprise network must interact with a subset of these CC nodes for a successful SSH connection. The problem of user authentication to CC nodes was also discussed in [12] and password-based variants were proposed for the same.
  The functionality of a password update mechanism for its users in an ESKM is an obvious requirement. As a concrete application, we implement PAS-TA-U to instantiate an ESKM that admits password update functionality for its users.
- The plain PASTA framework distributes the role of an identity provider IP among n identity servers $\{\mathtt{S}_1, \ldots, \mathtt{S}_n\}$. Consequently, the registration phase in PASTA involves a direct interaction between a user

and the n identity servers. We propose to keep only a user and the identity provider IP as primary participants in the registration phase. It is during the registration phase that a user is provided with a set of identity servers $\{\mathtt{S}_1, \ldots, \mathtt{S}_n\}$ by the IP. The user consequently reaches out to all identity servers with necessary information. In doing so we are keeping the identity provider in charge who may choose to select different identity servers for different users.

1.2 Organization

In Section 2, preliminaries required for PAS-TA-U framework are given. In § 2.1 GapTOMDH problem is given; the details on secret sharing are given in § 2.2; and the details on threshold token generation (TTG) and threshold oblivious pseudo-random function (TOPRF) are given in § 2.3 and § 2.4 respectively. The details on password-based threshold authentication (PbTA) are given in Section 3. The PAS-TA-U framework is given in Section 4. The security definitions for the underlying primitives make the basis for PbTA security definitions such as password-safety and unforgeability. As PAS-TA-U modifies PASTA frame directly to enhance it with a password functionality, we define password-update safety in Definition 11. The password-update safety of PAS-TA-U is directly reduced to the unforgeability requirement in § 4.1. In Section 5 we use PAS-TA-U to construct a distributed SSH key manager for enterprise networks. The implementation details are given in § 5.1.

2 Preliminaries

Notations. $\mathbb{N}$ denotes the set of natural numbers. $\mathbb{Z}_p$ denotes the finite field having elements $\{0, 1, , \ldots, p-1\}$ with addition and multiplication modulo the prime p. For a natural number n, we let $[n] = \{1, 2, \ldots, n\}$. For a set S, $a \xleftarrow{\$} S$ denotes that a is selected from S at random. We use PPT as a shorthand for probabilistic polynomial time and negl to denote negligible functions. The $\mathsf{GroupGen}$ be a PPT algorithm that on input a security parameter 1^λ, outputs $(p, g, \mathbb{G})$ where p is a λ bit prime, $\mathbb{G}$ is a multiplicative group of order p, and g is a generator of $\mathbb{G}$.

2.1 Hardness Assumption

For $q_1, \ldots, q_n \in \mathbb{N}$ and $t', t \in \mathbb{N}$ where $t' < t \leq n$, define

$$\mathsf{MAX}_{t',t} = \mathsf{max}\{\ell \mid \exists\ \hat{u}_1, \ldots, \hat{u}_\ell \in \{0,1\}^n \text{ s.t. } \mathsf{HW}(\hat{u}_i) = t - t' \text{ and } \sum_{i=1}^{\ell} \hat{u}_i \leq (q_1, \ldots, q_n)\} \tag{1}$$

where $\mathsf{HW}(\hat{u})$ denotes the number of 1's in the binary vector $\hat{u}$, and all operations on vectors are component-wise integer operations.

Definition 1 (GapTOMDH) *A cyclic group generator* $\mathsf{GroupGen}$ *satisfies the Gap Threshold One-More Diffie-Hellman (GapTOMDH) assumption if for all* t', t, n, r *where* $t' < t \leq n$ *and for all* PPT *adversary* $\mathcal{A}$ *in* $\mathsf{OneMore}_{\mathcal{A}}(1^\lambda, t', t, n, r)$ *game as defined in Fig. 4, there exists a negligible function* negl *s.t.*

$$\mathsf{Pr}[1 \leftarrow \mathsf{OneMore}_{\mathcal{A}}(1^\lambda, t', t, n, r)] \leq \mathsf{negl}(\lambda)$$

2.2 Shamir Secret Sharing

In the following we recall Shamir secret sharing scheme [13] and related concepts.

$\mathsf{OneMore}_{\mathcal{A}}(1^\lambda, t', t, n, r)$

- $(p, g, \mathbb{G}) \xleftarrow{\$} \mathsf{GroupGen}(1^\lambda)$
- $g_1, \ldots, g_r \xleftarrow{\$} \mathbb{G}$
- $(\{(i, \alpha_i)\}_{i \in \mathcal{B}}, \mathsf{st}) \leftarrow \mathcal{A}(p, g, \mathbb{G}, g_1, \ldots, g_r)$, where $B \subseteq [n]$, $|B| = t'$
- $(k_0, k_1, \ldots, k_n) \xleftarrow{\$} \mathsf{GenShare}(p, t, n, \{(i, \alpha_i)\}_{i \in B})$
- $q_i := 0$ for $i \in [n]$
- $((g'_1, h_1), \ldots, (g'_\ell, h_\ell)) \leftarrow \mathcal{A}^{\langle \mathcal{O} \rangle}(\mathsf{st})$
- output 1 $\iff$ $\begin{cases} \ell > \mathsf{MAX}_{t',t}(q_1, \ldots, q_n) \\ g'_i \in \{g_1, \ldots, g_r\} \text{ and } h_i = g_i'^{k_0} \ \forall\ i \in [\ell] \\ g'_i \neq g'_j \ \forall\ 1 \leq i \neq j \leq \ell \end{cases}$

- $\mathcal{O}(i, x)$
  - Input: $i \in [n] \backslash B$, $x \in \mathbb{G}$
  - $q_i := q_i + 1$
  - Output: return x^{k_i}
- $\mathcal{O}_{\mathsf{DDH}}(x_1, x_2, y_1, y_2)$
  - Input: $x_1, x_2, y_1, y_2 \in \mathbb{G}$
  - Output: return 1 $\iff$ $\mathsf{dlog}_{x_1} x_2 = \mathsf{dlog}_{y_1} y_2$

Figure 4: $\mathsf{OneMore}_{\mathcal{A}}(1^\lambda, t', t, n, r)$

Definition 2 (Lagrange Interpolation) *Let t be a positive integer and $\mathbb{F}$ be a field. Given any t pairs of field elements $(x_1, y_1), \ldots, (x_t, y_t)$ with distinct x_i's, there exists a unique polynomial $f(x) \in \mathbb{F}[x]$ of degree at most $t-1$ such that $f(x_i) = y_i$ for $1 \leq i \leq t$. The polynomial can be obtained using the Lagrange interpolation formula as follows,*

$$f(x) = y_1 \lambda^S_{x_1}(x) + \cdots + y_t \lambda^S_{x_t}(x), \tag{2}$$

where $S = \{x_1, \ldots, x_t\}$ and $\lambda^S_{x_i}(x)$'s $(1 \leq i \leq t)$ are Lagrange basis polynomials, given by

$$\lambda^S_{x_i}(x) = \frac{\prod_{1 \leq j \leq t, j \neq i}(x - x_j)}{\prod_{1 \leq j \leq t, j \neq i}(x_i - x_j)} \tag{3}$$

Definition 3 (Shamir Secret Sharing) *Let $t, n \in \mathbb{N}$ such that $1 < t \leq n$. A (t, n)-threshold Shamir secret sharing allows a dealer $\mathcal{D}$ to share a secret among n participants $\{P_1, \ldots, P_n\}$ such that at t least of them are required to reconstruct the secret back, and any set of less than t participants will have no information on the secret. In particular, the scheme has two algorithms* $(\mathsf{ShamirShare}, \mathsf{ShamirRec})$ *and they work as follows:*

- ShamirShare. *On input a secret $\mathsf{sk} \in \mathbb{F}_q$ where $q > n$, the share generation algorithm* ShamirShare *outputs a list of n shares as follows.*
  - $f(x) \xleftarrow{\$} \mathbb{F}_q[x]$ *s.t.* $f(0) = \mathsf{sk}$ *and* $\mathsf{degree}(f) \leq t - 1$.
  - $\mathsf{sk}_i = f(i) \bmod p$, *for* $i \in [n]$.
  - *Each P_i gets sk_i, the ith share of* sk
- ShamirRec*: The secret reconstruction algorithm* ShamirRec *requires any t shares for correct secret reconstruction.*
  1. *Input:* $\{\mathsf{sk}_i\}_{i \in S}$, *where* $S \subseteq [n]$ *and* $|S| = t$
  2. *Output:* $\mathsf{sk} = \sum_{i \in S} \mathsf{sk}_i \lambda^S_i(0)$

2.3 Threshold Token Generation (TTG)

A threshold token generation scheme essentially refers to a threshold signature scheme. In a traditional public-key signature scheme, we have a signer who holds a secret signing key and a verifier who holds a public verification key. In a (t, n)-threshold signature scheme, where $t \leq n$, the secret key is split into n signers $\{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ such that at least t these parties must come together to sign a message. The collective signing is done in a way such that at no stage the secret key is revealed.

