PASTA: PASsword-based Threshold Authentication

Shashank Agrawal[1], Peihan Miao*[2], Payman Mohassel[1], and Pratyay Mukherjee[1]

[1]Visa Research
{shaagraw,pmohasse,pratmukh}@visa.com
[2]University of California, Berkeley
peihan@berkeley.edu

Abstract

Token-based authentication is commonly used to enable a single-sign-on experience on the web, in mobile applications and on enterprise networks using a wide range of open standards and network authentication protocols: clients sign on to an identity provider using their username/password to obtain a cryptographic token generated with a master secret key, and store the token for future accesses to various services and applications. The authentication server(s) are single point of failures that if breached, enable attackers to forge arbitrary tokens or mount offline dictionary attacks to recover client credentials.

Our work is the first to introduce and formalize the notion of password-based threshold token-based authentication which distributes the role of an identity provider among n servers. Any t servers can collectively verify passwords and generate tokens, while no $t-1$ servers can forge a valid token or mount offline dictionary attacks. We then introduce PASTA, a general framework that can be instantiated using any threshold token generation scheme, wherein clients can "sign-on" using a two-round (optimal) protocol that meets our strong notions of unforgeability and password-safety.

We instantiate and implement our framework in C++ using two threshold message authentication codes (MAC) and two threshold digital signatures with different trade-offs. Our experiments show that the overhead of protecting secrets and credentials against breaches in PASTA, i.e. compared to a naïve single server solution, is extremely low (1-5%) in the most likely setting where client and servers communicate over the internet. The overhead is higher in case of MAC-based tokens over a LAN (though still only a few milliseconds) due to public-key operations in PASTA. We show, however, that this cost is inherent by proving a symmetric-key only solution impossible.

1 Introduction

Token-based authentication is arguably the most common way we obtain authorized access to resources, services, and applications on the internet and on enterprise networks.

Open standards such as JSON Web Token (JWT) [jwt] and SAML [sam] are widely used to facilitate single-sign-on authentication by allowing clients to initially sign on using a standard mechanism such as username/password verification to obtain and locally store a token in a cookie or the local storage. The token can then be used for all future accesses to various applications without client involvement, until it expires.

*Work done as an intern at Visa Research.

A similar mechanism is used, via open standards such as OAuth [oau] and OpenID [ope], by many companies including Google, Facebook and Amazon [goo, fac, ama] to enable their users to share information about their accounts with (or authenticate themselves to) third party applications or websites without revealing their passwords to them.

Finally, network authentication protocols such as Kerberos [ker] are commonly used by enterprises (e.g. Active Directory in Windows Servers) to, periodically but infrequently, authenticate clients with their credentials and issue them a ticket-granting ticket (TGT) that they can use to request access to various services on the enterprise network such as printers, internal web and more.

It is therefore no surprise that most software-based secret management systems provide tokens as a primary method for authenticating clients. For example, consider the following statement from the popular open source solution Vault by Hashicorp [vau]:

> "The token auth method is built-in and is at the core of client authentication. Other auth methods may be used to authenticate a client, but they eventually result in the generation of a client token managed by the token backend."

In all these cases, the authentication flow is effectively the same. A client signs on with its username/password, typically by sending hash of its password to an identity provider. The identity server who stores the username along with its hashed passwords as part of a registration phase, verifies the client's credential by matching the hash during the sign-on process before issuing an authentication token using a master secret key (see Figure 1). The token is generated by computing a digital signature or a message authentication code (all the above-mentioned standards support both digital signatures and MACs) on a message that can contain client's information/attributes, expiration time and a policy that would control the nature of access. The token is later verified by an application server which holds the verification key (for MACs this is equal to the master secret key). See Figure 2 for a sample JWT authenticated using HMAC [FIP02]. Note that the only secret known to the client is its password, and the device the client uses for access stores the temporary authentication token on its behalf. Besides this temporary (and often restricted) token, client devices do not store any long term secrets that are used to authenticate the client.

However, such an identity provider is a *single point of failure* that if breached, enables an attacker to (i) recover the master secret key and forge arbitrary tokens that enable access to arbitrary resources and information in the system and (ii) obtain hashed passwords to use as part of an offline dictionary attack to recover client credentials.

Security against server breach. We propose the notion of Password-based Threshold Authentication (PbTA), for distributing the role of the identity provider among n servers who collectively verify clients' passwords and generate authentication tokens for them (see Figure 3 for a generic flow). PbTA enables any t ($2 \leq t \leq n$) servers to authenticate the client and generate valid tokens while any attacker who compromises at most $t-1$ servers cannot forge valid tokens *or* mount offline dictionary attacks, thus providing very strong *unforgeability* and *password-safety* properties.

1.1 Our Contributions

We formally introduce the notion of Password-based Threshold Authentication (PbTA) with the goal of making password-based token generation secure against server breaches that could compromise both long-term keys and user credentials. Our contributions are as follows:

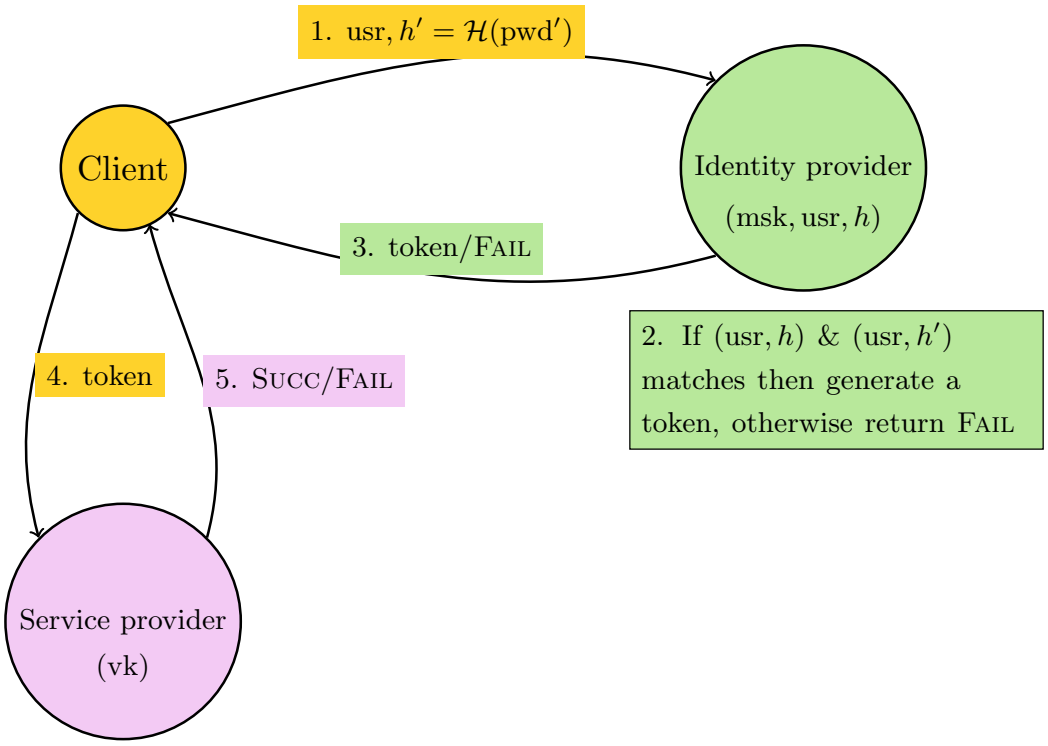


Figure 1: The generic flow diagram of commonly used password-based token generation solutions. The figure shows only the sign-on phase which is preceded by a one-time registration phase (not shown) where the client stores its username (usr) and the hash (h) of its password with the identity provider. msk is the secret key used for generating tokens and vk is used for verifying them.

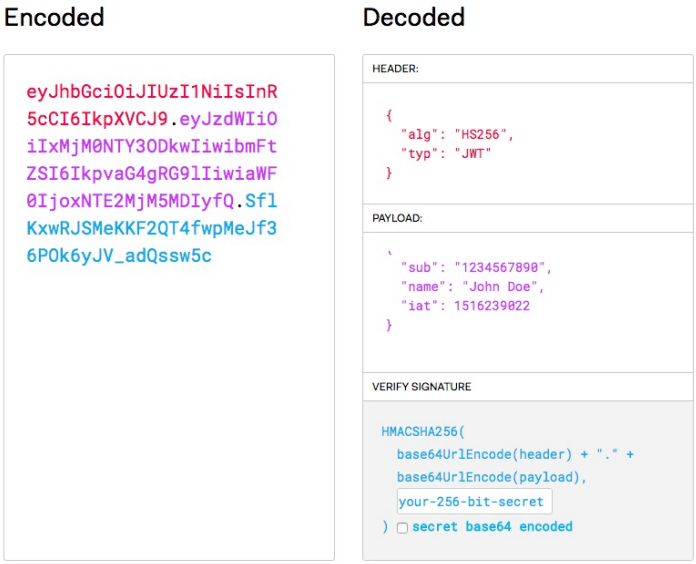
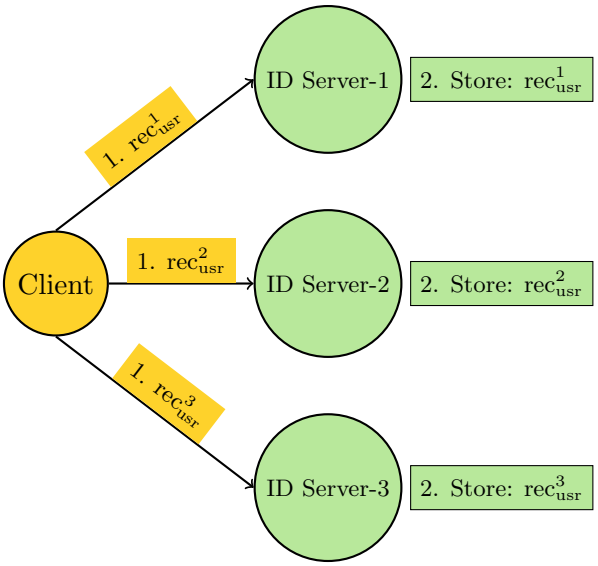
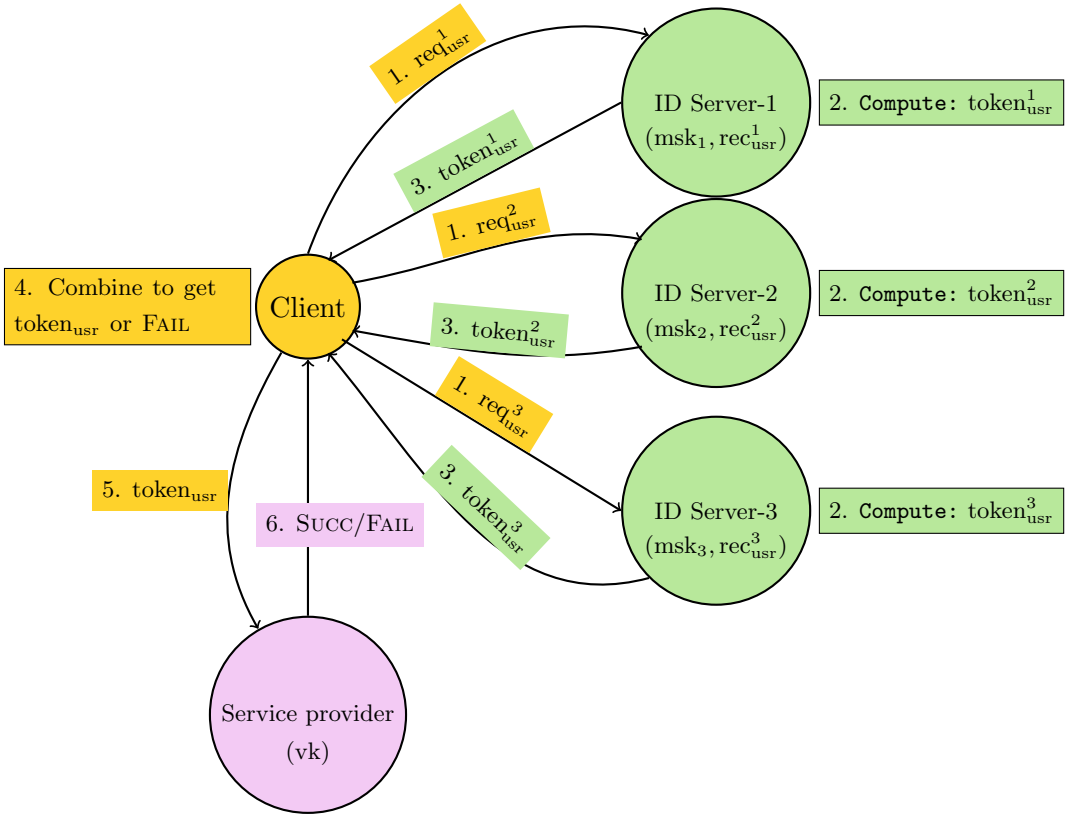


Figure 2: A sample JSON Web Token [jwt] that uses HMAC. The base64 encoded token on the left is what is sent and stored. When decoded, it contains a header with algorithm and token type, a payload that includes various attributes and a HMAC of header/payload.



(a) Generic flow diagram of a PbTA protocol in the registration phase.



(b) Generic flow diagram of a PbTA protocol in the sign-on phase.

Figure 3: Generic flow diagrams of a PbTA protocol. For more detailed explanation see Section 1.1.

Defining security for PbTA We formalize password-based threshold authentication, and establish the necessary security requirements of token unforgeability and password-safety in presence of an adversary who may breach a subset of the identity servers. Our game-based definitions are strong and intuitive, and consider security in a multi-client setting where many clients use the same identity provider. Adversary could corrupt clients in an adaptive fashion during the game. We note that an alternative approach would be to use the Universal Composability framework [Can01] as followed in some prior work involving password-based authentication (e.g. [JKKX17]) . We chose to focus on game-based definitions that are much simpler to work with but comprehensive enough to cover a very broad set of attack scenarios.

The PASTA framework We propose a general framework called PASTA, that uses as building blocks any threshold oblivious pseudorandom function (TOPRF) and any threshold token generation (TTG) scheme, i.e. a threshold MAC or digital signature. PASTA meets our stringent security requirements for PbTA.

After a one-time registration phase, a client just needs to remember its password. It can sign-on using a two-round protocol wherein the servers do not talk to each other (assuming only that the servers communicate to the client over an authenticated channel). Thus, PASTA requires minimal interaction.

The sign-on protocol ensures that if the client's password is correct, he obtains a valid authentication token: client sends a request message to a subset of the servers, and servers respond with messages of their own. If the client password is a match, it can combine server responses to obtain a valid token (see Figure 3) [1]. Otherwise, it does not learn anything.

At the first glance, it may seem unnatural to define a general framework that works for both symmetric-key tokens (i.e. MAC) and public-key tokens (i.e. digital signature). Though their verification procedures are different in terms of being private or public, note that their token generation procedures are both private. PASTA focuses on generating tokens, hence it works for both types of tokens.

Instantiations and Implementation We instantiate and implement our framework in C++ with four different threshold token generation schemes: block-cipher based and DDH-based threshold MACs of Naor et al. [NPR99], threshold RSA-based signature of Shoup [Sho00] and threshold pairing-based signature of Boldyreva [Bol03]. Each instantiation has its own advantages and disadvantages. When instantiated with a threshold MAC, we obtain a more efficient solution but the tokens are not publicly verifiable, i.e. vk in Figure 1 and Figure 3b stored in the application server would be the same as the master secret key msk, since the verifier needs the secret key for verification.[2] PASTA with RSA-based and pairing-based token generation are more expensive but are publicly verifiable. Among the signature-based solutions, the pairing-based one is faster since signing does not require pairings but the RSA-based solution has faster verification and produces signatures that are compatible with legacy applications. To the best of our knowledge, our work is also the first to implement

[1] Note that in this setting, as opposed to the naïve solution (Figure 1), no matching takes place on the identity provider side. In particular, an ID-server does not check against a record stored in the registration phase, because, if it did, one can easily see that offline attacks would be possible even if a single server is breached. See Section 2 for more details.

[2] To achieve better security of the secret key, the verification process can also be made distributed using a standard threshold MAC scheme. We omit the distributed verification in the rest of this paper because it is not our focus.

several of the threshold token generation schemes (*not* password-based) and report on their performance.

Our experiments show that the overhead of obtaining security against server breaches using PASTA, in the sign-on stage, is at most 5% compared to the naïve solution of using hashed passwords and a single-server token generation, in the most likely scenario where clients connect to servers over the internet (a WAN network). This is primarily due to the fact that in this case, network latency dominates the total runtime for all token types. The overhead is a bit higher in the LAN setting but the total runtime of sign-on (steps 1-4 in Figure 3b) is still very fast, ranging from 1.3 ms for $(n, t) = (3, 2)$ with a symmetric-key MAC token to 23 ms for $(n, t) = (10, 10)$ with an RSA-based token, where n is the number of servers and t is the threshold.

Necessity of public-key operations PASTA has its largest overhead compared to the naïve single-server solution, for symmetric-key based tokens in the LAN setting. This is because public-key operations dominate PASTA's runtime while the naïve solution only involves symmetric-key operations. Nevertheless, we show that this inefficiency is inherent by proving that public-key operations are necessary to achieve our notion of PbTA in Appendix D.

1.2 Related Work

Password-based techniques are the most common methods for authenticating users. However, the traditional approach of storing hashed passwords on the servers is susceptible to offline dictionary attacks [WW13, Dan]. Standard remedies such as *salting* or more advanced remedies such as *memory-hard functions* [Tar, ACP+17, ACK+16, DKAN, BZ17, BD16], pursued in the recent password-hashing competition [phc], surely make the task of the attacker harder, but do not resolve the fundamental issue of trusting a single server.

A large body of work considers distributed token generation through threshold digital signatures [DF90, DDFY94, GHKR08, DK01, Bol03, AMN01, GJKR96, GGN16, Sho00, BS01] and threshold message authentication codes [BLMR13, NPR99, MPSN+03] which can protect the master key against $t-1$ breached servers. A separate line of work on threshold password-authenticated key exchange (T-PAKE) [MSJ02, DG03, AFP05, ACFP05, CLN15, KMTG05] aims to prevent offline dictionary attacks in standard password-authenticated key exchange (PAKE) [BPR00, BMP00, KOY01, GK10, KV11, CHK+05, PS10, KOY03] by employing multiple servers.

While PAKE and T-PAKE solve the problem of establishing a secret key between a server and a client, where the client authenticates with a password, they do not solve the problem posed in this paper of distributing trust in password-based token generation. Specifically, PbTA generates tokens and provides token unforgeability, which T-PAKE does not deal with. Moreover, PbTA works in a setting where multiple clients share the same token generation set-up, and guarantees that attacks on one client do not affect the security of others. Finally, PbTA has a per-client registration phase that further differentiates it from PAKE and T-PAKE.

It is also worth noting that a straightforward composition of a T-PAKE followed by a threshold signature/MAC meets neither the efficiency nor the security requirements for PbTA. For efficiency, recall that we require minimal interaction where servers need not communicate with each other after a one-time setup procedure, and both the password verification and the token generation can be performed simultaneously in two rounds. The most efficient T-PAKE schemes require at least three rounds of interaction between the client and servers and

additional communication among the servers (which could further increase when combined with threshold token generation). For security, it is unclear how to make such a composition meet our strong unforgeability and password-safety properties which we elaborate on shortly.

Another line of work focuses on constructing password-based server-aided signatures [CLNS16, XS03, Gan95, GT11, MR03]. However, they assume that apart from the password, a client also needs to use a secret state (e.g. a shared secret key) to generate a signature. In contrast, we focus on a solution in which a client only needs to use a password to generate a signature (more generally, a token).

