Threshold PAKE with Security against Compromise of all Servers*

Yanqi Gu[1], Stanislaw Jarecki[1], Pawel Kedzior[2], Phillip Nazarian[1], and Jiayu Xu[3]

[1] University of California Irvine, USA, {yanqig1, sjarecki, pnazaria}@uci.edu
[2] University of Warsaw, Poland, p.kedzior@mimuw.edu.pl
[3] Oregon State University, USA, xujiay@oregonstate.edu

Abstract. We revisit the notion of *threshold Password-Authenticated Key Exchange* (tPAKE), and we extend it to *augmented* tPAKE (atPAKE), which protects password information even in the case all servers are compromised, except for allowing an (inevitable) offline dictionary attack. Compared to prior notions of tPAKE this is analogous to replacing symmetric PAKE, where the server stores the user's password, with an *augmented* (or *asymmetric*) PAKE, like OPAQUE [44], where the server stores a password *hash*, which can be used only as a target in an offline dictionary search for the password. An atPAKE scheme also strictly improves on the security of an aPAKE, by secret-sharing the password hash among a set of servers. Indeed, our atPAKE protocol is a natural realization of *threshold* OPAQUE.

We formalize atPAKE in the framework of Universal Composability (UC), and show practical ways to realize it. All our schemes are generic compositions which interface to any aPAKE used as a sub-protocol, making them easier to adopt. Our main scheme relies on *threshold Oblivious Pseudorandom Function* (tOPRF), and our independent contribution fixes a flaw in the UC tOPRF notion of [41] and upgrades the tOPRF scheme therein to achieve the fixed definition while preserving its minimal cost and round complexity. The technique we use enforces implicit agreement on arbitrary context information within threshold computation, and it is of general interest.

1 Introduction

Passwords remain a dominant method of authentication of end-users on the Internet (and beyond),[4] and for decades the prime mechanism for client-server password authentication has been "password-over-TLS", where the user sends the password to the server over a secure channel authenticated via a Public Key Infrastructure (PKI), and the server compares the received password against the *salted*, i.e. randomized, *password hash* created during user registration.

* This is an extended version of the paper which appeared in [34].
[4] See e.g. [55, 56, 52] for usability review of alternatives to password authentication.

This scheme has multiple weaknesses, including password visibility on the server during authentication, accidental storage of passwords (for examples see e.g. [1, 2]), and password leakage if the user falls prey to the so-called "phishing" attack. To improve upon this the Internet Engineering Task Force (IETF) conducted a standardization process for a much stronger mechanism, a (strong) *asymmetric (or augmented) Password-Authenticated Key Exchange* (aPAKE), see e.g. [32, 44] and references there-in, where the server stores a randomized password hash for each user but the authentication protocol does not rely on PKI, except possibly for user registration. An aPAKE scheme eliminates all the above weaknesses of password-over-TLS, limiting attacks to the two unavoidable avenues, namely online password tests, and offline password tests in the case of server compromise. The IETF aPAKE competition chose the OPAQUE protocol [44, 13], which in addition to low computation and communication costs, uses an Authenticated Key Exchange (AKE) protocol as a black-box, which makes it easy to integrate with existing secure channel establishment protocols like TLS 1.3 [37, 62].

Strengthening password protection by server distribution. The online password tests, where the adversary tests a password by using it to authenticate, can be mitigated by setting limits on the number of unsuccessful authentication attempts.[5] The exposure to offline password tests after server compromise can be reduced as well, using Multi-Party Computation (MPC) [33, 10]: If the aPAKE server is emulated by n parties via an MPC protocol, then the offline testing attack is enabled only if the adversary corrupts more than some threshold $t < n$ of these parties and reconstructs server data, i.e. the password hash.

Employing generic MPC techniques makes the complexity of such a solution linear in the circuit description of the aPAKE server code, which for OPAQUE involves elliptic curve operations, symmetric ciphers, and CRH hashes, with a resulting circuit with tens of thousands of gates. A natural question is whether there are much more practical solutions to an aPAKE where the server-held password hash is secret-shared across a group of servers, and the offline password tests attack avenue is enabled only after corruption of $t+1$ servers.

Threshold *symmetric* PAKE. MPC-emulation of one party in a *symmetric* PAKE is known as *threshold PAKE* (tPAKE). It was addressed in many works starting from Ford-Kaliski [29] and Jablon [38], whose proposals included informal security arguments. MacKenzie, Shrimpton, and Jakobsson [53] showed the first tPAKE secure in a game-based model, for arbitrary t, assuming PKI. Gennaro and Raimondo [26] showed a password-only tPAKE for $t < n/3$, and Abdalla et al. [3] improved it to $t < n/2$ in the Random Oracle Model (ROM). Jarecki, Kiayias, and Krawczyk [39] constructed game-based tPAKE for any t using an intermediary tool of *password-protected secret-sharing* (PPSS), a.k.a. *password-authenticated secret-sharing* (PASS), [6, 19, 17, 40, 41]. In addition, several works focus on the case of 2 servers,

[5] Two-factor authentication *could* mitigate on-line password tests as well, but two-factor authentication is not commonly used in that way.

known as *2PAKE* [15, 63, 48, 66, 47, 19, 49, 11, 43], including support for proactive security and universal composability [16, 67, 50].

However, in these works the servers MPC-emulate a *symmetric* PAKE, and make no security guarantees after corruption of $t+1$ servers, i.e. the adversary who corrupts that threshold might reconstruct a plaintext password, as opposed to its (salted) hash. Current cryptographic literature thus gives the implementers two incomparable choices for protecting password-related information on the server: They can protect it by storing only password hashes, using either the password-over-TLS method or an aPAKE like OPAQUE, or they can protect it using secret-sharing among n servers and using tPAKE, but the latter choice is worse than the former if the attacker corrupts $t+1$ servers.

Contribution #1: UC augmented threshold PAKE model. We define a tPAKE notion which achieves the best of both worlds, i.e. where the compromise of $t+1$ servers leaks only a salted password hash, which enables offline password tests but does not reveal the password in the clear. We call such scheme an *augmented* threshold PAKE (*atPAKE*), and we define it in the Universal Composability (UC) framework [20], by extending the UC *(strong) aPAKE* notion [44] to the multi-server setting.[6]

We note that the UC framework for expressing security of password authentication protocols, beginning with the Canetti et al. model for UC PAKE [22], is much stronger than extensions of the game-based PAKE security notion of Bellare-Pointcheval-Rogaway (BPR) [9], because in addition to arbitrary interactions between protocol instances it can capture security in the face of arbitrary password correlations, password mistyping, and arbitrary password information leakage. Indeed, UC security was a requirement in the PAKE/aPAKE competition conducted by the IETF, and if distributing the server should strictly upgrade the security properties of UC aPAKE, then the notion that captures the security of such distribution must extend and strengthen the UC aPAKE notion.

We define the UC atPAKE model in a flexible way, distinguishing between two types of servers: A *target server*, who establishes a secure session with a password-authenticated client, and an *auxiliary server*, who holds a secret-share of the password hash but does not establish a secure session. Similar split of roles was considered for both tPAKE and 2PAKE, e.g. in [26, 3, 11], with the target server sometimes called a *gateway*. However, these solutions assumed a single party playing the target role and required secure channels between each pair of servers. By contrast, our protocols admit an arbitrary number of target and auxiliary servers, with no prior trust assumptions between them. (However, we imply security *only if* the client is uncorrupted during account initialization.) In our model the auxiliary servers do not learn whether authentication succeeds, but if they need this information, e.g. to implement rate-limiting on unsuccessful

[6] Here we use *aPAKE* to refer to *strong* aPAKE functionality of [44], denoted saPAKE therein, and we use *weak aPAKE* to refer to the original aPAKE functionality of [32]. The weak aPAKE model allows for speeding up the offline attack by precomputation performed before server compromise, while the (strong) aPAKE disallows it.

authentication attempts, it can be supported if each auxiliary server also plays a target server role.

We view our UC atPAKE model as a major part of our contribution. As a sanity check, we verify that any protocol that realizes the UC atPAKE notion, simplified to the case where auxiliary and target servers are equated, is also secure under the game-based tPAKE notion of MacKenzie, Shrimpton, and Jakobsson [53], upgraded using the real-or-random extension of the BPR game-based PAKE model introduced by Abdalla-Fouque-Pointcheval [4]. Recall that a similar check was done by Canetti et al. [22] who verified that their UC PAKE notion implies the BPR game-based PAKE notion of [9].

Contribution #2: *Augmented* threshold PAKE schemes. We show how to realize the atPAKE notion in a modular way, as a generic composition of universally composable sub-protocols, and we prove security of our schemes under the UC atPAKE definition discussed above. Our main proposal realizes UC atPAKE by generically combining *threshold Oblivious Pseudorandom Function* (tOPRF) [41] with an aPAKE scheme. The variants of this approach include replacing the tOPRF component with *threshold Partially Oblivious PRF* (tPOPRF) [28], or with *augmented Password-Protected Secret-Sharing* (aPPSS) [27]. We include an overview of these variants in Section 1.2 below.

Contribution #3: Threshold Oblivious PRF. Oblivious PRF (OPRF) [30] is a 2-party protocol between a server who holds key k of PRF F and a client who holds an argument x, s.t. the client outputs $y = F_k(x)$, but no information on x is leaked to the server. OPRF was defined in the UC framework in [39, 40], and it is realized by the "2HashDH" scheme [40] based on the Gap-OneMore-DH (Gap-OMDH) assumption in a prime-order group. For OPRF's based on other assumptions see e.g. [46, 39, 5, 23]. Threshold OPRF (tOPRF) [41] replaces the OPRF server with a group that secret-shares the PRF key.

We revisit the UC tOPRF notion of [41].[7] We point out a subtle flaw in their model which makes it ambiguous, and we propose a revised UC tOPRF definition which strengthens the model and fixes the ambiguity. The original tOPRF protocol of [41], based on secret-sharing of the PRF key in the 2HashDH OPRF of [40], does not seem to be provably secure in the fixed model, as we explain in the technical overview below. However, we show that adding an additional form of blinding to this tOPRF scheme – which adds only a very modest cost to the protocol – lets the modified protocol realize the fixed UC tOPRF notion, under the Gap-OMDH and DDH assumptions in ROM. Moreover, whereas the original tOPRF of [41] was analyzed under a complex interactive assumption, which was shown secure in the generic group model, amending that protocol with our blinding method not only lets the protocol realize a stronger security model, but it does so under much simpler security assumptions, adding only the DDH assumption to the Gap-OMDH assumption required by the underlying 2HashDH OPRF.

[7] Variants of UC tOPRF notion were also defined and constructed in [8] and [24], but these works target only the setting of n-out-of-n sharing.

We extend the UC tOPRF model and our protocol to threshold *Partially Oblivious* PRF (tPOPRF) [28]. This protocol variant has applications e.g. to an atPAKE scheme where the auxiliary servers share $O(1)$-sized state across all user accounts. (We discuss this extension in Section 1.2 below.)

UC tOPRF can also be used to implement augmented PPSS (aPPSS) [27] via the simple compiler of [41]. Augmented PPSS has further applications, e.g. to *password-protected cryptosystems* [27]. An advantage of aPPSS built from tOPRF via the compiler of [41] is that if tOPRF is *proactively* secure (see Section 1.2 below) then so is the resulting aPPSS. By contrast, the aPPSS construction shown in [27] does not seem easy to proactivize.

1.1 Technical Overview

The idea behind our atPAKE protocol is simple and indeed it has appeared in close variants before. Our tOPRF-based construction and its aPPSS-based modification, are close variants of the game-based tPAKE's of resp. [41] and [39]. They are also natural threshold extensions of resp. the "server learns first" variant of OPAQUE called OPAQUE' in [44, 35, 58], and of OPAQUE itself [44]. They can also be seen as extensions of the 2PAKE of [43] to general thresholds. Indeed, all these protocols are natural threshold extensions of the *password-hardening* idea of Ford-Kaliski [29] and Jablon [38], which originated all research on threshold PAKE's. The idea of [29, 38] is a blueprint for 2PAKE: The client on password pw computes a *hardened password* $\mathsf{rw} = F_k(\mathsf{pw})$ via OPRF with (an auxiliary) server #1 who holds key k, and then uses rw to authenticate to (a target) server #2. If the last step is a (weak) aPAKE instance, this scheme was shown as a game-based secure augmented 2PAKE [43], where *augmented* refers to the property that corruption of both servers enables offline password tests but does not leak the cleartext password. Indeed, if PRF F_k is appropriately constructed then leaking key k held by server #1 and $\mathsf{rw} = F_k(\mathsf{pw})$ held by server #2, does not leak argument pw except via brute-force attack, where an evaluation of $F_k(\cdot)$ on each argument requires some fixed amount of computation.

First, observe that using (strong) aPAKE [44] instead of PAKE in the last step strengthens security by eliminating advantages due to precomputation in the offline attack in case of all-parties compromise. Second, make it into a "threshold cryptosystem" by replacing OPRF with a (t, n)-threshold tOPRF involving n auxiliary servers. Finally, rather than use the tOPRF-derived value $\mathsf{rw} = F_k(\mathsf{pw})$ directly, let the user derive $\mathcal{T}$-specific password as $\mathsf{rw}_{\mathcal{T}} = \mathsf{KDF}_{\mathsf{rw}}(\mathcal{T})$. This forms our tOPRF-based atPAKE construction: The client on password pw computes $\mathsf{rw} = F_k(\mathsf{pw})$ via tOPRF with the auxiliary servers who hold a (t, n)-threshold secret-sharing of key k, and then uses $\mathcal{T}$-specific value derived from rw in an instance of aPAKE with the target server $\mathcal{T}$. We prove that this scheme is a UC atPAKE if the tOPRF subprotocol is a UC tOPRF.

UC atPAKE model. The UC atPAKE model we propose is a threshold extension of the UC (strong) aPAKE model [44], customized to a division of roles of auxiliary and target servers, where the former play the role of

"guardians", who must agree for an authentication instance to go through, while the latter are authentication end-points.

The security properties of our UC atPAKE model can be summarized as follows: If a user creates an account with a target server with an identifier *sid* and a set of n auxiliary servers, then an authentication attempt against the target server requires a unique password guess and participation of $t+1$ of these auxiliary servers who agree on an *sid*-dependent authentication attempt. Turning to active attacks against the client, if the attacker plays man-in-the-middle on client interaction with all servers then the security is as in PAKE, i.e. one user authentication instance allows the attacker one on-line password test. However, our model strengthens this basic guarantee s.t. if the adversary is passive in client's interaction with at least some auxiliary servers then the on-line password test is only possible using password guesses which the attacker used himself in on-line interactions with these servers. In other words, as in the case of the online attack against the target server, the attacker can on-line test a password only if it uses it in an *sid*-tagged interaction with auxiliary servers. This limits online attacks on the client who fails to authenticate the target server but correctly authenticates the auxiliary servers. Finally, offline password tests are enabled only if the adversary corrupts $t+1$ auxiliary servers *and* the target server, and there is no other avenue for learning information on the password.

Requirements on tOPRF and the flaw in the tOPRF model [41]. The above requirements impose the following contract on the tOPRF subprotocol: To get one online password test opportunity, either against the target server or the client, the adversary must engage $t+1$ auxiliary server instances under the target account identifier. We strengthen this contract further, so that all participating auxiliary server instances must run on the same sub-session identifier *ssid*. This allows more flexibility in the applications, e.g. *ssid* can include context information which must be approved by all these servers (and approved by the target server if the latter is in the auxiliary group).

The tOPRF of [41] is perfectly hiding for the client and does not enforce that the servers agree on a sub-session identifier *ssid* they use to service an interaction with a user. Indeed, it is unclear how a simulator in that protocol can identify which server executions pertain to a user computing consistently on some fixed password. This was observed by [41], who tried to solve this by letting the ideal tOPRF functionality $\mathcal{F}_{\mathsf{tOPRF}}$ pick an arbitrary subset of sessions with distinct $t+1$ servers which are "utilized" for computing one OPRF value. However, this turns out to create an ambiguous and effectively unrealizable model.

Consider a tOPRF instance with $n=3$ servers $\mathsf{S}_1, \mathsf{S}_2, \mathsf{S}_3$ and threshold $t=1$, so one needs 2 servers to compute $F_k(\cdot)$. Assume that the environment allowed each of servers $\mathsf{S}_1, \mathsf{S}_2, \mathsf{S}_3$ to engage in tOPRF, and the simulator SIM observes that the adversary computes $F_k(\cdot)$ on some argument x_1, so SIM sends the "evaluate $F_k(\cdot)$ on x_1" request to $\mathcal{F}_{\mathsf{tOPRF}}$, which in the model of [41] must decide which pair of server sessions to utilize for this evaluation. Now, whatever choice it makes, with 2/3 probability it will not match the subset which the real-world adversary used, e.g. if the latter picks its set of two servers at random. For

example, assume the adversary computes $F_k(\cdot)$ on x_1 via interaction with S_1 and S_2, whereas interaction with S_3 was directed at computing $F_k(\cdot)$ on x_2, and assume that functionality $\mathcal{F}_{\mathsf{tOPRF}}$ picked the set of utilized servers badly, e.g. it chose $\{\mathsf{S}_2, \mathsf{S}_3\}$. Unfortunately, this will prevent correct simulation afterwards. Assume the environment cooperates with the adversary, and after this $F_k(x_1)$ evaluation the environment allows S_1 to engage in one more tOPRF instance. The real-world adversary can evaluate $F_k(\cdot)$ on x_2 by using this last interaction with S_1 together with the interaction with S_3, but when SIM asks $\mathcal{F}_{\mathsf{tOPRF}}$ for the value of $F_k(\cdot)$ on x_2, functionality $\mathcal{F}_{\mathsf{tOPRF}}$ is stuck: The only two non-utilized server sessions are two sessions by a single server S_1, whereas an evaluation of $F_k(\cdot)$ on any new argument requires non-utilized sessions with 2 *different* servers.