Definition 4 (Threshold Token Generation (TTG)) *A threshold token generation scheme* Π *is a tuple* $(\mathsf{SetUp}, \mathsf{PartSign}, \mathsf{Combine}, \mathsf{Verify})$ *of four* PPT *algorithms and they work as follows:*

- $\mathsf{Setup}(1^\lambda, t, n)$ *The inputs to the setup algorithm are: a security parameter* λ*,* n *signing parties* $P_1, \ldots, P_n$*, and a number* $t \leq n$*. It proceeds as follows:*
  - *Generate a pair of singing and verification key* $(\mathsf{sk}, \mathsf{vk})$ *along with public parameters* pp
  - *Generate* n *shares* $(\mathsf{sk}_1, \ldots, \mathsf{sk}_n)$ *of the signing key* sk
  - *Each signing party* P_i *receives* $(\mathsf{sk}_i, \mathsf{pp})$
  - *The pair* $(\mathsf{vk}, \mathsf{pp})$ *is made publicly available.*
- $\mathsf{PartSign}(\mathsf{sk}_i, x)$ *This algorithm is run by individual signers. The* i*th signer* P_i *takes as input a message* x *and its share* sk_i*. It returns a partial signature* y_i *on* x*.*
- $\mathsf{Combine}(\{y_i\}_{i \in S})$*. This algorithm puts together* t *part signatures* $\{\mathsf{y}_i\}_{i \in S}$ *where* $|S| = t$ *and combine them to output a signature, also refer it as a token,* tk *on* x*.*
- $\mathsf{Verify}(\mathsf{vk}, \mathsf{tk}, x)$ *The verification algorithm takes as input the verification key* vk*, a token* tk*, and a message* x*. It outputs* 1 *if and only if* tk *is a valid signature on* x*.*

2.3.1 A TTG Construction

In the following we recall the famous TTG scheme due to Victor Shoup [6]. We begin by setting up a few notations. Let t, n be positive integers such that $1 < t \leq n$. Define $\Delta = n!$. In the following we now define modified Lagrange coefficients. The Lagrange coefficients are polynomials and the modified variants are defined as follows.

Definition 5 (Modified Lagrange Coefficients [6]) *Let* $S \subset \{1, 2, \ldots, n\}$*. For any* $j \in \{1, 2, \ldots, n\} \backslash S$*, define*

$$\tilde{\lambda}_j^S(x) = \Delta \lambda_j^S(x) = \Delta \cdot \frac{\prod_{k \in S \backslash \{j\}} (x - k)}{\prod_{k \in S \backslash \{j\}} (j - k)} \tag{4}$$

Clearly, $\tilde{\lambda}_j^S(x) \in \mathbb{Z}[x]$ for all $j \in \{1, 2, \ldots, n\} \backslash S$. In the following, we recall Lagrange interpolation over $\mathbb{Z}_N$ where N is an RSA modulus, i.e. $N = p \times q$ where p, q are distinct primes.

Theorem 1 ([6]) *Let* $f(x) = \sum_{k=0}^{t-1} a_k x^k \in \mathbb{Z}_N[X]$ *be of degree at most* $t - 1$*, where* $N > n$ *is an RSA modulus. Let* $\{(j, f(j))\}_{j \in S}$ *be* t *points over* $f(x)$*, where* $S \subseteq \{1, 2, \ldots, n\}$ *and* $\Delta = n!$*. Then,*

$$\Delta \cdot f(x) = \sum_{j \in S} f(j) \cdot \tilde{\lambda}_j^S(x) \bmod \phi(N) \tag{5}$$

In the following we now recall Shoup's [6] threshold signature, aka threshold token generation scheme.

- $\mathsf{Setup}(1^\lambda, t, n)$: The setup algorithm generates necessary parameters for the n signers as follows:

- Generate λ-bit primes p, q ($p \neq q$)
- $N = pq$
- $e, d \xleftarrow{\$} \mathbb{Z}_N$ s.t. $ed \equiv 1 \pmod{\phi(N)}$
- $(\mathsf{sk}_1, \ldots, \mathsf{sk}_n) \longleftarrow \mathsf{ShamirShare}(d, t, n)$
- Verification key $\mathsf{vk} = (e, N, H(\cdot))$, where $H : \{0,1\}^* \to \mathbb{Z}_N$ is a secure hash function.
- Each signer P_i receives sk_i.

- PartSign: A signer P_i runs this algorithm. It produces a partial signature on a message m using its secret share sk_i as follows:
  - Output: $y_i = H(m)^{2\Delta \mathsf{sk}_i} \bmod N$, where $\Delta = n!$
- $\mathsf{Combine}(\{y_i\}_{i \in S})$: The input to these algorithms are partial signatures $\{y_i\}_{i \in S}$ received from t signers, i.e. $|S| = t$ where $S \subseteq \{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$. This algorithm combines these partial signatures to obtain a full signature as follows.
  - Compute $w = \prod_{i \in S} y_i^{2\Delta \tilde{\lambda}_i^S(0)} \bmod N$
  - Find integers a, b such that $a4\Delta^2 + be = 1$
  - Compute $\sigma = w^a H(m)^b \bmod N$
  - Output (m, σ)
- Verify: The verification algorithm is used for checking signature validity. The input to this algorithm is a tuple (m, σ, vk) and it works as follows:
  - Output valid $\iff H(m) \stackrel{?}{=} \sigma^e \pmod N$.

Correctness: Note that, $w = \left(H(m)^{4\Delta^2 d}\right) \pmod N$. Finally,

$$\begin{aligned} \sigma &\equiv w^a H(m)^b \bmod N \\ &\equiv (H(m)^{4\Delta^2 d})^a (H(m)^{ed})^b \bmod N \\ &= (H(m)^d)^{4\Delta^2 a + eb} \bmod N \\ &= H(m)^d \bmod N \end{aligned}$$

2.4 Threshold Oblivious Pseudo-Random Function (TOPRF)

A pseudo-random function (PRF) family is a keyed family of deterministic functions. A function chosen at random from the family is indistinguishable from a random function. Oblivious PRF (OPRF) is an extension of PRF to a two-party setting where a server S holds the key and a party P holds an input. S can help P in computing the PRF value on the input but in doing so P should not get any other information and S should not learn P's input. Jarecki et al. [14] extend OPRF to a multi-server setting so that a threshold number t of the servers are needed to compute the PRF on any input. Furthermore, a collusion of at most $t-1$ servers learns no information about the input.

Definition 6 (Threshold Oblivious Pseudo-Random Function (TOPRF)) *A threshold oblivious pseudo-random function is a PRF* $\mathsf{TOP} : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ *along with a tuple of four* PPT *algorithms* (Setup, Encode, Eval, Combine) *described below. The server S is split into n servers $S_1, \ldots, S_n$. The party P must reach out to a subset of these servers to compute a* TOP *value.*

- $\mathsf{Setup}(1^\lambda, t, n) \to (\mathsf{sk}, \{\mathsf{sk}_i\}_{i=1}^n, \mathsf{pp})$. *It generates a secret key* sk*, n shares* $\mathsf{sk}_1, \ldots, \mathsf{sk}_n$ *of* sk*, and public parameters* pp*. Share* sk_i *is given to party S_i. The public parameters* pp *will be an implicit input in the algorithms below.*

– $\mathsf{Encode}(x, \rho)$. *It is run by the party* P. *It generates an encoding* c *of* x *using randomness* ρ.
– $\mathsf{Eval}(\mathsf{sk}_i, c)$ *It is run by a server* S_i. *It uses* sk_i *to generate a share* z_i *of TOPRF value* $\mathsf{TOP}(\mathsf{sk}, x)$ *from the encoding* c.
– $\mathsf{Combine}(x, \{(i, z_i)\}_{i \in S}, \rho)$ *It is run by the party* P. *It combines the shares received from any set of* t *servers* ($|S| = t$) *using randomness* ρ *to generate a value* h.

Correctness

For all $\lambda \in \mathbb{N}$, and $t, n \in \mathbb{N}$ where $t \leq n$, all $(\mathsf{sk}, \{\mathsf{sk}_i\}_{i=1}^n, \mathsf{pp}) \xleftarrow{\$} \mathsf{Setup}(1^\lambda, t, n)$, any value $x \in \mathcal{X}$, any randomness ρ, ρ', and any two sets $S, S' \subseteq [n]$ where $|S| = |S'| \geq t$, if $c = \mathsf{Encode}(x.\rho)$, $c' = \mathsf{Encode}(x.\rho')$, $z_i = \mathsf{Eval}(\mathsf{sk}_i, c)$ for $i \in S$, and $z'_j = \mathsf{Eval}(\mathsf{sk}_j, c')$ for $j \in S'$, then $\mathsf{Combine}(x, \{(i, z_i)\}_{i \in S}, \rho) = \mathsf{Combine}(x, \{(i, z'_j)\}_{j \in S'}, \rho')$.