Password-protected secret sharing (PPSS) [BJSL11, JKKX17, ACNP16, JKK14, YHCL15, CLLN14, CLN12, CEN15, JKKX16] considers the related problem of sharing a secret among multiple servers where t servers can reconstruct the secret if client's password verifies. This line of work does not meet our goal of keeping the master secret distributed at all times for use in a threshold token generation scheme. Moreover, PPSS is commonly studied in a single client setting where each client has its own unique secret. As we will see shortly, the multi-client setting and the common master-key used for all clients introduces additional technical challenges.

A very recent work of Harchol et al. [HAP18] implements and uses similar building blocks to ours, i.e. a threshold oblivious PRF [JKKX17] and a proactive variant of threshold RSA signature scheme [Sho00]. But it uses them for the different end goal of distributing server secret keys and protecting client secret keys with a password in SSH implementations. As such, it neither formalizes nor addresses the security/efficiency requirements of a password-based token generation scheme.

2 An Overview of PASTA

We start with a *plain* password-based token generation protocol that is insecure against server breaches. As mentioned in Section 1, the plain protocol works as follows. In the registration phase, a client registers with its username/password by storing its username and hashed password $h = \mathcal{H}(\text{password})$ on the identity server. In the sign-on phase, client sends its username and hashed password h' to the server; server checks if $h' = h$ for the username. If the check passes, server then uses a master secret key msk to compute a token $auth_{msk}(x)$ and sends it to client, where $auth$ is either a MAC or a digital signature and x is the data to be signed. In this solution, both the master secret key msk and the hashed password h are compromised if the server is breached. Hence clients' passwords could be recovered using offline dictionary attacks.

Threshold solution A natural approach for protecting the master secret key msk is to combine the above plain solution with a threshold token generation (TTG) scheme (i.e. a threshold MAC or threshold signature). TTG schemes enable us to secret share msk among n servers such that any t servers can jointly generate valid tokens while any subset of up to $t-1$ servers cannot forge valid tokens or recover msk. To combine with the plain solution, the client registers to every server by sending its username and hashed password h in the registration phase. Then in the sign-on phase, client sends to t servers its username and hashed password h'. Every server checks if the $h = h'$ for the username, and performs its portion of the TTG scheme if the check passes. This solution guarantees the security of msk when at most $t-1$ servers are breached, but clients' passwords are still vulnerable against offline dictionary attacks even if a single server is breached.

Changing secret information stored on servers The above two naïve solutions follow the same paradigm: server issues a token or executes the TTG scheme *only if client is using the correct password*. In order to check if client is using the correct password in the sign-on phase, server needs to store some "secret information" about client's password in the registration phase. In the above solutions, this secret information is the hashed password. A fundamental problem with this is that the secret information can be computed given only the password, hence enabling offline dictionary attacks on the password. To resolve this issue, we make the stored secret information also depend on a server-side secret.

This can be achieved by a threshold oblivious pseudorandom function (TOPRF) [FIPR05]. In a TOPRF protocol, a secret key k for a pseudorandom function F is initially shared among n servers. A client can obtain a PRF value of its password $h = F_k(\text{password})$ by interacting with t servers, without revealing any information about its password to servers. Moreover, the function $F_k(\cdot)$ is computable by any t servers, but cannot be computed by up to $t-1$ servers. To this end, the PRF value $h = F_k(\text{password})$ serves as our new secret information stored on servers, and the protocol is now secure against offline dictionary attacks.

From four rounds to two rounds A TOPRF protocol requires at least two rounds. Hence the sign-on phase in the above protocol requires at least four rounds: client and servers run the TOPRF protocol which requires two rounds for the client to obtain h; client then sends h back to servers as a third-round message; servers verify and respond with token shares of the TTG scheme as fourth-round messages. We would like to reduce the interaction to two rounds because network latency is a major bottleneck in the protocol especially over WAN networks (see Section 7.2 for details).

On the one hand, in order to prevent offline dictionary attack, we require that the "secret information" be computed jointly by client and servers, which requires at least two rounds. On the other hand, servers must ensure that generation of token is only performed after the secret information is checked, which also requires two rounds, so it seems that four rounds is necessary to achieve our goal.

We resolve this deadlock by observing that the check does not have to be done on the server side. Instead of checking the secret information and then participating in the TTG scheme to generate token shares, the servers generate token shares directly and encrypt them under the secret information h using a symmetric-key encryption scheme. The ciphertexts are sent along with the second-round message of the TOPRF protocol. Now the protocol only has two rounds, and the check is done on the client side: only if the client has used the correct password in the first round of TOPRF can it calculate the correct h and decrypt the ciphertexts to obtain t token shares, and combine them to recover the final token.

Mitigating client impersonation attacks There is still a subtle security problem. Consider an attacker who compromises a single server and retrieves the secret information h of a client, and then impersonates the client to which h belongs without knowing its password by participating in a sign-on protocol with the servers. The servers generate token shares, encrypt them under h, and send back to the attacker. Since the attacker already knows h, it can decrypt all the ciphertexts and combine the token shares to obtain a valid token without ever knowing the client's password or the master secret key. This issue occurs because when reducing the round complexity from four to two, we make the servers generate token shares without checking the secret information, but encrypt them using the secret information.

We address this issue by further modifying the secret information stored on servers. A

client who computes h in the registration phase only sends $h_i = \mathcal{H}'(h, i)$ to server i where $\mathcal{H}'$ is assumed to be a random oracle. In other words, h is never revealed to or stored by any server, and each server only learns its corresponding h_i. Later in the sign-on phase, token shares are encrypted under the h_is. The client impersonation attack no longer works since compromising certain servers only reveals the h_is of these servers to the attacker, while the remaining h_is are still kept secret.

Multi-client security In our final protocol, we require that the only allowed attack is to impersonate a certain client and try different passwords by participating in an online sign-on protocol. This type of online attack is easy to detect in practice (e.g. if the same client is trying to sign-on very frequently within a short period of time). But enforcing the same across a large set of clients is not possible. Hence an important security requirement is that attacking one client should not help in attacking any other client.

This is not true for the protocol we have described so far. Consider an attacker who does not compromise any server, and performs the above online attack on one client[3], trying all possible passwords. As a result, the attacker would obtain all the PRF values $h = PRF_k(\text{password})$ for all possible passwords. Then the attacker impersonates another client by participating in a single sign-on protocol with the servers. Since the attacker already knows all possible PRF values, it could try decrypting the ciphertexts sent from servers using the collected dictionary of PRF values (offline) to find the correct value and hence recover the password. In other words, he can leverage his online attack against one client to perform offline attacks (after a single online interaction) on many other clients. Note that including client username as part of the input to the PRF does not solve the problem either since servers have no way of checking what username the attacker incorporates in the TOPRF protocol without adding expensive zero-knowledge proofs to this effect to the construction.

One natural idea is to have a distinct TOPRF key for every client, so that PRF values learned from one client would be useless for any other client. This means that servers need to generate a sufficiently large number of TOPRF keys in the global setup phase, which is not practical. There is a simple and efficient fix: we let every client generate its own TOPRF key and secret share it between servers in the registration phase. This yields our final protocol which we formally prove to meet all our security requirements under the gap TOMDH assumption [JKKX17] in the random oracle model.

3 Preliminaries

We use κ to denote the security parameter. Let $\mathbb{Z}$ denote the set of all integers and $\mathbb{Z}_n$ the set $\{0, 1, 2, \ldots, n-1\}$. $\mathbb{Z}_n^*$ is defined as $\mathbb{Z}_n^* := \{x \in \mathbb{Z}_n | \gcd(x, n) = 1\}$. We use $[a, b]$ for $a, b \in \mathbb{Z}$, $a \leq b$, to denote the set $\{a, a+1, \ldots, b-1, b\}$. $[b]$ denotes the set $[1, b]$. $\mathbb{N}$ denotes the set of natural numbers.

We use $x \leftarrow_\$ S$ to denote that x is sampled uniformly at random from a set S. We use PPT as a shorthand for probabilistic polynomial time and negl to denote negligible functions.

We use $[\![a]\!]$ as a shorthand for $(a, a_1, \ldots, a_n)$ where $a_1, \ldots, a_n$ are *shares* of a. A concrete scheme will specify how the shares will be generated. The value of n will be clear from context.

[3] A smarter attacker would distribute its online attack across many clients to avoid detection. We use the single client in this example just to highlight the underlying multi-client security issue.

We use a 'require' statement in the description of an oracle to enforce some checks on the inputs. If any of the checks fail, the oracle outputs $\perp$.

In a security game, we use $\langle \mathcal{O} \rangle$ to denote the collection of all the oracles defined in the game. For e.g., if a game defines oracles $\mathcal{O}_1, \ldots, \mathcal{O}_\ell$, then for an adversary Adv, $\mathsf{Adv}^{\langle \mathcal{O} \rangle}$ denotes that Adv has access to the collection $\langle \mathcal{O} \rangle := (\mathcal{O}_1, \ldots, \mathcal{O}_\ell)$.

Shamir's secret sharing. Shamir's secret sharing is a simple way to generates shares of a secret so that a threshold of the shares are sufficient to reconstruct the secret, while any smaller number hides it completely. We consider a slightly more general form of Shamir's sharing here. Let $\mathsf{GenShare}$ be an algorithm that takes inputs p, n, t, $\{(i, \alpha_i)\}_{i \in S}$ s.t. $t \leq n < p$, p is prime, $S \subseteq [0, n]$ and $|S| < t$. It picks a random polynomial f of degree at most $t-1$ over $\mathbb{Z}_p$ s.t. $f(i) = \alpha_i$ for all $i \in S$, and outputs $f(0), f(1), \ldots, f(n)$.

To generate a (t, n)-Shamir sharing of a secret $s \in \mathbb{Z}_p$, $\mathsf{GenShare}$ is given p, n, t and $(0, s)$ as inputs to produce shares $s_0, s_1, \ldots, s_n$. Using the shorthand defined above, one can write the output compactly as $[\![s]\!]$. Given any t or more of the shares, say $\{s_j\}_{j \in T}$ for $|T| \geq t$, one can efficiently find coefficients $\{\lambda_j\}_{j \in T}$ such that $s = f(0) = \sum_{j \in T} \lambda_j \cdot s_j$. However, knowledge of up to $t-1$ shares reveals no information about s if it is chosen at random from $\mathbb{Z}_p$.

Cyclic group generator. Let $\mathsf{GroupGen}$ be a PPT algorithm that on input 1^κ outputs $(p, g, \mathbb{G})$ where $p = \Theta(\kappa)$, p is prime, $\mathbb{G}$ is a group of order p, and g is a generator of $\mathbb{G}$. We will use multiplication to denote the group operation.

3.1 Hardness Assumption

Threshold oblivious PRF (TOPRF) was introduced by Jarecki et al. [JKKX17] in a recent work. They propose a simple TOPRF protocol called 2HashTDH and prove that it is UC-secure under the Gap Threshold One-More Diffie-Hellman (Gap-TOMDH) assumption in the random oracle model. They also show that Gap-TOMDH is hard in the generic group model.

Rather than modeling TOPRF as a functionality in the UC-sense, we will explicitly formalize two natural properties for it, obliviousness and unpredictability, in Section 4. We will show that Jarecki et al.'s 2HashTDH protocol satisfies these properties under the same assumption. Here, we formally state the assumption.

For $q_1, \ldots, q_n \in \mathbb{N}$ and $t', t \in \mathbb{N}$ where $t' < t \leq n$, define $\mathsf{MAX}_{t',t}(q_1, \ldots, q_n)$ to be the largest value of ℓ such that there exists *binary* vectors $\mathbf{u}_1, \ldots, \mathbf{u}_\ell \in \{0,1\}^n$ such that each $\mathbf{u}_i$ has $t - t'$ number of 1's in it and $(q_1, \ldots, q_n) \geq \sum_{i \in [\ell]} \mathbf{u}_i$. (All operations on vectors are component-wise integer operations.) Looking ahead, t and t' will be the parameters in the security definition of TOPRF and PbTA (t will be the threshold and t' the number of corrupted parties).

Definition 3.1 (Gap-TOMDH) *A cyclic group generator* $\mathsf{GroupGen}$ *satisfies the* Gap Threshold One-More Diffie-Hellman *(Gap-TOMDH) assumption if for all t', t, n, N such that $t' < t \leq n$ and for all PPT adversary* Adv*, there exists a negligible function* negl *s.t.* $\mathsf{One\text{-}More}_{\mathsf{Adv}}(1^\kappa, t', t, n, N)$ *(Figure 4) outputs* 1 *with probability at most* $\mathsf{negl}(\kappa)$.

In this game, a random polynomial of degree $t-1$ is picked but Adv gets to choose its value at t' points (steps 3 and 4). Adv gets access to two oracles:

- $\mathcal{O}$ allows it to compute x^{k_i}, where k_i is the value of the randomly chosen polynomial at i, for k_i that it does not know. A counter q_i is incremented for every such call.

$\underline{\mathsf{One\text{-}More}_{\mathsf{Adv}}(1^\kappa, t', t, n, N)}$:

1. $(p, g, \mathbb{G}) \leftarrow \mathsf{GroupGen}(1^\kappa)$
2. $g_1, \ldots, g_N \leftarrow_\$ \mathbb{G}$
3. $(\{(i, \alpha_i)\}_{i \in \mathcal{U}}, \mathsf{st}) \leftarrow \mathsf{Adv}(p, g, \mathbb{G}, g_1, \ldots, g_N)$, where $\mathcal{U} \subseteq [n]$, $|\mathcal{U}| = t'$
4. $(k_0, k_1, \ldots, k_n) \leftarrow \mathsf{GenShare}(p, n, t, \{(i, \alpha_i)\}_{i \in \mathcal{U}})$
5. $q_1, \ldots, q_n := 0$
6. $((g'_1, h_1), \ldots, (g'_\ell, h_\ell)) \leftarrow \mathsf{Adv}^{\langle \mathcal{O} \rangle}(\mathsf{st})$
7. output 1 iff
   - $\ell > \mathsf{MAX}_{t',t}(q_1, \ldots, q_n)$,
   - $\forall\ i \in [\ell]$, $g'_i \in \{g_1, \ldots, g_N\}$ and $h_i = {g'_i}^{k_0}$, and
   - $\forall\ i, j \in [\ell]$ s.t. $i \neq j$, $g'_i \neq g'_j$.

$\underline{\mathcal{O}(i, x)}$:

- require: $i \in [n] \setminus \mathcal{U}, x \in \mathbb{G}$
- increment q_i by 1
- return x^{k_i}

$\underline{\mathcal{O}_{\mathrm{DDH}}(g_1, g_2, h_1, h_2)}$

- require: $g_1, g_2, h_1, h_2 \in \mathbb{G}$
- return 1 iff $\exists\ a \in \mathbb{Z}_p$ s.t. $g_2 = g_1^a$ and $h_2 = h_1^a$

Figure 4: Gap-TOMDH game

- $\mathcal{O}_{\mathrm{DDH}}$ allows it to check if the discrete log of g_2 w.r.t. g_1 is the same as the discrete log of h_2 w.r.t. h_1.

Intuitively, to compute a pair of the form (g, g^{k_0}), Adv should somehow get access to k_0. It clearly knows k_i for $i \in \mathcal{U}$, but shares outside $\mathcal{U}$ can only be obtained in the exponent, with the help of oracle $\mathcal{O}$. One option for Adv is to invoke $\mathcal{O}$ with (i, g) for at least $t - t'$ different values of i outside of $\mathcal{U}$, and then combine them together along with the k_i it knows to obtain g^{k_0}.

If Adv sticks to this strategy, it would have to repeat it entirely to compute h^{k_0} for a different base h. It could invoke $\mathcal{O}$ on different subsets of $[n]$ for different basis, but $\mathsf{MAX}_{t',t}(q_1, \ldots, q_n)$ will be the maximum number of pairs of type (x, x^{k_0}) it will be able to generate through this process.

Certainly, an adversary is not restricted to producing pairs in the way described above. However, Gap-TOMDH assumes that no matter what strategy a PPT adversary takes, it can effectively do no better than this.

3.2 Threshold Token Generation

A threshold token generation (TTG) scheme distributes the task of generating tokens for authentication among a set of n servers, such that at least a threshold t number of servers must be contacted to compute a token. TTG provides a strong unforgeability guarantee: even if $t' < t$ of the servers are corrupt, any time a token on some new value x is needed, at least $t - t'$ servers must be contacted.

We formally define a TTG scheme and the unforgeability guarantee associated with it

below.

Definition 3.2 (Threshold Token Generation) *A threshold token generation scheme* TTG *is a tuple of four PPT algorithms (*Setup, PartEval, Combine, Verify*) that satisfies the consistency property below.*

- $\mathsf{Setup}(1^\kappa, n, t) \to (\llbracket \mathsf{sk} \rrbracket, \mathsf{vk}, \mathsf{pp})$. *It generates a secret key* sk*, shares* sk_1*,* sk_2*, ...,* sk_n *of the key, a verification key* vk*, and public parameters* pp*. Share* sk_i *is given to party* i*. (*pp *will be an implicit input in the algorithms below.)*
- $\mathsf{PartEval}(\mathsf{sk}_i, x) \to y_i$. *It generates shares of token for an input. Party* i *computes the* i*-th share* y_i *for* x *by running* PartEval *with* sk_i *and* x*.*
- $\mathsf{Combine}(\{i, y_i\}_{i \in S}) =: \mathsf{tk}/\perp$. *It combines the shares received from parties in the set* S *to generate a token* tk*. If the algorithm fails, its output is denoted by* $\perp$*.*
- $\mathsf{Verify}(\mathsf{vk}, x, \mathsf{tk}) =: 1/0$. *It verifies whether token* tk *is valid for* x *or not using the verification key* vk*. (Output* 1 *denotes validity.)*

Consistency. For all $\kappa \in \mathbb{N}$*, any* $n, t \in \mathbb{N}$ *such that* $t \leq n$*, all* $(\llbracket \mathsf{sk} \rrbracket, \mathsf{vk}, \mathsf{pp})$ *generated by* $\mathsf{Setup}(1^\kappa, n, t)$*, any value* x*, and any set* $S \subseteq [n]$ *of size at least* t*, if* $y_i \leftarrow \mathsf{PartEval}(\mathsf{sk}_i, x)$ *for* $i \in S$*, then* $\mathsf{Verify}(\mathsf{vk}, x, \mathsf{Combine}(\{(i, y_i)\}_{i \in S})) = 1$*.*

Definition 3.3 (Unforgeability) *A threshold token generation scheme* TTG $:=$ *(*Setup, PartEval, Combine, Verify*) is* unforgeable *if for all PPT adversaries* Adv*, there exists a negligible function* negl *such the probability that the following game outputs* 1 *is at most* $\mathsf{negl}(\kappa)$*.*

$\mathsf{Unforgeability}_{\mathsf{TOP},\mathsf{Adv}}(1^\kappa, n, t)$*:*

- *Initialize. Run* $\mathsf{Setup}(1^\kappa, n, t)$ *to get* $(\llbracket \mathsf{sk} \rrbracket, \mathsf{vk}, \mathsf{pp})$*. Give* pp *to* Adv*.*
- *Corrupt. Receive the set of corrupt parties* $\mathcal{U}$ *from* Adv*, where* $t' := |\mathcal{U}| < t$*. Give* $\{\mathsf{sk}_i\}_{i \in \mathcal{U}}$ *to* Adv*.*
- *Evaluate. In response to* Adv*'s query* $(\mathtt{Eval}, x, i)$ *for* $i \in [n] \backslash \mathcal{U}$*, return* $y_i := \mathsf{PartEval}(\mathsf{sk}_i, x)$*. Repeat this step as many times as* Adv *desires.*
- *Challenge.* Adv *outputs* $(x^\star, \mathsf{tk}^\star)$*. Check if*
  - $|\{i \mid \mathsf{Adv}$ *made a query* $(\mathtt{Eval}, x^\star, i)\}| < t - t'$ *and*
  - $\mathsf{Verify}(\mathsf{vk}, x^\star, \mathsf{tk}^\star) = 1$*.*

  Output 1 *if and only if both checks succeed.*

The unforgeability property captures the requirement that it must not be possible to generate a valid token on some value if less than $t - t'$ servers are contacted with that value.