Fixing the tOPRF model of [41]. A natural fix is to change $\mathcal{F}_{\mathsf{tOPRF}}$ s.t. simulator SIM must extract the set of servers which the real-world adversary uses to compute $F_k(\cdot)$ on a single argument. However, the tOPRF protocol of [41] does not seem to enable such simulation. As mentioned above, tOPRF of [41] is a threshold version of 2HashDH OPRF of [40]. In the latter F_k is defined as $F_k(x) = H_3(x, H_1(x)^k)$ where H_1, H_3 are RO hash functions, range of H_1 is a group $\mathbb{G}$ of prime order m, and key k is random in $\mathbb{Z}_m$. This is a PRF under the CDH assumption on group $\mathbb{G}$ in ROM. In the 2HashDH protocol for oblivious evaluation of this PRF, the client on input x picks $r \leftarrow_{\$} \mathbb{Z}_m$ and sends to the server a *blinded* form of its argument, $a = H_1(x)^r$. The server responds with $b = a^k$, which the client de-blinds and outputs $F_k(x)$ as $H_3(x, b^{1/r})$. In protocol 2HashTDH which is a tOPRF version of this scheme [41], k is (Shamir) secret-shared as $(k_1, ..., k_n)$, the client sends a to $t{+}1$ servers, each S_i responds with $b_i = a^{k_i}$, and the client recovers $H_1(x)^k = b^{k/r} = \prod_i (b_i)^{\lambda_i / r}$ where λ_i's are interpolation coefficients. However, a malicious client can send $a_i = H_1(x)^{r_i}$, for random r_i, to each S_i, and still recover $H_1(x)^k$ as $\prod_i (b_i)^{\lambda_i / r_i}$. Since each a_i is a random group element, the simulator's view is independent of which a_i's correspond to the same argument x.

Fixing protocol 2HashDH: enforcing agreement without interaction. Still, an honest sender sends the same a to each server, so if the servers preceded the above tOPRF with a round of agreement on $a, ssid$, this would bind $t{+}1$ server tOPRF sessions to a single $x, ssid$. This extra round of agreement introduces costs and delays, and might not be easy to implement e.g. if one tOPRF node is the user's own personal device, a cell phone or a USB dongle. We show protocol *3HashTDH*, which enforces $(a, ssid)$-binding on $t{+}1$ server sessions without sacrificing the optimal round-complexity of 2HashTDH, using a form of "label-based blinding", applicable to threshold exponentiation. Namely, in addition to sharing $(k_1, ..., k_n)$ of key k, the servers hold a random zero-sharing $(z_1, ..., z_n)$, i.e. shares of a random t-degree polynomial which evaluates to zero, and using another hash function H_2 onto $\mathbb{G}$, server S_i on input $ssid$ and client's message a set its response to $b_i = a^{k_i} \cdot (H_2(ssid, a))^{z_i}$. The client computes $H_1(x)^k$ in the same way, i.e. as $\prod_i (b_i)^{\lambda_i / r} = \prod_i (a)^{\lambda_i k_i / r} \cdot \prod_i (H_2(ssid, a))^{\lambda_i z_i / r}$: The first factor evaluates to $a^{k/r} = H(x)^k$ as before, while the second factor evaluates to 1 because z_i's

form a zero-sharing, hence $\sum_i \lambda_i z_i = 0$. If H_2 is an RO hash onto $\mathbb{G}$ then under the DDH assumption, the blinding factors $(H_2(ssid, a))^{z_i}$ used by any t servers S_i are indistinguishable from random group elements (after this threshold the blinding factors are correlated because z_i's lie on a t-degree polynomial). Consequently, unless $t+1$ servers use the same $(a, ssid)$ these blinding factors mask server's responses, making effective evaluation possible only if $t+1$ servers use the same $(a, ssid)$ pair.

1.2 Protocol Variants, Extensions, and Applications

Auxiliary servers with constant-sized state. We consider several further variants of our main tOPRF+(s)aPAKE construction. Firstly, note that in this tOPRF-based construction the auxiliary servers hold separate tOPRF keys for each user. However, the auxiliary servers can keep only a *single secret-shared key* if we replace tOPRF with threshold *Partially* Oblivious PRF (tPOPRF) [28]. *Partially Oblivious* PRF (POPRF) [28] extends OPRF to 2-party evaluation of a PRF whose arguments are pairs $(x_{\mathsf{priv}}, x_{\mathsf{pub}})$, where x_{priv} is a private input of the client while x_{pub} is known to both parties, and the protocol hides only x_{priv} from the server. The "Pythia" POPRF of [28] is secure under One-More BilinearDH (OMBDH) under a game-based definition. For POPRF's based on other assumptions see e.g. [42, 65].[8] Threshold POPRF (tPOPRF) [28, 42] replaces the POPRF server with a group that secret-shares the PRF key.

In the atPAKE application replacing tOPRF with tPOPRF means that a single secret-shared key can be re-used across all user accounts: If the servers set the public tPOPRF input x_{pub} to an account identifier UID, and the user's private input is $x_{\mathsf{priv}} = \mathsf{pw}$, then the user's output is $\mathsf{rw} = F_k(\mathsf{pw}, \mathsf{UID})$, which by the PRF property of F can be interpreted as $\mathsf{rw} = F'_{k[\mathsf{UID}]}(\mathsf{pw})$ where F' is a PRF and $k[\mathsf{UID}]$ is a user-specific PRF key. We show that a threshold version of the Pythia POPRF [28], amended by the same blinding technique as above, realizes the UC tPOPRF functionality in ROM under Gap-OMBDH and the DDH assumption on the target group.

Structured authentication data. We also consider a variant where tOPRF is replaced with aPPSS [27], i.e. a protocol that allows the client holding password pw to decrypt and authenticate an arbitrary secret rw which was secret-shared among the auxiliary servers in initialization. (Furthermore, the aPPSS is *augmented* in the same sense as atPAKE, i.e. corruption of $t+1$ servers allows only for an offline dictionary attack against the password.) A benefit of using aPPSS over tOPRF is that aPPSS can be built from (non-threshold) OPRF [40, 27], which might require weaker assumptions, e.g. only GapOMDH [40], or only DDH (with more protocols rounds) [17], or LWE

[8] POPRF can be implemented as $F'_k(x_{\mathsf{priv}}, x_{\mathsf{pub}}) = F_{F^*_k(x_{\mathsf{pub}})}(x_{\mathsf{priv}})$ using any OPRF F and PRF F^*, but this generic construction might not have properties like threshold implementation, updatability, or verifiability without x_{pub}-dependent keys [28, 65].

[5]. Moreover, aPPSS based on OPRF was shown to be *adaptively secure* for arbitrary t, n parameters [27] (see below).

A disadvantage of this protocol variant is that it enables offline password testing attack after compromise of t+1 auxiliary servers, without the compromise of a target server, because the aPPSS datastructure already allows for verification of the password guess. Indeed, our aPPSS-based atPAKE scheme can be seen as a threshold counterpart to OPAQUE, where the client authenticates the server-supplied data before using it to authenticate to the server, while our tOPRF-based scheme can be seen as a threshold counterpart to OPAQUE', where the (target) server is the first party that can verify an authentication result. Furthermore, we don't know how to make the OPRF-based aPPSS construction *proactive* (see below).

Adaptive and Proactive security. We show our t(P)OPRF protocols secure for arbitrary t, n parameters in the *static corruptions* model, i.e. if the environment corrupts all parties at the outset of the protocol. A variant of the same argument shows that these protocols remain secure against *adaptive corruptions*, but only if $\binom{n}{t}$ is polynomial in the security parameter. The same security statements carry to the atPAKE construction instantiated if t(P)OPRF is instantiated as above.

Another benefit of our t(P)OPRF-based atPAKE protocol is that it can be *proactivized*, i.e. made secure against proactive adversary, using standard techniques of distributed secret-sharing randomization [36]. We note that the same techniques do not extend to the aPPSS of [27], which leaves an open question of constructing a practical atPAKE which is both proactive and adaptively secure for arbitrary t, n parameters.

Practical Advantages and Applications. Our UC atPAKE protocol is highly practical: It involves a single round of low bandwidth interaction between the client and t+1 auxiliary servers, followed by an aPAKE instance between the client and the target server. The servers do not need to communicate directly, which makes the scheme flexible: The auxiliary servers can be implemented by different commercial entities offering a "password hardening" service, or they can be *user's own devices*, like a USB stick or a cell phone. Our scheme requires *no trust assumptions* (and no secure channels) between auxiliary servers or between the auxiliary servers and the target servers.

We note that for simplicity we present our protocols in non-robust versions, but *robustness* and *verifiability* can be added using well-known inexpensive ROM-based non-interactive zero-knowledge proofs. We note also that our atPAKE uses aPAKE as a black-box, and can interface with any existing target-server aPAKE implementation, like OPAQUE. (Indeed, its variants can interface with TLS-OPAQUE, password-over-TLS, and others, see Section 6.)

Our UC atPAKE model assumes that the user knows the list of target servers at initialization, but this is done purely to reduce model complexity, because all our protocol variants allow the user to add more target servers by reconstructing rw and computing new $\mathsf{rw}_{\mathcal{T}} = \mathsf{KDF}_{\mathsf{rw}}(\mathcal{T})$ values. Indeed, this

feature make our atPAKE scheme applicable to a (threshold) *password manager* application, implemented by the auxiliary servers.

Other related works. As mentioned above, our atPAKE can be thought of as a *threshold password manager* scheme, where the user recovers service-specific passwords from the master password. Our atPAKE protocols map to this application if the master password is pw, the $\mathcal{T}$-specific password is $\mathsf{rw}_{\mathcal{T}} = \mathsf{KDF}_{\mathsf{rw}}(\mathcal{T})$, and rw is recovered from pw via either tOPRF, tPOPRF, or PPSS. We note that similar usage of *non-threshold* OPRF or POPRF for outsourced password managers was previously considered e.g. in [28, 60, 61].

In other related work, a *password-protected storage* scheme of [24], a variant of PPSS which allows adaptive addition of records, analyzed a similar solution in the n-out-of-n case, i.e. distributed but not (general) threshold.

Another scheme that uses secret-sharing to protect server-stored password hashes is *Distributed Password Verification* [18, 28, 59, 51]. In these schemes compromise of *permanent storage* of all servers leaks only a salted password hash, as in atPAKE. However, these schemes implement only the verification step in the password-over-TLS authentication, i.e. the secret-shared password hash can be used for secure comparison with a cleartext password candidate, but not for authenticated key exchange of a remote entity holding that password.

Roadmap. Section 2 includes notation and security assumptions used across this work, and a brief overview of universal composability. In Section 3 we define UC Threshold Oblivious PRF (tOPRF) and show protocol 3HashTDH which realizes that notion. Section 4 introduces the UC Augmented Threshold PAKE (atPAKE) functionality. Section 5 includes our main construction of secure UC atPAKE from UC tOPRF. Section 6 overviews several variants and extensions of the above construction. We defer some material to the appendices.

Specifically, in the appendices we include a proof that UC atPAKE notion of Section 3 implies a game-based T-PAKE (Appendix A), a proof of that tOPRF scheme of Section 3 realizes UC tOPRF functionality (Appendix B), a proof that atPAKE construction of Section 5 realizes UC atPAKE functionality (Appendix C, and for completeness we include the UC (strong) asymmetric PAKE (aPAKE) functionality which is used in this proof in Appendix D), an extension of our 3HashTDH tOPRF to the threshold *Partially* Oblivious PRF (tPOPRF) (Appendix G), and we present three further variants of the atPAKE protocol: (1) replacing tOPRF with tPOPRF [28] (Appendix H), (2) replacing tOPRF with *augmented* password-protected secret sharing (aPPSS) [27] (Appendix F), and (3) replacing (strong) aPAKE with a weak aPAKE in the last protocol flow (Appendix E).

Acknowledgments. Yanqi Gu, Stanislaw Jarecki and Phillip Nazarian were supported by NSF SaTC TTP award 2030575. Pawel Kedzior was supported by funding from the European Research Council (ERC) under the European Union's Horizon 2020 innovation program (grant PROCONTRA-885666).

2 Preliminaries

Notation. We use τ to denote the security parameter. Given a finite set S, we write $x \leftarrow_\$ S$ to indicate that x is sampled uniformly at random from S. Throughout the paper we assume function $\mathsf{KDF} : \{0,1\}^\tau \times \{0,1\}^* \to \{0,1\}^\tau$ which is a PRF.

We recall the computational assumptions that we use in this paper:

Definition 1. *Let $(\mathbb{G}, \cdot)$ be a cyclic group of prime order m with generator g. The* Decisional Diffie-Hellman problem (DDH) *on $\mathbb{G}$ is to distinguish the following two distributions: $\{(g, g^a, g^b, g^c) : a, b, c \leftarrow_\$ \mathbb{Z}_m\}$ and $\{(g, g^a, g^b, g^{ab}) : a, b \leftarrow_\$ \mathbb{Z}_m\}$. The DDH assumption is that any PPT adversary $\mathcal{A}$ can only solve this problem with negligible advantage $\mathsf{negl}(\tau)$.*

Definition 2. *Let $(\mathbb{G}, \cdot)$ be a cyclic group of prime order m with generator g. The* Gap One-More Diffie Hellman problem (GapOMDH) *on $\mathbb{G}$ is that, given a vector $(y^*, h_1, \ldots, h_q)$ where $h_j \leftarrow_\$ \mathbb{G}$, and $y^* = g^s$ where $s \leftarrow_\$ \mathbb{Z}_m$, along with access to oracle $\mathsf{OMDH}(a)$ which returns a^s and oracle $\mathsf{DDH}(y, h, u)$ which returns 1 if and only if (g, y, h, u) is a Diffie-Hellman tuple, $\mathcal{A}$ wins if it outputs a set W of pairs (j, h_j^s) where $|W|$ is strictly greater than the number of (unique) queries $\mathcal{A}$ made to the OMDH oracle. The GapOMDH assumption is that any PPT adversary $\mathcal{A}$ can win this game with only negligible probability $\mathsf{negl}(\tau)$.*

For groups with bilinear maps, we denote $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ as cyclic groups of prime order m with generators g_1, g_2, and g_T respectively, and define $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ as a bilinear pairing. Recall the definition of a bilinear pairing: $e(g_1^\alpha, g_2^\beta) = g_T^{\alpha\beta}$ for all $\alpha, \beta \in \mathbb{Z}_m$.

Definition 3. *The Gap One-More Bilinear Diffie Hellman problem (GapOMBDH) on $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ is that, given a vector $(y^*, h_1, \ldots, h_q, h'_1, \ldots, h'_q)$ where $h_j \leftarrow_\$ \mathbb{G}_1$ and $h'_{j'} \leftarrow_\$ \mathbb{G}_2$, and $y^* = g_T^s$ where $s \leftarrow_\$ \mathbb{Z}_m$, along with access to oracle $\mathsf{OMDH}(a)$ which returns a^s given $a \in \mathbb{G}_T$, and oracle $\mathsf{DDH}(y, h, u)$ which returns 1 if and only if (y, h, u) is a Diffie-Hellman tuple in $\mathbb{G}_T$, $\mathcal{A}$ wins if it outputs a set W of triples $(j, j', e(h_j, h'_{j'})^s)$ where $|W|$ is strictly greater than the number of (unique) queries to the OMDH oracle. The GapOMBDH assumption is that any PPT adversary $\mathcal{A}$ can win this game with only negligible probability $\mathsf{negl}(\tau)$.*

Note that [28] uses OMBDH assumption instead of GapOMBDH we assume. Specifically, their reductions do not require access to a DDH oracle, because they argue security under under a game-based definition, while [8] and our protocol use simulation-based proofs, where the reduction uses the DDH oracle to maintain consistency of concurrent sessions.

Secure Channel. We assume secure channels, i.e. secure and authenticated communication, modeled as a UC functionality $\mathcal{F}_{\mathsf{channel}}$ in Figure 1. We stress that all our protocols use secure channels *only in initialization.*

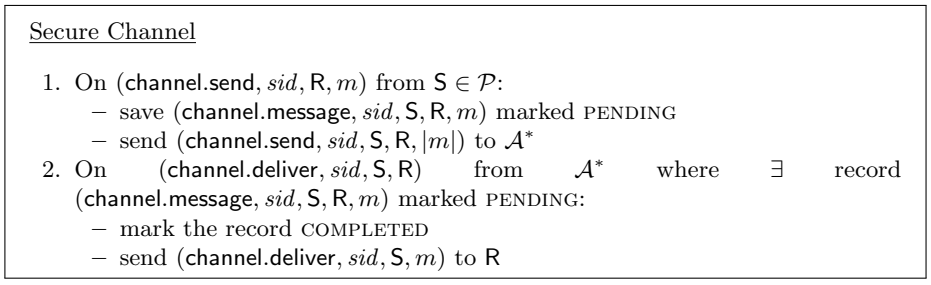

Fig. 1. $\mathcal{F}_{\mathsf{channel}}$: secure and authenticated communication functionality

Universal composability and related notation. In this paper we use the *Universal Composability* (UC) framework [20] to construct security proofs. UC follows the simulation-based paradigm where the security of a protocol is modeled by a machine called the ideal functionality $\mathcal{F}$, which interacts with a set of "dummy" parties and an ideal world adversary SIM, and does all computation in the ideal world. We say that protocol π securely realizes $\mathcal{F}$ if for any PPT $\mathcal{A}$, there is a simulator SIM s.t. for all environments $\mathcal{Z}$, the difference between the real-world view, i.e. an interaction of $\mathcal{Z}$ and $\mathcal{A}$ with parties executing π, and the ideal-world view, i.e. an interaction of $\mathcal{Z}$ and $\mathcal{A}$ with SIM and $\mathcal{F}$, is negligible in τ.

In the descriptions of ideal functionalities, e.g. Figure 2, Figure 4, and others, we specify that each functionality interacts with a set of honest parties $\mathcal{P}$ and an adversary $\mathcal{A}^*$, and we use notation $\mathcal{P}^* = \mathcal{P} \cup \{\mathcal{A}^*\}$. Each functionality assumes set **Corr** includes all initially corrupted parties, and by convention $\mathcal{A}^* \in \mathbf{Corr}$. In our functionalities, protocols, and simulators, we assume strings sid (or sid_{A}, sid_{T}) have the form $sid = (\ldots, \mathbf{S})$ where $\mathbf{S} = (\mathsf{S}_1, \ldots, \mathsf{S}_n)$ is a sequence in $\mathcal{P}$, and $\mathbf{S}_{sid}$ denotes the list $\mathbf{S}$ specified by string sid.