2.4.1 Security

As given in [11], in the following we list the security properties that a `TOPRF` must satisfy.

Definition 7 (Unpredictability [11]) *A* `TOPRF` *is unpredictable if for all* $t, n \in \mathbb{N}$ *where* $t \leq n$, *and* PPT *adversary* $\mathcal{A}$ *in* $\mathsf{Unpredictability}_{\mathsf{TOP},\mathcal{A}}(1^\lambda, t, n)$ *game as defined in Fig. 5, there exists a negligible function* negl *s.t.*

$$\Pr[1 \leftarrow \mathsf{Unpredictability}_{\mathsf{TOP},\mathcal{A}}(1^\lambda, t, n)] \leq \frac{\mathsf{MAX}_{|\mathcal{C}|,t}(q_1, \ldots, q_n)}{|\mathcal{X}|} + \mathsf{negl}(\lambda)$$

$\mathsf{Unpredictability}_{\mathsf{TOP},\mathcal{A}}(1^\lambda, t, n)$

- $([[\mathsf{sk}]], \mathsf{pp}) \xleftarrow{\$} \mathsf{Setup}(1^\lambda, t, n)$
- $\mathcal{C} \leftarrow \mathcal{A}(\mathsf{pp})$
- $x^* \xleftarrow{\$} \mathcal{X}$
- $q_i := 0$ for $i \in [n]$
- $h^* \leftarrow \mathcal{A}^{\langle \mathcal{O} \rangle}(\{\mathsf{sk}_i\}_{i \in \mathcal{C}})$
- output $1 \iff \mathsf{TOP}(\mathsf{sk}, x^*) = h^*$

- $\mathcal{O}_{\mathsf{enc} \wedge \mathsf{eval}}()$
  - $c := \mathsf{Encode}(x^*, \rho)$ for a randomness ρ
  - $z_i \leftarrow \mathsf{Eval}(\mathsf{sk}_i, c)$ for $i \in [n] \backslash \mathcal{C}$
  - return c, $\{z_i\}_{i \in [n] \backslash \mathcal{C}}$
- $\mathcal{O}_{\mathsf{eval}}(i, c)$
  - require: $i \in [n] \backslash \mathcal{C}$
  - $q_i = q_i + 1$
  - return $\mathsf{Eval}(\mathsf{sk}_i, c)$
- $\mathcal{O}_{\mathsf{check}}(h)$
  - return $\begin{cases} 1 & \text{if } h = \mathsf{TOP}(\mathsf{sk}, x^*) \\ 0 & \text{otherwise} \end{cases}$

Figure 5: $\mathsf{Unpredictability}_{\mathsf{TOP},\mathcal{A}}(1^\lambda, t, n)$

Definition 8 (Obliviousness [11]) *A* `TOPRF` *is unpredictable if for all* $t, n \in \mathbb{N}$ *where* $t \leq n$*, and* PPT *adversary* $\mathcal{A}$ *in* $\mathsf{Obliviousness}_{\mathsf{TOP},\mathcal{A}}(1^\lambda, t, n)$ *game as defined in Fig. 6, there exists a negligible function* negl *s.t.*

$$\mathsf{Pr}[1 \leftarrow \mathsf{Obliviousness}_{\mathsf{TOP},\mathcal{A}}(1^\lambda, t, n)] \leq \frac{\mathsf{MAX}_{|\mathcal{C}|,t}(q_1, \ldots, q_n) + 1}{|\mathcal{X}|} + \mathsf{negl}(\lambda)$$

$\mathsf{Obliviousness}_{\mathsf{TOP},\mathcal{A}}(1^\lambda, t, n)$

- $([[\mathsf{sk}]], \mathsf{pp}) \xleftarrow{\$} \mathsf{Setup}(1^\lambda, t, n)$
- $\mathcal{C} \leftarrow \mathcal{A}(\mathsf{pp})$
- $x^* \xleftarrow{\$} \mathcal{X}$
- $q_i := 0$ for $i \in [n]$
- $x^{**} \leftarrow \mathcal{A}^{\langle \mathcal{O} \rangle}(\{\mathsf{sk}_i\}_{i \in \mathcal{C}})$
- output $1 \iff x^* = x^{**}$
-

- $\mathcal{O}_{\mathsf{enc} \wedge \mathsf{eval}}()$
  - $c := \mathsf{Encode}(x^*, \rho)$ for a randomness ρ
  - $z_i \leftarrow \mathsf{Eval}(\mathsf{sk}_i, c)$ for $i \in [n] \backslash \mathcal{C}$
  - $h := \mathsf{Combine}(x^*, \{(i, z_i)\}_{i \in [n]}, \rho)$
  - return c, h, $\{z_i\}_{i \in [n] \backslash \mathcal{C}}$
- $\mathcal{O}_{\mathsf{eval}}(i, c)$
  - require: $i \in [n] \backslash \mathcal{C}$
  - $q_i = q_i + 1$
  - return $\mathsf{Eval}(\mathsf{sk}_i, c)$

Figure 6: $\mathsf{Obliviousness}_{\mathsf{TOP},\mathcal{A}}(1^\lambda, t, n)$

2.4.2 A TOPRF Construction

In the following we recall a `TOPRF` construction called `2HashTDH` [14] that PASTA uses. The underlying PRF for `2HashTDH` is $H : \mathcal{X} \times \mathbb{Z}_p \to \{0,1\}^\lambda$, where $\mathcal{X}$ is the message space, and $\mathbb{Z}_p$ the key space. To define H, we require two hash functions: $H_1 : \mathcal{X} \times \mathbb{G} \to \{0,1\}^\lambda$, and $H_2 : \mathcal{X} \to \mathbb{G}$, where $(p, g, \mathbb{G}) \xleftarrow{\$} \mathsf{GroupGen}(1^\lambda)$. Finally, for $x \in \mathcal{X}$, and key $\mathsf{sk} \in \mathbb{Z}_p$, define

$$H(x, \mathsf{sk}) = H_1(x, H_2(x)^{\mathsf{sk}}).$$

The `TOPRF` evaluation for H is described below, where P holds a message $x \in \mathcal{X}$, and the secret key sk is split between n servers.

- $\mathsf{Setup}(1^\lambda, t, n)$.
  - $(p, g, \mathbb{G}) \xleftarrow{\$} \mathsf{GroupGen}(1^\lambda)$
  - $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_p$
  - $(\mathsf{sk}_1, \ldots, \mathsf{sk}_n) \xleftarrow{\$} \mathsf{ShamirShare}(\mathsf{sk}, t, n)$
  - Set $\mathsf{pp} = (p, g, \mathbb{G}, t, n, H_1, H_2)$.
  - Give $(\mathsf{sk}_i, \mathsf{pp})$ to each server S_i
- $\mathsf{Encode}(x)$. This is run by P

  - $\rho \xleftarrow{\$} \mathbb{Z}_p$
  - Output $c = H_2(x)^\rho$
- $\mathsf{Eval}(\mathsf{sk}_i, c)$. This is run by a server S_i
  - Output $z_i = c^{\mathsf{sk}_i}$
- $\mathsf{Combine}(x, \{(i, z_i)\}_{i \in S}, \rho)$. This is run by P
  - Output $\perp$ if $|S| \leq t - 1$ where $S \subseteq [n]$
  - Compute $z = \prod_{i \in S} z_i^{\lambda_i}$, where λ_i are Lagrange coefficients computed over the set S ($|S| = t$)
  - Output $H_1(x, z^{\rho^{-1}})$

`2HashTDH` Security

In the work of [11], it was shown that `2HashTDH` scheme as described above satisfies both Unpredictability and Obliviousness properties.

3 Password-based Threshold Authentication with Password Update

A password-based threshold authentication scheme allows a set of related application servers to provide a seamless single sign-on (SSO) service to its users. The system components include:

- a primary identity provider `IP`,
- n identity servers $\{\mathtt{S}_1, \ldots, \mathtt{S}_n\}$ managing, between them, user's credentials,
- application servers `AS`'s,
- users who want to gain access to all related yet independent applications servers `AS`s without the need for them to repeatedly prove their identities to these servers and hold different credentials for each of them, and
- a client application that facilitate user's interaction with the `IP`, identity servers $\mathtt{S}_i$'s, and application servers `AS`'s.

The algorithms for a `PbTA` system with password-update functionality are given below, along with their workflow.