4 Threshold Oblivious Pseudo-Random Function

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 [FIPR05]. 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. [JKKX17] 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. They propose a functionality for TOPRF and show how to realize it in a UC-secure way. We instead treat TOPRF as a set of algorithms that must satisfy two natural properties, unpredictability and obliviousness.

4.1 Definition

Definition 4.1 (Threshold Oblivious Pseudo-Random Function) *An $(\mathcal{X}, \mathcal{R})$-threshold oblivious pseudo-random function (TOPRF)* TOP *is a tuple of four PPT algorithms* $(\mathsf{Setup}, \mathsf{Encode}, \mathsf{Eval}, \mathsf{Combine})$ *that satisfies the consistency property below.*

- $\mathsf{Setup}(1^\kappa, n, t) \to (\llbracket \mathsf{sk} \rrbracket, \mathsf{pp})$. *It generates n secret key shares $\mathsf{sk}_1, \mathsf{sk}_2, \ldots, \mathsf{sk}_n$ and public parameters* pp. *Share sk_i is given to party i. (*pp *will be an implicit input in the algorithms below.)*
- $\mathsf{Encode}(x, \rho) =: c$. *It generates an encoding c of $x \in \mathcal{X}$ using randomness $\rho \in \mathcal{R}$.*
- $\mathsf{Eval}(\mathsf{sk}_i, c) =: z_i$. *It generates shares of TOPRF value from an encoding. Party i computes the i-th share z_i from c by running* Eval *with sk_i and c.*
- $\mathsf{Combine}(x, \{(i, z_i)\}_{i \in S}, \rho) =: h/\perp$. *It combines the shares received from parties in the set S using randomness ρ to generate a value h. If the algorithm fails, its output is denoted by $\perp$.*

Consistency. For all $\kappa \in \mathbb{N}$, any $n, t \in \mathbb{N}$ such that $t \leq n$, all $(\llbracket \mathsf{sk} \rrbracket, \mathsf{pp})$ generated by $\mathsf{Setup}(1^\kappa, n, t)$, any value $x \in \mathcal{X}$, any randomness $\rho, \rho' \in \mathcal{R}$, and any two sets $S, S' \subseteq [n]$ of size at least 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 $\mathsf{Combine}(x, \{(i, z_i)\}_{i \in S}, \rho) = \mathsf{Combine}(x, \{(j, z'_j)\}_{j \in S'}, \rho') \neq \perp$.

Thus, irrespective of the randomness used to encode an x and the set of parties whose shares are combined, the output of $\mathsf{Combine}$ does not change (as long as $\mathsf{Combine}$ is given the same randomness used for encoding). We call this output the output of the TOPRF on x, and denote it by $\mathsf{TOP}(\mathsf{sk}, x)$.

Public combine. We also consider a public combine algorithm $\mathsf{PubCombine}$ that could be run by anyone with access to just the partial evaluations. It would be used to check if a purported set of evaluations can lead to the right PRF value or not. Formally, for $Z := \{(i, z_i)\}_{i \in S}$ generated in the same manner as in the consistency property, and any arbitrary $Z^\star := \{(i, z_i^\star)\}_{i \in S}$, if $\mathsf{PubCombine}(Z) = \mathsf{PubCombine}(Z^\star)$ then $\mathsf{Combine}(x, Z, \rho) = \mathsf{Combine}(x, Z^\star, \rho)$. More importantly though, if the former equality does not hold then the later must not hold either (with high probability).

4.2 Security properties

We want a TOPRF scheme to satisfy two properties, unpredictability and obliviousness. Unpredictability mandates that it must be difficult to predict TOPRF output on a random value, and obliviousness mandates that the random value itself is hard to guess even if the TOPRF output is available.

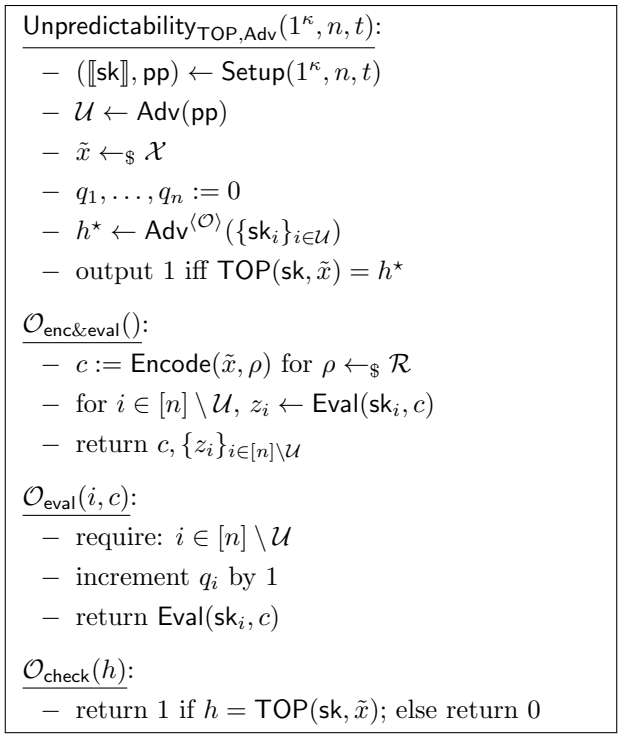


Figure 5: Unpredictability game

Definition 4.2 (Unpredictability) *A* $(\mathcal{X}, \mathcal{R})$*-TOPRF* TOP := (Setup, Encode, Eval, Combine) *is* unpredictable *if for all* $n, t \in \mathbb{N}$*,* $t \leq n$*, and PPT adversaries* Adv*, there exists a negligible function* negl *s.t.*

$$\mathsf{Pr}[\mathsf{Unpredictability}_{\mathsf{TOP},\mathsf{Adv}}(1^\kappa, n, t) = 1] \leq \frac{\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n)}{|\mathcal{X}|} + \mathsf{negl}(\kappa), \tag{4.1}$$

where Unpredictability *is defined in Figure 5.*

Our unpredictability definition provides several interfaces to an adversary Adv. Oracle $\mathcal{O}_{\mathsf{enc\&eval}}$ can be called any number of times to get different sets of partial evaluations on the challenge input $\tilde{x}$, but the randomness used in this process is not revealed to Adv. If no query is made to $\mathcal{O}_{\mathsf{eval}}$, so that none of the q_i change, then Adv's probability of guessing the TOPRF output on $\tilde{x}$ should be negligible (see Eq. (4.1)). In other words, any number of partial evaluations by themselves should not help at all.

Adv could, however, encode an arbitrary input itself, get partial evaluations through $\mathcal{O}_{\mathsf{eval}}$, and then combine them to learn the TOPRF output. It could also check if this output is same as the TOPRF output on the challenge input through $\mathcal{O}_{\mathsf{check}}$. Thus, by repeatedly querying $\mathcal{O}_{\mathsf{eval}}$, adversary can increase its chances of making the right guess. Eq. (4.1) requires that the probability of success should be no more than the maximum number of TOPRF outputs Adv can learn through this process over the size of password space. In some sense, this is the best we can hope to achieve.

Definition 4.3 (Obliviousness) *An* $(\mathcal{X}, \mathcal{R})$*-TOPRF* TOP := (Setup, Encode, Eval, Combine) *is* oblivious *if for all* $n, t \in \mathbb{N}$*,* $t \leq n$*, and all PPT adversaries* Adv*, there exists a negligible*

$\underline{\mathsf{Obliviousness}_{\mathsf{TOP},\mathsf{Adv}}(1^\kappa, n, t)}$:

- $([\![\mathsf{sk}]\!], \mathsf{pp}) \leftarrow \mathsf{Setup}(1^\kappa, n, t)$
- $\mathcal{U} \leftarrow \mathsf{Adv}(\mathsf{pp})$
- $\tilde{x} \leftarrow_\$ \mathcal{X}$
- $q_1, \ldots, q_n := 0$
- $x^\star \leftarrow \mathsf{Adv}^{\langle \mathcal{O} \rangle}(\{\mathsf{sk}_i\}_{i \in \mathcal{U}})$
- output 1 iff $x^\star = \tilde{x}$

$\underline{\mathcal{O}_{\mathsf{enc\&eval}}()}$:

- $c := \mathsf{Encode}(\tilde{x}, \rho)$ for $\rho \leftarrow_\$ \mathcal{R}$
- for $i \in [n]$, $z_i \leftarrow \mathsf{Eval}(\mathsf{sk}_i, c)$
- $h := \mathsf{Combine}(\tilde{x}, \{(i, z_i)\}_{i \in [n]}, \rho)$
- return $c, \{z_i\}_{i \in [n] \setminus \mathcal{U}}, h$

$\underline{\mathcal{O}_{\mathsf{eval}}(i, c)}$:

- require: $i \in [n] \setminus \mathcal{U}$
- increment q_i by 1
- return $\mathsf{Eval}(\mathsf{sk}_i, c)$

Figure 6: Obliviousness game

function negl *s.t.*

$$\Pr[\mathsf{Obliviousness}_{\mathsf{TOP},\mathsf{Adv}}(1^\kappa, n, t) = 1] \leq \frac{\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) + 1}{|\mathcal{X}|} + \mathsf{negl}(\kappa), \tag{4.2}$$

where Obliviousness *is defined in Figure 6.*

The obliviousness definition differs from unpredictability in small but crucial ways. Unlike the unpredictability game, Adv directly gets h from $\mathcal{O}_{\mathsf{enc\&eval}}$ because our goal is not to challenge the adversary on guessing the TOPRF output. Adv can still use $\mathcal{O}_{\mathsf{eval}}$ to learn new TOPRF outputs and check (by itself) it they match with h or not. Thus it can improve its chances of guessing $\tilde{x}$. The bound of Eq. 4.2 differs slightly from that of Eq. 4.1 though: there is an extra additive factor of 1 in the former case. This is to take into account that Adv can output a guess for $\tilde{x}$ different from the ones it has tried in the game.

4.3 Construction

We recall here the TOPRF construction, 2HashTDH, of Jarecki et al. [JKKX17, Section 3], for an input space $\mathcal{X}$. We refer to this construction as TOP from here onwards.

- $\mathsf{Setup}(1^\kappa, n, t)$. Run $\mathsf{GroupGen}(1^\kappa)$ to get $(p, g, \mathbb{G})$. Pick an sk at random from $\mathbb{Z}_p$. Let $[\![\mathsf{sk}]\!] \leftarrow \mathsf{GenShare}(p, n, t, (0, \mathsf{sk}))$ be a (t, n)-Shamir sharing of sk. Let $\mathcal{H}_1 : \mathcal{X} \times \mathbb{G} \to \{0,1\}^\kappa$ and $\mathcal{H}_2 : \mathcal{X} \to \mathbb{G}$ be hash functions. Output $[\![\mathsf{sk}]\!]$ and $\mathsf{pp} := (p, g, \mathbb{G}, n, t, \mathcal{H}_1, \mathcal{H}_2)$.
- $\mathsf{Encode}(x, \rho)$. Output $\mathcal{H}_2(x)^\rho$.
- $\mathsf{Eval}(\mathsf{sk}_i, c)$. Output c^{sk_i}.
- $\mathsf{Combine}(x, \{(i, z_i)\}_{i \in S}, \rho)$. If $|S| < t - 1$, output $\perp$. Else, use S to find coefficients $\{\lambda_i\}_{i \in S}$, compute $z := \prod_{i \in S} z_i^{\lambda_i}$, and output $\mathcal{H}_1(x \| z^{\rho^{-1}})$.

It is easy to see that TOP is a consistent $(\mathcal{X}, \mathbb{Z}_p^*)$-TOPRF scheme.

We prove unpredictability and obliviousness of TOP in Appendix A.1 and A.2, respectively.

Public combine. One can define $\mathsf{PubCombine}(\{(i, z_i)\}_{i \in S})$ for 2HashTDH to output z, the intermediate value in Combine. Given x, z and ρ, the output of Combine is fixed. So, if an arbitrary set of partial evaluations produces the same z, Combine would output the same thing. Moreover, if PubCombine produces a $z^\star$ different from z, then $z^{\star\rho^{-1}} \neq z^{\rho^{-1}}$, and output of Combine will be different with high probability (assuming that $\mathcal{H}_1$ is collision-resistant).

5 Password-based Threshold Authentication

In a password-based threshold authentication (PbTA) system, there are n servers and any number of clients. PbTA is naturally split into four phases: (i) during a global set-up phase, a master secret key is shared among the servers, which they later use to generate authentication tokens, (ii) in the registration phase, a client C computes sign-up messages (one for each server) based on its username and password and sends them to the servers. Each server processes the message it receives and stores a unique record for that client. (iii) in the sign-on phase, a client initiates authentication by sending a request message that incorporates its username/password and additional information to be included in the token. Each server computes a response using its record for the client. This response contains shares of the authentication token the client eventually wants to obtain. If client's password is a match he is able to combine and finalize the token shares into a single valid token for future accesses. (iv) The finalized token can be verified using a verification algorithm that takes a public or private (depending on the token type) verification key to validate that the token was generated using the unique master secret key. The verification process can also be distributed among multiple servers (may be required for MAC-based tokens) but for simplicity we use a centralized verification phase.

We also note that in a PbTA scheme, clients need not store any persistent secret information. The only secret they need to sign-on is their password which would not be stored anywhere. The device(s) a client uses to sign-on can store certain public parameters of the system (e.g. the identities of the servers).

For simplicity, we assume that clients choose passwords uniformly at random from a space P. Our definitions can be extended to the general case.

5.1 Algorithms

Definition 5.1 (Password-based Threshold Authentication) *A PbTA scheme* Π *is a tuple of seven PPT algorithms (*GlobalSetup, SignUp, Store, Request, Respond, Finalize, Verify*) that satisfies the correctness requirement below.*

- $\mathsf{GlobalSetup}(1^\kappa, n, t, \mathsf{P}) \to (\llbracket \mathsf{sk} \rrbracket, \mathsf{vk}, \mathsf{pp})$. *It takes the security parameter, number of servers* n*, a threshold* t *and the space of passwords* P *as inputs. It outputs a secret key* sk*, shares* $\mathsf{sk}_1, \mathsf{sk}_2, \ldots, \mathsf{sk}_n$ *of the key, and a verification key* vk*. The public parameters* pp *include all the inputs to* GlobalSetup *and some other information if needed.*

  pp *will be an implicit input in the algorithms below. The* n *servers will be denoted by* $S_1, \ldots, S_n$*.* S_i *receives* $(\mathsf{sk}_i, \mathsf{pp})$ *and initializes a set of records* $\text{REC}_i := \emptyset$*, for* $i \in [n]$*.*

Registration phase.

- $\mathsf{SignUp}(C, \mathsf{pwd}) \to \{(C, \mathsf{msg}_i)\}_{i\in[n]}$. *It takes as inputs a client id* C *and a password* $\mathsf{pwd} \in \mathsf{P}$*, and outputs a message for each server.*
- $\mathsf{Store}(C, \mathsf{msg}_i) =: \mathsf{rec}_{i,C}$. *It takes as input a client id* C *and a message* msg_i*, and outputs a record* $\mathsf{rec}_{i,C}$. S_i *stores* $(C, \mathsf{rec}_{i,C})$ *in its list of records* REC_i *if no record for* C *exists; otherwise, it does nothing.*

Sign-on phase.

- $\mathsf{Request}(C, \mathsf{pwd}, x, \mathcal{T}) \to (\mathsf{st}, \{(C, x, \mathsf{req}_i)\}_{i\in\mathcal{T}})$. *It takes as inputs a client id* C*, a password* pwd*, a value* x*, and a set* $\mathcal{T} \subseteq [n]$*, and outputs a secret state* st *and request messages* $\{\mathsf{req}_i\}_{i\in\mathcal{T}}$. *For* $i \in \mathcal{T}$*,* (C, x, req_i) *is sent to* S_i.
- $\mathsf{Respond}(\mathsf{sk}_i, \mathrm{REC}_i, C, x, \mathsf{req}_i) \to \mathsf{res}_i$. *It takes as inputs the secret key share* sk_i*, the record set* REC_i*, a client id* C*, a value* x *and a request message* req_i*, and outputs a response message* res_i.
- $\mathsf{Finalize}(\mathsf{st}, \{\mathsf{res}_i\}_{i\in\mathcal{T}}) =: \mathsf{tk}$. *It takes as input a secret state* st *and response messages* $\{\mathsf{res}_i\}_{i\in\mathcal{T}}$*, and outputs a token* tk.

Verification.

- $\mathsf{Verify}(\mathsf{vk}, C, x, \mathsf{tk}) \to \{0, 1\}$. *It takes as inputs the verification key* vk*, a client id* C*, a value* x *and a token* tk*, and outputs* 1 *(denotes validity) or* 0.

Correctness. *For all* $\kappa \in \mathbb{N}$*, any* $n, t \in \mathbb{N}$ *such that* $t \le n$*, any password space* P*, all* $([\![\mathsf{sk}]\!], \mathsf{vk}, \mathsf{pp})$ *generated by* $\mathsf{Setup}(1^\kappa, n, t, \mathsf{P})$*, any client id* C*, any password* $\mathsf{pwd} \in \mathsf{P}$*, any value* x*, and any* $\mathcal{T} \subseteq [n]$ *of size at least* t*, if*

- $((C, \mathsf{msg}_1), \ldots, (C, \mathsf{msg}_n)) \leftarrow \mathsf{SignUp}(C, \mathsf{pwd})$,
- $\mathsf{rec}_{i,C} := \mathsf{Store}(C, \mathsf{msg}_i)$ *for* $i \in [n]$,
- $(\mathsf{st}, \{(C, x, \mathsf{req}_i)\}_{i\in\mathcal{T}}) \leftarrow \mathsf{Request}(C, \mathsf{pwd}, x, \mathcal{T})$,
- $\mathsf{res}_i \leftarrow \mathsf{Respond}(\mathsf{sk}_i, \mathrm{REC}_i, C, x, \mathsf{req}_i)$ *for* $i \in \mathcal{T}$*, and*
- $\mathsf{tk} := \mathsf{Finalize}(\mathsf{st}, \{\mathsf{res}_i\}_{i\in\mathcal{T}})$,

then $\mathsf{Verify}(\mathsf{vk}, C, x, \mathsf{tk}) = 1$.

5.2 Security properties

We define security properties for PbTA with the help of a security game, described in Figure 7 in detail. In the security game, an adversary Adv gets access to a number of oracles, which run PbTA algorithms and do some bookkeeping. [4]

We do not allow the adversary to interfere with the registration phase. We assume that registration happens over secure channels. In practice, a client would establish a TLS connection with the servers over which it will send the sign-up messages. (Thus, the *actual* number of rounds in registration could be several.) The sign-on phase, however, is completely under the control of the adversary. Adversary can insert, delete or modify messages sent between clients and servers, even if client/server is not corrupt. This is captured by providing Adv access to three oracles for the three different algorithms of the sign-on phase (as opposed to just one oracle for registration). Adv can give any input to these oracles.

At the start of the game, $\mathsf{GlobalSetup}$ is run to generate shares of the master secret, verification key, public parameters and decryption keys. Public parameters are given to Adv.

[4] During a run of an oracle, if an algorithm does not produce a valid output, then the oracle stops immediately and returns $\perp$. We do not make this explicit in Figure 7 for simplicity.