3 Threshold Oblivious PRF

An oblivious pseudorandom function (OPRF) is a protocol with two parties, a server and an evaluator. The server holds the key to a pseudorandom function, and the evaluator holds an input to be evaluated by that pseudorandom function. The OPRF protocol allows the evaluator to evaluate the function obliviously (i.e. without revealing the input to the server) without learning the server's secret key. Since the PRF key is kept secret, evaluators can only compute the function with the online participation of the server, who could, for example, enforce a rate-limiting policy on evaluators.

3.1 Threshold Oblivious PRF Model

Threshold OPRF (tOPRF) is an extension of OPRF introduced by [41] which distributes server across n parties, s.t. any $t + 1$ of them must participate for an evaluation to succeed. As explained in detail in Section 1.1, the tOPRF ideal

functionality of [41] has a subtle flaw that seems to make it unrealizable. In Figure 2 we show $\mathcal{F}_{\mathsf{tOPRF}}$, our modification of the UC tOPRF functionality for the (t, n) threshold. Our functionality $\mathcal{F}_{\mathsf{tOPRF}}$ is a modification of the tOPRF functionality in [41] (see Fig. 1 therein), and it involves several refinements, including the critical fix to the flaw mentioned above. Below we explain the workings of our functionality, including the ways in which it differs from [41].

The initialization phase. The initialization is done between an initializer P_0 and a group of n servers $\mathbf{S}_{sid}$. At the end of the process, each server $\mathsf{S} \in \mathbf{S}_{sid}$ is supposed to record a key share, to be used later in evaluation. In the functionality, this is modeled as having a record $\mathsf{toprf.sinit}$ for each server S, which is marked either ACTIVE or COMPR; the latter denotes the adversary knowing S's key share, which happens if S is compromised, or P_0 is the adversary (in which case the adversary can compute all key shares on its own).

Additionally, P_0 can specify a vector of PRF inputs $(x_1, ..., x_k)$, and obtain their PRF outputs $(F_{sid}(x_1), ..., F_{sid}(x_k))$ during initialization. Though somewhat atypical, this "eval-during-init" feature is natural in our initialization setting. Since P_0 is responsible for creating all key shares and sending them to the servers, there is no reason why P_0 shouldn't be able to locally evaluate the PRF during initialization. Looking ahead, our tOPRF-based atPAKE construction uses this feature to simplify its initialization phase; specifically, it eliminates the need for the client to perform a second round of communication with the tOPRF servers after initializing them.

The evaluation phase. In this phase, a user U begins its evaluation by specifying a PRF input x. Any server whose record is ACTIVE, as well as any corrupted server (whose record is COMPR), may choose to participate; each server S is associated with a *ticket counter* $\mathsf{tx}[sid, ssid_{\mathsf{S}}, i]$ (where S is the i-th server in $\mathbf{S}_{sid}$), which increments if S participates (the mechanics of $ssid_{\mathsf{S}}$ will be explained below). Finally, the ideal adversary specifies an index sid^* which may or may not be the intended index for evaluation sid, together with a set of $t + 1$ servers $\mathbf{C}$ which may or may not be a subset of the intended set of n servers for evaluation $\mathbf{S}_{sid}$. (Giving the adversary the ability to directly specify the evaluation set $\mathbf{C}$ fixes the flaw in the tOPRF functionality of [41].) After that, U receives $F_{sid^*}(x)$ *provided that none of the servers in* $\mathbf{C}$ *has ticket counter 0 (w.r.t. index* sid^**)*, and all those ticket counters decrement. This models a man-in-the-middle adversary that might impersonate $t + 1$ servers and let the user evaluate on a "wrong" PRF key.

As a specific case, the ideal adversary can make the user evaluate an unintended function with index $sid^* \neq sid$: the adversary can act as P_0 and init tOPRF sid^* with itself as all n servers; then it can print tickets for servers corresponding to sid^* at will, evaluate function F_{sid^*} locally on its own inputs, and use index sid^* to respond dishonestly to honest evaluators who are intending to evaluate a different function. This all simply represents the real adversary's ability to locally sample PRFs. To enhance readability, we provide the adversary with a $\mathsf{sndrcomplete}^*$ interface that is simply a shortcut for the

Notation

Initially $\mathsf{tx}[sid, ssid_{\mathsf{S}}, i] := 0$ and $F_{sid}(x)$ is undefined for all sid,$ssid_{\mathsf{S}}, i, x$. When $F_{sid}(x)$ is first referenced $\mathcal{F}_{\mathsf{tOPRF}}$ assigns $F_{sid}(x) \leftarrow_{\$} \{0,1\}^l$.

Initialization

1. On $(\mathsf{toprf.init}, sid, x_1, \ldots, x_k)$ from $\mathsf{P}_0 \in \mathcal{P}^*$, if sid is new (abort otherwise):
   - send $(\mathsf{toprf.init}, sid, \mathsf{P}_0)$ to $\mathcal{A}^*$
   - send $(\mathsf{toprf.initeval}, sid, F_{sid}(x_1), \ldots, F_{sid}(x_k))$ to P_0
   - save $(\mathsf{toprf.init}, sid, \mathsf{P}_0)$ and mark it TAMPERED if $\mathsf{P}_0 \in \mathbf{Corr}$
2. On $(\mathsf{toprf.sinit}, sid, i, \mathsf{P}_0)$ from S where $\mathsf{S} = \mathbf{S}_{sid}[i]$ or ($\mathsf{S} = \mathcal{A}^*$ and $\mathbf{S}_{sid}[i] \in \mathbf{Corr}$), save record $(\mathsf{toprf.sinit}, sid, i, \mathsf{P}_0)$ marked INACTIVE
3. On $(\mathsf{toprf.fininit}, sid, i)$ from $\mathcal{A}^*$ where $\exists$ record $\mathsf{urec} = (\mathsf{toprf.init}, sid, \mathsf{P}_0)$ and record $\mathsf{srec} = (\mathsf{toprf.sinit}, sid, i, \mathsf{P}_0)$ marked INACTIVE:
   - send $(\mathsf{toprf.sinit}, sid, i)$ to $\mathbf{S}_{sid}[i]$
   - if urec is TAMPERED then mark srec TAMPERED
   - else if $\mathbf{S}_{sid}[i] \in \mathbf{Corr}$ then mark srec COMPR
   - else (i.e. urec is not TAMPERED and $\mathbf{S}_{sid}[i] \notin \mathbf{Corr}$) mark srec ACTIVE

Corruption *(in the static corruption model disallowed after any other queries)*

4. On $(\mathsf{toprf.corrupt}, \mathsf{P})$ from $\mathcal{A}^*$ (with permission from $\mathcal{Z}$):
   - set $\mathbf{Corr} := \mathbf{Corr} \cup \{\mathsf{P}\}$
   - mark every ACTIVE record $(\mathsf{toprf.sinit}, sid, i, \mathsf{P}_0)$ COMPR where $\mathsf{P} = \mathbf{S}_{sid}[i]$

Evaluation

5. On $(\mathsf{toprf.eval}, sid, ssid_{\mathsf{U}}, x)$ from $\mathsf{U} \in \mathcal{P}^*$, if this is the first call from U for sid and $ssid_{\mathsf{U}}$:
   - send $(\mathsf{toprf.eval}, sid, ssid_{\mathsf{U}}, \mathsf{U})$ to $\mathcal{A}^*$
   - save $(\mathsf{toprf.eval}, sid, ssid_{\mathsf{U}}, \mathsf{U}, x)$ marked FRESH
6. On $(\mathsf{toprf.sndrcomplete}, sid, i, ssid_{\mathsf{S}})$ from S where $\exists$ record $\mathsf{srec} = (\mathsf{toprf.sinit}, sid, i, \mathsf{P}_0)$ not marked INACTIVE and ($\mathsf{S} = \mathbf{S}_{sid}[i]$ or ($\mathsf{S} = \mathcal{A}^*$ and srec is marked COMPR or TAMPERED)):
   - send $(\mathsf{toprf.sndrcomplete}, sid, i, ssid_{\mathsf{S}})$ to $\mathcal{A}^*$
   - set $\mathsf{tx}[sid, ssid_{\mathsf{S}}, i]{+}{+}$

6*. On $(\mathsf{toprf.sndrcomplete}^*, sid, i, ssid_{\mathsf{S}})$ from $\mathcal{A}^*$ where not $\exists$ record $(\mathsf{toprf.init}, sid, \mathsf{P}_0)$, set $\mathsf{tx}[sid, ssid_{\mathsf{S}}, i]{+}{+}$

7. On $(\mathsf{toprf.rcvcomplete}, sid, ssid_{\mathsf{U}}, sid^*, ssid^*_{\mathsf{S}}, \mathbf{C})$ from $\mathcal{A}^*$ where $|\mathbf{C}| = t{+}1$ and $\exists$ record $(\mathsf{toprf.eval}, sid, ssid_{\mathsf{U}}, \mathsf{U}, x)$ marked FRESH:
   - if $\exists j \in \mathbf{C}$ such that $\mathsf{tx}[sid^*, ssid^*_{\mathsf{S}}, j] = 0$, then abort
   - otherwise mark the record COMPLETED, set $\mathsf{tx}[sid^*, ssid^*_{\mathsf{S}}, j]{-}{-}$ for all $j \in \mathbf{C}$, and send $(\mathsf{toprf.eval}, sid, ssid_{\mathsf{U}}, F_{sid^*}(x))$ to U

Fig. 2. $\mathcal{F}_{\mathsf{tOPRF}}$: threshold OPRF functionality, parameterized by threshold t, number of servers n, and output length l.

above series of actions. This shortcut is never actually needed by the adversary, and indeed the simulator for our tOPRF realization does not make use of it.

The "ticketing mechanism" — inherited from various prior works on (t)OPRF [39], [41] — ensures that in order to compute the PRF value, at least $t+1$ servers must participate in the evaluation process. In our functionality the action of printing tickets comes from the environment, which models the fact that servers can choose when they wish to participate in an evaluation. In contrast, [41, Fig. 1] models ticket-printing as an adversarial action, effectively reducing tOPRF servers to powerless entities that blindly allow tOPRF evaluations whenever they are asked to.

Our model strengthens the ticketing mechanism by further associating each ticket with the server subsession identifier $ssid_{\mathsf{S}}$ used to print it. To complete an evaluation, the adversary specifies not only the tOPRF instance sid^* and server set $\mathbf{C}$ to use, but also the particular $ssid^*_{\mathsf{S}}$ whose tickets to use up. A successful evaluation requires not only that $t+1$ servers agree to participate, but that they agree to participate using the same $ssid_{\mathsf{S}}$. These $ssid_{\mathsf{S}}$ values might, therefore, be used to bind tickets to a particular context (for example, an evaluation timestamp). If this feature is unneeded, the $ssid_{\mathsf{S}}$ field can simply be left blank by servers, in which case it will play no role in ticketing.

Server corruption. Finally, the ideal adversary may corrupt a server P and steal its key share. This is modeled by marking any ACTIVE $\mathsf{toprf.sinit}$ record for P COMPR. Note that the only consequence of corrupting a server is that the adversary can now print tickets for it at will.

3.2 3HashTDH

Figure 3 shows our 3HashTDH protocol Π_{3HashTDH}. Protocol Π_{3HashTDH} relies on secure authenticated channels *during initialization*, to allow for secure communication between the initializer party and the n servers participating in the scheme. In Figure 3 this is modeled via a secure channel functionality $\mathcal{F}_{\mathsf{channel}}$ shown in Figure 1.

3HashTDH uses a prime-order group $\mathbb{G}$ of size m and three hash functions, $H_1 : \{0,1\}^* \to \mathbb{G}$, $H_2 : \{0,1\}^* \to \mathbb{G}$, and $H_3 : \{0,1\}^* \to \{0,1\}^l$. The PRF it defined as $F_k(x) = H_3(x, H_1(x)^k)$, exactly the same as the 2HashDH OPRF of [39]. 2HashDH's (single-server) oblivious evaluation protocol for this PRF is the foundation of our protocol. In 2HashDH, the evaluator first picks $r \leftarrow_{\$} \mathbb{Z}_m$ and sends the blinded input $a := H_1(x)^r$ to the server. The server exponentiates using its secret key and sends $b := a^k$ back to the evaluator. The evaluator de-blinds and computes the final hash, outputting $H_3(x, b^{1/r})$.

The 2HashTDH protocol of [41] extends 2HashDH to the threshold, multi-server setting. Each server S_i now holds a (Shamir) secret share k_i of the PRF key k. The evaluator sends $a := H_1(x)^r$ to $t+1$ servers, and they each respond with $b_i := a^{k_i}$. To combine these responses, the evaluator uses polynomial interpolation in the exponent to compute $b := \prod_i b_i^{\lambda_i} = a^k$, where the λ_i's are Lagrange interpolation coefficients. Finally the evaluator can compute the final output $H_3(x, b^{1/r})$ as before.

Initialization

1. On input $(\mathsf{toprf.init}, sid, x_1, ..., x_k)$, initializer P_0 does:
   - pick $\alpha_0, ..., \alpha_t \leftarrow_\$ \mathbb{Z}_m$ and define polynomial $k(x) := \alpha_0 + \alpha_1 x + ... + \alpha_t x^t$
   - pick $\beta_1, ..., \beta_t \leftarrow_\$ \mathbb{Z}_m$ and define polynomial $z(x) := \beta_1 x + \beta_2 x^2 + ... + \beta_t x^t$
   - for each $i \in [n]$, send $(\mathsf{channel.send}, [sid||i], \mathbf{S}_{sid}[i], (i, k(i), z(i)))$ to $\mathcal{F}_{\mathsf{channel}}$,
   - output $(\mathsf{toprf.initeval}, sid, y_1, ..., y_k)$ s.t. $y_j := H_3(x_j, H_1(x_j)^{\alpha_0})$ for all $j \in [k]$
2. On input $(\mathsf{toprf.sinit}, sid, i, \mathsf{P}_0)$, server $\mathbf{S}_{sid}[i]$ does:
   - await $(\mathsf{channel.deliver}, [sid||i], \mathsf{P}_0, (i, k_i, z_i))$ from $\mathcal{F}_{\mathsf{channel}}$;
   - then save record $(\mathsf{toprf.share}, sid, i, k_i, z_i)$
   - output $(\mathsf{toprf.fininit}, sid, i)$

Evaluation

3. On input $(\mathsf{toprf.eval}, sid, ssid_{\mathsf{U}}, x)$, evaluator U does:
   - pick $r \leftarrow_\$ \mathbb{Z}_m$ and compute $a := H_1(x)^r$
   - for each $i \in [n]$, send $(sid, i, ssid_{\mathsf{U}}, a)$ to $\mathbf{S}_{sid}[i]$
   - await responses $(sid, i, ssid_{\mathsf{U}}, b_i)$ from $\mathbf{S}_{sid}[i]$ for all $i \in \mathbf{C}$, for any set $\mathbf{C} \subseteq [n]$ of size $t+1$;
   - then compute $b := \prod_{i \in \mathbf{C}} b_i^{\lambda_i}$ where λ_i is the Lagrange interpolation coefficient for index i and index set $\mathbf{C}$
   - output $(\mathsf{toprf.eval}, ssid_{\mathsf{U}}, H_3(x, b^{1/r}))$
4. On input $(\mathsf{toprf.sndrcomplete}, sid, i, ssid_{\mathsf{S}})$, server $\mathbf{S}_{sid}[i]$ does:
   - retrieve record $(\mathsf{toprf.share}, sid, i, k_i, z_i)$ (abort if not found)
   - await $(sid, i, ssid_{\mathsf{U}}, a)$ from any U (if it hasn't already been received);
   - then compute $b_i := a^{k_i} \cdot H_2(ssid_{\mathsf{S}}, a)^{z_i}$
   - send response $(sid, i, ssid_{\mathsf{U}}, b_i)$ to U

Fig. 3. Protocol Π_{3HashTDH} which realizes $\mathcal{F}_{\mathsf{tOPRF}}$ in the $\mathcal{F}_{\mathsf{channel}}$-hybrid world.

As explained in Section 1.1, though, this 2HashTDH protocol does not seem to realize the fixed $\mathcal{F}_{\mathsf{tOPRF}}$ functionality (nor does it realize the non-fixed functionality, which it seems no protocol can realize). To fully simulate the environment's real-world view, the ideal-world simulator must be able to observe exactly which servers the man-in-the-middle adversary is using for each evaluation. If the evaluator honestly sends the same $a = H_1(x)^r$ to all servers, then the simulator can indeed make this observation. However, in 2HashTDH, there is nothing preventing a dishonest evaluator from picking $t+1$ different blinding exponents r_i and sending information-theoretically random messages $a_i := H_1(x)^{r_i}$ to each server as part of a single evaluation on x.

With 3HashTDH, we aim to fill this simulatability gap by forcing dishonest evaluators to use a single a with all servers for each evaluation, just as honest evaluators do. 3HashTDH accomplishes this by having the servers apply an a-specific blinding factor to their responses; the evaluator can only remove this blinding factor by combining $t+1$ server responses computed on the same a. In particular, the servers in 3HashTDH hold a Shamir secret sharing $(z_1, ..., z_n)$ of

zero (in addition to their sharing of the PRF key k). Given an evaluator query a, server i responds with $b_i := a^{k_i} \cdot (H_2(a))^{z_i}$. The evaluator combines the responses as $b := \prod_i b_i^{\lambda_i} = \prod_i a^{k_i \lambda_i} \cdot \prod_i (H_2(a))^{z_i \lambda_i}$, the second part of which interpolates to $(H_2(a))^0 = 1$ and disappears. Meanwhile, an adversary who sees only t or less responses for the same a cannot distinguish them from random group elements (under the DDH assumption). Thus, even dishonest evaluators are forced to send the same a to all servers, and simulation succeeds.