- $\mathsf{GlobalSetup}(1^\lambda)$: In this phase the primary identity provider `IP`, on input a system wide security parameter λ, number of identity servers n, and a threshold parameter t ($1 < t \leq n$), generates a pair of signing and verification key $(\mathsf{sk}, \mathsf{vk})$ along with system public parameters pp. The `IP` then chooses and initialises n identity servers $\{\mathtt{S}_1, \ldots, \mathtt{S}_n\}$ with the necessary parameters. In particular,
  - each identity server $\mathtt{S}_i$ is given a secret share sk_i of the signing secret key sk and the public parameters pp,
  - each application server `AS` is given the corresponding master verification key vk and the public parameters pp.

  The GlobalSetup phase also initialises user client application with the necessary public parameters.
- Registration: This one-time phase allows a user `U` to open an account and register its user credentials with the n identity servers $\{\mathtt{S}_1, \ldots, \mathtt{S}_n\}$ *by interacting with the identity provider* `IP`. It begins with the `U` choosing a pair of identity and password $(\mathtt{id}_\mathtt{U}, \mathtt{pwd})$, and it ends with the each server $\mathtt{S}_i$ receiving and storing a distinct user record $\mathsf{rec}_{i,\mathtt{U}}$ in its record database REC_i.

- TokenGeneration: The token generation algorithm is an interactive algorithm run between a user $\mathtt{U}$'s client application and a set of at t identity servers, say $\{\mathtt{S}_1, \ldots, \mathtt{S}_t\}$.
  - TG.Request. Run by a client application.
    - **Input** - $(\mathtt{id}_\mathtt{U}, \mathsf{pwd}, \mathsf{pld}, \mathcal{T})$. As an input it takes a user identity $\mathtt{id}_\mathtt{U}$, a user password pwd, a payload pld, and a set $\mathcal{T} \subseteq [n]$.
    - **Output** - $(\mathtt{st}_{\mathsf{pld}}, \{(\mathtt{id}_\mathtt{U}, \mathsf{pld}, \mathsf{req}_i)\}_{i\in\mathcal{T}})$. It outputs a secret state $\mathtt{st}_{\mathsf{pld}}$ and request messages $\{(\mathsf{req}_i)\}_{i\in\mathcal{T}}$. For $i \in \mathcal{T}$, $(\mathtt{id}_\mathtt{U}, \mathsf{pld}, \mathsf{req}_i)$ is sent to $\mathtt{S}_i$.
  - TG.Respond. Run by an identity server $\mathtt{S}_i$.
    - **Input** - $(\mathsf{sk}_i, \mathsf{REC}_i, (\mathtt{id}_\mathtt{U}, \mathsf{pld}, \mathsf{req}_i))$. As an input it takes a secret key share sk_i, a record set REC_i, a user identity $\mathtt{id}_\mathtt{U}$, a payload pld and a request message req_i.
    - **Output** - res_i. The output is a response message res_i which is sent back to the connecting client.
  - TG.Finalize. Run by a client application.
    - **Input** - $(\mathtt{st}_{\mathsf{pld}}, \{\mathsf{res}_i\}_{i\in\mathcal{T}})$. As input it takes a secret state $\mathtt{st}_{\mathsf{pld}}$ and response messages $\{\mathsf{res}_i\}_{i\in\mathcal{T}}$.
    - **Output** - $(\mathsf{tk}_{\mathsf{pld}})$. The output is a token $(\mathsf{tk}_{\mathsf{pld}})$ for the payload pld. The token will be used by $\mathtt{U}$ to gain access to $\mathtt{AS}$ services.
- Verification: The verification algorithm checks the validity of a tk. The inputs to the verification algorithm are: the master verification key vk, a payload pld, and a token $\mathsf{tk}_{\mathsf{pld}}$. The verification algorithm is run by the $\mathtt{AS}$ services every time they receive a login request.
- PasswordUpdate It allows users to update their passwords. The interactive algorithm PasswordUpdate is run between a user $\mathtt{U}$ and all identity servers, $\{\mathtt{S}_1, \ldots, \mathtt{S}_n\}$. $\mathtt{U}$'s inputs are its current password pwd and a new password pwd'. The algorithm ends with the each server $\mathtt{S}_i$ updating their respective $\mathtt{rec}_{i,\mathtt{U}}$ for $\mathtt{U}$. The updated records are now valid against the new password pwd'. It is important to note that the PasswordUpdate is not run between $\mathtt{U}$ and the application server $\mathtt{AS}$.

3.1 Security [11]

In the following we recall the security game $\mathtt{SecGame}_{\Pi,\mathcal{A}}(1^\lambda, t, n)$ for a password-based threshold authentication scheme Π with password update as given in [11].

$\mathtt{SecGame}_{\Pi,\mathcal{A}}(1^\lambda, t, n)$

- $(\mathsf{pp}, \mathsf{vk}, (\mathsf{sk}_1, \ldots, \mathsf{sk}_n)) \xleftarrow{\$} \mathsf{GlobalSetup}(1^\lambda, t, n)$
- $(\mathcal{C}, \mathtt{U}^*, \mathsf{st}_\mathcal{A}) \xleftarrow{\$} \mathcal{A}(\mathsf{pp})$ *# $\mathcal{C}$: corrupt servers, $\mathtt{U}^*$: target user*
- $\mathcal{V} := \phi$ *# set of corrupt users*
- $\mathsf{PwdList} := \phi$ *# list of $(\mathtt{id}_\mathtt{U}, \mathsf{pwd})$ pairs, indexed by $\mathtt{id}_\mathtt{U}$*
- $\mathsf{ReqList}_{\mathtt{U},i} := \phi$ for $i \in [n]$ *# token requests $\mathtt{U}$ makes to $\mathtt{S}_i$*
- $\mathsf{ct} := 0$, $\mathsf{LiveSessions} = []$ *# LiveSessions is indexed by ct*
- $\mathsf{TokList} := \phi$ *# list of tokens generated through $\mathcal{O}_{\mathsf{final}}$*
- $Q_{\mathtt{U},i} := 0$ for all $\mathtt{U}$ and $i \in [n]$
- $Q_{\mathtt{U},\mathsf{pld}} := 0$ for all $\mathtt{U}$ and pld
- $\mathsf{out} \leftarrow \mathcal{A}^{\langle\mathcal{O}\rangle}(\{\mathsf{sk}_i\}_{i\in\mathcal{C}}, \mathsf{st}_\mathcal{A})$

$\mathcal{O}_{\mathtt{corrupt}}(\mathtt{U})$

- Input: $\mathtt{id}_\mathtt{U}$
- $\mathcal{V} := \mathcal{V} \cup \{\mathtt{id}_\mathtt{U}\}$

– Output: if $(\mathtt{id}_{\mathtt{U}}, *) \in \mathsf{PwdList}$, return $\mathsf{PwdList}[\mathtt{id}_{\mathtt{U}}]$

$\mathcal{O}_{\mathtt{register}}(\mathtt{U})$

– Input: $\mathtt{id}_{\mathtt{U}}$, such that $\mathsf{PwdList}[\mathtt{id}_{\mathtt{U}}] = \perp$
– $\mathsf{pwd} \xleftarrow{\$} \mathbb{P}$
– $\mathsf{PwdList} := \mathsf{PwdList} \cup \{(\mathtt{id}_{\mathtt{U}}, \mathsf{pwd})\}$
– $(\mathsf{rec}_{1,\mathtt{U}}, \ldots, \mathsf{rec}_{n,\mathtt{U}}) \xleftarrow{\$} \mathsf{Register}(\mathtt{id}_{\mathtt{U}}, \mathsf{pwd})$
– $\mathsf{REC}_i := \mathsf{REC}_i \cup \{\mathsf{rec}_{i,\mathtt{U}}\}$ for $i \in [n]$

$\mathcal{O}_{\mathtt{req}}(\mathtt{U}, \mathsf{pld}, \mathcal{T})$

– Input: User identity $\mathtt{id}_{\mathtt{U}}$ with $\mathsf{PwdList}[\mathtt{id}_{\mathtt{U}}] \neq \perp$, payload pld, and a set $\mathcal{T} \subseteq [n]$.
– $(\mathsf{st}_{\mathsf{pld}}, \{\mathsf{req}_i\}_{i\in\mathcal{T}}) \xleftarrow{\$} \mathsf{TG.Request}(\mathtt{id}_{\mathtt{U}}, \mathsf{PwdList}[\mathtt{id}_{\mathtt{U}}], \mathsf{pld}, \mathcal{T})$
– $\mathsf{LiveSessions}[\mathsf{ct}] := \mathsf{st}_{\mathsf{pld}}$
– $\mathsf{ReqList}_{\mathtt{U},i} = \mathsf{ReqList}_{\mathtt{U},i} \cup \{\mathsf{req}_i\}$ for $i \in \mathcal{T}$
– $\mathsf{ct} = \mathsf{ct} + 1$
– Output: return $\{\mathsf{req}_i\}_{i\in\mathcal{T}}$