$\underline{\mathsf{SecGame}_{\Pi,\mathsf{Adv}}(1^\kappa, n, t, \mathsf{P})}$:

- $([\![\mathsf{sk}]\!], \mathsf{vk}, \mathsf{pp}, (\mathsf{SK}_1, \ldots, \mathsf{SK}_n)) \leftarrow \mathsf{GlobalSetup}(1^\kappa, n, t, \mathsf{P})$
- $(\mathcal{U}, C^\star, \mathsf{st}_{\mathsf{adv}}) \leftarrow \mathsf{Adv}(\mathsf{pp})$ *# $\mathcal{U}$: corrupt servers, $C^\star$: targeted client*
- $\mathcal{V} := \emptyset$ *# set of corrupt clients*
- $\mathsf{PwdList} := \emptyset$ *# list of (C, pwd) pairs, indexed by C*
- $\mathsf{ReqList}_{C,i} := \emptyset$ for $i \in [n]$ *# token requests C makes to S_i*
- $\mathsf{ct} := 0$, $\mathsf{LiveSessions} = []$ *# LiveSessions is indexed by ct*
- $\mathsf{TokList} := \emptyset$ *# list of tokens generated through $\mathcal{O}_{\mathsf{final}}$*
- $Q_{C,i} := 0$ for all C and $i \in [n]$
- $Q_{C,x} := 0$ for all C and x
- $\mathsf{out} \leftarrow \mathsf{Adv}^{\langle \mathcal{O} \rangle}(\{\mathsf{sk}_i\}_{i \in \mathcal{U}}, \{\mathsf{SK}_i\}_{i \in \mathcal{U}}, \mathsf{st}_{\mathsf{adv}})$

$\underline{\mathcal{O}_{\mathsf{corrupt}}(C)}$.

- $\mathcal{V} := \mathcal{V} \cup \{C\}$
- if $(C, \star) \in \mathsf{PwdList}$, return $\mathsf{PwdList}[C]$

$\underline{\mathcal{O}_{\mathsf{register}}(C)}$.

- require: $\mathsf{PwdList}[C] = \perp$
- $\mathsf{pwd} \leftarrow_{\$} \mathsf{P}$
- add (C, pwd) to $\mathsf{PwdList}$
- $((C, \mathsf{msg}_1), \ldots, (C, \mathsf{msg}_n)) \leftarrow \mathsf{SignUp}(C, \mathsf{pwd})$
- $\mathsf{rec}_{i,C} := \mathsf{Store}(C, \mathsf{msg}_i)$ for all $i \in [n]$
- add $\mathsf{rec}_{i,C}$ to REC_i for all $i \in [n]$

$\underline{\mathcal{O}_{\mathsf{req}}(C, x, \mathcal{T})}$.

- require: $\mathsf{PwdList}[C] \neq \perp$
- $(\mathsf{st}, \{\mathsf{req}_i\}_{i \in \mathcal{T}}) \leftarrow \mathsf{Request}(C, \mathsf{PwdList}[C])$
- $\mathsf{LiveSessions}[\mathsf{ct}] := \mathsf{st}$
- add req_i to $\mathsf{ReqList}_{C,i}$ for $i \in \mathcal{T}$
- increment ct by 1
- return $\{\mathsf{req}_i\}_{i \in \mathcal{T}}$

$\underline{\mathcal{O}_{\mathsf{resp}}(i, C, x, \mathsf{req}_i)}$.

- $\mathsf{res}_i \leftarrow \mathsf{Respond}(\mathsf{sk}_i, \mathrm{REC}_i, C, x, \mathsf{req}_i)$
- if $\mathsf{req}_i \notin \mathsf{ReqList}_{C,i}$, increment $Q_{C,i}$ by 1
- increment $Q_{C,x}$ by 1
- return res_i

$\underline{\mathcal{O}_{\mathsf{final}}(\mathsf{ct}, \{\mathsf{res}_i\}_{i \in S})}$.

- $\mathsf{st} := \mathsf{LiveSessions}[\mathsf{ct}]$
- $\mathsf{tk} := \mathsf{Finalize}(\mathsf{st}, \{\mathsf{res}_i\}_{i \in S})$
- add tk to TokList
- return tk

$\underline{\mathcal{O}_{\mathsf{verify}}(C, x, \mathsf{tk})}$.

- return $\mathsf{Verify}(\mathsf{vk}, C, x, \mathsf{tk})$

Figure 7: Security game for PbTA

It outputs the set of servers $\mathcal{U}$ it wants to corrupt and the client $C^\star$ it wants to target.

A number of variables are initialized before Adv is given access to the oracles. $\mathcal{V}$ keeps track of clients as they are corrupted in the game, through $\mathcal{O}_{\mathsf{corrupt}}$ oracle. $\mathsf{PwdList}$ stores clients' passwords in the form of $(id, password)$ pairs, indexed by id, as they sign-up. $\mathsf{ReqList}_{C,i}$ stores the requests generated by C for the i-th server. These requests will not be counted against the adversary, as we will see later.

$\mathcal{O}_{\mathsf{req}}(C, x, \mathcal{T})$ allows Adv to start a sign-on session of C—who may not be corrupt—with servers in $\mathcal{T}$ to generate a token on x. $\mathcal{O}_{\mathsf{req}}$ runs $\mathsf{Request}$ to generate request messages, using the password of C stored in $\mathsf{PwdList}$. While these messages are revealed to Adv, C's intermediate state st is stored in $\mathsf{LiveSessions}$ at position ct. Adv can resume this sign-on session at any point in the future by invoking $\mathcal{O}_{\mathsf{final}}$ with ct and any arbitrary responses from the servers in $\mathcal{T}$.

$\mathcal{O}_{\mathsf{resp}}$ can be invoked to get responses from a server as part of the sign-on phase. Adv can invoke $\mathcal{O}_{\mathsf{resp}}$ with any message (C, x, req_i) of its choice. $\mathcal{O}_{\mathsf{server}}$ does not check if req_i was indeed generated by C or not; a response is generated anyway, and returned to the adversary. However, if the request req_i was not generated by C before, then this could give some advantage to Adv in attacking C; so we increment a counter $Q_{C,i}$ in this case. A different counter $Q_{C,x}$ is incremented even if req_i was generated by C. This is just to count the number of times different servers are invoked on C and x. If this number is less than even $t - |\mathcal{U}|$, then Adv should not be able to generate a token on (C, x) except with negligible probability (see Def. 5.3, first point).

Note that the counters Q are separate for each client. If $\mathcal{O}_{\mathsf{server}}$ is invoked with a certain client id, then the counters for just that id are updated. When we define the security properties for PbTA below, only the counters for $C^\star$ (the target client) are taken into account. Thus, we consider Adv to be attacking $C^\star$ only when it reveals this id to the servers. In other words, we do not allow Adv to gain any advantage in attacking $C^\star$ if it pretends to be someone else.

$\mathcal{O}_{\mathsf{final}}$, as mentioned before, can be used to resume a sign-on session. Client's state st is retrieved from $\mathsf{LiveSessions}$, and $\mathsf{Finalize}$ is run on st and the server responses given as input. Adv can provide any arbitrary response on behalf of any server—even the ones that are not corrupt. The token generated through $\mathsf{Finalize}$ is given to Adv and added to $\mathsf{TokList}$. Finally, Adv can use $\mathcal{O}_{\mathsf{verify}}$ to check if a token is valid or not.

We are now ready to formally state the two security properties we would like any PbTA scheme to satisfy.

Definition 5.2 (Password Safety) *A PbTA scheme* Π *is* password safe *if for all* $n, t \in \mathbb{N}$, $t \leq n$, *all password space* P *and all PPT adversary* Adv *in* $\mathsf{SecGame}_{\Pi,\mathsf{Adv}}(1^\kappa, n, t, \mathsf{P})$ *(Figure 7), there exists a negligible function* negl *s.t.*

$$\Pr[C^\star \notin \mathcal{V} \wedge \mathsf{out} = \mathsf{PwdList}[C^\star] \neq \bot] \leq \frac{\mathsf{MAX}_{|\mathcal{U}|,t}(Q_{C^\star,1}, \ldots, Q_{C^\star,n}) + 1}{|\mathsf{P}|} + \mathsf{negl}(\kappa). \quad (5.1)$$

To get some intuition into the above definition, consider the following attack. Adv guesses a password for $C^\star$, generates request messages on its own (so that it knows the intermediate state), invokes $\mathcal{O}_{\mathsf{resp}}$ to get the corresponding responses, combines them using $\mathsf{Finalize}$ to get a token, and finally checks if the token is valid or not. If the password guess was correct, then the token would be valid by the correctness property of PbTA.

As such, Adv is not restricted to attacking a PbTA scheme in the above manner. However, we require that, essentially, this is the best it can do. $\mathsf{MAX}_{|\mathcal{U}|,t}(Q_{C^\star,1}, \ldots, Q_{C^\star,n})$ in Eq. 5.1 gives a bound on the number of password attempts Adv can make through the above attack.

We do not penalize Adv for just replaying the requests generated by C itself by not incrementing $Q_{C,i}$ in those cases. The additive factor of 1 captures the possibility that Adv can output a new guess at the end of the game (similar to the obliviousness property for TOPRF, see Def. 4.3).

Definition 5.3 (Unforgeability) *A PbTA scheme* Π *is* unforgeable *if for all* $n, t \in \mathbb{N}$, $t \leq n$, *all password space* P *and all PPT adversary* Adv *in* $\mathsf{SecGame}_{\Pi,\mathsf{Adv}}(1^\kappa, n, t, \mathsf{P})$ *(Figure 7), there exists a negligible function* negl *s.t.*

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

$$\Pr[\mathsf{Verify}(\mathsf{vk}, C^\star, x^\star, \mathsf{tk}^\star) = 1] \leq \mathsf{negl}(\kappa); \tag{5.2}$$

– *else*

$$\Pr[C^\star \notin \mathcal{V} \wedge \mathsf{tk}^\star \notin \mathsf{TokList} \wedge \mathsf{Verify}(\mathsf{vk}, C^\star, x^\star, \mathsf{tk}^\star) = 1] \leq \frac{\mathsf{MAX}_{|\mathcal{U}|,t}(Q_{C^\star,1}, \ldots, Q_{C^\star,n})}{|\mathsf{P}|} + \mathsf{negl}(\kappa), \tag{5.3}$$

where Adv*'s output* out *is parsed as* $(x^\star, \mathsf{tk}^\star)$.

The security game for unforgeability is the same as password-safety (Figure 7) but Adv produces a token $\mathsf{tk}^\star$ now. The probability of it being valid on $(C^\star, x^\star)$ depends on several cases. First, if the value of $Q_{C^\star,x^\star}$ is smaller than even $t - |\mathcal{U}|$, then Adv didn't even contact *enough* servers on $(C^\star, x^\star)$. So we would like its probability of producing a valid token to be negligible. (Eq. 5.2 also captures that querying servers on (C, x) for a different C or x than $C^\star$ and $x^\star$ should not help.)

If Adv does contact enough servers and $C^\star$ was corrupted, then Adv can easily generate a valid token; so this case is not interesting. However, if $C^\star$ is not corrupt but Adv is able to guess its password, then it can also produce a valid token (with respect to $C^\star$ only). Comparing Eq. 5.3 and 5.1, one can see that unforgeability property basically requires that this is the best Adv can do.

6 PASTA: Our Construction

In this section we present PASTA, our framework for building PbTA schemes. PASTA provides a way to combine any threshold token generation scheme (TTG) and any threshold oblivious PRF (TOP) in a black-box way to build a PbTA scheme that provides strong password-safety and unforgeability guarantees. Figure 8 provides a complete description of the framework.

PASTA uses the two main underlying primitives, TTG and TOP, in a fairly light-weight manner. The sign-on phase, which consists of Request, Respond and Finalize, does not add any public-key operations on top of what the primitives may have. Request runs TOP.Encode once; Respond runs both TOP.Eval and TTG.PartEval, but only once each; and, Finalize runs TOP.Combine and TTG.Combine once each. Even though number of decryptions in Finalize is proportional to t, these operations are very fast symmetric-key operations. Thus, PASTA

makes minimal use of the two primitives that it builds on and its performance is mainly governed by the efficiency of these primitives.

PASTA needs a *key-binding* symmetric-key encryption scheme so that when a ciphertext is decrypted with a wrong key, decryption fails [Fis99]. Key-binding can be obtained very efficiently in the random oracle model, for e.g., by appending a hash of the secret key to every ciphertext.

For the sign-on phase, PASTA assumes that the servers communicate to clients over authenticated channels so that an adversary cannot send arbitrary messages to a client on behalf of honest servers. PASTA does *not* assume that these channels provide any confidentiality. Observe that if there is an authenticated channel in the other direction, namely the servers can identity the sender of every message they receive, then passwords are not needed, and hence a PbTA scheme is moot.

An important feature of PASTA, especially from the point of view of proving security, is that the use of TOP and TTG overlaps very slightly. The output of TOP is used to encrypt the partial evaluations of TTG but, apart from that, they operate independently. Thus, even if TTG is broken in some manner, it would not affect the safety of clients' passwords. Furthermore, even if TOP is broken, a threshold number of servers would still be needed to generate a token. However, PASTA must prevent against several other attack scenarios, as captured by the game in Figure 7. The formal security guarantee of PASTA is stated as follows.

Theorem 6.1 (Security of PASTA) *If* TTG *is an unforgeable threshold token generation scheme (Def. 3.3),* TOP *is an unpredictable (Def. 4.2) and oblivious (Def. 4.3) TOPRF, and* SKE *is a key-binding CPA-secure symmetric-key encryption scheme, then the PbTA scheme PASTA as described in Figure 8 is password-safe (Def. 5.2) and unforgeable (Def. 5.3) when* $\mathcal{H}$ *is modeled as a random oracle.*

Password-safety and unforgeability properties are proved in Appendix B.1 and B.2 respectively.

7 Performance Evaluation

We implement PASTA for four types of threshold token generation schemes: a block-cipher based MAC [NPR99], a DDH-based (requires exponentiations) MAC [NPR99], a pairing based signature [Bol03] and an RSA based signature [Sho00]. In this section we report on the performance of these instantiations.

7.1 Implementation Details

PASTA is a generic construction consisting of two building blocks: a threshold oblivious pseudo-random function and a threshold token generation scheme. We implement PASTA with the 2HashTDH TOPRF protocol of Jarecki et al. [JKKX17] and the aforementioned TTG schemes (see Appendix C for their descriptions) to obtain four types of tokens. To the best of our knowledge, most of these TTG schemes were not implemented before.

We implement pseudorandom functions (PRFs) using AES-NI and hash functions using Blake2 [bla]. The elliptic curve operations, pairing operations, and RSA operations are implemented using the Relic library [AG]. The key length in AES-NI is 128 bits. The cyclic group used in 2HashTDH TOPRF and the DDH based MAC is the group $\mathbb{G}_1$ on 256-bit

Ingredients.

- A threshold token generation scheme

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

- A threshold oblivious PRF

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

- A symmetric-key encryption scheme

$$\mathsf{SKE} := (\mathsf{SKE.Encrypt}, \mathsf{SKE.Decrypt}).$$

- A hash function $\mathcal{H}$.

$\underline{\mathsf{GlobalSetup}(1^\kappa, n, t, \mathsf{P}) \to (\llbracket \mathsf{sk} \rrbracket, \mathsf{vk}, \mathsf{pp})}$.

- Run $\mathsf{TTG.Setup}(1^\kappa, n, t)$ to get $(\llbracket \mathsf{tsk} \rrbracket, \mathsf{tvk}, \mathsf{tpp})$.
- Set $\mathsf{sk}_i := \mathsf{tsk}_i$ for all $i \in [n]$, $\mathsf{vk} := \mathsf{tvk}$ and $\mathsf{pp} := (\kappa, n, t, \mathsf{P}, \mathsf{tpp})$.

$\underline{\mathsf{SignUp}(C, \mathsf{pwd}) \to ((C, \mathsf{msg}_1), \ldots, (C, \mathsf{msg}_n))}$.

- Run $\mathsf{TOP.Setup}(1^\kappa, n, t)$ to get $(\llbracket k \rrbracket, \mathsf{opp})$.
- Compute $h := \mathsf{TOP}(k, \mathsf{pwd})$ and $h_i = \mathcal{H}(h \| i)$ for $i \in [n]$.
- Set $\mathsf{msg}_i := (k_i, h_i)$ for $i \in [n]$.

$\underline{\mathsf{Store}(\mathsf{SK}_i, C, \mathsf{msg}_i) =: \mathsf{rec}_{i,C}}$.

- Parse msg_i as (k_i, h_i).
- Set $\mathsf{rec}_{i,C} := (k_i, h_i)$

$\underline{\mathsf{Request}(C, \mathsf{pwd}, x, \mathcal{T}) \to (\{(C, x, \mathsf{req}_i)\}_{i \in \mathcal{T}}, \mathsf{st})}$.

- If $|\mathcal{T}| < t$, output $\perp$.
- Pick a ρ at random. Run $\mathsf{TOP.Encode}(\mathsf{pwd}, \rho)$ to get c.
- Set $\mathsf{req}_i := c$ for all $i \in [n]$ and $\mathsf{st} := (C, \mathsf{pwd}, \rho, \mathcal{T})$.

$\underline{\mathsf{Respond}(\mathsf{sk}_i, \mathrm{REC}_i, C, x, \mathsf{req}_i) \to \mathsf{res}_i}$.

- If $\mathsf{rec}_{i,C} \notin \mathrm{REC}_i$, output $\perp$. Else, parse $\mathsf{rec}_{i,C}$ as (k_i, h_i).
- Run $\mathsf{TOP.Eval}(k_i, \mathsf{req}_i)$ to get z_i.
- Run $\mathsf{TTG.PartEval}(\mathsf{tsk}_i, C \| x)$ to get y_i.
- Set $\mathsf{res}_i := (z_i, \mathsf{SKE.Encrypt}(h_i, y_i))$.

$\underline{\mathsf{Finalize}(\mathsf{st}, \{\mathsf{res}_i\}_{i \in S}) \to \mathsf{tk}}$.

- Parse res_i as (z_i, ctxt_i) and st as $(C, \mathsf{pwd}, \rho, \mathcal{T})$.
- If $S \neq \mathcal{T}$, output $\perp$.
- Run $\mathsf{TOP.Combine}(\mathsf{pwd}, \{(i, z_i)\}_{i \in \mathcal{T}}, \rho)$ to get h.
- For all $i \in \mathcal{T}$, compute $h_i := \mathcal{H}(h \| i)$ and $y_i := \mathsf{SKE.Decrypt}(h_i, \mathsf{ctxt}_i)$.
- Finally, set tk to be $\mathsf{TTG.Combine}(\{i, y_i\}_{i \in \mathcal{T}})$.

(If any of TOP.Combine, SKE.Decrypt or TTG.Combine fail, $\perp$ is output.)

$\underline{\mathsf{Verify}(\mathsf{vk}, C, x, \mathsf{tk}) \to \{0, 1\}}$.

- Output $\mathsf{TTG.Verify}(\mathsf{tvk}, C \| x, \mathsf{tk})$.

Figure 8: A complete description of PASTA

Barreto-Naehrig curves (BN-curves) [BN06]. Pairing is implemented on 256-bit BN-curves. The key length in RSA based signature is 2048 bits.

In order to evaluate the performance, we implement various settings described below. The experiments are run on a single server with 2x 24-core 2.2 GHz CPUs and 64 GB of RAM. We run all the parties on different cores of the same server (1 core per server), and simulate network connections using the Linux `tc` command: a LAN setting with 10 Gbps network bandwidth and 0.1 ms round-trip latency; a WAN setting with 40 Mbps network bandwidth and a simulated 80 ms round-trip latency.

7.2 Token Generation Time

Table 1 shows the total runtime for a client to generate a single token in the sign-on phase after registration in our PASTA protocol. We show experiments for various types of tokens in the LAN and WAN settings and different values of (n, t) where n is the number of servers and t is the threshold. The reported time is an average of 10,000 token requests. We discuss a few observations below.