As a further feature, we have each server include its subsession identifier $ssid_{\mathsf{S}}$ alongside a in the H_2 input. Then, only server responses corresponding to the same $(ssid_{\mathsf{S}}, a)$ can be combined. Servers can use this $ssid_{\mathsf{S}}$ field to bind evaluations to some context-specific data, which $t+1$ servers must agree upon in order to have a successful evaluation.

In a concurrent work, Das and Ren [25] have used blinding factors formed in the same way as ours, though for a different purpose: achieving adaptive security for a threshold BLS signature scheme. In their case, the message being signed acts as the "binding data" used as input to the random oracle. Both of our works are preceded by Canetti and Goldwasser [21], who employed a similar blinding factor in a CCA-secure threshold encryption scheme. They also used a Shamir sharing of zero in the exponent, but with a fixed base rather than a random oracle output. Therefore, their scheme requires a fresh zero sharing for each execution.

3.3 Security Analysis of 3HashTDH

If corruptions are static, then the 3HashTDH protocol in Figure 3 is secure in the random oracle model under the Gap One-More Diffie Hellman (GapOMDH) and Decisional Diffie Hellman (DDH) assumptions.[9]

Theorem 1. *Protocol 3HashTDH realizes functionality $\mathcal{F}_{\mathsf{tOPRF}}$ with parameters t and n in the $\mathcal{F}_{\mathsf{channel}}$-hybrid model, assuming static corruptions, hash functions H_1, H_2, and H_3 modeled as random oracles, and the GapOMDH and DDH assumptions on group $\mathbb{G}$.*

Specifically, for any efficient adversary $\mathcal{A}$ against protocol 3HashTDH, there exists a simulator SIM such that no efficient environment $\mathcal{Z}$ can distinguish the view of $\mathcal{A}$ interacting with the real 3HashTDH protocol and the view of SIM interacting with the ideal functionality $\mathcal{F}_{\mathsf{tOPRF}}$ with advantage better than $q_I^2/m + q_I \cdot (t \cdot \mathbf{Adv}_{\mathcal{Q}}^{\mathsf{DDH}} + \mathbf{Adv}_{\mathcal{R}}^{\mathsf{GapOMDH}}) \cdot (t+1)$ where q_I is the number of tOPRF instances, $m = |\mathbb{G}|$, and $\mathbf{Adv}_{\mathcal{R}}^{\mathsf{GapOMDH}}$ and $\mathbf{Adv}_{\mathcal{Q}}^{\mathsf{DDH}}$ are bounds on the probability that any efficient algorithm violates the GapOMDH and DDH assumptions, respectively.

[9] Though somewhat unusual, the combination of the GapOMDH and DDH assumptions on group $\mathbb{G}$ is not theoretically problematic. It has been proven that the DDH assumption [12] and the GapOMDH assumption [41] each hold for generic groups. Therefore, our security statement is, at a minimum, sound in the Generic Group Model. We also note that there are precedents for making a Gap assumption alongside the DDH assumption on the same group, e.g. [45].

If corruptions are adaptive, then the 3HashTDH protocol remains secure under the additional assumption that $\binom{n}{t'}$ is a polynomial function of the security parameter for all $0 \leq t' \leq t$.

Theorem 2. *In the case of adaptive corruptions, the statement from Theorem 1 still holds under the additional assumption that* $\binom{n}{t'}$ *is a polynomial function of the security parameter for all* $0 \leq t' \leq t$.

Specifically, no efficient adversary $\mathcal{A}$ *against 3HashTDH has distinguishing advantage better than* $q_I^2/m + q_I \cdot (t \cdot \mathbf{Adv}_{\mathcal{Q}}^{\mathsf{DDH}} + \mathbf{Adv}_{\mathcal{R}}^{\mathsf{GapOMDH}}) \cdot \sum_{t'=0}^{t} \binom{n}{t'}$.

Proof of Theorems 1 and 2 is deferred to Appendix B. We provide a high-level sketch here.

The essential steps of the proof involve the blinding factors $H_2(ssid_{\mathsf{S}}, a)^{z_i}$ that are applied to the server responses. We construct a GapOMDH reduction that simulates the real 3HashTDH behavior, but picks random key shares to send to the adversary for the corrupted servers (denote the number of corrupted servers as t'). For the uncorrupted servers, the reduction responds to evaluation requests with uniformly random group elements *until* the same $(ssid_{\mathsf{S}}, a)$ is queried to $t - t' + 1$ servers. At that point, the reduction queries the OMDH oracle to find a^k, where k is the OMDH secret (here it acts as the PRF key). The reduction then knows enough values to perform interpolation in the exponent between a^k, the t' key shares attributed to the corrupted servers, and the randomly chosen first $t-t'$ responses. Thus, the $(t-t'+1)$th response (and any future responses) to $(ssid_{\mathsf{S}}, a)$ are correctly formed from the point of view of an adversary who wishes to interpolate a^k. The reduction embeds the OMDH challenge group elements in the H_1 responses, and uses the H_3 random oracle queries as an opportunity to intercept completed evaluations. The "Gap" DDH oracle is used to verify whether or not an H_3 query represents a correct evaluation. If the adversary ever exceeds their $\mathcal{F}_{\mathsf{tOPRF}}$-allowed number of evaluations, then the reduction exceeds its OMDH-allowed number of k exponentiations and thereby wins the GapOMDH game.

In this GapOMDH reduction, the first $t - t'$ uncorrupted server evaluation requests for any $(ssid_{\mathsf{S}}, a)$ return random group elements, rather than $a^{k_i} \cdot H_2(ssid_{\mathsf{S}}, a)^{z_i}$ for consistent (secret) (k_i, z_i) as in the real world. This is the only difference between the adversary's views in the reduction and in the real world. We use a series of hybrids to prove that it is not detectable.

Our proof first picks any arbitrary set of $t-t'$ uncorrupted servers. In a series of incremental hybrids, we replace the blinding factor $H_2(ssid_{\mathsf{S}}, a)^{z_i}$ at each of these servers with a randomly chosen group element (for each $(ssid_{\mathsf{S}}, a)$), one by one. For the uncorrupted servers outside of this set, the blinding factors are computed by interpolation in the exponent between the t' key shares attributed to the corrupted servers, the $t-t'$ randomly chosen blinding factors, and the fact that $\{z_i\}$ are a sharing of 0. By a one-time pad argument, a uniformly random blinding factor creates a uniformly random overall server response. Therefore, once all blinding factors are randomized in this way, the adversary's view is identical to that in the GapOMDH reduction. At each hybrid, we use a DDH

reduction to prove that replacing one more server's blinding factors with random values is not detectable by the adversary. In sum, then, the adversary's behavior in the GapOMDH reduction must differ only negligibly from the real world.

The factor of $\sum_{t'=0}^{t} \binom{n}{t'}$ that appears in the adaptive-case security bound is a consequence of the GapOMDH reduction simply guessing (at the start of execution) which t' servers the adversary will eventually corrupt. As long as n and t are small, this guess will succeed with non-negligible probability. It is not sufficient for the reduction to guess a superset (e.g. a t-size superset) of the servers that will eventually be corrupted. The reduction relies on the fact that adversarial computation of a PRF output without having queried $t - t' + 1$ uncorrupted servers always corresponds to a win in the GapOMDH game. This is not true unless the reduction's guess set is exactly the same as the actual corrupted set at the moment of this adversarial computation.

3.4 Extension to Threshold Partially Oblivious PRF

Partially Oblivious Pseudorandom Function (POPRF) [28] is a generalization of OPRF where the argument to the PRF is split into two parts, x_{priv} and x_{pub}. Function evaluation is only partially oblivious because the x_{pub} part of the input is visible to both parties, while x_{priv} is visible only to the evaluator and is hidden from the server. Note that if POPRF $F_k(x_{\mathsf{priv}}, x_{\mathsf{pub}})$ is evaluated s.t. x_{pub} is always $\perp$ (or any other constant), then POPRF behaves exactly like a standard OPRF, hence POPRF can be seen as a generalization of OPRF.

The techniques we use above to implement a UC threshold OPRF (tOPRF) extend to a UC threshold POPRF (tPOPRF). In Appendix G we define tPOPRF via the UC functionality $\mathcal{F}_{\mathsf{tPOPRF}}$, a generalization of our $\mathcal{F}_{\mathsf{tOPRF}}$ functionality in Figure 2, and we show that this functionality is realized by protocol P3HashTDH, which combines the blinding technique used in our 3HashTDH tOPRF protocol with the natural threshold implementation of the pairing-based (single-server) POPRF protocol of Pythia [28]. Protocol P3HashTDH uses only two flows, just like 3HashTDH, and it implements the PRF of Pythia, i.e. $F_k(x_{\mathsf{priv}}, x_{\mathsf{pub}}) = H_3(x_{\mathsf{priv}}, x_{\mathsf{pub}}, e(H_1(x_{\mathsf{priv}}), H_1'(x_{\mathsf{pub}}))^k)$.

We note that [42] and [64] showed alternative POPRF constructions that do not rely on pairings. Both of these constructions should have efficient threshold implementations which realize functionality $\mathcal{F}_{\mathsf{tPOPRF}}$ without bilinear maps. The threshold version of the POPRF of [64] would require additional rounds of communication, while POPRF of [42] is a generic construction from any OPRF, and its threshold implementation can be instantiated using our 2-flows tOPRF protocol 3HashTDH. However, the disadvantage of the latter tPOPRF, just like the POPRF of [42], is that it is efficient only for small groups of servers and it offers no verifiability.

4 Augmented Threshold PAKE Model

Figures 4, 5, and 6 are $\mathcal{F}_{\mathsf{atPAKE}}$, the UC functionality for *augmented threshold PAKE (atPAKE)*. As explained in the introduction, this functionality is flexible

in that it models target servers and auxiliary servers separately. The target servers are the entities that ultimately wish to establish keys with password-authenticated users. The auxiliary servers distribute the secret information in such a way that it requires the participation of $t+1$ of them for a user to establish a session with a target server. If this separation of responsibilities is undesired, atPAKE can simply be instantiated such that the auxiliary and target server lists partially or wholly overlap.

The shadowed text in $\mathcal{F}_{\mathsf{atPAKE}}$ corresponds to relaxations introduced by [44] to the (non-threshold) $\mathcal{F}_{\mathsf{saPAKE}}$ model (included for reference in Appendix D). The OPAQUE protocol [44] realizes $\mathcal{F}_{\mathsf{saPAKE}}$ only with these slight relaxations, but, as argued in [44], the relaxations do not reduce the functionality's practical security properties in any significant way. Since our $\mathcal{F}_{\mathsf{atPAKE}}$ is a threshold generalization of [44]'s $\mathcal{F}_{\mathsf{saPAKE}}$, we inherit the same relaxations.

Initialization. A user U may initialize with a group of auxiliary servers and a group of target servers on password pw, represented by sid_{A} and sid_{T} respectively, using a userinit call to the functionality. Similarly, a server (auxiliary or target) may initialize with a user U using an auxinit or targetinit call. The (ideal) adversary finishes initialization for an auxiliary server $\mathbf{S}_{sid_{\mathsf{A}}}[i]$ by sending finishauxiliaryinit, which establishes the server's file record; the file record is COMPR (i.e., the adversary knows its content) if the server is corrupt. Similarly, a finishtargetinit call finishes initialization for a target server $\mathbf{S}_{sid_{\mathsf{T}}}[j]$ and establishes a corresponding record, which is COMPR if that server is corrupt. However, a target server's record might additionally be TAMPERED if the user it communicates with is corrupt, and in this case we allow the adversary to change the password in the target server's file from pw to some pw^* of the adversary's choice; the adversary can also overwrite the auxiliary server instance sid_{A} with which the target server's file is associated. As will be apparent in the other sections of the functionality, this TAMPERED case models the intuitive notion that many atPAKE security properties are lost if the original initializing user is dishonest.

Corruption, file compromise, and offline password tests. The adversary may corrupt any party (by sending corrupt) or compromise any server and steal its file without corrupting it (by sending stealauxiliaryfile or stealtargetfile); either way, the corresponding server's file will become COMPR.

Obtaining a target server's file allows the adversary to run an offline dictionary attack on it, which is modeled by the offlinetestpwd call in which the adversary specifies a password guess pw^*. Though offline with regard to the target server, the adversary still requires the participation of $t + 1$ auxiliary servers to perform this attack (some or all of them may be compromised, in which case the adversary can also emulate their participation offline via auxsession, explained below). In each offlinetestpwd call, the adversary can specify an auxiliary server i with which it wishes to evaluate pw^*. Only once $t + 1$ auxiliary servers have participated in the evaluation of pw^* can the adversary test the password guess pw^* against the compromised target server file and learn whether or not it is correct.

Notation
Integer t is a threshold parameter.
Set $\mathsf{tx}[x] := 0$ and $\mathsf{TS}[x] := \{\}$ for all x.

Initialization Phase

1. On $(\mathsf{userinit}, sid_\mathsf{A}, sid_\mathsf{T}, \mathsf{pw})$ from $\mathsf{U} \in \mathcal{P}$:
   - send $(\mathsf{userinit}, \mathsf{U}, sid_\mathsf{A}, sid_\mathsf{T})$ to $\mathcal{A}^*$
   - save $(\mathsf{userinit}, \mathsf{U}, sid_\mathsf{A}, sid_\mathsf{T}, \mathsf{pw})$ and set $\mathsf{cflag}[sid_\mathsf{A}] :=$ UNCOMPROMISED
   - ignore future $\mathsf{userinit}$ calls for same sid_A or sid_T
2. On $(\mathsf{auxinit}, sid_\mathsf{A}, i, \mathsf{U})$ from $\mathsf{S} = \mathbf{S}_{sid_\mathsf{A}}[i]$:
   - send $(\mathsf{auxinit}, sid_\mathsf{A}, i, \mathsf{U})$ to $\mathcal{A}^*$
   - save $(\mathsf{auxinit}, sid_\mathsf{A}, i, \mathsf{U})$ marked PENDING
   - ignore future $\mathsf{auxinit}$ calls for same (sid_A, i)
3. On $(\mathsf{targetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, \mathsf{U})$ from $\mathsf{T} = \mathbf{S}_{sid_\mathsf{T}}[j]$:
   - send $(\mathsf{targetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, \mathsf{U})$ to $\mathcal{A}^*$
   - save $(\mathsf{targetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, \mathsf{U})$ marked PENDING
   - ignore future $\mathsf{targetinit}$ calls for same sid_A or sid_T and j
4. On $(\mathsf{finishauxiliaryinit}, sid_\mathsf{A}, i)$ from $\mathcal{A}^*$:
   - find $(\mathsf{userinit}, [\mathsf{U}], sid_\mathsf{A}, [sid_\mathsf{T}, \mathsf{pw}])$ (abort if missing)
   - find $(\mathsf{auxinit}, sid_\mathsf{A}, i, \mathsf{U})$ marked PENDING (abort if missing) and change its mark to COMPLETED
   - save $(\mathsf{auxiliaryfile}, sid_\mathsf{A}, i)$, and if $\mathbf{S}_{sid_\mathsf{A}}[i] \in \mathbf{Corr}$ then mark it COMPR
   - output $(\mathsf{finishauxiliaryinit}, sid_\mathsf{A})$ to $\mathsf{S} = \mathbf{S}_{sid_\mathsf{A}}[i]$
5. On $(\mathsf{finishtargetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, sid^*_\mathsf{A}, \mathsf{pw}^*)$ from $\mathcal{A}^*$:
   - find $\mathsf{rec} = (\mathsf{targetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, [\mathsf{U}])$ marked PENDING (abort if missing)
   - if $\mathsf{U} \notin \mathbf{Corr}$ then find $(\mathsf{userinit}, \mathsf{U}, sid_\mathsf{A}, sid_\mathsf{T}, [\mathsf{pw}])$ (abort if missing) and save $(\mathsf{targetfile}, sid_\mathsf{T}, j, sid_\mathsf{A}, \mathsf{pw}, \text{UNTAMPERED})$
   - otherwise (i.e. if $\mathsf{U} \in \mathbf{Corr}$) save $(\mathsf{targetfile}, sid_\mathsf{T}, j, sid^*_\mathsf{A}, \mathsf{pw}^*, \text{TAMPERED})$
   - output $(\mathsf{finishtargetinit}, sid_\mathsf{A}, sid_\mathsf{T})$ to $\mathsf{T} = \mathbf{S}_{sid_\mathsf{T}}[j]$, mark rec COMPLETED
   - if $\mathbf{S}_{sid_\mathsf{T}}[j] \in \mathbf{Corr}$ then mark the $\mathsf{targetfile}$ COMPR

Fig. 4. $\mathcal{F}_{\mathsf{atPAKE}}$: atPAKE functionality (1): Initialization Phase.