$\mathcal{O}_{\mathtt{resp}}(i, \mathtt{U}, \mathsf{pld}, \mathsf{req}_i)$

– Input: The ith identity server is provided with a user identity $\mathtt{id}_{\mathtt{U}}$, payload pld, and a req_i for token generation
– $\mathsf{res}_i \leftarrow \mathsf{TG.Respond}(\mathsf{sk}_i, \mathsf{REC}_i, (\mathtt{id}_{\mathtt{U}}, \mathsf{pld}, \mathsf{req}_i))$
– $Q_{\mathtt{U},i} := Q_{\mathtt{U},i} + 1$ if $\mathsf{req}_i \notin \mathsf{ReqList}_{\mathtt{U},i}$
– $Q_{\mathtt{U},\mathsf{pld}} := Q_{\mathtt{U},\mathsf{pld}} + 1$
– Output: return res_i

$\mathcal{O}_{\mathtt{final}}(\mathsf{ct}, \{\mathsf{res}_i\}_{\mathcal{S}})$

– Input: It is presented with a counter leading to a specific state, and identity server partial responses to a token generation request
– $\mathsf{st}_{\mathsf{pld}} := \mathsf{LiveSessions}[\mathsf{ct}]$
– $\mathsf{tk} := \mathsf{TG.Finalize}(\mathsf{st}_{\mathsf{pld}}, \{\mathsf{res}_i\}_{i\in\mathcal{S}})$
– $\mathsf{TokList} := \mathsf{TokList} \cup \{\mathsf{tk}\}$
– Output: return tk

Definition 9 (Password Safety[11]) *A PbTA scheme Π is password safe if all $t, n \in \mathbb{N}$, $(t \leq n)$, all password space $\mathbb{P}$ and all* PPT *adversary $\mathcal{A}$ in $\mathsf{SecGame}_{\Pi,\mathcal{A}}(1^\lambda, t, n)$, there exists a negligible function* negl *such that*

$$\Pr[(\mathcal{U}^* \notin \mathcal{V}) \wedge (\mathsf{out} = \mathsf{PwdList}[id_{\mathcal{U}^*}] \neq \perp)] \leq \frac{\mathsf{MAX}_{|\mathcal{C}|,t}(Q_{\mathcal{U}^*,1}, \ldots, Q_{\mathcal{U}^*,n}) + 1}{|\mathbb{P}|} + \mathsf{negl}(\lambda)$$

Definition 10 (Unforgeability[11]) *A PbTA scheme is unforgeable if for all $t, n \in \mathbb{N}$, $(t \leq n)$, all password space $\mathbb{P}$ and all* PPT *adversary $\mathcal{A}$ in $\mathsf{SecGame}_{\Pi,\mathcal{A}}(1^\lambda, t, n)$, there exists a negligible function* negl *such that*

– *if $Q_{\mathcal{U}^*,\mathsf{pld}^*} < t - |\mathcal{C}|$,*

$$\Pr[\mathsf{Verify}(\mathsf{vk}, id_{\mathcal{U}^*}, \mathsf{pld}^*, \mathsf{tk}^*) = 1] \leq \mathsf{negl}(\lambda)$$

– *else*

$$\Pr[(\mathcal{U}^* \notin \mathcal{V}) \wedge (\mathsf{tk}^* \notin \mathsf{TokList}) \wedge (\mathsf{Verify}(\mathsf{vk}, id_{\mathcal{U}^*}, \mathsf{pld}^*, \mathsf{tk}^*) = 1)] \leq \frac{\mathsf{MAX}_{|\mathcal{C}|,t}(Q_{\mathcal{U}^*,1}, \ldots, Q_{\mathcal{U}^*,n})}{|\mathbb{P}|} + \mathsf{negl}(\lambda)$$

*where $\mathcal{A}$'s output "*out*" in the $\mathsf{SecGame}_{\Pi,\mathcal{A}}(1^\lambda, t, n)$ is parsed as $(\mathsf{pld}^*, \mathsf{tk}^*)$.*

Definition 11 (Password Update Safety) *A PbTA scheme is password update safe if for all $t, n \in \mathbb{N}$, $(t \leq n)$, all password space $\mathbb{P}$ and all* PPT *adversary $\mathcal{A}$ in* $\mathsf{SecGame}_{\Pi,\mathcal{A}}(1^\lambda, t, n)$*, there exists a negligible function* negl *such that*

$$\mathsf{Pr}[(\mathcal{U}^* \notin \mathcal{V}) \wedge (i \in [n] \backslash \mathcal{C}) \wedge (F_i \text{ is true})] \leq \mathsf{negl}(\lambda)$$

*where $\mathcal{A}$'s output "*out*" in the* $\mathsf{SecGame}_{\Pi,\mathcal{A}}(1^\lambda, t, n)$ *is parsed as* $(i, \mathsf{rec}^*_{i,\mathcal{U}^*})$*, and F_i denotes the event that $\mathcal{S}_i$ updates the existing record* $\mathsf{rec}_{i,\mathsf{U}^*}$ *by replacing it with* $\mathsf{rec}^*_{i,\mathcal{U}^*}$*.*

4 PAS-TA-U: PASTA with Password Update

In the following we present PAS-TA-U framework. It is obtained by providing a password update functionality to the PASTA framework of [11]. Additionally, the registration phase in PAS-TA-U proposes, unlike the registration phase in plain PASTA, to keep a user and an identity provider IP as the primary participants. It is during the registration phase that a user is provided with a set of identity servers $\{\mathtt{S}_1, \ldots, \mathtt{S}_n\}$ by the IP. The user consequently reaches out to all identity servers with the necessary information. In doing so we are keeping the identity provider in charge who may choose to select different identity servers for different users. The user uses a client application for all its interactions with the IP, identity servers and application servers.

The other algorithms in PAS-TA-U framework remain exactly as described in plain PASTA. The underlying cryptographic primitives used are:

1. a threshold token generation scheme:

   $\mathsf{TTG} = (\mathsf{TTG.Setup}, \mathsf{TTG.PartSign}, \mathsf{TTG.Combine}, \mathsf{TTG.Verify})$,

2. a threshold oblivious pseudo-random function:

   $\mathsf{TOP} = (\mathsf{TOP.Setup}, \mathsf{TOP.Encode}, \mathsf{TOP.Eval}, \mathsf{TOP.Combine})$,

3. a symmetric-key encryption scheme: $\mathsf{SKE} = (\mathsf{SKE.Enc}, \mathsf{SKE.Dec})$, and a hash function H.

- GlobalSetup: Both servers and clients are initialised with their respective setup parameters in the GlobalSetup phase:
  - TTG.Setup: Each server S_i, $1 \leq i \leq n$, receives $(\mathsf{sk}_i, \mathsf{pp}_{\mathtt{ttg}})$, where

    $$(\mathsf{pp}_{\mathtt{ttg}}, [[\mathsf{sk}]] = (\mathsf{sk}_1, \ldots, \mathsf{sk}_n), \mathsf{vk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{TTG.Setup}(1^{\lambda_1}, n, t)$$

    The verification key vk (along with $\mathsf{pp}_{\mathtt{ttg}}$) is made available to all services managed by the AS. The master signing key sk is discarded.
  - TOP.Setup: A client generates and saves $\mathsf{pp}_{\mathtt{top}}$, where

    $$\mathsf{pp}_{\mathtt{top}} = (p, g, \mathbb{G}, \mathcal{P}, H_1 : \mathcal{P} \times \mathbb{G} \to \{0,1\}^{\lambda_2}, H_2 : \mathcal{P} \to \mathbb{G}) \xleftarrow{\$} \mathsf{TOP.Setup}(1^{\lambda_2})$$

- Registration: The registration phase is used by a user U to create its login credentials. This involves a user client, an identity provider IP and n identity servers selected by the IP. The registration is a three-step process: In step1, the user client reaches out to the identity provider IP; in step2, IP responds to the client and also reaches out to all identity servers; and finally in step3, the user client reaches out to all identity servers. The details for each of these steps are as follows:
  - **Step1** ($\mathtt{U} \to \mathtt{IP}$): The user credentials for a user client U is a pair of user identity and a password $(\mathsf{id}_U, \mathsf{pwd})$. The registration begins with U computing and sharing the following with the IP:

* $r \xleftarrow{\$} \mathbb{Z}_p^*$.
* $\mathsf{reg.id_U} = H_2(\mathsf{pwd})^r$
* Send $(\mathsf{id_U}, \mathsf{reg.id_U})$ to $\mathtt{IP}$

– **Step2** ($\mathtt{IP} \rightarrow \{\mathtt{U}, \{\mathtt{S}_i\}_{i=1}^n\}$): Upon receiving $(\mathsf{id_U}, \mathsf{reg.id_U})$ from $\mathtt{U}$, the identity provider $\mathtt{IP}$ responds as follows:

* $\mathtt{IP} \rightarrow \mathtt{U}$
  · $k \xleftarrow{\$} \mathbb{Z}_p^*$
  · $\mathsf{res.id_U} = (H_2(\mathsf{pwd})^r)^k$
  · Sends back $\mathsf{res.id_U}$ to $\mathtt{U}$
* $\mathtt{IP} \rightarrow \{\mathtt{S}_i\}_{i=1}^n$
  · $(k_1, \ldots, k_n) \xleftarrow{\$} \mathsf{ShamirShare}(k, t, n)$
  · Sends $(\mathsf{id_U}, k_i)$ to $\mathtt{S}_i$, $1 \leq i \leq n$,

– **Step3** ($\mathtt{U} \rightarrow \{\mathtt{S}_i\}_{i=1}^n$): Upon receiving the response $\mathsf{res.id_U}$ from $\mathtt{IP}$, $\mathtt{U}$ reaches out to identity servers with the following to complete the registration.