	(n,t)	Sym-MAC	Public-MAC	Pairing-Sig	RSA-Sig
LAN	$(2,2)$	1.3	1.7	1.7	14.5
	$(3,2)$	1.3	1.7	1.7	14.5
	$(6,2)$	1.3	1.7	1.7	14.5
	$(10,2)$	1.3	1.7	1.7	14.5
	$(10,3)$	1.6	2.1	2.1	15.1
	$(10,5)$	2.3	3.0	3.0	16.8
	$(10,7)$	3.0	3.9	3.9	19.1
	$(10,10)$	4.1	5.4	5.4	22.6
WAN	$(2,2)$	81.4	81.8	81.8	94.6
	$(3,2)$	81.4	81.8	81.8	94.6
	$(6,2)$	81.4	81.8	81.8	94.6
	$(10,2)$	81.4	81.9	81.9	94.6
	$(10,3)$	81.7	82.2	82.2	95.0
	$(10,5)$	82.4	83.1	83.1	96.9
	$(10,7)$	83.1	83.9	83.9	99.2
	$(10,10)$	84.2	85.4	85.4	102.8

Table 1: The total runtime (in milliseconds) of our PASTA protocol for generating a single token for the number of servers n and threshold t in LAN and WAN settings.

Notice that for the same threshold $t = 2$ and the same type of token, different values of n result in similar runtime. This is aligned with our construction: for a threshold t, the client only needs to communicate with t servers, and the communication and computation cost for every server is the same, hence the total runtime should also be the same. Therefore, the total runtime is independent of n and only depends on the threshold t. For other values of threshold t, we only report the runtime for $n = 10$; the runtime for other values of n would be roughly the same.

Also notice that for the same (n, t) and same type of token, the runtime in the WAN setting is roughly the runtime in the LAN setting plus 80 ms round-trip latency. This is because in our protocol, the client sends a request to t servers and receive their responses in parallel. The communication complexity is very small, hence the bulk of communication overhead is roughly the round-trip latency. It is worth noting that the PASTA protocol has

the minimal two rounds of interaction, and hence this overhead is inevitable in the WAN setting.

The runtime of public-key based MAC and pairing based signature are almost the same under the same setting. This is because in our implementation, TTG schemes for public-key based MAC and pairing based signature are both implemented on the 256-bit Barreto-Naehrig curves (BN-curves) [BN06] in group $\mathbb{G}_1$. This group supports Type-3 pairing operation and is believed to satisfy the Decisional Diffie-Hellman (DDH) assumption, hence a good fit for both primitives.

We do not report the runtime for user registration because (i) it is done only once for every user and (ii) it is more efficient than obtaining a token.

7.3 Time Breakdown

We show the runtime breakdown for three different (n, t) values in Table 2 in the LAN setting. For each value of (n, t) in the table, the first row is the total runtime, and the second and third rows are the computation time on the client side and on a single server, respectively.

	Sym-MAC	Public-MAC	Pairing-Sig	RSA-Sig
$(10, 2)$	1.3	1.7	1.7	14.5
Client	1.0	1.2	1.2	2.8
Server	0.2	0.4	0.4	11.4
$(10, 5)$	2.3	3.0	3.0	16.8
Client	1.9	2.4	2.4	5.2
Server	0.2	0.4	0.4	11.4
$(10, 10)$	4.1	5.4	5.4	22.6
Client	3.7	4.6	4.6	10.7
Server	0.2	0.4	0.4	11.5

Table 2: Breakdown of runtime (in milliseconds) in LAN setting.

As shown in the table, for the same token type the computation time on a single server does not vary. On the other hand, the computation on the client grows with the threshold. Figure 9 shows the dependence of the computation time at the client side on the threshold t. For all four types of tokens, the computation time grows linearly in the threshold t.

7.4 Comparison with Naïve Solutions

We implement two naïve solutions to compare with our PASTA protocol:

– *Plain Solution:* The client signs on to a single server with its username/password. The server verifies its credential and then issues an authentication token using a master secret key.

– *Threshold Solution:* This solution utilizes a threshold token generation scheme. The secret key shares $\mathsf{sk}_1, \mathsf{sk}_2, \ldots, \mathsf{sk}_n$ of the threshold scheme are distributed among the n servers. The client signs on with its username/password to t servers, where each server verifies its credential and then issues a share of the token. The client combines the shares received from the servers to generate the final token.

In the plain solution, a breached server would enable the attacker to (i) recover the master secret key and (ii) perform offline dictionary attacks to recover users' passwords. Comparing

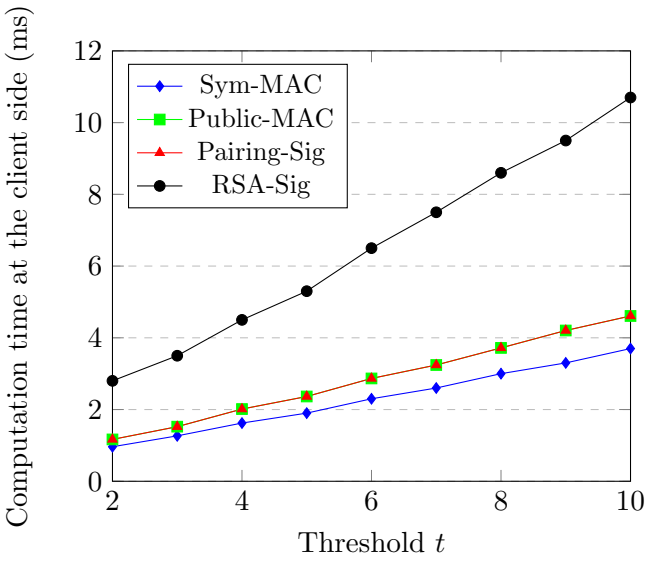


Figure 9: Growth of computation time (in milliseconds) at client's side with threshold t. (n is fixed to 10.)

our solution with the plain solution presents the extra cost of protecting both the master secret key and users' passwords. In the threshold solution, if up to $t-1$ servers are breached, then the master secret key stays secure, but users' passwords are vulnerable to offline dictionary attacks. Comparing our solution with the threshold solution gives the extra cost of protecting users' passwords.

		(n,t)	Sym-MAC	Public-MAC	Pairing-Sig	RSA-Sig
LAN	Plain		0.1	0.4	0.4	11.3
	Threshold	$(10,2)$	0.1	0.6	0.6	13.2
		$(10,3)$	0.1	0.6	0.6	13.3
		$(10,5)$	0.2	0.9	0.9	14.4
		$(10,7)$	0.3	1.2	1.2	16.0
		$(10,10)$	0.4	1.5	1.5	18.6
WAN	Plain		80.2	80.5	80.5	91.4
	Threshold	$(10,2)$	80.2	80.7	80.7	93.3
		$(10,3)$	80.2	80.7	80.7	93.4
		$(10,5)$	80.3	81.0	81.0	94.5
		$(10,7)$	80.4	81.3	81.3	96.1
		$(10,10)$	80.5	81.6	81.6	98.8

Table 3: The total time (in milliseconds) it takes to generate a single token through naïve solutions, for various settings in LAN and WAN networks.

Table 3 shows the total runtime for a client to generate a single token after registration using the plain solution and the threshold solution for different values of (n,t). The reported time is an average of 10,000 token requests in LAN and WAN settings. For the same setting and the same type of token, the runtime in the WAN network is roughly the runtime in the LAN network plus 80 ms round-trip latency, for the same reason discussed above for the PASTA protocol. Notice that in the threshold solution, the total runtime is independent of n and only depends on the threshold t. Hence we only report the runtime for the same $n = 10$

and different thresholds.

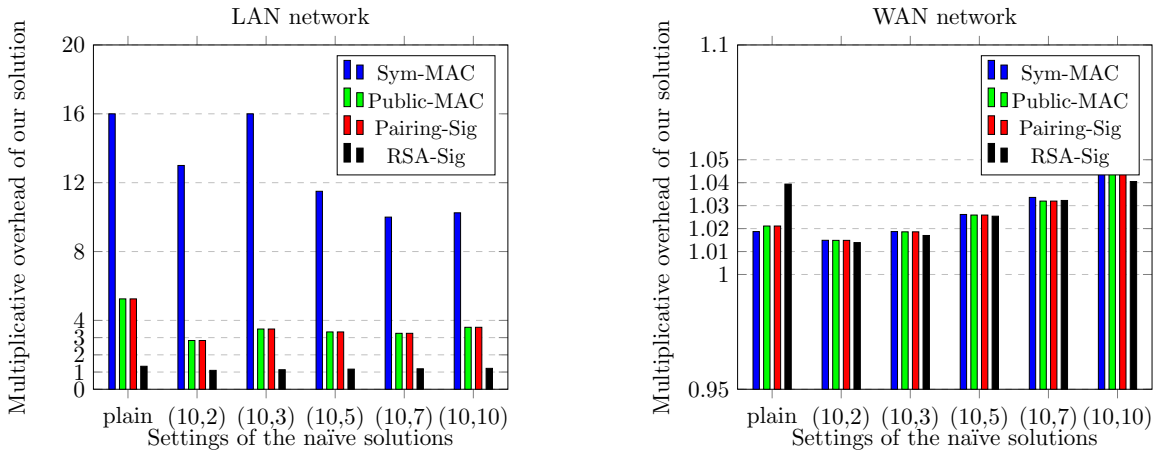


Figure 10: Multiplicative overhead of our solution in runtime compared to naïve solutions in LAN and WAN networks.

We compare our solution with the two naïve solutions and show the multiplicative overhead of our solution in Figure 10. The two figures represent the comparison in the LAN and WAN network, respectively. Different types of tokens are represented in different colors. In each picture, the first set of four bars represent the overhead of our solution compared to the plain solution. Note that there is no notion of (n, t) in the plain solution, hence we pick a setting $(5, 3)$ for our solution to compare with the plain solution. If comparing the plain solution with other (n, t) settings of our solution, the results would be slightly different. The remaining five sets of bars in each figure represent the overhead of our solution compared to the threshold solution for various values of (n, t). When comparing with those, we use the same (n, t) setting of our solution.

In the LAN network, notice that there is nearly no overhead for the RSA-based token generation. The overhead for public-key based MAC and pairing based signature is a small constant. There is a higher overhead for symmetric-key based MAC. This is because the naïve solutions only involve symmetric-key operations while our solution involves public-key operations, which is much more expensive. This overhead is necessary as we prove in Appendix D that public-key operations are necessary to achieve a password-based threshold authentication (PbTA) system.

In the WAN network, since the most time-consuming component is the network latency in our protocol as well as the naïve solutions, the overhead of our solution compared with the naïve solutions is fairly small. As shown in Figure 10, the overhead is less than 5% in all the settings and all token types.

References

[ACFP05] Michel Abdalla, Olivier Chevassut, Pierre-Alain Fouque, and David Pointcheval. A simple threshold authenticated key exchange from short secrets. In Bimal K. Roy, editor, *ASIACRYPT 2005*, volume 3788 of *LNCS*, pages 566–584. Springer, Heidelberg, December 2005.

[ACK+16] Joël Alwen, Binyi Chen, Chethan Kamath, Vladimir Kolmogorov, Krzysztof Pietrzak, and Stefano Tessaro. On the complexity of scrypt and proofs of space in the parallel random oracle model. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 358–387. Springer, Heidelberg, May 2016.

[ACNP16] Michel Abdalla, Mario Cornejo, Anca Nitulescu, and David Pointcheval. Robust password-protected secret sharing. In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows, editors, *ESORICS 2016, Part II*, volume 9879 of *LNCS*, pages 61–79. Springer, Heidelberg, September 2016.

[ACP+17] Joël Alwen, Binyi Chen, Krzysztof Pietrzak, Leonid Reyzin, and Stefano Tessaro. Scrypt is maximally memory-hard. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 33–62. Springer, Heidelberg, May 2017.

[AFP05] Michel Abdalla, Pierre-Alain Fouque, and David Pointcheval. Password-based authenticated key exchange in the three-party setting. In Serge Vaudenay, editor, *PKC 2005*, volume 3386 of *LNCS*, pages 65–84. Springer, Heidelberg, January 2005.

[AG] D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic.

[ama] Amazon OpenID. https://docs.aws.amazon.com/cognito/latest/developerguide/open-id.html. Accessed on September 19, 2018.

[AMN01] Michel Abdalla, Sara Miner, and Chanathip Namprempre. Forward-secure threshold signature schemes. In *Cryptographers Track at the RSA Conference*, pages 441–456. Springer, 2001.

[BD16] Jeremiah Blocki and Anupam Datta. CASH: A cost asymmetric secure hash algorithm for optimal password protection. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 371–386, 2016.

[BJSL11] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 11*, pages 433–444. ACM Press, October 2011.

[bla] BLAKE2 - fast secure hashing. https://blake2.net/. Accessed on September 19, 2018.

[BLMR13] Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. Key homomorphic PRFs and their applications. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 410–428. Springer, Heidelberg, August 2013.

[BMP00] Victor Boyko, Philip D. MacKenzie, and Sarvar Patel. Provably secure password-authenticated key exchange using Diffie-Hellman. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 156–171. Springer, Heidelberg, May 2000.

[BN06] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In Bart Preneel and Stafford Tavares, editors, *SAC 2005*, volume 3897 of *LNCS*, pages 319–331. Springer, Heidelberg, August 2006.

[Bol03] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In Yvo Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 31–46. Springer, Heidelberg, January 2003.

[BPR00] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 139–155. Springer, Heidelberg, May 2000.

[BS01] Mihir Bellare and Ravi Sandhu. The security of practical two-party RSA signature schemes. Cryptology ePrint Archive, Report 2001/060, 2001. http://eprint.iacr.org/2001/060.

[BZ17] Jeremiah Blocki and Samson Zhou. On the depth-robustness and cumulative pebbling cost of Argon2i. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 445–465. Springer, Heidelberg, November 2017.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[CEN15] Jan Camenisch, Robert R. Enderlein, and Gregory Neven. Two-server password-authenticated secret sharing UC-secure against transient corruptions. In Jonathan Katz, editor, *PKC 2015*, volume 9020 of *LNCS*, pages 283–307. Springer, Heidelberg, March / April 2015.

[CHK+05] Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. Universally composable password-based key exchange. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 404–421. Springer, Heidelberg, May 2005.

[CLLN14] Jan Camenisch, Anja Lehmann, Anna Lysyanskaya, and Gregory Neven. Memento: How to reconstruct your secrets from a single password in a hostile environment. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 256–275. Springer, Heidelberg, August 2014.

[CLN12] Jan Camenisch, Anna Lysyanskaya, and Gregory Neven. Practical yet universally composable two-server password-authenticated secret sharing. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12*, pages 525–536. ACM Press, October 2012.

[CLN15] Jan Camenisch, Anja Lehmann, and Gregory Neven. Optimal distributed password verification. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 182–194. ACM Press, October 2015.

[CLNS16] Jan Camenisch, Anja Lehmann, Gregory Neven, and Kai Samelin. Virtual smart cards: How to sign with a password and a server. In Vassilis Zikas and Roberto De Prisco, editors, *SCN 16*, volume 9841 of *LNCS*, pages 353–371. Springer, Heidelberg, August / September 2016.

[Dan] Daniel Sewell. Offline Password Cracking: The Attack and the Best Defense Against It. https://www.alpinesecurity.com/blog/offline-password-cracking-the-attack-and-the-best-defense-against-it. Accessed on September 19, 2018.

[DDFY94] Alfredo De Santis, Yvo Desmedt, Yair Frankel, and Moti Yung. How to share a function securely. In *26th ACM STOC*, pages 522–533. ACM Press, May 1994.

[DF90] Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 307–315. Springer, Heidelberg, August 1990.

[DG03] Mario Di Raimondo and Rosario Gennaro. Provably secure threshold password-authenticated key exchange. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 507–523. Springer, Heidelberg, May 2003.

[DK01] Ivan Damgård and Maciej Koprowski. Practical threshold RSA signatures without a trusted dealer. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 152–165. Springer, Heidelberg, May 2001.

[DKAN] Daniel Dinu, Dmitry Khovratovich, Jean-Philippe Aumasson, and Samuel Neves. Argon2. https://github.com/P-H-C/phc-winner-argon2. Github Repository: Accessed on September 19, 2018.

[fac] Facebook Login. https://developers.facebook.com/docs/facebook-login. Accessed on September 19, 2018.

[FIP02] NIST FIPS. 198: The keyed-hash message authentication code (hmac). *National Institute of Standards and Technology, Federal Information Processing Standards*, page 29, 2002.

[FIPR05] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Heidelberg, February 2005.

[Fis99] Marc Fischlin. Pseudorandom function tribe ensembles based on one-way permutations: Improvements and applications. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 432–445. Springer, Heidelberg, May 1999.

[Gan95] Ravi Ganesan. Yaksha: augmenting kerberos with public key cryptography. In *1995 Symposium on Network and Distributed System Security, (S)NDSS '95, San Diego, California, February 16-17, 1995*, pages 132–143, 1995.

[GGN16] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In Mark Manulis, Ahmad-Reza Sadeghi, and Steve Schneider, editors, *ACNS 16*, volume 9696 of *LNCS*, pages 156–174. Springer, Heidelberg, June 2016.

[GHKR08] Rosario Gennaro, Shai Halevi, Hugo Krawczyk, and Tal Rabin. Threshold RSA for dynamic and ad-hoc groups. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 88–107. Springer, Heidelberg, April 2008.

[GJKR96] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Robust threshold DSS signatures. In Ueli M. Maurer, editor, *EUROCRYPT'96*, volume 1070 of *LNCS*, pages 354–371. Springer, Heidelberg, May 1996.

[GK10] Adam Groce and Jonathan Katz. A new framework for efficient password-based authenticated key exchange. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS 10*, pages 516–525. ACM Press, October 2010.

[goo] Google Identity Platform – OpenID Connect. https://developers.google.com/identity/protocols/OpenIDConnect. Accessed on September 19, 2018.

[GT11] Kristian Gjosteen and Oystein Thuen. Password-based signatures. In *Public Key Infrastructures, Services and Applications - 8th European Workshop, EuroPKI 2011, Leuven, Belgium, September 15-16, 2011, Revised Selected Papers*, pages 17–33, 2011.

[HAP18] Yotam Harchol, Ittai Abraham, and Benny Pinkas. Distributed ssh key management with proactive rsa threshold signatures. Cryptology ePrint Archive, Report 2018/389, 2018. https://eprint.iacr.org/2018/389.

[IR90] Russell Impagliazzo and Steven Rudich. Limits on the provable consequences of one-way permutations. In Shafi Goldwasser, editor, *CRYPTO'88*, volume 403 of *LNCS*, pages 8–26. Springer, Heidelberg, August 1990.

[JKK14] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 233–253. Springer, Heidelberg, December 2014.

[JKKX16] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 276–291, 2016.

[JKKX17] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17*, volume 10355 of *LNCS*, pages 39–58. Springer, Heidelberg, July 2017.

[jwt] JSON Web Tokens. https://jwt.io/. Accessed on September 19, 2018.

[ker] Kerberos: The Network Authentication Protocol. https://web.mit.edu/kerberos/. Accessed on September 19, 2018.

[KMTG05] Jonathan Katz, Philip MacKenzie, Gelareh Taban, and Virgil Gligor. Two-server password-only authenticated key exchange. In John Ioannidis, Angelos

Keromytis, and Moti Yung, editors, *Applied Cryptography and Network Security*, pages 1–16, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[KOY01] Jonathan Katz, Rafail Ostrovsky, and Moti Yung. Efficient password-authenticated key exchange using human-memorable passwords. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 475–494. Springer, Heidelberg, May 2001.

[KOY03] Jonathan Katz, Rafail Ostrovsky, and Moti Yung. Forward secrecy in password-only key exchange protocols. In Stelvio Cimato, Clemente Galdi, and Giuseppe Persiano, editors, *SCN 02*, volume 2576 of *LNCS*, pages 29–44. Springer, Heidelberg, September 2003.

[KV11] Jonathan Katz and Vinod Vaikuntanathan. Round-optimal password-based authenticated key exchange. In Yuval Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 293–310. Springer, Heidelberg, March 2011.

[MPSN+03] Keith M. Martin, Josef Pieprzyk, Rei Safavi-Naini, Huaxiong Wang, and Peter R. Wild. Threshold macs. In Pil Joong Lee and Chae Hoon Lim, editors, *Information Security and Cryptology — ICISC 2002*, pages 237–252, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[MR03] Philip MacKenzie and Michael K Reiter. Networked cryptographic devices resilient to capture. *International Journal of Information Security*, 2(1):1–20, 2003.