Authentication. In the authentication phase, a user U' may start an online session with a target server $\mathsf{T} = \mathbf{S}_{sid_\mathsf{T}}[j]$ using a $\mathsf{usersession}$ call (which specifies a password pw'); this call also implicitly defines the auxiliary servers by specifying sid_A. This establishes a session record for U' marked PRELIM. Similarly, a target server T may start a session with a user U' using a $\mathsf{targetsession}$ call; since the target server's password is included in its file record, it is not explicitly specified in the $\mathsf{targetsession}$ message. This establishes a session record for T marked FRESH. Next, an auxiliary server $\mathsf{S}_i = \mathbf{S}_{sid_\mathsf{A}}[i]$ may choose to participate via a $\mathsf{auxsession}$ call, which also increments its ticket count $\mathsf{tx}[sid_\mathsf{A}, i, ssid_\mathsf{A}]$. If the auxiliary server's file is COMPR, then the adversary can call $\mathsf{auxsession}$ on its behalf and thereby print tickets for the server at will. In order to progress a user session from PRELIM to FRESH, the adversary must use an $\mathsf{auxproceed}$ call to connect the user with a set $\mathbf{C}$ of $t+1$ auxiliary servers running on identifier

Party Corruption, File Compromise, Offline Password Tests

6. On $(\mathsf{corrupt}, \mathsf{P})$ from $\mathcal{A}^*$ (permitted by $\mathcal{Z}$), set $\mathbf{Corr} := \mathbf{Corr} \cup \{\mathsf{P}\}$
   - if $\exists$ $(\mathsf{auxiliaryfile}, sid_\mathsf{A}, i)$ for $\mathbf{S}_{sid_\mathsf{A}}[i] = \mathsf{P}$ mark it COMPR
   - if $\exists$ $(\mathsf{targetfile}, sid_\mathsf{T}, j, sid_\mathsf{A}, [\mathsf{pw}, \mathsf{tflag}])$ for $\mathbf{S}_{sid_\mathsf{T}}[j] = \mathsf{P}$ mark it COMPR
7. On $(\mathsf{stealauxiliaryfile}, sid_\mathsf{A}, i)$ from $\mathcal{A}^*$ (permitted by $\mathcal{Z}$):
   - if $\exists$ $(\mathsf{auxiliaryfile}, sid_\mathsf{A}, i)$ mark it COMPR
8. On $(\mathsf{stealtargetfile}, sid_\mathsf{T}, j)$ from $\mathcal{A}^*$ (permitted by $\mathcal{Z}$):
   - if $\exists$ $(\mathsf{targetfile}, sid_\mathsf{T}, j, [sid_\mathsf{A}, \mathsf{pw}, \mathsf{tflag}])$ mark it COMPR
9. On $(\mathsf{offlinetestpwd}, sid_\mathsf{A}, i, ssid_\mathsf{A}, sid_\mathsf{T}, j, \mathsf{pw}^*)$ from $\mathcal{A}^*$:
   - if $\mathsf{tx}[sid_\mathsf{A}, i, ssid_\mathsf{A}] > 0$ add i to $\mathsf{TS}[sid_\mathsf{A}, \mathsf{pw}^*, ssid_\mathsf{A}]$, set $\mathsf{tx}[sid_\mathsf{A}, i, ssid_\mathsf{A}]$−−
   - retrieve $\mathsf{rec} = (\mathsf{targetfile}, sid_\mathsf{T}, j, sid_\mathsf{A}, [\mathsf{pw}, \mathsf{tflag}])$ (abort if not found)
   - if $|\mathsf{TS}[sid_\mathsf{A}, \mathsf{pw}^*, ssid_\mathsf{A}]| \geq t{+}1$ and rec is marked COMPR then return "correct guess" to $\mathcal{A}^*$ and set $\mathsf{cflag}[sid_\mathsf{A}] :=$ COMPR if $\mathsf{pw}^* = \mathsf{pw}$, else return "wrong guess" to $\mathcal{A}^*$

Authentication Phase (I): Session Initialization, Passive Transmission

10. On $(\mathsf{usersession}, sid_\mathsf{A}, sid_\mathsf{T}, j, ssid, \mathsf{pw}')$ from $\mathsf{U}' \in \mathcal{P}$:
    - send $(\mathsf{usersession}, \mathsf{U}', sid_\mathsf{A}, sid_\mathsf{T}, j, ssid)$ to $\mathcal{A}^*$
    - save $(\mathsf{session}, \mathsf{U}', \mathbf{S}_{sid_\mathsf{T}}[j], sid_\mathsf{A}, sid_\mathsf{T}, j, ssid, \mathsf{pw}')$ marked PRELIM
    - ignore future $\mathsf{usersession}$ calls for same $ssid$
11. On $(\mathsf{auxsession}, sid_\mathsf{A}, i, ssid_\mathsf{A})$ from $\mathsf{S} = \mathbf{S}_{sid_\mathsf{A}}[i]$ or $\mathcal{A}^*$:
    - retrieve $(\mathsf{auxiliaryfile}, sid_\mathsf{A}, i)$ (abort if record not found)
    - if sender is $\mathcal{A}^*$ then abort unless the retrieved record is marked COMPR
    - send $(\mathsf{auxsession}, sid_\mathsf{A}, i, ssid_\mathsf{A})$ to $\mathcal{A}^*$
    - set $\mathsf{tx}[sid_\mathsf{A}, i, ssid_\mathsf{A}]$++
12. On $(\mathsf{targetsession}, sid_\mathsf{T}, j, \mathsf{U}', ssid)$ from $\mathsf{S} = \mathbf{S}_{sid_\mathsf{T}}[j]$:
    - retrieve $(\mathsf{targetfile}, sid_\mathsf{T}, j, [sid_\mathsf{A}, \mathsf{pw}, \mathsf{tflag}])$ (abort if record not found)
    - send $(\mathsf{targetsession}, sid_\mathsf{T}, j, \mathsf{U}', ssid)$ to $\mathcal{A}^*$
    - save $(\mathsf{session}, \mathsf{S}, \mathsf{U}', sid_\mathsf{A}, sid_\mathsf{T}, j, ssid, \mathsf{pw})$ marked FRESH
    - ignore future $\mathsf{targetsession}$ calls for same $ssid$
13. On $(\mathsf{auxproceed}, \mathsf{U}', ssid, sid^*_\mathsf{A}, ssid_\mathsf{A}, \mathbf{C})$ s.t. $|\mathbf{C}| = t + 1$ from $\mathcal{A}^*$:
    - retrieve $\mathsf{rec} = (\mathsf{session}, \mathsf{U}', \mathsf{S}, [sid_\mathsf{A}, sid_\mathsf{T}, j], ssid, [\mathsf{pw}'])$ marked PRELIM (abort if record not found)
    - reset field sid_A in record rec to sid^*_A
    - abort if $\exists_{i \in \mathbf{C}}$ $\mathsf{tx}[sid_\mathsf{A}, i, ssid_\mathsf{A}] = 0$, else $\forall_{i \in \mathbf{C}}$ set $\mathsf{tx}[sid_\mathsf{A}, i, ssid_\mathsf{A}]$−−
    - change rec's mark to FRESH
14. On $(\mathsf{testabort}, \mathsf{U}', sid_\mathsf{T}, j, ssid)$ from $\mathcal{A}^*$:
    - retrieve $\mathsf{rec} = (\mathsf{session}, \mathsf{U}', \mathbf{S}_{sid_\mathsf{T}}[j], [sid'_\mathsf{A}], sid_\mathsf{T}, j, ssid, [\mathsf{pw}'])$ marked FRESH and $(\mathsf{targetfile}, sid_\mathsf{T}, j, [sid_\mathsf{A}, \mathsf{pw}, \mathsf{tflag}])$ (abort if either not found)
    - if $(sid'_\mathsf{A}, \mathsf{pw}') = (sid_\mathsf{A}, \mathsf{pw})$ send "success" to $\mathcal{A}^*$; else mark rec COMPLETED, send "fail" to $\mathcal{A}^*$, and output $(\mathsf{abort}, ssid)$ to U'

Fig. 5. $\mathcal{F}_{\mathsf{atPAKE}}$: atPAKE functionality (2): Compromises, Authentication (I).

Authentication Phase (II): Active Attacks, Session Termination

15. On $(\mathsf{auxactive}, \mathsf{U}', ssid)$ from $\mathcal{A}^*$:
    - retrieve $(\mathsf{session}, \mathsf{U}', \mathsf{S}, [sid_\mathsf{A}, sid_\mathsf{T}, j], ssid, [\mathsf{pw}'])$ marked PRELIM and mark it COUNTERFEIT (abort if record not found)
16. On $(\mathsf{interrupt}, sid_\mathsf{T}, j, ssid)$ from $\mathcal{A}^*$:
    - retrieve $(\mathsf{session}, \mathbf{S}_{sid_\mathsf{T}}[j], [\mathsf{U}', sid_\mathsf{A}], sid_\mathsf{T}, j, ssid, [\mathsf{pw}])$ marked FRESH, mark it INTERRUPTED and set $\mathsf{dPT}[ssid] := 1$.
17. On $(\mathsf{testpwd}, \mathsf{P}, ssid, \mathsf{pw}^*)$ from $\mathcal{A}^*$:
    - retrieve $\mathsf{rec} = (\mathsf{session}, \mathsf{P}, [\mathsf{P}', sid_\mathsf{A}, sid_\mathsf{T}, j], ssid, [\mathsf{pw}])$ (abort if not found)
    - if $\mathsf{dPT}[ssid] = 1$, then set $\mathsf{dPT}[ssid] := 0$; else if rec is not marked FRESH or COUNTERFEIT, abort
    - if $\mathsf{pw}^* = \mathsf{pw}$ and any of the following conditions hold:
      (a) $\exists ssid_\mathsf{A}$ s.t. $|\mathsf{TS}[sid_\mathsf{A}, \mathsf{pw}^*, ssid_\mathsf{A}]| \geq t{+}1$
      (b) or rec marked COUNTERFEIT
      (c) or $\mathsf{P} = \mathbf{S}_{sid_\mathsf{T}}[j]$ and $\exists$ record $(\mathsf{targetfile}, sid_\mathsf{T}, j, sid_\mathsf{A}, \mathsf{pw}, \text{TAMPERED})$
      then mark rec as COMPR, send "correct guess" to $\mathcal{A}^*$, and (if $\mathsf{P} = \mathbf{S}_{sid_\mathsf{T}}[j]$) set $\mathsf{cflag}[sid_\mathsf{A}] := \text{COMPR}$; else mark rec as INTERRUPTED and send "wrong guess" to $\mathcal{A}^*$
18. On $(\mathsf{impersonate}, sid_\mathsf{T}, j, ssid)$ from $\mathcal{A}^*$:
    - retrieve $\mathsf{rec} = (\mathsf{session}, \mathsf{U}', \mathbf{S}_{sid_\mathsf{T}}[j], [sid_\mathsf{A}], sid_\mathsf{T}, j, ssid, [\mathsf{pw}])$ marked FRESH (abort if not found)
    - if $\exists$ record $(\mathsf{targetfile}, sid_\mathsf{T}, j, sid_\mathsf{A}, \mathsf{pw}, [\mathsf{tflag}'])$ marked COMPR then mark rec as COMPR and send "correct guess" to $\mathcal{A}^*$; else mark rec as INTERRUPTED and send "wrong guess" to $\mathcal{A}^*$
19. On $(\mathsf{newkey}, \mathsf{P}, ssid, K^*)$ from $\mathcal{A}^*$:
    - retrieve $\mathsf{rec} = (\mathsf{session}, \mathsf{P}, [\mathsf{P}', sid_\mathsf{A}, sid_\mathsf{T}, j], ssid, [\mathsf{pw}])$ not marked PRELIM or COMPLETED (abort if record not found) and do:
      - if rec is marked COMPR, then set $K \leftarrow K^*$
      - if $\mathsf{P} = \mathbf{S}_{sid_\mathsf{T}}[j]$, rec is marked INTERRUPTED, and ($\mathsf{cflag}[sid_\mathsf{A}] = \text{COMPR}$ or $\exists$ record $(\mathsf{targetfile}, sid_\mathsf{T}, j, sid_\mathsf{A}, \mathsf{pw}, \text{TAMPERED})$), then set $K \leftarrow K^*$
      - if rec is FRESH and $\exists\ \mathsf{rec}' = (\mathsf{session}, \mathsf{P}', \mathsf{P}, sid_\mathsf{A}, sid_\mathsf{T}, j, ssid, \mathsf{pw})$ s.t. P' received $(\mathsf{newkey}, ssid, K')$ when rec' was FRESH, then set $K \leftarrow K'$
      - else pick $K \leftarrow_\$ \{0,1\}^\tau$
    - finally, mark rec as COMPLETED, output $(\mathsf{newkey}, ssid, K)$ to P

Fig. 6. $\mathcal{F}_{\mathsf{atPAKE}}$: atPAKE functionality (3): Authentication (II).

sid_{A}. Those auxiliary servers must have all agreed to participate in an evaluation, which is tracked via the ticket mechanism.

Once both U' and T sessions are FRESH, the adversary lets a party output a session key using the newkey call. If both sides use the same pw and the same subsession identifier *ssid*, then they will receive the same key. Otherwise, they will output independently random keys for the session. Figure 7 diagrams the state transitions that user and server session records can move through (including those corresponding to attack scenarios).

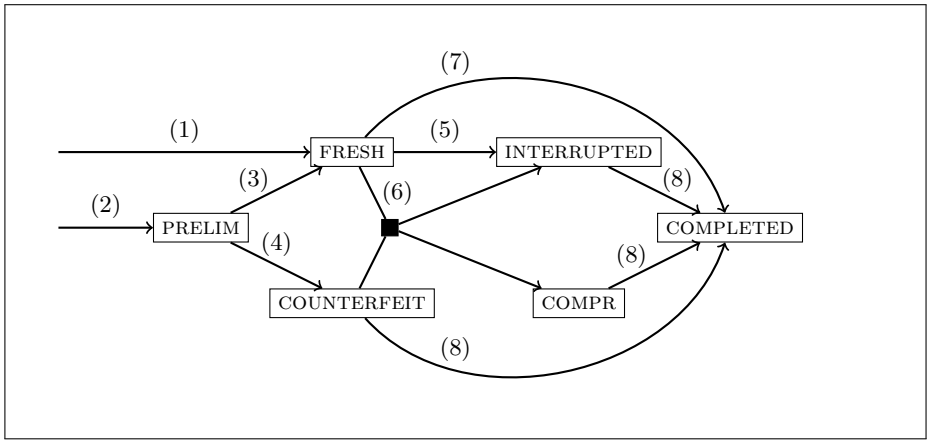

Fig. 7. $\mathcal{F}_{\mathsf{atPAKE}}$ session record state diagram. (1) is targetsession; (2) is usersession; (3) is auxproceed; (4) is auxactive; (5) is interrupt (only for server sessions); (6) is impersonate (only for FRESH user sessions) or testpwd; (7) is testabort (only for user sessions) or newkey; (8) is newkey.

Passive attacks. The adversary has two "passive" attack avenues that correspond to simply transmitting messages between parties. With auxproceed, the adversary might choose to connect the user with a different auxiliary session sid^*_{A} than expected; in that case the user will not be able to successfully authenticate to the intended target server (unless that server was dishonestly initialized with the same spurious sid^*_{A}). With the testabort call, the adversary can connect a FRESH user session to any target server and observe whether or not they successfully authenticate.

Active attacks. For active session attacks, if the user U''s session record is PRELIM (i.e. it has not completed its communication with the auxiliary servers), the adversary can run the auxiliary servers' algorithms on its own and communicate with U'. In this situation, modeled by the auxactive call, the adversary effectively controls all auxiliary servers that communicate with U', so we mark U''s session record COUNTERFEIT. Such a session can never successfully authenticate to a target server, but it is vulnerable to an online password-guessing attack.

Using testpwd, the adversary can perform an active session attack with some password guess pw^*. There are three possible cases:

(a) the adversary has evaluated pw^* with $t+1$ auxiliary servers (using offlinetestpwd);
(b) the adversary is attacking a user session that is COUNTERFEIT;
(c) the adversary is attacking a server that was dishonestly initialized (i.e. its file is TAMPERED).

If the password guess is successful, then the attacked session is marked COMPR. Otherwise, it is marked INTERRUPTED, indicating that it is no longer possible for the session to successfully complete. In either case, the adversary learns whether or not the password guess was correct. After stealing a target server's file, the adversary can also compromise user sessions meant to connect with that server via the impersonate call. If the user's password does not match the one in the saved file, then this impersonate attack will fail and the user's session will become INTERRUPTED.

Once a session is COMPR, the adversary has done a successful attack, so the adversary is able to choose that session's output key in newkey. If both the session and its countersession are FRESH, this models an unattacked pair, so the two parties output the same random key (if their passwords and sid_{AS}s match). In all other cases (i.e. COUNTERFEIT and INTERRUPTED), the functionality samples an independent random key for the session.

Comparison to game-based tPAKE. As a sanity check, we verify that $\mathcal{F}_{\mathsf{atPAKE}}$ is at least as strong as the game-based tPAKE definition of MacKenzie, Shrimpton, and Jakobsson [53]. Appendix A contains a proof that any protocol realizing $\mathcal{F}_{\mathsf{atPAKE}}$ (in the case that the auxiliary and target server lists are identical) is also secure under that notion.

5 Augmented Threshold PAKE Construction

Figure 8 shows protocol $\Pi_{\mathsf{tOPRF-atPAKE}}$, our main atPAKE construction which is a generic composition of UC tOPRF and UC (strong) aPAKE. In Figure 8 we show this protocol in the hybrid model assuming functionalities $\mathcal{F}_{\mathsf{tOPRF}}$ and $\mathcal{F}_{\mathsf{saPAKE}}$ which model respectively UC tOPRF and UC (strong) aPAKE (they are shown respectively in Figures 2 and 19), but in an implementation these functionalities will be replaced by sub-protocols that realize them. Protocol $\Pi_{\mathsf{tOPRF-atPAKE}}$ also uses secure channels modeled by functionality $\mathcal{F}_{\mathsf{channel}}$ (shown in Figure 1), but it uses them only in the initialization. Note that our realization of the threshold OPRF functionality $\mathcal{F}_{\mathsf{tOPRF}}$, i.e. protocol 3HashTDH of Section 3.2, also relies on secure channels in the initialization.