* $H_2(\mathsf{pwd})^k = (\mathsf{res.id_U})^{r^{-1}}$
* $h = \mathsf{TOP}_k(\mathsf{pwd}) = H_1(\mathsf{pwd}, H_2(\mathsf{pwd}^k))$
* $h_i = H(h \| i)$, $1 \leq i \leq n$
* Sends $(\mathsf{id_U}, h_i)$ to $\mathtt{S}_i$, $1 \leq i \leq n$.

– Each $\mathtt{S}_i$, $1 \leq i \leq n$, make a record of $\mathtt{U}$'s registration credentials by storing $[\mathsf{id_U}, (h_i, k_i)]$.

- TokenGeneration: This is run is by a user $\mathtt{U}$ to generate tokens (signed messages) used for accessing application services. As an input it takes a payload pld on which a token (signature) will be generated eventually and $\mathtt{U}$'s pwd. Consequently, the user client reaches out to any t out of n identity servers with the token generation request. Upon receiving all t responses, they are combined to produce a valid token.

  – $\mathsf{TG.Request}$ ($\mathtt{U} \rightarrow \{\mathtt{S}_i\}_{i \in \mathcal{T}}$):
    * $\rho \in \mathbb{Z}_p^*$
    * $\mathsf{req_{ttg}} = \mathsf{TOP.Encode}(\mathsf{pwd}, \rho) = H_2(\mathsf{pwd})^\rho$
    * Sends $(\mathsf{id_U}, \mathsf{pld}, \mathsf{req_{ttg}})$ to a set of t identity servers, without loss of generality, say $\mathtt{S}_1, \ldots \mathtt{S}_t$.
  – $\mathsf{TG.Respond}$ ($\mathtt{S}_i \rightarrow \mathtt{U}$): Upon receiving $(\mathsf{id_U}, \mathsf{pld}, \mathsf{req_{ttg}})$, the ith identity server $\mathtt{S}_i$ retrieves the corresponding record $[\mathsf{id_U}, (h_i, k_i)]$ and responds as follows:
    * $z_i = \mathsf{TOP.Eval}(k_i, \mathsf{req_{ttg}}) = (\mathsf{req_{ttg}})^{k_i} = H_2(\mathsf{pwd})^{\rho k_i}$
    * $y_i = \mathsf{TTG.PartSign}(\mathsf{sk}_i, \mathsf{pld} \| \mathsf{id_U})$
    * $v_i = \mathsf{SKE.Enc}(h_i, y_i)$
    * Return $\mathtt{ttg.res}_i = (z_i, v_i)$ to $\mathtt{U}$
  – $\mathsf{TG.Finalize}$: The received responses $\{\mathtt{ttg.res}_i = (z_i, v_i) \mid 1 \leq i \leq t\}$ are combined by the user client to produce a token on the payload pld as follows:
    * $\theta = H_2(\mathsf{pwd})^{k\rho} = \prod_{i=1}^t z_i^{\lambda_i}$
    * $h = \mathsf{TOP}_k(\mathsf{pwd}) = H_1(\mathsf{pwd}, H_2(\mathsf{pwd}^k)) = H_1(\mathsf{pwd}, \theta^{\rho^{-1}})$
    * $h_i = H(h \| i)$, $1 \leq i \leq t$
    * $y_i = \mathsf{SKE.Dec}(h_i, v_i)$, $1 \leq i \leq t$
    * $\mathsf{tk_{pld}} = \mathsf{TTG.Combine}(\{y_i\}_{i=1}^t)$

- Verification: The token verification algorithm is run by application services every time they receive a token $\mathsf{tk_{pld}}$ on a payload pld. The access to the service is allowed only if $1 \xleftarrow{\$} \mathsf{TTG.Verify}(\mathsf{pld}, \mathsf{tk_{pld}}, \mathsf{vk})$.

- PasswordUpdate: This is run by a user $\mathtt{U}$, every time it wants to update its current password pwd and set it to a new password pwd'. The process requires $\mathtt{U}$ to run TOP evaluations twice: once on the input pwd and later on pwd'. As TOP is implicitly built into the $\mathsf{TokenGeneration}$ algorithm, $\mathtt{U}$ therefore runs the later twice (note that payload is not important here). In the final step $\mathtt{U}$ runs the $\mathsf{TokenGeneration}$ algorithm for the third time on a special payload. This token is finally shared with all identity servers, who then use it to update their respective records associated with $\mathtt{U}$. The details are as follows:
  - **Step1**:
    - $\{\mathtt{ttg}.\mathsf{res}_i = (z_i, v_i) \mid 1 \leq i \leq t\} \xleftarrow{\$} \mathsf{TG.Respond}(\mathsf{TG.Request}(\mathsf{pwd}, \mathsf{pld}_1))$
    - Use $\{z_i\}_{i=1}^t$ to compute $h = H_1(\mathsf{pwd}, H_2(\mathsf{pwd})^k)$
    - Compute $h_i = H(h\|i), 1 \leq i \leq n$
  - **Step2**:
    - $\{\mathtt{ttg}.\mathsf{res}_i = (z'_i, v'_i) \mid 1 \leq i \leq t\} \xleftarrow{\$} \mathsf{TG.Respond}(\mathsf{TG.Request}(\mathsf{pwd}', \mathsf{pld}_2))$
    - Use $\{z'_i\}_{i=1}^t$ to compute $h' = H_1(\mathsf{pwd}', H_2(\mathsf{pwd}')^k)$
    - Compute $h'_i = H(h'\|i), 1 \leq i \leq n$
  - **Step3**:
    - Set $\mathsf{pld}_3 = \mathsf{SKE.Enc}(h_1, h_1||h'_1)\| \cdots \|\mathsf{SKE.Enc}(h_n, h_n||h'_n)\|\mathsf{id}_\mathtt{U}$
    - Compute $\mathsf{tk}_{\mathsf{pld}_3} \xleftarrow{\$} \mathsf{TokenGeneration}(\mathsf{pwd}, \mathsf{pld}_3)$ (the current password is used)
  - **Step4**: The user client for $\mathtt{U}$ finally sends $(\mathsf{pld}_3, \mathsf{tk}_{\mathsf{pld}_3})$ to all identity servers for password update.
  - PasswordUpdate: Upon receiving a token $(\mathsf{pld}_3, \mathsf{tk}_{\mathsf{pld}_3})$ requesting password update, each identity server S_i proceeds as follows:
    - S_i ensures that $\mathsf{TTG.Verify}(\mathsf{pld}_3, \mathsf{tk}_{\mathsf{pld}_3}, \mathsf{vk})$ returns 1
    - Extract $L_i = \mathsf{SKE.Enc}(h_i, h_i\|h'_i)$ from pld_3
    - Compute $h_i\|h'_i = \mathsf{SKE.Dec}(h_i, L_i)$
    - Update the record for $\mathtt{id}_\mathtt{U}$ as $[\mathtt{id}_\mathtt{U}, h'_i, k_i]$ (replacing h_i by h'_i)

4.1 Security: Password Update

In the following we discuss password update security for `PAS-TA-U` as per the definition 11. A successful password update in `PAS-TA-U` requires a user to obtain a valid token using the existing credentials. Consequently, an unauthorised password update is directly reduced to token forgery. The following theorem provides the necessary details.