[MSJ02] Philip D. MacKenzie, Thomas Shrimpton, and Markus Jakobsson. Threshold password-authenticated key exchange. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 385–400. Springer, Heidelberg, August 2002.

[NPR99] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and KDCs. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 327–346. Springer, Heidelberg, May 1999.

[oau] The OAuth 2.0 Authorization Framework: Bearer Token Usage. https://tools.ietf.org/html/rfc6750. Accessed on September 19, 2018.

[ope] The OpenID Connect. http://openid.net/connect/.

[phc] Password Hashing Competition. https://password-hashing.net/. Accessed on September 19, 2018.

[PS10] Kenneth G. Paterson and Douglas Stebila. One-time-password-authenticated key exchange. In Ron Steinfeld and Philip Hawkes, editors, *ACISP 10*, volume 6168 of *LNCS*, pages 264–281. Springer, Heidelberg, July 2010.

[sam] SAML Toolkits. https://developers.onelogin.com/saml. Accessed on September 19, 2018.

[Sho00] Victor Shoup. Practical threshold signatures. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 207–220. Springer, Heidelberg, May 2000.

[Tar] Tarsnap. Scrypt. https://github.com/Tarsnap/scrypt. Github Repository: Accessed on September 19, 2018.

[vau] Vault by Hashicorp. https://www.vaultproject.io/docs/internals/token.html. Accessed on September 19, 2018.

[WW13] Ding Wang and Ping Wang. Offline dictionary attack on password authentication schemes using smart cards. 2014:1–16, 01 2013.

[XS03] Shouhuai Xu and Ravi S. Sandhu. Two efficient and provably secure schemes for server-assisted threshold signatures. In Marc Joye, editor, *CT-RSA 2003*, volume 2612 of *LNCS*, pages 355–372. Springer, Heidelberg, April 2003.

[YHCL15] Xun Yi, Feng Hao, Liqun Chen, and Joseph K. Liu. Practical threshold password-authenticated secret sharing protocol. In Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl, editors, *ESORICS 2015, Part I*, volume 9326 of *LNCS*, pages 347–365. Springer, Heidelberg, September 2015.

A TOPRF: Proofs

A.1 Unpredictability

In this section we prove output unpredictability of our construction. Suppose there exists a PPT adversary Adv such that

$$\mathsf{Pr}[\mathsf{Unpredictability}_{\mathsf{TOP},\mathsf{Adv}}(1^\kappa, n, t) = 1] \geq \frac{\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n)}{|\mathcal{X}|} + \mathsf{non\text{-}negl}(\kappa). \tag{A.1}$$

We will consider two cases of Adv. In the first case, there exists $k \in \mathbb{N}$ such that when Adv calls $\mathcal{H}_1$ with k distinct valid (x, y) pairs, $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) < k$. In this case, we will use Adv to break the gap TOMDH assumption. In the second case, for any $k \in \mathbb{N}$, when Adv calls $\mathcal{H}_1$ with k valid (x, y) pairs, $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) \geq k$. In this case, we will prove that information theoretically Formula A.1 does not hold.

First Case There exists $k \in \mathbb{N}$ such that when Adv calls $\mathcal{H}_1$ with k distinct valid (x, y) pairs, $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) < k$. Then we construct an adversary $\mathcal{B}$ that breaks the gap TOMDH assumption (see Definition 3.1).

We construct $\mathcal{B}$ as follows. It first receives $(p, g, \mathbb{G}, g_1, \ldots, g_N)$ from the TOMDH game $\mathsf{One\text{-}More}_{\mathcal{B}}(1^\kappa, t', t, n, N)$, presents $\mathsf{pp} := (p, g, \mathbb{G}, n, t, \mathcal{H}_1, \mathcal{H}_2)$ to Adv and gets back $\mathcal{U}$. It then generates $\{\alpha_i\}_{i\in\mathcal{U}}$ at random, sends $\{(i, \alpha_i)\}_{i\in\mathcal{U}}$ to the TOMDH game, and sends $\{\alpha_i\}_{i\in\mathcal{U}}$ to Adv. It then samples $\tilde{x} \leftarrow_\$ \mathcal{X}$, set $\mathcal{L} := []$, set $\mathsf{LIST} := \emptyset$, and set $k := 0$. Then $\mathcal{B}$ computes $g_1^{\alpha_i}$ for $i \in \mathcal{U}$, calls $\mathcal{O}(i, g_1)$ to get $g_1^{\alpha_i}$ for all $i \in [n] \setminus \mathcal{U}$, and computes $y_1 := g_1^{\mathsf{sk}}$. It adds (g_1, y_1) to LIST, sets $q = 1$, and handles Adv's oracle queries as follows:

- On Adv's call to $\mathcal{O}_{\mathsf{enc\&eval}}()$: Pick an unused g_j where $j \in [N]$, set $c := g_j$, compute $z_i \leftarrow \mathsf{Eval}(\mathsf{sk}_i, c)$ for $i \in \mathcal{U}$ and call $\mathcal{O}(i, c)$ to get z_i for all $i \in [n] \setminus \mathcal{U}$. Use $[n]$ to find coefficients $\{\lambda_i\}_{i\in[n]}$ and compute $y_j := \prod_{i\in[n]} z_i^{\lambda_i}$. Add (g_j, y_j) to LIST, and increment q by 1. Compute $h := \mathcal{H}_1(\tilde{x}, y_j)$, and return $(c, \{z_i\}_{i\in[n]\setminus\mathcal{U}}, h)$ to Adv.
- On Adv's call to $\mathcal{O}_{\mathsf{eval}}(i, c)$: Call $\mathcal{O}(i, c)$ in the TOMDH game and return the output to Adv.

– On Adv's call to $\mathcal{H}_1(x, y)$: If $x \notin \mathcal{L}$, compute $\mathcal{H}_1(x, y)$ honestly and return to Adv. Otherwise, let $g_j := \mathcal{L}[x]$. If $\log_{g_j} y = \log_{g_1} y_1$ and $(g_j, \star) \notin \mathsf{LIST}$ (i.e., (g_j, y) is a new valid pair), increment k by 1, add (g_j, y) to LIST, and output LIST in the TOMDH game if $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) < k$. Compute $\mathcal{H}_1(x, y)$ honestly and return to Adv.

– On Adv's call to $\mathcal{H}_2(x)$: If $x \in \mathcal{L}$, return $\mathcal{L}[x]$; otherwise, pick an unused g_j where $j \in [N]$, set $\mathcal{L}[x] := g_j$ and return g_j to Adv.

Adv's view in the game $\mathsf{Unpredictability}_{\mathsf{TOP},\mathsf{Adv}}(1^\kappa, n, t)$ is information theoretically indistinguishable from the view simulated by $\mathcal{B}$ in the random oracle model. This can be proved via a hybrid argument:

Hyb_0: The first hybrid is Adv's view in the real-world game $\mathsf{Unpredictability}_{\mathsf{TOP},\mathsf{Adv}}(1^\kappa, n, t)$.

Hyb_1: This hybrid is the same as Hyb_0 except that in the response to $\mathcal{O}_{\mathsf{enc\&eval}}()$, c is randomly sampled as $c \leftarrow_\$ \mathbb{G}$. This hybrid is information theoretically indistinguishable from Hyb_0 because $\mathbb{G}$ is a cyclic group of prime order.

Hyb_2: This hybrid is the same as Hyb_1 except that the output of $\mathcal{H}_2(\cdot)$ is a truly random group element in $\mathbb{G}$. The indistinguishability of Hyb_1 and Hyb_2 follows from the random oracle model.

Hyb_3: This hybrid is Adv's view simulated by $\mathcal{B}$. It is the same as Hyb_2 except that the random group elements are replaced by g_j's where $j \in [N]$. Since g_j's are also randomly sampled from $\mathbb{G}$ in the TOMDH game, the two hybrids are indistinguishable.

From the construction of $\mathcal{B}$, we know that

$$\mathsf{MAX}_{t',t}(q'_1, \ldots, q'_n) = \mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) + q,$$

where $\mathsf{MAX}_{t',t}(q'_1, \ldots, q'_n)$ is from the TOMDH game, and $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n)$ is from the unpredictability game. Since $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) < k$, the output of $\mathcal{B}$ has the following number of valid pairs:

$$\begin{aligned} |\mathsf{LIST}| &= k + q \\ &> \mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) + q \\ &= \mathsf{MAX}_{t',t}(q'_1, \ldots, q'_n). \end{aligned}$$

Therefore, $\mathcal{B}$ breaks the gap TOMDH assumption.

Second Case For any $k \in \mathbb{N}$, when Adv calls $\mathcal{H}_1$ with k valid (x, y) pairs, $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) \geq k$.

We define a predicting game in Figure 11. Information theoretically we have that for any PPT adversary Adv, there exists a negligible function negl s.t.

$$\mathsf{Pr}[\mathsf{Predicting}_{\mathsf{Adv}}(1^\kappa) = 1] \leq \frac{k}{|\mathcal{X}|} + \mathsf{negl}(\kappa).$$

We will use Adv to construct an adversary $\mathcal{B}$ that breaks the predicting game. The construction of $\mathcal{B}$ is the following. It first runs $\mathsf{Setup}(1^\kappa, n, t)$ to generate $([\![\mathsf{sk}]\!], \mathsf{pp})$, presents pp to Adv and gets back $\mathcal{U}$. It thens gives $\{\mathsf{sk}_i\}_{i \in \mathcal{U}}$ to Adv. It sets $\mathcal{L} := []$, and then handles Adv's oracle queries as follows:

– On Adv's call to $\mathcal{O}_{\mathsf{enc\&eval}}()$: Sample $c \leftarrow_\$ \mathbb{G}$, compute $z_i \leftarrow \mathsf{Eval}(\mathsf{sk}_i, c)$ for $i \in [n]$, and return $(c, \{z_i\}_{i \in [n] \setminus \mathcal{U}})$ to Adv.

– On Adv's call to $\mathcal{O}_{\mathsf{eval}}(i, c)$: Return $\mathsf{Eval}(\mathsf{sk}_i, c)$.

$\underline{\mathsf{Predicting}_{\mathsf{Adv}}(1^\kappa)}$:

1. for every $x \in \mathcal{X}$: $h \leftarrow_\$ \{0,1\}^\kappa$, $\mathcal{M}[x] := h$
2. $\tilde{x} \leftarrow_\$ \mathcal{X}$, $\tilde{h} := \mathcal{M}[\tilde{x}]$
3. $k := 0$
4. $h^\star \leftarrow \mathsf{Adv}^{\langle \mathcal{O} \rangle}(1^\kappa)$
5. output 1 iff $h^\star = \tilde{h}$

$\underline{\mathcal{O}_{\mathsf{compute}}(x)}$:

- increment k by 1
- return $\mathcal{M}[x]$

$\underline{\mathcal{O}_{\mathsf{compare}}(h)}$:

- return 1 if $h = \tilde{h}$; else return 0

Figure 11: Predicting game

- On Adv's call to $\mathcal{O}_{\mathsf{check}}(h)$: Call $\mathcal{O}_{\mathsf{compare}}(h)$ and return the output to Adv.
- On Adv's call to $\mathcal{H}_1(x, y)$:

  a. If $(x, y) \in \mathcal{L}$, let $h := \mathcal{L}[(x, y)]$.
  b. If $(x, y) \notin \mathcal{L}$ and $y \neq \mathcal{H}_2(x)^{\mathsf{sk}}$, then sample $h \leftarrow_\$ \{0,1\}^\kappa$ and set $\mathcal{L}[(x, y)] := h$.
  c. If $(x, y) \notin \mathcal{L}$ and $y = \mathcal{H}_2(x)^{\mathsf{sk}}$ (i.e., (x, y) is a valid pair), call $\mathcal{O}_{\mathsf{compute}}(x)$ to get h. Set $\mathcal{L}[(x, y)] := h$.

  Return h to Adv.
- On Adv's call to $\mathcal{H}_2(x)$: Compute $\mathcal{H}_2(x)$ honestly and return the output.

Adv's view in the game $\mathsf{Unpredictability}_{\mathsf{TOP},\mathsf{Adv}}(1^\kappa, n, t)$ is information theoretically indistinguishable from the view simulated by $\mathcal{B}$ in the random oracle model. This can be proved via a hybrid argument:

Hyb_0: The first hybrid is Adv's view in the real-world game $\mathsf{Unpredictability}_{\mathsf{TOP},\mathsf{Adv}}(1^\kappa, n, t)$.

Hyb_1: This hybrid is the same as Hyb_0 except that in the response to $\mathcal{O}_{\mathsf{enc\&eval}}()$, c is randomly sampled as $c \leftarrow_\$ \mathbb{G}$. This hybrid is information theoretically indistinguishable from Hyb_0 because $\mathbb{G}$ is a cyclic group of prime order.

Hyb_2: This hybrid is the same as Hyb_1 except that the output of $\mathcal{H}_1(\cdot)$ is a truly random string. The indistinguishability of Hyb_1 and Hyb_2 follows from the random oracle model.

Hyb_3: This hybrid is Adv's view simulated by $\mathcal{B}$. It is the same as Hyb_2 except that $\tilde{x}$ is not sampled in the game, but sampled in the predicting game $\mathsf{Predicting}_{\mathcal{B}}(1^\kappa)$, and that $\mathcal{H}_1(x, y)$ for valid (x, y) pairs are sampled in predicting game. Since these values are randomly sampled in both hybrids, they are indistinguishable.

Therefore, if Adv breaks the game $\mathsf{Unpredictability}_{\mathsf{TOP},\mathsf{Adv}}(1^\kappa, n, t)$, then $\mathcal{B}$ breaks the

predicting game:

$$\begin{aligned}&\mathsf{Pr}[\mathsf{Predicting}_{\mathcal{B}}(1^\kappa) = 1]\\ &\qquad \geq \frac{\mathsf{MAX}_{|\mathcal{U}|,t}(q_1,\ldots,q_n)}{|\mathcal{X}|} + \mathsf{non\text{-}negl}(\kappa)\\ &\qquad \geq \frac{k}{|\mathcal{X}|} + \mathsf{non\text{-}negl}(\kappa).\end{aligned}$$

This is information theoretically impossible, leading to a contradiction, and hence concludes the proof.

A.2 Input Obliviousness

In this section we prove input obliviousness of our construction. Suppose there exists a PPT adversary Adv such that

$$\mathsf{Pr}[\mathsf{Obliviousness}_{\mathsf{TOP},\mathsf{Adv}}(1^\kappa, n, t) = 1] \geq \frac{\mathsf{MAX}_{|\mathcal{U}|,t}(q_1,\ldots,q_n)+1}{|\mathcal{X}|} + \mathsf{non\text{-}negl}(\kappa). \tag{A.2}$$

We will consider two cases of Adv. In the first case, there exists $k \in \mathbb{N}$ such that when Adv calls $\mathcal{H}_1$ with k distinct valid (x, y) pairs, $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1,\ldots,q_n) < k$. In this case, we will use Adv to break the gap TOMDH assumption. In the second case, for any $k \in \mathbb{N}$, when Adv calls $\mathcal{H}_1$ with k valid (x, y) pairs, $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1,\ldots,q_n) \geq k$. In this case, we will prove that information theoretically Formula A.2 does not hold.

First Case There exists $k \in \mathbb{N}$ such that when Adv calls $\mathcal{H}_1$ with k distinct valid (x, y) pairs, $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1,\ldots,q_n) < k$. Then we construct an adversary $\mathcal{B}$ that breaks the gap TOMDH assumption (see Definition 3.1). The proof is the same as the first case in Section A.1.

Second Case Whenever Adv calls $\mathcal{H}_1$ with a valid (x, y) pair for the k-th time, $\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) \geq k$ at that time.

We define a guessing game in Figure 12. Information theoretically we have that for any PPT adversary Adv, there exists a negligible function negl s.t.

$$\mathsf{Pr}[\mathsf{Guessing}_{\mathsf{Adv}}(1^\kappa) = 1] \leq \frac{k+1}{|\mathcal{X}|} + \mathsf{negl}(\kappa).$$

We will use Adv to construct an adversary $\mathcal{B}$ that breaks the guessing game. The construction of $\mathcal{B}$ is the following. It first runs $\mathsf{Setup}(1^\kappa, n, t)$ to generate $(\llbracket \mathsf{sk} \rrbracket, \mathsf{pp})$, presents pp to Adv and gets back $\mathcal{U}$. It thens gives $\{\mathsf{sk}_i\}_{i\in\mathcal{U}}$ to Adv. It samples $\tilde{h} \leftarrow_\$ \{0,1\}^\kappa$, set $\mathcal{L} := []$, and then handles Adv's oracle queries as follows:

- On Adv's call to $\mathcal{O}_{\mathsf{enc\&eval}}()$: Sample $c \leftarrow_\$ \mathbb{G}$, compute $z_i \leftarrow \mathsf{Eval}(\mathsf{sk}_i, c)$ for $i \in [n]$, and return $(c, \{z_i\}_{i\in[n]\setminus\mathcal{U}}, \tilde{h})$ to Adv.
- On Adv's call to $\mathcal{O}_{\mathsf{eval}}(i, c)$: Return $\mathsf{Eval}(\mathsf{sk}_i, c)$.
- On Adv's call to $\mathcal{H}_1(x, y)$:
  a. If $(x, y) \in \mathcal{L}$, let $h := \mathcal{L}[(x, y)]$.
  b. If $(x, y) \notin \mathcal{L}$ and $y \neq \mathcal{H}_2(x)^{\mathsf{sk}}$, then sample $h \leftarrow_\$ \{0,1\}^\kappa$ and set $\mathcal{L}[(x, y)] := h$.

$\underline{\mathsf{Guessing}_{\mathsf{Adv}}(1^\kappa)}$:

1. $\tilde{x} \leftarrow_\$ \mathcal{X}$
2. $k := 0$
3. $x^\star \leftarrow \mathsf{Adv}^{\langle \mathcal{O} \rangle}(1^\kappa)$
4. output 1 iff $x^\star = \tilde{x}$

$\underline{\mathcal{O}_{\mathsf{guess}}(x)}$:

- increment k by 1
- return 1 if $x = \tilde{x}$; else return 0

Figure 12: Guessing game

c. If $(x, y) \notin \mathcal{L}$ and $y = \mathcal{H}_2(x)^{\mathsf{sk}}$ (i.e., (x, y) is a valid pair), call $\mathcal{O}_{\mathsf{guess}}(x)$. If the output if 1, then $h := \tilde{h}$; otherwise sample $h \leftarrow_\$ \{0,1\}^\kappa$. Set $\mathcal{L}[(x, y)] := h$.

Return h to Adv.

– On Adv's call to $\mathcal{H}_2(x)$: Compute $\mathcal{H}_2(x)$ honestly and return the output.

Adv's view in the game $\mathsf{Obliviousness}_{\mathsf{TOP},\mathsf{Adv}}(1^\kappa, n, t)$ is information theoretically indistinguishable from the view simulated by $\mathcal{B}$ in the random oracle model. This can be proved via a hybrid argument:

Hyb_0: The first hybrid is Adv's view in the real-world game $\mathsf{Obliviousness}_{\mathsf{TOP},\mathsf{Adv}}(1^\kappa, n, t)$.

Hyb_1: This hybrid is the same as Hyb_0 except that in the response to $\mathcal{O}_{\mathsf{enc\&eval}}()$, c is randomly sampled as $c \leftarrow_\$ \mathbb{G}$. This hybrid is information theoretically indistinguishable from Hyb_0 because $\mathbb{G}$ is a cyclic group of prime order.

Hyb_2: This hybrid is the same as Hyb_1 except that the output of $\mathcal{H}_1(\cdot)$ is a truly random string. The indistinguishability of Hyb_1 and Hyb_2 follows from the random oracle model.