In protocol $\Pi_{\mathsf{tOPRF-atPAKE}}$, authentication between a user and a target server T, indexed as the j-th server in the target server list $\mathbf{S}_{sid_{\mathsf{T}}}$, proceeds in two steps. First, the user interacts with $t+1$ tOPRF servers, i.e. the auxiliary servers, in order to convert their password guess pw' into a "hardened"

Initialization

1. On input $(\mathsf{atpake.userinit}, sid_\mathsf{A}, sid_\mathsf{T}, \mathsf{pw})$, user U does:
   - send $(\mathsf{toprf.init}, sid_\mathsf{A}, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{tOPRF}}$
   - await response $(\mathsf{toprf.initeval}, sid_\mathsf{A}, \mathsf{rw})$
   - for every $j \in \{1, \ldots, |\mathbf{S}_{sid_\mathsf{T}}|\}$, compute $\mathsf{rw}_j : \mathsf{KDF}(\mathsf{rw}, \mathbf{S}_{sid_\mathsf{T}}[j])$ and send $(\mathsf{channel.send}, (sid_\mathsf{A}||sid_\mathsf{T}||j), \mathbf{S}_{sid_\mathsf{T}}[j], \mathsf{rw}_j)$ to $\mathcal{F}_{\mathsf{channel}}$
2. On input $(\mathsf{atpake.auxinit}, sid_\mathsf{A}, i, \mathsf{U}')$, auxiliary server $\mathbf{S}_{sid_\mathsf{A}}[i]$ does:
   - send $(\mathsf{toprf.sinit}, sid_\mathsf{A}, i, \mathsf{U}')$ to $\mathcal{F}_{\mathsf{tOPRF}}$
   - await response $(\mathsf{toprf.fininit}, sid_\mathsf{A}, i)$
   - then output $(\mathsf{atpake.finishauxiliaryinit}, sid_\mathsf{A})$
3. On input $(\mathsf{atpake.targetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, \mathsf{U}')$, target server $\mathbf{S}_{sid_\mathsf{T}}[j]$ does:
   - await $(\mathsf{channel.deliver}, (sid_\mathsf{A}||sid_\mathsf{T}||j), \mathsf{U}', \mathsf{rw}_j)$ from $\mathcal{F}_{\mathsf{channel}}$
   - send $(\mathsf{sapake.storepwdfile}, (sid_\mathsf{T}||j), \mathsf{U}', \mathsf{rw}_j)$ to $\mathcal{F}_{\mathsf{saPAKE}}$
   - output $(\mathsf{atpake.finishtargetinit}, sid_\mathsf{A}, sid_\mathsf{T})$

Authentication

4. On input $(\mathsf{atpake.usersession}, sid_\mathsf{A}, sid_\mathsf{T}, j, ssid, \mathsf{pw}')$, user U' does:
   - send $(\mathsf{toprf.eval}, sid_\mathsf{A}, ssid, \mathsf{pw}')$ to $\mathcal{F}_{\mathsf{tOPRF}}$
   - await response $(\mathsf{toprf.eval}, sid_\mathsf{A}, ssid, \mathsf{rw}')$
   - compute $\mathsf{rw}'_j := \mathsf{KDF}(\mathsf{rw}', \mathbf{S}_{sid_\mathsf{T}}[j])$
   - send $(\mathsf{sapake.usrsession}, (sid_\mathsf{T}||j), ssid, \mathbf{S}_{sid_\mathsf{T}}[j], \mathsf{rw}'_j)$ to $\mathcal{F}_{\mathsf{saPAKE}}$
   - upon response $(\mathsf{sapake.newkey}, (sid_\mathsf{T}||j), ssid, K)$, output $(\mathsf{atpake.newkey}, ssid, K)$
   - upon response $(\mathsf{sapake.abort}, (sid_\mathsf{T}||j), ssid)$, output $(\mathsf{atpake.abort}, ssid)$
5. On input $(\mathsf{atpake.auxsession}, sid_\mathsf{A}, i, ssid_\mathsf{A})$, auxiliary server $\mathbf{S}_{sid_\mathsf{A}}[i]$ sends $(\mathsf{toprf.sndrcomplete}, sid_\mathsf{A}, i, ssid_\mathsf{A})$ to $\mathcal{F}_{\mathsf{tOPRF}}$
6. On input $(\mathsf{atpake.targetsession}, sid_\mathsf{T}, j, \mathsf{U}', ssid)$, target server $\mathbf{S}_{sid_\mathsf{T}}[j]$ does:
   - send $(\mathsf{sapake.svrsession}, (sid_\mathsf{T}||j), ssid)$ to $\mathcal{F}_{\mathsf{saPAKE}}$
   - await response $(\mathsf{atpake.newkey}, (sid_\mathsf{T}||j), ssid, K)$
   - output $(\mathsf{atpake.newkey}, ssid, K)$

Fig. 8. Protocol $\Pi_{\mathsf{tOPRF-atPAKE}}$ which realizes $\mathcal{F}_{\mathsf{atPAKE}}$ using $\mathcal{F}_{\mathsf{tOPRF}}$ and $\mathcal{F}_{\mathsf{saPAKE}}$

password $\mathsf{rw}' = F_k(\mathsf{pw}')$. Then, the user uses a pseudorandom function to derive a T-specific password $\mathsf{rw}'_j = \mathsf{KDF}(\mathsf{rw}', \mathsf{T})$, and uses rw'_j as the password in an underlying (strong) aPAKE instance between the user and the target server.

When registering a new user with password pw, the client must initialize a new tOPRF instance $F_k(\cdot)$ with the auxiliary servers. At the same time the client computes $\mathsf{rw}_j = \mathsf{KDF}(F_k(\mathsf{pw}), \mathsf{T}_j)$ for every target server $\mathsf{T}_j = \mathbf{S}_{sid_\mathsf{T}}[j]$, and sends rw_j to T_j over a secure channel. Upon receiving rw_j, server T_j uses it to create a password file for this user.

Theorem 3. *Protocol $\Pi_{\mathsf{tOPRF-atPAKE}}$ realizes functionality $\mathcal{F}_{\mathsf{atPAKE}}$ with parameters t and n in the $(\mathcal{F}_{\mathsf{tOPRF}}, \mathcal{F}_{\mathsf{saPAKE}}, \mathcal{F}_{\mathsf{channel}})$-hybrid model.*

Specifically, for any efficient adversary $\mathcal{A}$ against protocol $\Pi_{\mathsf{tOPRF-atPAKE}}$, there exists a simulator SIM such that no efficient environment $\mathcal{Z}$ can distinguish the view of $\mathcal{A}$ interacting with the real $\Pi_{\mathsf{tOPRF-atPAKE}}$ protocol and the view of SIM interacting with the ideal functionality $\mathcal{F}_{\mathsf{atPAKE}}$ with advantage better than $(q_{\mathsf{T}} \cdot q_{\mathsf{eval}}^2 + q_{\mathsf{test}} + q_{\mathsf{T}}^* \cdot q_{\mathsf{eval}})/2^\tau$ where q_{T} is the number of target server instances, q_{eval} is the number of tOPRF evaluations, q_{test} is the number of online and offline password-guessing attacks against $\mathcal{F}_{\mathsf{saPAKE}}$, q_{T}^* is the number of dishonestly initialized target server instances, and security parameter τ is the tOPRF output length.

Proof of Theorem 3 is deferred to Appendix C.

Concrete Instantiation. In Figure 9 we show a concrete instantiation of protocol $\Pi_{\mathsf{tOPRF-atPAKE}}$, with $\mathcal{F}_{\mathsf{tOPRF}}$ realized with the 3HashTDH protocol of Section 3.2. The user and the target server interact via an arbitrary (strong) aPAKE, which can be realized with any realization of $\mathcal{F}_{\mathsf{saPAKE}}$, including OPAQUE [44], OPAQUE' [44, 35, 58], TLS-OPAQUE [37, 62], or other aPAKE constructions [14, 54]. As discussed in Section 6, the (strong) aPAKE can be replaced with weaker building blocks, including weak aPAKE [32], the "envelope+AKE" building block used within OPAQUE, or even password-over-TLS, although the resulting protocol could realize modified (and sometimes weakened) versions of the atPAKE functionality $\mathcal{F}_{\mathsf{atPAKE}}$.

6 Protocol Variants and Extensions

We considered several possible variants of the tOPRF+aPAKE construction of atPAKE shown in Section 5. Indeed, both the UC tOPRF subprotocol and the (strong) aPAKE protocol can be substituted by other building blocks, and the resulting protocols implement variants of the atPAKE functionality $\mathcal{F}_{\mathsf{atPAKE}}$. We summarize the security properties of the protocol variants we considered in Table 1 below. Table entries marked with a special symbol $^{(*)}$ are verified formally in this paper. All the other table entries are not formally verified, but these are our hypotheses based on extrapolating the formally verified cases.

Implementing atPAKE with variants of tOPRF. The columns of the table include three variants of the threshold OPRF (tOPRF) protocol, namely tOPRF itself, the threshold *Partial* OPRF (tPOPRF) (see Section 3.4), and the *augmented* Password-Protected Secret-Sharing (aPPSS) [27], an extension of PPSS [6] to security up to offline dictionary attack upon compromise of all servers. Recall that PPSS is a protocol which secret-shares a secret s among n servers and protects it under password pw, s.t. no t parties can learn any information on s, and $t+1$ parties suffice to reconstruct s but only if the reconstruction protocol client enters the same password pw which was used to initialize this secret-sharing.[10]

[10] As shown in [41], tOPRF implies PPSS: One simply uses an authenticated encryption to encrypt secret s under $\mathsf{rw} = F_k(\mathsf{pw})$, where F_k is a tOPRF implemented by the

Notation As in Section 2, τ is a security parameter, $\mathsf{KDF} : \{0,1\}^\tau \times \{0,1\}^* \to \{0,1\}^\tau$ is a PRF, $\mathbb{G}$ is a group of prime order m. H_1, H_2, H_3 are hash functions with ranges $\mathbb{G}$, $\mathbb{G}$, and $\{0,1\}^l$ where l is a parameter. SC denotes communication over a secure and authenticated channel. "aPAKE" denotes arbitrary (strong) aPAKE. (In other protocol variants aPAKE can be replaced with TLS-OPAQUE, weak aPAKE, envelope+AKE, or password-over-TLS.)

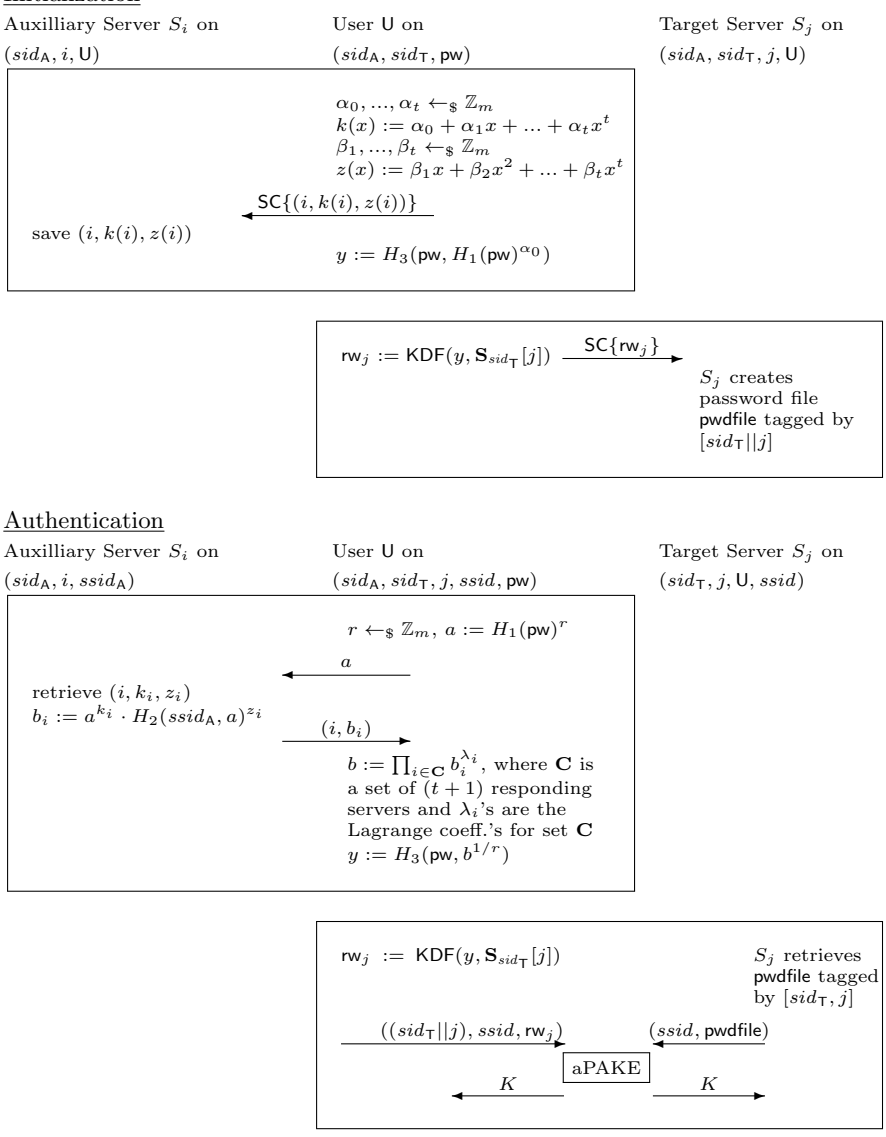

Fig. 9. Concrete instantiation of $\Pi_{\mathsf{tOPRF-atPAKE}}$ using arbitrary (strong) aPAKE

These three protocol variants have the following security characteristics: Replacing tOPRF with tPOPRF creates a very mild and essentially negligible difference in the atPAKE security model, denoted $\mathcal{F}_{\mathsf{atPAKE}'}$. However, replacing t(P)OPRF with aPPSS changes the resulting atPAKE notion to a variant of UC atPAKE notion we denote $\mathcal{F}_{\mathsf{atPAKE(weak)}}$, which is weaker than $\mathcal{F}_{\mathsf{atPAKE}}$ in the following sense: In the latter model an offline dictionary attack (ODA) is enabled only if the adversary corrupts $t+1$ of n auxiliary servers $\mathsf{S}_1, ..., \mathsf{S}_n$ *and* the target server T, whereas in $\mathcal{F}_{\mathsf{atPAKE(weak)}}$ the ODA requires corruption of $t+1$ auxiliary servers, but it can be done without corrupting a target server. However, note that one can implement the security contract of $\mathcal{F}_{\mathsf{atPAKE}}$ using $\mathcal{F}_{\mathsf{atPAKE(weak)}}$ if the atPAKE scheme involves a single target server: If the target server T is part of the auxiliary server group, and it holds a "blocking" set of shares, then corruption of a sufficient number of (virtual) auxiliary servers becomes possible only if one corrupts $t+1$ (real) auxiliary server *and* the target server T.

Table 1. Summary of security properties of atPAKE implementation variants

U-to-T subprotocol	tOPRF	tPOPRF	aPPSS
(strong) aPAKE	$\mathcal{F}_{\mathsf{atPAKE}}{}^{(*)}$	$\mathcal{F}_{\mathsf{atPAKE}'}{}^{(*)}$	$\mathcal{F}_{\mathsf{atPAKE(weak)}}{}^{(*)}$
TLS-OPAQUE	$\mathcal{F}_{\mathsf{atPAKE-EA}}$	$\mathcal{F}_{\mathsf{atPAKE}'-\mathsf{EA}}$	$\mathcal{F}_{\mathsf{atPAKE(weak)-EA}}$
password-over-TLS	$\mathcal{F}_{\mathsf{atPAKE-PKI}}$	$\mathcal{F}_{\mathsf{atPAKE}'-\mathsf{PKI}}$	$\mathcal{F}_{\mathsf{atPAKE(weak)-PKI}}$
weak aPAKE	$\mathcal{F}_{\mathsf{atPAKE(medium)}}{}^{(*)}$	$\mathcal{F}_{\mathsf{atPAKE}'\mathsf{(medium)}}$	$\mathcal{F}_{\mathsf{atPAKE(weak)}}$
AKE	N/A	N/A	$\mathcal{F}_{\mathsf{atPAKE(weak)}}$

Implementing atPAKE with variants of (strong) aPAKE. Table rows include five variants of the U-to-T authentication protocol, i.e. the way user U uses rw retrieved using its password to authenticate to the target server T. In the protocol analyzed in Section 5 this is handled by (strong) aPAKE. The same holds for the variants, where tOPRF is replaced by tPOPRF and aPPSS. However, one can consider replacing the (strong) aPAKE sub-protocol with TLS-OPAQUE, i.e. the *Exported Authenticator* (EA) extension of TLS which implements augmented password authentication over existing TLS connection, rather than creating new password-authenticated session key [37, 62]. The mechanics of these "EA" extensions of our $\mathcal{F}_{\mathsf{atPAKE}}$ variants will be similar as in [37], and their security properties should be the same as in $\mathcal{F}_{\mathsf{atPAKE}}/\mathcal{F}_{\mathsf{atPAKE}'}/\mathcal{F}_{\mathsf{atPAKE(weak)}}$ except the final U-T authentication would pertain to a pre-established secure channel between these two parties.

Another possible variant considers U-T interaction implemented as "password-over-TLS". Such "PKI" extensions of our $\mathcal{F}_{\mathsf{atPAKE}}$ variants would have weaker security: First, U's login sessions need to include T's identity as

servers. We believe that this implements UC *augmented* PPSS (aPPSS) [27], but we leave formal verification of this to future work.

input. Second, if this identity is wrong, i.e. U fails to authenticate the proper T counterparty, or if it is right but T is compromised, then (1) the adversary learns rw_T and can authenticate to T as U, and (2) the adversary can stage ODA if $t+1$ auxiliary servers are corrupted. (In other words, PKI error in U authenticating T has the same consequences regarding ODA attack as corruption of T.)

The final two possibilities include replacing (strong) aPAKE with weak aPAKE of [32] or with pfs-AKE, i.e. AKE with perfect forward secrecy, usually achieved by key confirmation flows. Using the first option allows for atPAKE with lower round complexity (only 3 protocol flows) if weak aPAKE is implemented with a 2-flow protocol of [31]. However, the resulting functionality is slightly weaker in the case of $\mathcal{F}_{\mathsf{atPAKE}}$ and $\mathcal{F}_{\mathsf{atPAKE}'}$, denoted resp. $\mathcal{F}_{\mathsf{atPAKE(medium)}}$ and $\mathcal{F}_{\mathsf{atPAKE}'\mathsf{(medium)}}$. The weakening is that after compromise of $t+1$ auxiliary servers the attacker can precompute the ODA before the compromise of a target server. This forms a mid-point between $\mathcal{F}_{\mathsf{atPAKE}}$, where ODA can start only at compromise of $t+1$ auxiliary servers and a target server, and $\mathcal{F}_{\mathsf{atPAKE(weak)}}$ where it can start at compromise of just the auxiliary servers (see Appendix E). Using pfs-AKE can further lower the U-T subprotocol cost, but it can be done only with aPPSS, because U needs to reconstruct structured data, not a pseudorandom string, to run AKE, namely its own private key and server T's public key. Using either weak aPAKE or pfs-AKE option the aPPSS-based protocol should achieve the same atPAKE notion variant $\mathcal{F}_{\mathsf{atPAKE(weak)}}$, but we recommend that the last option, i.e. aPPSS and pfs-AKE, should be formally verified before anyone implements it.