Theorem 2 *For all $t, n \in \mathbb{N}$, $(t \leq n)$, all password space $\mathbb{P}$ and all* PPT *adversary $\mathcal{A}$ in $\mathsf{SecGame}_{\Pi,\mathcal{A}}(1^\lambda, t, n)$ (as given in Section 3.1), there exists a* PPT *adversary B in $\mathsf{SecGame}_{\Pi,\mathcal{A}}(1^\lambda, t, n)$ such that*

$$\Pr[(\mathtt{U}^* \notin \mathcal{V}) \wedge (i \in [n] \backslash \mathcal{C}) \wedge (F_i \text{ is true})] \leq \mathsf{Adv}^{\mathcal{B}}_{\mathsf{Unforgreability}}(\mathsf{SecGame}_{\Pi,\mathcal{B}}(1^\lambda, t, n)) \tag{6}$$

where Π denotes `PAS-TA-U`, $\mathcal{A}$'s output "out" in the $\mathsf{SecGame}_{\Pi,\mathcal{A}}(1^\lambda, t, n)$ is parsed as $(i, \mathsf{rec}^*_{i,\mathtt{U}^*})$, and F_i denotes the event that $\mathtt{S}_i$ updates the existing record $\mathsf{rec}_{i,\mathtt{U}^*}$ by replacing it with the new record $\mathsf{rec}^*_{i,\mathtt{U}^*}$. The $\mathsf{Adv}^{\mathcal{B}}_{\mathsf{Unforgreability}}(\mathsf{SecGame}_{\Pi,\mathcal{B}}(1^\lambda, t, n))$ denotes $\mathcal{B}$'s advantage in forging a token in the $\mathsf{SecGame}_{\Pi,\mathcal{B}}(1^\lambda, t, n)$.

Proof: Assume, $\mathcal{A} = \mathcal{A}_{\mathsf{pu}}$ be a PPT adversary who can make any identity server $\mathtt{S}_i$ to update a target user $\mathtt{U}^*$'s existing record $\mathsf{rec}_{i,\mathtt{U}^*}$ by a record $\mathsf{rec}^*_{i,\mathtt{U}^*}$. We then construct a PPT adversary $\mathcal{B} = \mathcal{A}_{\mathsf{f}}$ who can forge token's for arbitrary users. The $\mathcal{A}_{\mathsf{f}}$ runs $\mathcal{A}_{\mathsf{pu}}$ as a subroutine. The details are as follows.

Let $\mathtt{U}^*$ be the target user for $\mathcal{A}_f$ in the $\mathsf{SecGame}_{\Pi,\mathcal{A}_{\mathsf{f}}}(1^\lambda, t, n)$. Let $\mathcal{C} \subset [n]$ be the subset of identity servers that are corrupted by $\mathcal{A}_f$ such that $|\mathcal{C}| = t - 1$. Let $\mathsf{pwd}^{\mathsf{e}}_{\mathtt{U}^*}$ be the existing password for $\mathtt{U}^*$. $\mathcal{A}_f$ chooses an

identity server $\mathtt{S}_i$ such that $i \in [n] \backslash \mathcal{C}$ and a new password $\mathsf{pwd}^{\mathsf{n}}_{\mathtt{U}^*}$ for $\mathtt{U}^*$. $\mathcal{A}_f$ then makes a call to $\mathcal{A}_{\mathsf{pu}}$ and present it with the input $(i, \mathsf{pwd}^{\mathsf{n}}_{\mathtt{U}^*}, \mathsf{h}^*_i, \{\mathsf{rec}^{\mathsf{e}}_{j,\mathtt{U}^*}\}_{j \in \mathcal{C}})$ where $h^*_i = H(h^* \| i)$, $h^* = H_1(\mathsf{pwd}^{\mathsf{n}}_{\mathtt{U}^*}, H_2(\mathsf{pwd}^{\mathsf{n}}_{\mathtt{U}^*})^k)$. The h^* can be computed by $\mathcal{A}_{\mathsf{f}}/\mathcal{A}_{\mathsf{pu}}$ by reaching out to any t identity servers on inputs $\mathsf{id}_{\mathtt{U}^*}$, $\mathsf{pwd}^{\mathsf{n}}_{\mathtt{U}^*}$.

The adversary $\mathcal{A}_{\mathsf{pu}}$ subsequently mounts an attack on the target identity server $\mathtt{S}_i$. If successful, $\mathtt{S}_i$ would update its existing record $\mathsf{rec}_{i,\mathtt{U}^*} = (\mathsf{id}_{\mathtt{U}^*}, h_i, k_i)$ with the new record $\mathsf{rec}^*_{i,\mathtt{U}^*} = (\mathsf{id}_{\mathtt{U}^*}, h^*_i, k_i)$ where h^*_i is provided by $\mathcal{A}_{\mathsf{pu}}$.

On the other hand, $\mathcal{A}_{\mathsf{f}}$ by itself could update the records of each of the $t-1$ corrupted identity servers $\mathtt{S}_j$'s, $j \in \mathcal{C}$, with the respective h^*_j's where $h^*_j = H(h^* \| j)$, $h^* = H_1(\mathsf{pwd}^{\mathsf{n}}_{\mathtt{U}^*}, H_2(\mathsf{pwd}^{\mathsf{n}}_{\mathtt{U}^*})^k)$. Finally, let $\mathcal{C}' = \mathcal{C} \cup \{i\}$. As $|\mathcal{C}'| = t$, and as each identity server in $\mathcal{C}'$ now have the valid h^*_j's corresponding to the new password $\mathsf{pwd}^{\mathsf{n}}_{\mathtt{U}^*}$, $\mathcal{A}_{\mathsf{f}}$ can obtain valid tokens issued on identity $\mathsf{id}_{\mathtt{U}^*}$. As we see a direct reduction here in terms of the problem instance for $\mathcal{A}_{\mathsf{f}}$ being directly reduced to the problem instance for $\mathcal{A}_{\mathsf{pu}}$, this proves the claim of equation 6.

5 A Concrete Application of PAS-TA-U: Practical SSH key manager for enterprise networks

In the following We show how PAS-TA-U framework can be used to implement an SSH (Secure shell) key manager (ESKM) - an enterprise level solution for managing SSH keys. The concept of an ESKM was first introduced by Harchol et al. in [12] and was instantiated therein using threshold cryptographic primitives. ESKM is a secure and fault-tolerant logically-centralised SSH key manager whose advantages include: (1) the ability to observe, approve and log all SSH connections origination from the home network, and (2) clients do not need to store secret keys and instead only need to authenticate themselves to the ESKM system. The ESKM central key server of [12] was implemented in a distributed fashion using n key servers (also know as *control cluster* (CC) nodes) such that users in the enterprise network must interact with a subset of these CC nodes for a successful SSH connection. The problem of user authentication to CC nodes was also discussed in [12] and password-based variants were proposed for the same. The functionality of a password update mechanism for its users in an ESKM is an obvious requirement. As a concrete application, we implement PAS-TA-U to instantiate an ESKM that admits password update functionality for its users. The different entities in PAS-TA-U are mapped to ESKM components as follows: (1) the key manager in ESKM play the role of an identity provider IP, and n control nodes represent the identity servers $\{\mathtt{S}_1, \ldots, \mathtt{S}_n\}$, (2) enterprise network has SSH key pairs $(\mathsf{sk}, \mathsf{vk})$ and SSH servers storing vk represent the application servers AS, and (3) users in the enterprise network must obtain valid tokens by connecting to a subset of t ($t \leq n$) identity servers before they can gain access to these SSH servers. In the following we present a working flow of the system.