Hyb_3: This hybrid is Adv's view simulated by $\mathcal{B}$. It is the same as Hyb_2 except that $\tilde{x}$ is not sampled in the game, but sampled in the guessing game $\mathsf{Guessing}_{\mathcal{B}}(1^\kappa)$. Since $\tilde{x}$ is randomly sampled as $\tilde{x} \leftarrow_\$ \mathcal{X}$ in both hybrids, they are indistinguishable.

Therefore, if Adv breaks the game $\mathsf{Obliviousness}_{\mathsf{TOP},\mathsf{Adv}}(1^\kappa, n, t)$, then $\mathcal{B}$ breaks the guessing game:

$$\begin{aligned}
&\mathsf{Pr}[\mathsf{Guessing}_{\mathcal{B}}(1^\kappa) = 1] \\
&\qquad \geq \frac{\mathsf{MAX}_{|\mathcal{U}|,t}(q_1, \ldots, q_n) + 1}{|\mathcal{X}|} + \mathsf{non\text{-}negl}(\kappa) \\
&\qquad \geq \frac{k+1}{|\mathcal{X}|} + \mathsf{non\text{-}negl}(\kappa).
\end{aligned}$$

This is information theoretically impossible, leading to a contradiction, and hence concludes the proof.

B PASTA: proofs

B.1 Password safety

PASTA's password safety primarily relies on the obliviousness of TOP. Intuitively, if we use a TOPRF on clients' passwords, then obliviousness of TOPRF would make it hard to guess

them. Formally, we build an adversary $\mathcal{B}$ that can translate an adversary Adv's advantage in the password-safety game into a similar advantage in the TOPRF obliviousness game $\mathsf{Obliviousness}$ (Figure 6). $\mathcal{B}$ will run Adv *internally* simulating the password-safety for it, while playing the role of adversary *externally* in $\mathsf{Obliviousness}$.

$\mathcal{B}$ can implicitly set the targeted client $C^\star$'s password to be the random value chosen in $\mathsf{Obliviousness}$. If Adv guesses the password, $\mathcal{B}$ can output the same guess. However, to simulate $\mathsf{SecGame}$ properly for Adv, $\mathcal{B}$ needs to run the oracles in a way that Adv cannot tell the difference. In particular, $\mathcal{B}$ needs partial TOPRF evaluations z_i on the password for $\mathcal{O}_{\mathsf{resp}}$, the final TOPRF value for $\mathcal{O}_{\mathsf{register}}$ and the randomness ρ used for encoding for $\mathcal{O}_{\mathsf{final}}$. $\mathcal{B}$ can take help of the oracles $\mathcal{O}_{\mathsf{eval}}$ and $\mathcal{O}_{\mathsf{enc\&eval}}$ provided by $\mathsf{Obliviousness}$ to handle the first two problems, but there is no way to get ρ in $\mathsf{Obliviousness}$.

Intermediate hybrid. We tackle the latter problem first by going through a hybrid. We refer to the original game as Hyb_0 and the new game as Hyb_1. Hyb_0 is described in Figure 13; it basically replaces Π in Figure 7 with PASTA. Hyb_1 is described in Figure 14. In Hyb_1, several oracles behave differently for the targeted client $C^\star$. $\mathcal{O}_{\mathsf{req}}$ evaluates the TOPRF in advance for $C^\star$. It stores the partial evaluations z_i and the final result h in $\mathsf{LiveSessions}$ itself. Importantly, it does *not* store ρ. When $\mathcal{O}_{\mathsf{resp}}$ is invoked, it checks if $C^\star$ generated req_i for S_i before ($\mathsf{req}_i \in \mathsf{ReqList}_{C,i}$). If yes, then z_i is picked up from $\mathsf{LiveSessions}$. Now, whether a z_i computed in advance is used in $\mathcal{O}_{\mathsf{resp}}$ or not makes no difference from the point of the adversary because z_i is derived deterministically from k_i and req_i.

Oracle $\mathcal{O}_{\mathsf{final}}$ also behaves differently for $C^\star$. First, note that if $\mathsf{TOP.PubCombine}(\{z_i\}_{i\in\mathcal{T}})$ is equal to $\mathsf{TOP.PubCombine}(\{z'_i\}_{i\in\mathcal{T}})$, then combining either set will lead to the same value. The only difference in Hyb_1 is that h was computed beforehand. Once again, for the same reason as above, this makes no difference.

The crucial step where Hyb_0 and Hyb_1 differ is when the two outputs of $\mathsf{PubCombine}$ do not match. While Hyb_0 does not do any test of this kind, Hyb_1 simply outputs $\perp$. For these hybrids to be indistinguishable, we need to argue that *had* the outputs of $\mathsf{PubCombine}$ not matched in Hyb_0, it would have output $\perp$ as well (at least with a high probability).

Note that the *right* z_i and h are well-defined for Hyb_0; they can be derived from pwd and ρ. If one were to do the public combine test in this hybrid and it fails, then $h' \neq h$ with high probability. Therefore, using the collision resistance of $\mathcal{H}$, one can argue that $h'_i \neq h_i$. Now, observe that there must be an honest S_j in $\mathcal{T}$, so ctxt'_j could only have been generated by S_j (recall our authenticated channels assumption). When ciphertext ctxt'_j, which was encrypted under h_j, is decrypted with $h'_j \neq h_j$, decryption fails with high probability due to the key-binding property of SKE. Thus, Hyb_0 returns $\perp$ just like Hyb_1.

Reduction. Now that we know that absence of encoding randomness ρ would not prevent a successful simulation of $\mathcal{O}_{\mathsf{final}}$, we come back to the task of exploiting TOPRF obliviousness to hide the targeted client's password. Towards this, adversary $\mathcal{B}$ is formally described in Figure 15. When $\mathcal{B}$ outputs a message, it should be interpreted as sending the message to the obliviousness game. Let's now go through the differences between Hyb_1 and $\mathcal{B}$'s simulation of it one by one.

Simulation of $\mathcal{O}_{\mathsf{register}}$ differs only for $C = C^\star$. In Hyb_1, a randomly chosen password for $C^\star$ is used to compute h, while in $\mathcal{B}$'s simulation, $C^\star$'s password is implicitly set to be the random input $\tilde{x}$ chosen by $\mathsf{Obliviousness}$ and $\mathcal{O}_{\mathsf{enc\&eval}}$ is called to get h. Clearly, this difference does not affect Adv. There is one other difference though: while all of $k_1, \ldots, k_n$

are known in Hyb_1, $\mathcal{B}$ knows k_i for corrupt servers only. As a result, $\mathcal{B}$ defines $\mathsf{rec}_{i,C^\star}$ to be $(\mathbf{0}, h_i)$ for $i \in [n] \setminus \mathcal{U}$.

Like the registration oracle, request oracle behaves differently only when $C = C^\star$. However, one can easily see that the difference is insignificant: while Hyb_1 computes c, z_i and h using $\mathsf{PwdList}[C^\star]$, $\mathcal{B}$ invokes $\mathcal{O}_{\mathsf{enc\&eval}}$ to get them, which uses $\tilde{x}$.

Finally, $\mathcal{B}$ invokes $\mathcal{O}_{\mathsf{eval}}$ to get z_i in the simulation of $\mathcal{O}_{\mathsf{resp}}$ (because it does not know k_i for honest servers) but it is computed directly in Hyb_1. This does not make any difference either. The important thing to note is that the counter $Q_{C^\star,i}$ is incremented if and only if the counter q_i of $\mathsf{Obliviousness}$ is incremented. As a result, the final value of $Q_{C^\star,i}$ will be the same as q_i. Therefore, $\mathcal{B}$ will successfully translate Adv's probability of guessing $C^\star$'s password to guessing $\tilde{x}$.

B.2 Unforgeability

First we handle the easier case of $Q_{C^\star,x^\star} < t - |\mathcal{U}|$. Here $C^\star$ could even be corrupt, so Adv may know its password. Note that $Q_{C^\star,x^\star}$ is incremented on every invocation of $\mathcal{O}_{\mathsf{resp}}(i, C^\star, x^\star, \mathsf{req}_i)$ irrespective of the value of i and whether or not $\mathsf{req}_i \in \mathsf{ReqList}_{i,C^\star}$. So if $Q_{C^\star,x^\star} < t - |\mathcal{U}|$, Adv simply doesn't have enough shares to generate a valid token, irrespective of whether $C^\star$ is corrupt or not. One can formally prove unforgeability in this case by invoking the unforgeability of the threshold token generation scheme TTG (Definition 3.3). We skip the details.

When $Q_{C^\star,x^\star} \geq t - |\mathcal{U}|$, unforgeability can only be expected when $C^\star$ is never corrupted. We need to show that generating a valid token for $(C^\star, x^\star)$ for any $x^\star$ effectively amounts to guessing $C^\star$'s password. Indistinguishability of Hyb_0 (Figure 13) and Hyb_1 (Figure 14) still holds because it just relies on the properties of $\mathsf{PubCombine}$ and authenticated channels.

We now wish to build an adversary $\mathcal{B}'$ that can use an adversary Adv who breaks the unforgeability guarantee of PASTA to break the unpredictability of TOPRF. The first natural question to ask is whether $\mathcal{B}'$ can break unpredictability of TOPRF in the same way as $\mathcal{B}$ broke obliviousness. Not quite, because there are some key differences in the two settings:

- Even though both $\mathcal{B}$ and $\mathcal{B}'$ get access to an oracle $\mathcal{O}_{\mathsf{enc\&eval}}$ that both encodes and evaluates, $\mathcal{B}$'s oracle returns the final TOPRF output h while $\mathcal{B}'$'s oracle doesn't. So it is not clear how h_i will be generated by $\mathcal{O}_{\mathsf{register}}$ and $\mathcal{O}_{\mathsf{final}}$ for $C^\star$.
- $\mathcal{B}$ was able to use the output of Adv for the password-safety game directly into the obliviousness game, but $\mathcal{B}'$ cannot. Adv now outputs a token for the authentication scheme while $\mathcal{B}'$ is supposed to guess the TOPRF output h on the (hidden) password of $C^\star$.

As a result, $\mathcal{B}'$'s behavior differs from $\mathcal{B}$ in the following manner. At the start of the simulation, $\mathcal{B}'$ picks random numbers $r_1, \ldots, r_n$ and will use them instead of $h_1, \ldots, h_n$ in the registration oracle. $\mathsf{LiveSessions}$ cannot contain h anymore, so when it is needed in the finalize oracle, $r_1, \ldots, r_n$ will be used once again. If Adv queries $\mathcal{H}$ on $h' \| i$ at any time, $\mathcal{B}'$ will query $\mathcal{O}_{\mathsf{check}}$ on h' to check if $h' = h$ or not. If $\mathcal{O}_{\mathsf{check}}$ returns 1, then $\mathcal{B}'$ sends r_i to Adv.

This also gives a way for $\mathcal{B}'$ to guess h. If Adv queries $\mathcal{H}$ for some $h' \| i$ and $\mathcal{O}_{\mathsf{check}}$ returns 1 on h', then $\mathcal{B}'$ just outputs h' in the unpredictability game. However, we don't have the guarantee that Adv will make such a query. All we know is that Adv can produce a valid token. Hence, we must argue that Adv can produce a valid token only if it queries $\mathcal{H}$ on h.

Any token share sent by the i-th server is encrypted with h_i. At a high level, Adv needs to decrypt at least one token share from an honest server, say j-th, to construct a token. The

$\underline{\mathsf{SecGame}_{\mathrm{PASTA},\mathsf{Adv}}(1^\kappa, n, t, \mathsf{P})}$:

- $(\llbracket \mathsf{tsk} \rrbracket, \mathsf{tvk}, \mathsf{tpp}) \leftarrow \mathsf{TTG.Setup}(1^\kappa, n, t)$
- set $\mathsf{sk}_i := \mathsf{tsk}_i$ for all $i \in [n]$, $\mathsf{vk} := \mathsf{tvk}$ and $\mathsf{pp} := (\kappa,\ n,\ t,\ \mathsf{P},\ \mathsf{tpp})$.
- $(\mathcal{U}, C^\star, \mathsf{st}_{\mathsf{adv}}) \leftarrow \mathsf{Adv}(\mathsf{pp})$
- $\mathcal{V}, \mathsf{PwdList}, \mathsf{TokList} := \emptyset$, $\mathsf{ReqList}_{C,i} := \emptyset$ for $i \in [n]$
- $\mathsf{ct} := 0$, $\mathsf{LiveSessions} = []$
- $Q_{C,i}, Q_{C,x} := 0$ for all C, $i \in [n]$ and x
- $\mathsf{out} \leftarrow \mathsf{Adv}^{\langle \mathcal{O} \rangle}(\{\mathsf{sk}_i\}_{i\in\mathcal{U}}, \{\mathsf{SK}_i\}_{i\in\mathcal{U}}, \mathsf{st}_{\mathsf{adv}})$

$\underline{\mathcal{O}_{\mathsf{corrupt}}(C)}$.

- $\mathcal{V} := \mathcal{V} \cup \{C\}$
- if $(C, \star) \in \mathsf{PwdList}$, return $\mathsf{PwdList}[C]$

$\underline{\mathcal{O}_{\mathsf{register}}(C)}$.

- require: $\mathsf{PwdList}[C] = \bot$
- $\mathsf{pwd} \leftarrow_{\$} \mathsf{P}$
- add (C, pwd) to $\mathsf{PwdList}$
- $(\llbracket k \rrbracket, \mathsf{opp}) \leftarrow \mathsf{TOP.Setup}(1^\kappa, n, t)$
- $h := \mathsf{TOP}(k, \mathsf{pwd})$ and $h_i := \mathcal{H}(h \| i)$ for $i \in [n]$
- $\mathsf{rec}_{i,C} := (k_i, h_i)$ for all $i \in [n]$
- add $\mathsf{rec}_{i,C}$ to REC_i for all $i \in [n]$

$\underline{\mathcal{O}_{\mathsf{req}}(C, x, \mathcal{T})}$.

- if $\mathsf{PwdList}[C] = \bot$ or $|\mathcal{T}| < t$, output $\bot$
- $c := \mathsf{TOP.Encode}(\mathsf{PwdList}[C], \rho)$ for a random ρ
- set $\mathsf{req}_i := c$ for $i \in [n]$
- $\mathsf{LiveSessions}[\mathsf{ct}] := (C, \mathsf{PwdList}[C], \rho, \mathcal{T})$
- add req_i to $\mathsf{ReqList}_{C,i}$ for $i \in \mathcal{T}$
- increment ct by 1
- return $\{\mathsf{req}_i\}_{i\in\mathcal{T}}$

$\underline{\mathcal{O}_{\mathsf{resp}}(i, C, x, \mathsf{req}_i)}$.

- require: $i \in [n] \setminus \mathcal{U}$
- if $\mathsf{rec}_{i,C} \notin \mathrm{REC}_i$, return $\bot$; else, parse $\mathsf{rec}_{i,C}$ as (k_i, h_i)
- if $\mathsf{req}_i \notin \mathsf{ReqList}_{C,i}$, increment $Q_{C,i}$ by 1
- $z_i := \mathsf{TOP.Eval}(k_i, \mathsf{req}_i)$
- $y_i \leftarrow \mathsf{TTG.PartEval}(\mathsf{tsk}_i, C \| x)$
- set $\mathsf{res}_i := (z_i, \mathsf{SKE.Encrypt}(h_i, y_i))$
- increment $Q_{C,x}$ by 1
- return res_i

$\underline{\mathcal{O}_{\mathsf{final}}(\mathsf{ct}, \{\mathsf{res}_i\}_{i\in S})}$.

- $\mathsf{st} := \mathsf{LiveSessions}[\mathsf{ct}]$
- parse res_i as (z'_i, ctxt'_i) and st as $(C, \mathsf{pwd}, \rho, \mathcal{T})$.
- if $S \neq \mathcal{T}$, output $\bot$
- $h' := \mathsf{TOP.Combine}(\mathsf{pwd}, \{(i, z'_i)\}_{i\in\mathcal{T}}, \rho)$
- for $i \in \mathcal{T}$, $h'_i := \mathcal{H}(h' \| i)$ and $y'_i := \mathsf{SKE.Decrypt}(h'_i, \mathsf{ctxt}'_i)$
- set $\mathsf{tk} := \mathsf{TTG.Combine}(\{i, y'_i\}_{i\in\mathcal{T}})$
- add tk to $\mathsf{TokList}$
- return tk

$\underline{\mathcal{O}_{\mathsf{verify}}(C, x, \mathsf{tk})}$.

- return $\mathsf{TTG.Verify}(\mathsf{tvk}, C \| x, \mathsf{tk})$

Figure 13: Hyb_0: SecGame for PASTA

$\mathsf{SecGame}_{\mathrm{PASTA},\mathsf{Adv}}(1^\kappa, n, t, \mathsf{P})$:

Same as Hyb_0*, except the following oracles behave differently when* $C = C^\star$*. Below, we describe their behavior for this case only, highlighting the differences in red. When* $C \neq C^\star$*, they behave in the same way as* Hyb_0*.*

$\mathcal{O}_{\mathsf{req}}(C^\star, x, \mathcal{T})$.

- if $\mathsf{PwdList}[C^\star] = \perp$ or $|\mathcal{T}| < t$, output $\perp$
- $c := \mathsf{TOP.Encode}(\mathsf{PwdList}[C^\star], \rho)$ for a random ρ
- set $\mathsf{req}_i := c$ for $i \in [n]$
- $z_i := \mathsf{TOP.Eval}(k_i, \mathsf{req}_i)$
- $h := \mathsf{TOP.Combine}(\mathsf{PwdList}[C^\star], \{(i, z_i)\}_{i\in\mathcal{T}}, \rho)$
- $\mathsf{LiveSessions}[\mathsf{ct}] := (C^\star, c, \{(i, z_i)\}_{i\in\mathcal{T}}, h)$
- add req_i to $\mathsf{ReqList}_{C^\star,i}$ for $i \in \mathcal{T}$
- increment ct by 1
- return $\{\mathsf{req}_i\}_{i\in\mathcal{T}}$

$\mathcal{O}_{\mathsf{resp}}(i, C^\star, x, \mathsf{req}_i)$.

- require: $i \in [n] \setminus \mathcal{U}$
- if $\mathsf{rec}_{i,C^\star} \notin \mathrm{REC}_i$, return $\perp$; else, parse $\mathsf{rec}_{i,C^\star}$ as (k_i, h_i)
- if $\mathsf{req}_i \notin \mathsf{ReqList}_{C^\star,i}$, increment $Q_{C^\star,i}$ by 1
- if $(\mathsf{req}_i \notin \mathsf{ReqList}_{C^\star,i})$:
  - $z_i := \mathsf{TOP.Eval}(k_i, \mathsf{req}_i)$
- else:
  - let z_i be the value associated with i in the entry $(C^\star, \mathsf{req}_i, \ldots, (i, z_i) \ldots)$ in $\mathsf{LiveSessions}$
- $y_i \leftarrow \mathsf{TTG.PartEval}(\mathsf{tsk}_i, C^\star \| x)$
- set $\mathsf{res}_i := (z_i, \mathsf{SKE.Encrypt}(h_i, y_i))$
- increment $Q_{C^\star,x}$ by 1
- return res_i

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

- $(C^\star, c, \{(i, z_i)\}_{i\in\mathcal{T}}, h) := \mathsf{LiveSessions}[\mathsf{ct}]$
- parse res_i as (z'_i, ctxt'_i)
- if $S \neq \mathcal{T}$, output $\perp$
- if $(\mathsf{TOP.PubCombine}(\{z_i\}_{i\in\mathcal{T}}) \neq \mathsf{TOP.PubCombine}(\{z'_i\}_{i\in\mathcal{T}}))$:
  - return $\perp$
- for $i \in \mathcal{T}$, $h'_i := \mathcal{H}(h \| i)$ and $y'_i := \mathsf{SKE.Decrypt}(h'_i, \mathsf{ctxt}'_i)$
- set $\mathsf{tk} := \mathsf{TTG.Combine}(\{i, y'_i\}_{i\in\mathcal{T}})$
- add tk to $\mathsf{TokList}$
- return tk

Figure 14: Hybrid Hyb_1

$\mathcal{B}_{\mathsf{Adv}}(1^\kappa, n, t, \mathsf{P})$:
Same as Hyb_1*, except the following oracles are simulated differently when* $C = C^\star$*. Below, these oracles are described for this case only, with the differences highlighted in red. Output whatever* Adv *does.*

$\mathcal{O}_{\mathsf{register}}(C^\star)$.