Proactive Security. In our tOPRF and tPOPRF protocols, the only state held by each server is two Shamir secret shares, one of the PRF key and one of zero. Therefore, these protocols can be naturally extended to support proactive key refresh by using standard techniques for proactively refreshing a secret sharing. For example, [36] presents a scheme where all share-holders broadcast verifiable secret sharings of zero, which are then added to the secret sharing to be refreshed. After carrying out this refresh procedure, the share-holders hold a fresh sharing of their original (unchanged) secret value. If the servers in our t(P)OPRF were to adopt this behavior, it would effectively partition their evaluation tickets into "epochs" separated by key-refresh events. To successfully evaluate, one would not only need the participation of $t+1$ servers, but the participation of $t+1$ servers *within the time of one epoch*. Similarly, the adversary would gain nothing by stealing server files across multiple epochs; only by stealing $t+1$ servers' data within the time of one epoch could the adversary unlock the power to perform entirely offline evaluations.

Our atPAKE constructions that are based on t(P)OPRF inherit these same proactive security properties. We note that the tOPRF-based atPAKE protocol requires auxiliary servers to hold a separate tOPRF instance for each user, whereas the tPOPRF-based construction reuses a single tPOPRF instance across all users. Since it would likely be impractical for servers to proactively refresh a separate key sharing for every registered user, the tPOPRF-based

atPAKE has an additional benefit: Besides reducing each server's storage load, it would also make proactive key refresh much more practical.

References

1. Facebook stored hundreds of millions of passwords in plain text, `https://www.theverge.com/2019/3/21/18275837/facebook-plain-text-password-storage-hundreds-millions-users`. 2019.
2. Google stored some passwords in plain text for fourteen years, `https://www.theverge.com/2019/5/21/18634842/google-passwords-plain-text-g-suite-fourteen-years`. 2019.
3. 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, Berlin, Heidelberg, December 2005.
4. 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, Berlin, Heidelberg, January 2005.
5. Martin R. Albrecht, Alex Davidson, Amit Deo, and Nigel P. Smart. Round-optimal verifiable oblivious pseudorandom functions from ideal lattices. In Juan Garay, editor, *PKC 2021, Part II*, volume 12711 of *LNCS*, pages 261–289. Springer, Cham, May 2021.
6. Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011*, pages 433–444. ACM Press, October 2011.
7. Feng Bao, Robert H. Deng, and HuaFei Zhu. Variations of diffie-hellman problem. In *Information and Communications Security*, pages 301–312, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
8. Carsten Baum, Tore Kasper Frederiksen, Julia Hesse, Anja Lehmann, and Avishay Yanai. Pesto: Proactively secure distributed single sign-on, or how to trust a hacked server. In *Proceedings - 5th IEEE European Symposium on Security and Privacy, Euro S and P 2020*, pages 587–606. IEEE, 2020. 2020 IEEE European Symposium on Security and Privacy (EuroS&P) ; Conference date: 07-09-2020 Through 11-09-2020.
9. 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, Berlin, Heidelberg, May 2000.
10. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.
11. Olivier Blazy, Céline Chevalier, and Damien Vergnaud. Mitigating server breaches in password-based authentication: Secure and efficient solutions. In Kazue Sako, editor, *CT-RSA 2016*, volume 9610 of *LNCS*, pages 3–18. Springer, Cham, February / March 2016.
12. Dan Boneh. The decision diffie-hellman problem. Stanford Cryptography Group webpage, 1998. https://crypto.stanford.edu/ dabo/pubs/papers/DDH.pdf.

13. D. Bourdrez, H. Krawczyk, K. Lewi, and C. Wood. The OPAQUE Asymmetric PAKE Protocol, draft-irtf-cfrg-opaque, `https://tools.ietf.org/id/draft-irtf-cfrg-opaque`, July 2022.
14. Tatiana Bradley, Stanislaw Jarecki, and Jiayu Xu. Strong asymmetric PAKE based on trapdoor CKEM. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 798–825. Springer, Cham, August 2019.
15. John G. Brainard, Ari Juels, Burt Kaliski, and Michael Szydlo. A new two-server approach for authentication with short secrets. In *USENIX Security 2003*. USENIX Association, August 2003.
16. 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, Berlin, Heidelberg, March / April 2015.
17. Jan Camenisch, Anja Lehmann, Anna Lysyanskaya, and Gregory Neven. Memento: How to reconstruct your secrets from a single password in a hostile environment. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 256–275. Springer, Berlin, Heidelberg, August 2014.
18. Jan Camenisch, Anja Lehmann, and Gregory Neven. Optimal distributed password verification. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 182–194. ACM Press, October 2015.
19. 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 2012*, pages 525–536. ACM Press, October 2012.
20. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.
21. Ran Canetti and Shafi Goldwasser. An efficient threshold public key cryptosystem secure against adaptive chosen ciphertext attack. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 90–106. Springer, Berlin, Heidelberg, May 1999.
22. 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, Berlin, Heidelberg, May 2005.
23. Sílvia Casacuberta, Julia Hesse, and Anja Lehmann. Sok: Oblivious pseudorandom functions. Cryptology ePrint Archive, Paper 2022/302, 2022. `https://eprint.iacr.org/2022/302`.
24. Poulami Das, Julia Hesse, and Anja Lehmann. Dpase: Distributed password-authenticated symmetric encryption. Cryptology ePrint Archive, Paper 2020/1443, 2020. `https://eprint.iacr.org/2020/1443`.
25. Sourav Das and Ling Ren. Adaptively secure BLS threshold signatures from DDH and co-CDH. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part VII*, volume 14926 of *LNCS*, pages 251–284. Springer, Cham, August 2024.
26. 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, Berlin, Heidelberg, May 2003.
27. Stefan Dziembowski, Stanislaw Jarecki, Pawel Kedzior, Hugo Krawczyk, Nam Ngo, and Jiayu Xu. Password-protected threshold signatures. IACR Cryptology ePrint Archive, 2024.

28. Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The pythia PRF service. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 547–562. USENIX Association, August 2015.
29. Warwick Ford and Burton S. Kaliski Jr. Server-assisted generation of a strong secret from a password. In *9th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2000)*, pages 176–180, Gaithersburg, MD, USA, June 4–16, 2000. IEEE Computer Society.
30. 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, Berlin, Heidelberg, February 2005.
31. Bruno Freitas Dos Santos, Yanqi Gu, Stanislaw Jarecki, and Hugo Krawczyk. Asymmetric pake with low computation and communication. In *EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2022.
32. Craig Gentry, Philip D. Mackenzie, and Zulfikar Ramzan. Password authenticated key exchange using hidden smooth subgroups. In Vijayalakshmi Atluri, Catherine Meadows, and Ari Juels, editors, *ACM CCS 2005*, pages 299–309. ACM Press, November 2005.
33. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.
34. Yanqi Gu, Stanislaw Jarecki, Pawel Kedzior, Phillip Nazarian, and Jiayu Xu. Threshold pake with security against compromise of all servers. In *Advances in Cryptology – ASIACRYPT 2024*, 2024.
35. Yanqi Gu, Stanislaw Jarecki, and Hugo Krawczyk. KHAPE: Asymmetric PAKE from key-hiding key exchange. In *Advances in Cryptology - Crypto 2021*, pages 701–730, 2021. `https://ia.cr/2021/873`.
36. Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In Don Coppersmith, editor, *CRYPTO'95*, volume 963 of *LNCS*, pages 339–352. Springer, Berlin, Heidelberg, August 1995.
37. Julia Hesse, Stanislaw Jarecki, Hugo Krawczyk, and Christopher Wood. Password-authenticated TLS via OPAQUE and post-handshake authentication. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 98–127. Springer, Cham, April 2023.
38. David P. Jablon. Password authentication using multiple servers. In David Naccache, editor, *CT-RSA 2001*, volume 2020 of *LNCS*, pages 344–360. Springer, Berlin, Heidelberg, April 2001.
39. Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 233–253. Springer, Berlin, Heidelberg, December 2014.
40. 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*, pages 276–291. IEEE, 2016.
41. Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17International Conference on Applied Cryptography and Network Security*, volume 10355 of *LNCS*, pages 39–58. Springer, Cham, July 2017.

42. Stanislaw Jarecki, Hugo Krawczyk, and Jason Resch. Threshold partially-oblivious PRFs with applications to key management. Cryptology ePrint Archive, Report 2018/733, 2018.
43. Stanislaw Jarecki, Hugo Krawczyk, Maliheh Shirvanian, and Nitesh Saxena. Device-enhanced password protocols with optimal online-offline protection. In Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang, editors, *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 177–188. ACM, 2016.
44. Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 456–486. Springer, Cham, April / May 2018.
45. Stanislaw Jarecki and Xiaomin Liu. Affiliation-hiding envelope and authentication schemes with efficient support for multiple credentials. In *Automata, Languages and Programming*, pages 715–726, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
46. Stanislaw Jarecki and Xiaomin Liu. Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 577–594. Springer, Berlin, Heidelberg, March 2009.
47. Haimin Jin, Duncan S. Wong, and Yinlong Xu. An efficient password-only two-server authenticated key exchange system. In Sihan Qing, Hideki Imai, and Guilin Wang, editors, *ICICS 07*, volume 4861 of *LNCS*, pages 44–56. Springer, Berlin, Heidelberg, December 2008.
48. Jonathan Katz, Philip D. MacKenzie, Gelareh Taban, and Virgil D. Gligor. Two-server password-only authenticated key exchange. In John Ioannidis, Angelos Keromytis, and Moti Yung, editors, *ACNS 05International Conference on Applied Cryptography and Network Security*, volume 3531 of *LNCS*, pages 1–16. Springer, Berlin, Heidelberg, June 2005.
49. Franziskus Kiefer and Mark Manulis. Distributed smooth projective hashing and its application to two-server password authenticated key exchange. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *ACNS 14International Conference on Applied Cryptography and Network Security*, volume 8479 of *LNCS*, pages 199–216. Springer, Cham, June 2014.
50. Franziskus Kiefer and Mark Manulis. Universally composable two-server PAKE. In Matt Bishop and Anderson C. A. Nascimento, editors, *ISC 2016*, volume 9866 of *LNCS*, pages 147–166. Springer, Cham, September 2016.
51. Russell W. F. Lai, Christoph Egger, Dominique Schröder, and Sherman S. M. Chow. Phoenix: Rebirth of a cryptographic password-hardening service. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017*, pages 899–916. USENIX Association, August 2017.
52. Leona Lassak, Annika Hildebrandt, Maximilian Golla, and Blase Ur. "it's stored, hopefully, on an encrypted server": Mitigating users' misconceptions about fido2 biometric webauthn. In *Proc. USENIX Security*, 2021.
53. 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, Berlin, Heidelberg, August 2002.
54. Ian McQuoid and Jiayu Xu. An efficient strong asymmetric PAKE compiler instantiable from group actions. In *Advances in Cryptology - ASIACRYPT 2023 - 29th International Conference on the Theory and Application of Cryptology and*

Information Security, Guangzhou, China, December 4-8, 2023, Proceedings, Part VIII, volume 14445 of *Lecture Notes in Computer Science*, pages 176–207. Springer, 2023.

55. Kentrell Owens, Olabode Anise, Amanda Krauss, and Blase Ur. User perceptions of the usability and security of smartphones as fido2 roaming authenticators. In *SOUPS*, pages 57–76, 2021.
56. Hirak Ray, Flynn Wolf, Ravi Kuber, and Adam J Aviv. Why older adults (don't) use password managers. In *USENIX*, 2021.
57. Lawrence Roy and Jiayu Xu. A universally composable PAKE with zero communication cost - (and why it shouldn't be considered UC-secure). In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *PKC 2023, Part I*, volume 13940 of *LNCS*, pages 714–743. Springer, Cham, May 2023.
58. Bruno Freitas Dos Santos, Yanqi Gu, Stanislaw Jarecki, and Hugo Krawczyk. Asymmetric PAKE with low computation and communication. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 - June 3, 2022, Proceedings, Part II*, volume 13276 of *Lecture Notes in Computer Science*, pages 127–156. Springer, 2022.
59. Jonas Schneider, Nils Fleischhacker, Dominique Schröder, and Michael Backes. Efficient cryptographic password hardening services from partially oblivious commitments. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1192–1203. ACM Press, October 2016.
60. Maliheh Shirvanian, Stanislaw Jarecki, Hugo Krawczyk, and Nitesh Saxena. SPHINX: A password store that perfectly hides passwords from itself. In Kisung Lee and Ling Liu, editors, *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, pages 1094–1104. IEEE Computer Society, 2017.
61. Maliheh Shirvanian, Christopher Robert Price, Mohammed Jubur, Nitesh Saxena, Stanislaw Jarecki, and Hugo Krawczyk. A hidden-password online password manager. In Chih-Cheng Hung, Jiman Hong, Alessio Bechini, and Eunjee Song, editors, *SAC '21: The 36th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, Republic of Korea, March 22-26, 2021*, pages 1683–1686. ACM, 2021.
62. N. Sullivan, H. Krawczyk, O. Friel, and R. Barnes. OPAQUE with TLS 1.3, draft-sullivan-tls-opaque-01, `https://datatracker.ietf.org/doc/html/draft-sullivan-tls-opaque`, February 2021.
63. Michael Szydlo and Burton S. Kaliski Jr. Proofs for two-server password authentication. In Alfred Menezes, editor, *CT-RSA 2005*, volume 3376 of *LNCS*, pages 227–244. Springer, Berlin, Heidelberg, February 2005.
64. Nirvan Tyagi, Sofía Celi, Thomas Ristenpart, Nick Sullivan, Stefano Tessaro, and Christopher A. Wood. A fast and simple partially oblivious prf, with applications. In *Advances in Cryptology – EUROCRYPT 2022: 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 – June 3, 2022, Proceedings, Part II*, page 674–705, Berlin, Heidelberg, 2022. Springer-Verlag.
65. Nirvan Tyagi, Sofía Celi, Thomas Ristenpart, Nick Sullivan, Stefano Tessaro, and Christopher A. Wood. A fast andÂ simple partially oblivious prf, withÂ applications. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances*

in Cryptology – EUROCRYPT 2022, pages 674–705, Cham, 2022. Springer International Publishing.
66. Yanjiang Yang, Robert Deng, and Feng Bao. A practical password-based two-server authentication and key exchange system. *Dependable and Secure Computing, IEEE Transactions on*, 3:105–114, 05 2006.
67. Lin Zhang, Zhenfeng Zhang, and Xuexian Hu. UC-secure two-server password-based authentication protocol and its applications. In Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang, editors, *ASIACCS 16*, pages 153–164. ACM Press, May / June 2016.

A UC T-PAKE implies game-based T-PAKE

In this section, we prove that our UC-security model for atPAKE (Section 4) implies the game-based security model in [53]. In fact we will prove a stronger statement, where we consider a "real-or-random" (RoR)-style definition instead of the "find-then-guess" (FtG)-style definition in [53]. Concretely,

- The adversary cannot send Reveal commands;
- The adversary can send any number of Test commands — recall that there are two types of them, $\mathsf{Test}(\mathsf{U}, i, \mathsf{S}_j)$ and $\mathsf{Test}(\mathsf{S}_j, i)$ — instead of one (in [53] the adversary can only send a single query of *either type*), with the following caveats: a bit b is chosen at the beginning, and whenever a Test command is sent (no matter which type), if $b = 1$ then the corresponding real session key is sent, and if $b = 0$ then a random string is sent. In other words, the outputs of Test commands are either all real or all random. Furthermore, if $b = 0$ (the all random case) $\mathsf{Test}(\mathsf{U}, i, \mathsf{S}_j)$ and $\mathsf{Test}(\mathsf{S}_j, i')$ return the same string if Π_i^{U} and $\Pi_{i'}^{\mathsf{S}_j}$ are partnered instances.

The RoR-style definition was first introduced in the context of symmetric PAKE in [4].

One discrepancy between the game-based model in [53] and our UC model is that the former does not require the session IDs to be agreed upon between the parties in advance, while in UC the session ID is part of the input of the protocol and thus must be determined before the protocol execution begins. This could be done via, e.g., parties exchanging nonces as the session ID in advance. Below we ignore this technical distinction and always assume the "session ID-enhanced" version while referring to a protocol.

In the following we assume that the (UC-secure) tPAKE protocol has a *honest-but-curious (HBC)-respecting simulator*, i.e., a PPT simulator that for any PPT environment sends $\mathsf{testpwd}$ to $\mathcal{F}_{\mathsf{atPAKE}}$ with negligible probability in sessions where the real adversary passes all messages without any modification. This is analogous to the concept of "reasonable simulator" in [57], which is necessary for the result that UC-security implies game-based security in symmetric PAKE.

Theorem 4. *Consider any protocol Π that realizes functionality $\mathcal{F}_{\mathsf{atPAKE}}$ (Figures 4, 5, and 6). Then Π is secure under the game-based security definition as described above.*

Proof. Let $\tilde{\mathcal{A}}$ be any PPT adversary against game-based security of Π; we want to show that the advantage of $\tilde{\mathcal{A}}$ is negligible. We use $\tilde{\mathcal{A}}$ to construct an environment $\mathcal{Z}$ against the UC-security of Π:
Initialization:

- $\mathcal{Z}$ samples a password $\mathsf{pw} \leftarrow_{\$} D$.
- $\mathcal{Z}$ sends $(\mathsf{userinit}, sid_{\mathsf{A}}, sid_{\mathsf{T}}, \mathsf{pw})$ to U.
- $\mathcal{Z}$ sends $(\mathsf{auxinit}, sid_{\mathsf{A}}, i, \mathsf{U})$ and $(\mathsf{targetinit}, sid_{\mathsf{A}}, sid_{\mathsf{T}}, i, \mathsf{U})$ to S_i (for $i = 1, \ldots, n$).
- For $\mathsf{S} \in \mathbf{Corr}$, $\mathcal{Z}$ instructs $\mathcal{A}$ to corrupt S.