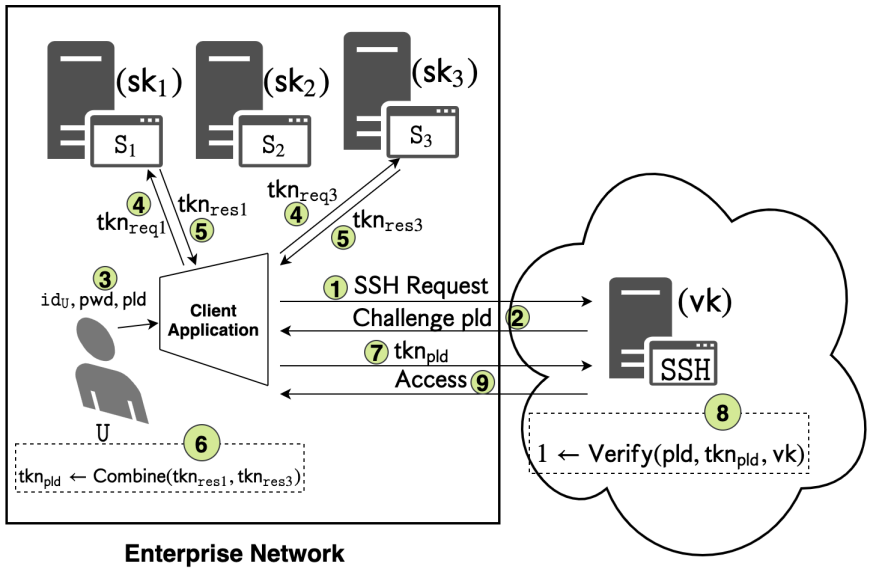

Figure 7: Distributed SSH key manager for enterprise networks

- **Step 1** The key manager (IP) runs the PAS-TA-U GlobalSetup to obtain $(\mathsf{sk}, \mathsf{vk})$ and n shares of $\{\mathsf{sk}_1, \ldots, \mathsf{sk}_n\}$ of sk. It submits vk with the application servers AS's (SSH servers). Finally, it sets up n identity servers $\{\mathtt{S}_1, \ldots, \mathtt{S}_n\}$ locally and initialises them with the respective shares: $\mathtt{S}_i \leftarrow \mathsf{sk}_i$.
- **Step 2** Users in the enterprise network run PAS-TA-U Registration with the IP to create and store credentials with the identity servers $\{\mathtt{S}_1, \ldots, \mathtt{S}_n\}$.
- **Step 3** This step shows how a registered user make an SSH connection into application servers. A user client initiates an SSH connection in to an application server AS. As part of user authentication, the SSH handshake phase presents to the client a challenge payload pld to be signed using the signing key sk. Consequently, the client runs the PAS-TA-U TokenGeneration on payload pld by reaching out to a set of t identity servers using U's password pwd. A set of t correct responses are combined to produce a signature token tk on the payload pld under the signing key sk. The generated tk is sent back to the application server which then verifies it using the stored verification key vk. A correctly generated token will give U an access to the application server.
- **Step 4** Our password update mechanism allows any user to update its password by reaching out to a set of any t identity servers directly. The updated matching user credentials are then uploaded to all identity servers.

5.1 ESKM Implementation

In this section we provide implementation details and consequently report on the performance of our instantiation of an ESKM using PAS-TA-U under practical network scenarios. We developed separate client applications for (a) users (b) identity provider IP, and (c) identity servers $\mathtt{S}_i$'s. All client applications are developed in Python. The entire source code for our implementation is hosted at https://anonymous.4open.science/r/198222d4-a335-4b93-a219-5a0bf5a4ec23/.

The cryptographic algorithms implemented for client applications include a hash function, Shamir secret sharing scheme, a symmetric-key encryption scheme, a TOPRF, and a TTG scheme. We used SHA-256 as our hash function. For symmetric key encryption, we used AES-128 with OFB mode. Both SHA-256 and AES-128 implementations were called from Python Cryptography library. For TOPRF, we implemented 2HashTDH scheme of Jarecki et al. [14]. For TTG, we implemented the scheme of Victor Shoup [6]. We used openssl library for 2HashTDH TOPRF and Shoup's TTG implementations.

The user client authentication in the SSH connection to the application server is done using public-key

based method. We used RSA signature scheme to generate a 2048 bits key-pair $(\mathsf{sk}, \mathsf{vk})$ and configure the application server with the vk. The `openssh` version used is 8.2p1 and it internally uses `openssl` version 1.1.1f. In the handshake phase of the SSH, the user client is presented with a challenge payload pld for it to be RSA signed using the secret signing key sk. The signing part is handled by the `libcrypto` library of `openssl`. In our case the secret key is not available with the client and instead it must reach out, using our python user client, to t identity servers for a threshold RSA signature generation (token) on the pld. To make this happen, we introduced a patch in `libcrypto` which can alter the usual signing flow. We replaced the secret key with an equivalent dummy key. If the `libcrypto`'s signing routine detects a dummy key, it calls our Python routine which computes signature by contacting t servers. Otherwise, the usual signing flow continues.

Experimental Setup: In order to evaluate the performance, we implemented various network settings described below. The experiments are run on a single laptop running Ubuntu 20.04 with 2-core 2.2 GHz Intel i5-5200U CPU and 8 GB of RAM. All identity providers are run on the same machine. We simulate the following network connections using the Linux `tc` command: (1) a local area network setting with 4 ms round-trip latency, and (2) an Internet setting with 80 ms round-trip latency.

Token Generation Time: Table 1 shows the total run-time for a user client to generate a single token in the `TokenGeneration` phase. We show experiments, both in the LAN and the Internet setting, for a different set of (t, n) parameters where n is the total number of identity servers and t $(t \leq n)$ is the threshold set of identity servers which are contacted by the user client. The reported time is an average of 100 token requests. The `Plain` setting in the table corresponds to the direct peer-to-peer client-server SSH connection (without the threshold token generation).

Setting	(t, n)	Time (ms)
LAN	`Plain`	15.1
	(2,3)	22.8
	(3,6)	25.1
	(5,10)	28.2
	(10,10)	32.5
Internet	`Plain`	101.3
	(2,3)	109.5
	(3,6)	112.4
	(5,10)	115.8
	(10,10)	119.9

Table 1: Token Generation Time for User Clients in `ESKM`

Password Update Time Table 2 shows the total run-time for the client to update password in our PbTA protocol. The reported time is an average of 100 password change requests.

Setting	(t, n)	Time (ms)
Local	(2,3)	65.3
	(3,6)	74.4
	(5,10)	82.7
	(10,10)	95.1
Internet	(2,3)	325.5
	(3,6)	333.1
	(5,10)	346.9
	(10,10)	357.6

Table 2: Password Update Time for User Clients in `ESKM`

6 Conclusion

We presented `PAS-TA-U` to propose a password-update functionality into `PASTA` - a generic framework for password-based threshold authentication (PbTA) schemes. Secure password update is a practical requirement for PbTA applications such as single-sign-on systems. Existing PbTA schemes, including the `PASTA` framework, do not admit password update functionality.

As a concrete application, we instantiated `PAS-TA-U` to implement a distributed enterprise SSH key manager that features a secure password-update functionality for its clients. Our experiments show that the overhead of protecting secret credentials against breaches in our system compared to a traditional single server setup is low (average 119 ms in a 10-out-of-10 server setting on Internet with 80 ms round trip latency). The entire source code for our implementation is hosted at `https://anonymous.4open.science/r/198222d4-a335-4b93-a219-5a0bf5a4ec23/`.

References

[1] Y. Desmedt and Y. Frankel, "Threshold cryptosystems," in *CRYPTO*, ser. LCNS, vol. 435. Springer, 1989, pp. 307–315.

[2] A. D. Santis, Y. Desmedt, Y. Frankel, and M. Yung, "How to share a function securely," in *ACM*. ACM, 1994, pp. 522–533.

[3] R. Gennaro, S. Halevi, H. Krawczyk, and T. Rabin, "Threshold RSA for dynamic and ad-hoc groups," in *EUROCRYPT*, ser. LCNS, vol. 4965. Springer, 2008, pp. 88–107.

[4] I. Damgård and M. Koprowski, "Practical threshold RSA signatures without a trusted dealer," in *EUROCRYPT*, ser. LCNS, vol. 2045. Springer, 2001, pp. 152–165.

[5] R. Gennaro, S. Goldfeder, and A. Narayanan, "Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security," in *ACNS*, ser. LCNS, vol. 9696. Springer, 2016, pp. 156–174.

[6] V. Shoup, "Practical threshold signatures," in *EUROCRYPT*, ser. LNCS, vol. 1807. Springer, 2000, pp. 207–220.

[7] P. D. MacKenzie, T. Shrimpton, and M. Jakobsson, "Threshold password-authenticated key exchange," in *CRYPTO*, ser. LCNS, vol. 2442. Springer, 2002, pp. 385–400.

[8] M. D. Raimondo and R. Gennaro, "Provably secure threshold password-authenticated key exchange," in *EUROCRYPT*, vol. 2656. Springer, 2003, pp. 507–523.

[9] M. Abdalla, P. Fouque, and D. Pointcheval, "Password-based authenticated key exchange in the three-party setting," in *PKC*, ser. LCNS, vol. 3386. Springer, 2005, pp. 65–84.

[10] K. G. Paterson and D. Stebila, "One-time-password-authenticated key exchange," in *ACISP*, ser. LCNS, vol. 6168. Springer, 2010, pp. 264–281.

[11] S. Agrawal, P. Miao, P. Mohassel, and P. Mukherjee, "PASTA: password-based threshold authentication," in *ACM CCS*. ACM, 2018, pp. 2042–2059.

[12] Y. Harchol, I. Abraham, and B. Pinkas, "Distributed SSH key management with proactive RSA threshold signatures," in *ACNS*, ser. LNCS, vol. 10892. Springer, 2018, pp. 22–43.

[13] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[14] S. Jarecki, A. Kiayias, H. Krawczyk, and J. Xu, "TOPPSS: cost-minimal password-protected secret sharing based on threshold OPRF," in *ACNS*, ser. LCNS, vol. 10355. Springer, 2017, pp. 39–58.