- require: $\mathsf{PwdList}[C^\star] = \bot$
- add $(C^\star, \mathsf{unknown})$ to PwdList
- output $\mathcal{U}$, get back $\{k_i\}_{i\in\mathcal{U}}$
- query $\mathcal{O}_{\mathsf{enc\&eval}}$ to get $(c, \{z_i\}_{i\in[n]}, h)$
- $h_i := \mathcal{H}(h\|i)$ for $i \in [n]$
- for $i \in [n] \setminus \mathcal{U}$, $\mathsf{rec}_{i,C^\star} := (\mathbf{0}, h_i)$
- for $i \in \mathcal{U}$, $\mathsf{rec}_{i,C^\star} := (k_i, h_i)$
- add $\mathsf{rec}_{i,C^\star}$ to REC_i for all $i \in [n]$

$\mathcal{O}_{\mathsf{req}}(C^\star, x, \mathcal{T})$.

- if $\mathsf{PwdList}[C^\star] = \bot$ or $|\mathcal{T}| < t$, output $\bot$
- query $\mathcal{O}_{\mathsf{enc\&eval}}$ to get $(c, \{z_i\}_{i\in[n]}, h)$
- set $\mathsf{req}_i := c$ for $i \in [n]$
- $\mathsf{LiveSessions}[\mathsf{ct}] := (C, c, \{i, z_i\}_{i\in\mathcal{T}}, h)$
- add req_i to $\mathsf{ReqList}_{C^\star,i}$ for $i \in \mathcal{T}$
- increment ct by 1
- return $\{\mathsf{req}_i\}_{i\in\mathcal{T}}$

$\mathcal{O}_{\mathsf{resp}}(i, C^\star, x, \mathsf{req}_i)$.

- require: $i \in [n] \setminus \mathcal{U}$
- if $\mathsf{rec}_{i,C^\star} \notin \mathrm{REC}_i$, return $\bot$; else, parse $\mathsf{rec}_{i,C^\star}$ as (k_i, h_i)
- if $\mathsf{req}_i \notin \mathsf{ReqList}_{C^\star,i}$, increment $Q_{C^\star,i}$ by 1
- if $(\mathsf{req}_i \notin \mathsf{ReqList}_{C^\star,i})$:
  - query $\mathcal{O}_{\mathsf{eval}}(i, \mathsf{req}_i)$ to get z_i
- else:
  - let z_i be the value associated with i in the entry $(C^\star, \mathsf{req}_i, \ldots, (i, z_i)\ldots)$ in LiveSessions
- $y_i \leftarrow \mathsf{TTG.PartEval}(\mathsf{tsk}_i, C^\star\|x)$
- set $\mathsf{res}_i := (z_i, \mathsf{SKE.Encrypt}(h_i, y_i))$
- increment $Q_{C^\star,x}$ by 1
- return res_i

Figure 15: Adversary $\mathcal{B}$

only way to get this key is by querying $\mathcal{H}$ on $h\|j$.

We defer a formal proof to the full version.

C Threshold Authentication Schemes

In this section, we describe the threshold authentication schemes we implemented. We implemented the following schemes:

- The DDH-based DPRF scheme of Naor, Pinkas and Reingold [NPR99] as a public-key threshold MAC (Figure 16).
- The PRF-only DPRF scheme of Naor, Pinkas and Reingold [NPR99] as a symmetric-key MAC (Figure 17).
- The threshold RSA-signature scheme of Shoup [Sho00] as a threshold signature scheme based on RSA assumption (Figure 18).
- The pairing-based signature scheme of Boldyreva [Bol03] as a threshold signature scheme based on the gap-DDH assumption (Figure 19).

Ingredients: Let $\mathbb{G} = \langle g \rangle$ be a multiplicative cyclic group of prime order p in which the DDH assumption holds and $\mathcal{H} : \{0,1\}^* \to \mathbb{G}$ be a hash function modeled as a random oracle. Let $\mathsf{GenShare}$ be Shamir's secret sharing scheme.

- $\mathsf{Setup}(1^\kappa, n, t) \to (\llbracket \mathsf{sk} \rrbracket, \mathsf{vk}, \mathsf{pp})$. Sample $s \leftarrow_\$ \mathbb{Z}_p$ and get $(s, s_1, \ldots, s_n) \leftarrow \mathsf{GenShare}(n, t, p, (0, s))$. Set $\mathsf{pp} := (p, g, \mathbb{G})$, $\mathsf{sk}_i := s_i$ and $\mathsf{vk} := s$. Give $(\mathsf{sk}_i, \mathsf{pp})$ to party i. (pp will be an implicit input in the algorithms below.).
- $\mathsf{PartEval}(\mathsf{sk}_i, x) \to y_i$. Compute $w := \mathcal{H}(x)$, $h_i := w^{\mathsf{sk}_i}$ and output h_i.
- $\mathsf{Combine}(\{i, y_i\}_{i \in S}) =: \mathsf{tk}/\perp$. If $|S| < t$ output $\perp$. Otherwise parse y_i as h_i for $i \in S$ and output $\prod_{i \in S} h_i^{\lambda_{i,S}}$
- $\mathsf{Verify}(\mathsf{vk}, x, \mathsf{tk}) =: 1/0$. Return 1 if and only if $\mathcal{H}(x)^{\mathsf{vk}} = \mathsf{tk}$.

Figure 16: The DDH-based DPRF construction of Naor et al. [NPR99] (public-key threshold MAC).

D Necessity of Public-Key Operations

In both the registration phase and sign-on phase of PASTA, we instantiate the TOPRF component with the 2HashTDH protocol of Jarecki et al. [JKKX17] which uses public-key operations. Therefore, all the instantiations of PASTA use public-key operations even if the threshold token generation scheme is purely symmetric-key. This could become a significant overhead in some cases compared to the naïve insecure solutions (see Section 7.4 for details). So the natural question is whether public-key operations can be avoided, or, put differently, is it just an artifact of PASTA and its instantiations? In this section we prove that public-key operations are indeed necessary to construct any secure PbTA scheme.

In more detail, we prove that if one can construct PbTA that only makes black-box use of one-way functions, then a secure two-party key agreement protocol can also be constructed by only making black-box use of one-way functions, which would imply $P \neq NP$ [IR90].

Ingredients: Let $f : \{0,1\}^\kappa \times \{0,1\}^* \to \{0,1\}^*$ be a pseudo-random function.

- $\mathsf{Setup}(1^\kappa, n, t) \to (\llbracket \mathsf{sk} \rrbracket, \mathsf{vk}, \mathsf{pp})$. Pick $d := \binom{n}{n-t+1}$ keys $k_1, \ldots, k_d \leftarrow_\$ \{0,1\}^\kappa$ for f. Let $D_1, \ldots, D_d$ be the d distinct $(n-t+1)$-sized subsets of $[n]$. For $i \in [n]$, let $\mathsf{sk}_i := \{k_j \mid i \in D_j \text{ for all } j \in [d]\}$ and $\mathsf{vk} := (k_1, \ldots, k_d)$. Set $\mathsf{pp} := f$ and give $(\mathsf{sk}_i, \mathsf{pp})$ to party i.
- $\mathsf{PartEval}(\mathsf{sk}_i, x) \to y_i$. Compute $h_{i,k} := f_k(x)$ for all $k \in \mathsf{sk}_i$ and output $\{h_{i,k}\}_{k \in \mathsf{sk}_i}$.
- $\mathsf{Combine}(\{i, y_i\}_{i \in S}) =: \mathsf{tk}/\perp$. If $|S| < t$ output $\perp$. Otherwise parse y_i as $\{h_{i,k}\}_{k \in \mathsf{sk}_i}$ for $i \in S$. Let $\{\mathsf{sk}'_i\}_{i \in S}$ be mutually disjoint sets such that $\cup_{i \in S} \mathsf{sk}'_i = \{k_1, \ldots, k_d\}$ and $\mathsf{sk}'_i \subseteq \mathsf{sk}_i$ for every i. Output $\oplus_{k \in \mathsf{sk}'_i, i \in S}(h_{i,k})$.
- $\mathsf{Verify}(\mathsf{vk}, x, \mathsf{tk}) =: 1/0$. Return 1 if and only if $\oplus_{i \in [d]}(f_{k_i}(x)) = \mathsf{tk}$ where $\mathsf{vk} = \{k_1, \ldots, k_d\}$.

Figure 17: The PRF-based DPRF of Naor et al. [NPR99] (symmetric-key threshold MAC).

Ingredients: Let $\mathsf{GenShare}$ be a Shamir's secret sharing scheme and $\mathcal{H} : \{0,1\}^* \to \mathbb{Z}_N^*$ be a hash function modeled as a random oracle.

- $\mathsf{Setup}(1^\kappa, n, t) \to (\llbracket \mathsf{sk} \rrbracket, \mathsf{vk}, \mathsf{pp})$. Let p', q' be two randomly chosen large primes of equal length and set $p := 2p' + 1$ and $q = 2q' + 1$. Set $N := pq$. Choose another large prime e at random and compute $d \equiv e^{-1} \bmod \Phi(N)$ where $\Phi(\cdot) : \mathbb{N} \to \mathbb{N}$ is the Euler's totient function. Then $(d, d_1, \ldots, d_n) \leftarrow \mathsf{GenShare}(n, t, \Phi(N), (0, d))$. Let $\mathsf{sk}_i := d_i$ and $\mathsf{vk} := (N, e)$. Set $\mathsf{pp} := \Delta$ where $\Delta := n!$. Give $(\mathsf{pp}, \mathsf{vk}, \mathsf{sk}_i)$ to party i.
- $\mathsf{PartEval}(\mathsf{sk}_i, x) \to y_i$. Output $y_i := \mathcal{H}(x)^{2\Delta d_i}$.
- $\mathsf{Combine}(\{i, y_i\}_{i \in S}) =: \mathsf{tk}/\perp$. If $|S| < t$ output $\perp$, otherwise compute $z := \prod_{i \in S} y_i^{2\lambda'_{i,S}} \bmod N$ where $\lambda'_{i,S} := \lambda_{i,S}\Delta \in \mathbb{Z}$. Find integer (a, b) by Extended Euclidean GCD algorithm such that $4\Delta^2 a + eb = 1$. Then compute $\mathsf{tk} := z^a \cdot \mathcal{H}(x)^b \bmod N$. Output tk.
- $\mathsf{Verify}(\mathsf{vk}, x, \mathsf{tk}) = 1/0$. Return 1 if and only if $\mathsf{tk}^e = \mathcal{H}(x) \bmod N$.

Figure 18: The threshold RSA-signature scheme of Shoup [Sho00].

Hence this gives us evidence that it is unlikely to construct PbTA using only symmetric-key operations.

Overview We now give an overview of our proof technique. At a high level, we construct a secure key-agreement protocol from PbTA in a black-box way. As a result, if one can construct PbTA that only makes black-box use of one-way functions, then our construction is a secure key-agreement protocol that only makes black-box use of one-way functions, contradicting the impossibility result of Impagliazzo and Rudich [IR90].

To construct the secure key-agreement protocol, think of the two parties P_1 and P_2 in the key-agreement protocol as a client C and the set of all servers in the PbTA protocol, respectively. We set the password space to contain only one password pwd, which means the password of C is known to both parties. Thus P_2, which represents the set of all servers, can run $\mathsf{GlobalSetup}$ and the registration phase of C on its own. Then the two parties run the sign-on phase so that P_1 obtains a token for $x = 0$. Since both parties know the password,

Ingredients: Let $\mathbb{G} = \langle g \rangle$ be a multiplicative cyclic group of prime order p that supports pairing and in which CDH is hard. In particular, there is an efficient algorithm $\mathsf{VerDDH}(g^a, g^b, g^c, g)$ that returns 1 if and only if $c = ab \bmod p$ for any $a, b, c \in \mathbb{Z}_p^*$ and 0 otherwise. Let $\mathcal{H} : \{0,1\}^* \to \mathbb{G}$ be a hash function modeled as a random oracle. Let $\mathsf{GenShare}$ be the Shamir's secret sharing scheme.

- $\mathsf{Setup}(1^\kappa, n, t) \to ([\![\mathsf{sk}]\!], \mathsf{vk}, \mathsf{pp})$. Sample $s \leftarrow_\$ \mathbb{Z}_p^*$ and get $(s, s_1, \ldots, s_n) \leftarrow \mathsf{GenShare}(n, t, p, (0, s))$. Set $\mathsf{pp} := (p, g, \mathbb{G})$, $\mathsf{sk}_i := s_i$ and $\mathsf{vk} := g^s$. Give $(\mathsf{sk}_i, \mathsf{pp})$ to party i.
- $\mathsf{PartEval}(\mathsf{sk}_i, x) \to y_i$. Compute $w := \mathcal{H}(x)$, $h_i := w^{\mathsf{sk}_i}$ and output h_i.
- $\mathsf{Combine}(\{i, y_i\}_{i \in S}) =: \mathsf{tk}/\perp$. If $|S| < t$ output $\perp$. Otherwise parse y_i as h_i for $i \in S$ and output $\prod_{i \in S} h_i^{\lambda_{i,S} \bmod p}$
- $\mathsf{Verify}(\mathsf{vk}, x, \mathsf{tk}) =: 1/0$. Return 1 if and only if $\mathsf{VerDDH}(\mathcal{H}(x), \mathsf{vk}, \mathsf{tk}, g) = 1$.

Figure 19: The threshold pairing-based signature scheme of Boldyreva [Bol03].

P_2 can emulate the sign-on phase on its own to generate a token for $x = 0$. The resulting token is the agreed key by both parties.

Notice that the generated token might not be a valid output for the key agreement protocol, but the two parties can apply randomness extractor to the token and obtain randomness to generate a valid key. We omit the details here.

The security of the key-agreement protocol relies on the unforgeability of the PbTA scheme. Intuitively speaking, if there exists a PPT adversary that outputs the agreed token by P_1 and P_2 with non-negligible probability, then the adversary is able to generate a valid token in the PbTA scheme with non-negligible probability, without making any fake requests to the servers (thus keeping all $Q_{C,i}$ to zero), contradicting the unforgeability property. Next we give provide a formal proof.

Theorem D.1 *A secure two-party key agreement protocol can be constructed from any PbTA scheme in a black-box way.*

Proof. Let $\Pi = (\mathsf{GlobalSetup}, \mathsf{SignUp}, \mathsf{Request}, \mathsf{Respond}, \mathsf{Finalize}, \mathsf{Verify})$ be a PbTA scheme. The secure two-party key agreement protocol is presented in Figure 20.

The protocol uses PbTA in a black-box way. Since the tokens $\mathsf{tk}, \mathsf{tk}'$ are generated using the same C, x, and secret key, they are equivalent. Hence the two parties agree on a token (which can be used to extract randomness to generate a key). Now we show that if there exists a PPT adversary Adv that outputs the agreed token by P_1, P_2 in the key-agreement protocol, then we can construct another adversary $\mathcal{B}$ that breaks unforgeability of the PbTA scheme.

$\mathcal{B}$ does not corrupt any server or client. It then calls $\mathcal{O}_{\mathsf{signup}}(C)$ to obtain $\{\mathsf{msg}_i\}_{i \in [n]}$, and calls $\mathcal{O}_{\mathsf{server}}(i, \mathsf{store}, \mathsf{msg}_i)$ to register C for all $i \in \mathcal{T}$. Then it calls $\mathcal{O}_{\mathsf{req}}(C, \mathsf{pwd}, 0, \mathcal{T})$ to obtain $\{\mathsf{req}_i\}_{i \in \mathcal{T}}$, and calls $\mathcal{O}_{\mathsf{server}}(i, \mathsf{respond}, \mathsf{req}_i)$ to obtain res_i for all $i \in \mathcal{T}$. $\mathcal{B}$ runs Adv with input being the transcript of the key-agreement protocol, consisting of $\{(\mathsf{pp}, C), \{\mathsf{req}_i\}_{i \in \mathcal{T}}, \{\mathsf{res}_i\}_{i \in \mathcal{T}}\}$, and obtains an output $\widetilde{\mathsf{tk}}$ from Adv. Then $\mathcal{B}$ simply outputs $(C, 0, \widetilde{\mathsf{tk}})$.

In the security game $\mathsf{SecGame}_{\Pi, \mathsf{Adv}}(1^\kappa, n, t, \mathsf{P})$ (Figure 7) for $\mathcal{B}$, we have $Q_{C,i} = 0$ for all i. By the unforgeability definition, there exists a negligible function negl such that

$$\mathsf{Pr}[\mathsf{Verify}(\mathsf{vk}, C, 0, \widetilde{\mathsf{tk}}) = 1] \leq \mathsf{negl}(\kappa).$$

Let the password space be $\mathsf{P} := \{\mathsf{pwd}\}$, set $n := 2, t := 2$, let $\mathcal{T}$ be the set of all servers in Π_{sym}, and set $x := 0$.

1. P_2 executes the following:
   - Run $\mathsf{GlobalSetup}(1^\kappa, n, t, \mathsf{P}) \to (\llbracket \mathsf{sk} \rrbracket, \mathsf{vk}, \mathsf{pp})$.
   - Run $\mathsf{SignUp}(C, \mathsf{pwd}) \to ((C, \mathsf{msg}_1), \ldots, (C, \mathsf{msg}_n))$.
   - Set $\mathsf{rec}_{i,C} := \mathsf{msg}_i$.
   - Send (pp, C) to P_1.
2. On receiving the first message from P_2, P_1 does the following:
   - Run $\mathsf{Request}(C, \mathsf{pwd}, x, \mathcal{T}) \to (\{(C, x, \mathsf{req}_i)\}_{i\in\mathcal{T}}, \mathsf{st})$.
   - Send $\{\mathsf{req}_i\}_{i\in\mathcal{T}}$ to P_2.
3. On receiving the message from P_1, P_2 does the following:
   - Run $\mathsf{Respond}(\mathsf{sk}_i, \mathrm{REC}_i, C, x, \mathsf{req}_i) \to \mathsf{res}_i$ for $i \in \mathcal{T}$.
   - Send $\{\mathsf{res}_i\}_{i\in\mathcal{T}}$ to P_1.
   - Emulate the protocol:
     a. $\mathsf{Request}(C, \mathsf{pwd}, x, \mathcal{T}) \to (\{(C, x, \mathsf{req}'_i)\}_{i\in\mathcal{T}}, \mathsf{st}')$.
     b. $\mathsf{Respond}(\mathsf{sk}_i, \mathrm{REC}_i, C, x, \mathsf{req}'_i) \to \mathsf{res}'_i$ for $i \in \mathcal{T}$.
     c. $\mathsf{Finalize}(\mathsf{st}', \{\mathsf{res}'_i\}_{i\in\mathcal{T}}) \to \mathsf{tk}'$.
   - Output tk'.
4. On receiving the second message from P_2, P_1 executes the following:
   - Run $\mathsf{Finalize}(\mathsf{st}, \{\mathsf{res}_i\}_{i\in\mathcal{T}}) \to \mathsf{tk}$.
   - Output tk.

Figure 20: Secure two-party key agreement protocol

However, Adv's token $\widetilde{\mathsf{tk}}$ is valid with non-negligible probability, leading to a contradiction. ■

Combining the above theorem with the result of Impagliazzo and Rudich [IR90], we have the following corollary:

Corollary D.2 *If there exists a PbTA scheme that only makes black-box use of one-way functions, then $P \neq NP$.*

This corollary provides us with evidence that it is unlikely to construct PbTA schemes that only makes black-box use of one-way functions. Notice that we did not rule out the possibility of getting around this problem by making non-black-box use of one-way functions. We leave that as an interesting open problem.