Protocol execution:

- $\mathcal{Z}$ samples a bit $b \leftarrow_{\$} \{0,1\}$ (to be used in Test queries).
- On $\mathsf{Execute}(\mathsf{U}, i, ((\mathsf{S}_{j_1}, \ell_{j_1}), \ldots, (\mathsf{S}_{j_k}, \ell_{j_k})))$ from $\tilde{\mathcal{A}}$, $\mathcal{Z}$ initiates the corresponding parties' sessions by sending $(\mathsf{usersession}, sid_{\mathsf{A}}, sid_{\mathsf{T}}, j_1, ssid, \mathsf{pw}), \ldots, (\mathsf{usersession}, sid_{\mathsf{A}}, sid_{\mathsf{T}}, j_k, ssid, \mathsf{pw})$ to U, $(\mathsf{targetsession}, sid_{\mathsf{T}}, j_1, \mathsf{U}, ssid)$ to S_{j_1}, $\ldots$, and $(\mathsf{targetsession}, sid_{\mathsf{T}}, j_k, \mathsf{U}, ssid)$ to S_{j_k}. After that, $\mathcal{Z}$ instructs the UC adversary $\mathcal{A}$[11] to pass messages between U and $\mathsf{S}_{j_1}, \ldots, \mathsf{S}_{j_k}$ without any modification.
- On $\mathsf{Send}((\mathsf{S}_{j_1}, \ldots, \mathsf{S}_{j_k}), i, m)$ from $\tilde{\mathcal{A}}$, if this query initializes a session between U and $\mathsf{S}_{j_1}, \ldots, \mathsf{S}_{j_k}$, $\mathcal{Z}$ initiates the corresponding session by sending $(\mathsf{usersession}, sid_{\mathsf{A}}, sid_{\mathsf{T}}, j_1, ssid, \mathsf{pw}), \ldots, (\mathsf{usersession}, sid_{\mathsf{A}}, sid_{\mathsf{T}}, j_k, ssid, \mathsf{pw})$ to U and instructing $\mathcal{A}$ to send m to $\mathsf{S}_{j_1}, \ldots, \mathsf{S}_{j_k}$. For later Send queries from $\tilde{\mathcal{A}}$, $\mathcal{Z}$ simply instructs $\mathcal{A}$ to send the corresponding messages to the corresponding party.
- On $\mathsf{Test}(\mathsf{U}, i, \mathsf{S}_j)$ from $\tilde{\mathcal{A}}$, $\mathcal{Z}$ observes U's session key K in the i-th session between U and S_j. (If there is no such session key, i.e., the session has not completed, then $\mathcal{Z}$ ignores this query.) If $b = 1$ then $\mathcal{Z}$ sends K to $\mathcal{A}$. If $b = 0$, $\mathcal{Z}$ checks if a K' has been sent to $\mathcal{A}$ as the result of $\mathcal{A}$'s $\mathsf{Test}(\mathsf{S}_j, i)$ query. If so, $\mathcal{Z}$ sends K' to $\mathcal{A}$; otherwise it sends $K' \leftarrow_{\$} \{0,1\}^{\tau}$ to $\mathcal{A}$.
- Similarly, on $\mathsf{Test}(\mathsf{S}_j, i)$ from $\tilde{\mathcal{A}}$, $\mathcal{Z}$ observes S_j's session key K in the i-th session between U and S_j, and sends K or a random K' (subject to the restriction that K' must be equal to the string sent to $\mathcal{A}$ as the result of a previous $\mathsf{Test}(\mathsf{U}, i, \mathsf{S}_j)$ query) to $\mathcal{A}$ according to b.

Since Π realizes $\mathcal{F}_{\mathsf{atPAKE}}$, there exists a successful PPT simulator SIM. Let $\mathsf{CorrectGuess}$ be the event that SIM sends at least one $\mathsf{testpwd}$ to $\mathcal{F}_{\mathsf{atPAKE}}$ resulting in "correct guess". Since SIM is HBC-respecting, it sends $\mathsf{testpwd}$ to $\mathcal{F}_{\mathsf{atPAKE}}$ with negligible probability in all eavesdropping sessions (in other words, in all sessions where $\tilde{\mathcal{A}}$ queries $\mathsf{Execute}$ to $\mathcal{Z}$). Furthermore, for each of the q sessions where

[11] We stress that $\mathcal{A}$ is the real adversary in the UC-security game, whereas $\tilde{\mathcal{A}}$ is an adversary against the game-based security of Π but invoked by the UC environment $\mathcal{Z}$.

$\tilde{\mathcal{A}}$ queries Send to $\mathcal{Z}$, SIM may send a $\mathsf{testpwd}$ to $\mathcal{F}_{\mathsf{atPAKE}}$ resulting in "correct guess". Overall, we have that

$$\Pr[\mathsf{CorrectGuess}] \leq \frac{q}{|D|} + \mathrm{negl}(\tau).$$

On the other hand, according to the syntax of $\mathcal{F}_{\mathsf{atPAKE}}$, if *none* of the sessions has even been COMPR, then each session outputs a random key. This means that if $\mathsf{CorrectGuess}$ does not happen, b is independent of $\tilde{\mathcal{A}}$'s view (no matter whether b is 0 or 1, $\tilde{\mathcal{A}}$ sees random strings from all sessions). So

$$\Pr[\mathsf{Succ}_{\Pi}(\tilde{\mathcal{A}}) = 1 \mid \mathsf{CorrectGuess}] = \frac{1}{2}.$$

Combining the above two, we get that

$$\Pr[\mathsf{Succ}_{\Pi}(\tilde{\mathcal{A}}) = 1] \leq \frac{1}{2} + \frac{q}{2|D|} + \mathrm{negl}(\tau),$$

which completes the proof. □

B Proof of 3HashTDH Security

In this section we prove Theorems 1 and 2 from Section 3.3, i.e. static and adaptive security of protocol 3HashTDH. These two proofs are provided as one; the sections in which they diverge are noted by the tags "STATIC" and "ADAPTIVE".

First, we provide the full version of $\mathcal{F}_{\mathsf{tOPRF}}$ (Figure 10), which includes an added transcript integrity feature. For every PRF evaluation, the evaluator and all $t+1$ servers output transcripts of the adversary's choosing. If the transcripts seen by both sides of the interaction match up, then it is guaranteed that the evaluator has correctly received an output from the function (i.e. PRF key) intended by the servers. If the servers were initialized by a dishonest party, then this guarantee is nullified (because, for example, the servers may not even hold a valid key sharing). This transcript model captures the natural intuition that if the man-in-the-middle adversary is passive (i.e. the messages sent between the honest parties are delivered without modification), then PRF evaluation should behave as expected. If the honest parties have a means of comparing their transcripts, they can then verify after the fact that an evaluation was correct.

We provide a corresponding full version of Π_{3HashTDH} (Figure 11), which achieves this transcript integrity feature. Our proof is carried out for this stronger model and protocol even though transcript integrity is not required by our higher-level atPAKE construction.

Proof. For any adversary $\mathcal{A}^*$, we construct simulator $\mathsf{SIM}_{\mathsf{3HashTDH}}$ as shown in Figures 13, 14, and 15. Without loss of generality, we assume that $\mathcal{A}^*$ is a

Notation

Initially $\mathsf{tx}[sid, ssid_{\mathsf{S}}, i] := 0$ and $F_{sid}(x)$ is undefined for all sid,$ssid_{\mathsf{S}}, i, x$. When $F_{sid}(x)$ is first referenced $\mathcal{F}_{\mathsf{tOPRF}}$ assigns $F_{sid}(x) \leftarrow_{\$} \{0,1\}^l$.

Initialization

1. On $(\mathsf{toprf.init}, sid, x_1, \ldots, x_k)$ from $\mathsf{P}_0 \in \mathcal{P}^*$, if sid is new (abort otherwise):
   - send $(\mathsf{toprf.init}, sid, \mathsf{P}_0)$ to $\mathcal{A}^*$
   - send $(\mathsf{toprf.initeval}, sid, F_{sid}(x_1), \ldots, F_{sid}(x_k))$ to P_0
   - save $(\mathsf{toprf.init}, sid, \mathsf{P}_0)$ and mark it TAMPERED if $\mathsf{P}_0 \in \mathbf{Corr}$
2. On $(\mathsf{toprf.sinit}, sid, i, \mathsf{P}_0)$ from S where $\mathsf{S} = \mathbf{S}_{sid}[i]$ or ($\mathsf{S} = \mathcal{A}^*$ and $\mathbf{S}_{sid}[i] \in \mathbf{Corr}$), save record $(\mathsf{toprf.sinit}, sid, i, \mathsf{P}_0)$ marked INACTIVE
3. On $(\mathsf{toprf.fininit}, sid, i)$ from $\mathcal{A}^*$ where $\exists$ record $\mathsf{urec} = (\mathsf{toprf.init}, sid, \mathsf{P}_0)$ and record $\mathsf{srec} = (\mathsf{toprf.sinit}, sid, i, \mathsf{P}_0)$ marked INACTIVE:
   - send $(\mathsf{toprf.sinit}, sid, i)$ to $\mathbf{S}_{sid}[i]$
   - if urec is TAMPERED then mark srec TAMPERED
   - else if $\mathbf{S}_{sid}[i] \in \mathbf{Corr}$ then mark srec COMPR
   - else (i.e. urec is not TAMPERED and $\mathbf{S}_{sid}[i] \notin \mathbf{Corr}$) mark srec ACTIVE

Corruption *(in the static corruption model disallowed after any other queries)*

4. On $(\mathsf{toprf.corrupt}, \mathsf{P})$ from $\mathcal{A}^*$ (with permission from $\mathcal{Z}$):
   - set $\mathbf{Corr} := \mathbf{Corr} \cup \{\mathsf{P}\}$
   - mark every ACTIVE record $(\mathsf{toprf.sinit}, sid, i, \mathsf{P}_0)$ COMPR where $\mathsf{P} = \mathbf{S}_{sid}[i]$

Evaluation

5. On $(\mathsf{toprf.eval}, sid, ssid_{\mathsf{U}}, x)$ from $\mathsf{U} \in \mathcal{P}^*$, if this is the first call from U for sid and $ssid_{\mathsf{U}}$:
   - send $(\mathsf{toprf.eval}, sid, ssid_{\mathsf{U}}, \mathsf{U})$ to $\mathcal{A}^*$
   - save $(\mathsf{toprf.eval}, sid, ssid_{\mathsf{U}}, \mathsf{U}, x)$ marked FRESH
6. On $(\mathsf{toprf.sndrcomplete}, sid, i, ssid_{\mathsf{S}})$ from S where $\exists$ record $\mathsf{srec} = (\mathsf{toprf.sinit}, sid, i, \mathsf{P}_0)$ not marked INACTIVE and ($\mathsf{S} = \mathbf{S}_{sid}[i]$ or ($\mathsf{S} = \mathcal{A}^*$ and srec is marked COMPR or TAMPERED)):
   - send $(\mathsf{toprf.sndrcomplete}, sid, i, ssid_{\mathsf{S}})$ to $\mathcal{A}^*$
   - and await $(\mathsf{toprf.sndrtrans}, sid, i, ssid_{\mathsf{S}}, \mathtt{tr}_i)$ from $\mathcal{A}^*$;
   - then send $(\mathsf{toprf.sndrtrans}, sid, i, ssid_{\mathsf{S}}, \mathtt{tr}_i)$ to S
   - if srec is not TAMPERED, then save $(\mathsf{toprf.sndrtrans}, sid, i, \mathtt{tr}_i)$
   - (regardless of the above) set $\mathsf{tx}[sid, ssid_{\mathsf{S}}, i]{+}{+}$
7. On $(\mathsf{toprf.rcvcomplete}, sid, ssid_{\mathsf{U}}, sid^*, ssid^*_{\mathsf{S}}, \mathbf{C}, \mathtt{tr}_{\mathsf{U}})$ from $\mathcal{A}^*$ where $|\mathbf{C}| = t+1$ and $\exists$ record $(\mathsf{toprf.eval}, sid, ssid_{\mathsf{U}}, \mathsf{U}, x)$ marked FRESH:
   - if (i) $\exists j \in \mathbf{C}$ such that $\mathsf{tx}[sid^*, ssid^*_{\mathsf{S}}, j] = 0$, or (ii) $\exists$ a set of records $\{(\mathsf{toprf.sndrtrans}, sid', j, \mathtt{tr}_{\mathsf{U}}[j])\}_{j \in \mathbf{C}}$ such that $sid' \neq sid^*$, then abort
   - otherwise mark the record COMPLETED, set $\mathsf{tx}[sid^*, ssid^*_{\mathsf{S}}, j]{-}{-}$ for all $j \in \mathbf{C}$, and send $(\mathsf{toprf.eval}, sid, ssid_{\mathsf{U}}, F_{sid^*}(x), \mathtt{tr}_{\mathsf{U}})$ to U

Fig. 10. $\mathcal{F}_{\mathsf{tOPRF}}$: threshold OPRF functionality, parameterized by threshold t, number of servers n, and output length l. Shadowed text can be ignored if transcript integrity is unneeded.

Initialization

1. On input $(\mathsf{toprf.init}, sid, x_1, ..., x_k)$, initializer P_0 does:
   - pick $\alpha_0, ..., \alpha_t \leftarrow_\$ \mathbb{Z}_m$ and define polynomial $k(x) := \alpha_0 + \alpha_1 x + ... + \alpha_t x^t$
   - pick $\beta_1, ..., \beta_t \leftarrow_\$ \mathbb{Z}_m$ and define polynomial $z(x) := \beta_1 x + \beta_2 x^2 + ... + \beta_t x^t$
   - for each $i \in [n]$, send $(\mathsf{channel.send}, [sid||i], \mathbf{S}_{sid}[i], (i, k(i), z(i)))$ to $\mathcal{F}_{\mathsf{channel}}$,
   - output $(\mathsf{toprf.initeval}, sid, y_1, ..., y_k)$ s.t. $y_j := H_3(x_j, H_1(x_j)^{\alpha_0})$ for all $j \in [k]$
2. On input $(\mathsf{toprf.sinit}, sid, i, \mathsf{P}_0)$, server $\mathbf{S}_{sid}[i]$ does:
   - await $(\mathsf{channel.deliver}, [sid||i], \mathsf{P}_0, (i, k_i, z_i))$ from $\mathcal{F}_{\mathsf{channel}}$;
   - then save record $(\mathsf{toprf.share}, sid, i, k_i, z_i)$
   - output $(\mathsf{toprf.fininit}, sid, i)$

Evaluation

3. On input $(\mathsf{toprf.eval}, sid, ssid_{\mathsf{U}}, x)$, evaluator U does:
   - pick $r \leftarrow_\$ \mathbb{Z}_m$ and compute $a := H_1(x)^r$
   - for each $i \in [n]$, send $(sid, i, ssid_{\mathsf{U}}, a)$ to $\mathbf{S}_{sid}[i]$
   - await responses $(sid, i, ssid_{\mathsf{U}}, \boxed{ssid_{\mathsf{S}},}\ b_i)$ from $\mathbf{S}_{sid}[i]$ for all $i \in \mathbf{C}$, for any set $\mathbf{C} \subseteq [n]$ of size $t+1$;
   - then compute $b := \prod_{i \in \mathbf{C}} b_i^{\lambda_i}$ where λ_i is the Lagrange interpolation coefficient for index i and index set $\mathbf{C}$
   - output $(\mathsf{toprf.eval}, ssid_{\mathsf{U}}, H_3(x, b^{1/r}), \boxed{(\mathtt{tr}_1, ..., \mathtt{tr}_n)})$ where $\boxed{\mathtt{tr}_i := (ssid_{\mathsf{S}}, a, b_i) \text{ for all } i \in \mathbf{C} \text{ and } \mathtt{tr}_i := \perp \text{ for all } i \notin \mathbf{C}}$
4. On input $(\mathsf{toprf.sndrcomplete}, sid, i, ssid_{\mathsf{S}})$, server $\mathbf{S}_{sid}[i]$ does:
   - retrieve record $(\mathsf{toprf.share}, sid, i, k_i, z_i)$ (abort if not found)
   - await $(sid, i, ssid_{\mathsf{U}}, a)$ from any U (if it hasn't already been received);
   - then compute $b_i := a^{k_i} \cdot H_2(ssid_{\mathsf{S}}, a)^{z_i}$
   - $\boxed{\text{output } (\mathsf{toprf.sndrtrans}, sid, i, ssid_{\mathsf{S}}, (ssid_{\mathsf{S}}, a, b_i))}$
   - send response $(sid, i, ssid_{\mathsf{U}}, \boxed{ssid_{\mathsf{S}},}\ b_i)$ to U

Fig. 11. Protocol Π_{3HashTDH} which realizes $\mathcal{F}_{\mathsf{tOPRF}}$ in the $\mathcal{F}_{\mathsf{channel}}$-hybrid world. Shadowed text can be ignored if transcript integrity is unneeded.

"dummy" adversary that merely passes messages to and from the environment $\mathcal{Z}$.

Figure 12 diagrams the interactions that occur in the real and simulated worlds. We now show that, for any efficient (i.e. PPT) $\mathcal{Z}$, the distinguishing advantage of $\mathcal{Z}$ between these two worlds is negligible. The argument proceeds by a series of game changes, starting from the real world $\mathbf{G}_0$ and ending at the simulated world $\mathbf{G}_7$. By $\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G},\mathbf{H}}$ we denote distinguisher $\mathcal{Z}$'s distinguishing advantage between world $\mathbf{G}$ and world $\mathbf{H}$. Specifically, $\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G},\mathbf{H}} = |\Pr_{\mathcal{Z} \leftrightarrow \mathbf{G}}[\mathcal{Z} \text{ outputs } 1] - \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}}[\mathcal{Z} \text{ outputs } 1]|$.

In this proof, $\mathcal{Z}$'s distinguishing advantage is upper-bounded using the advantages of PPT reductions in the Gap One-More Diffie Hellman

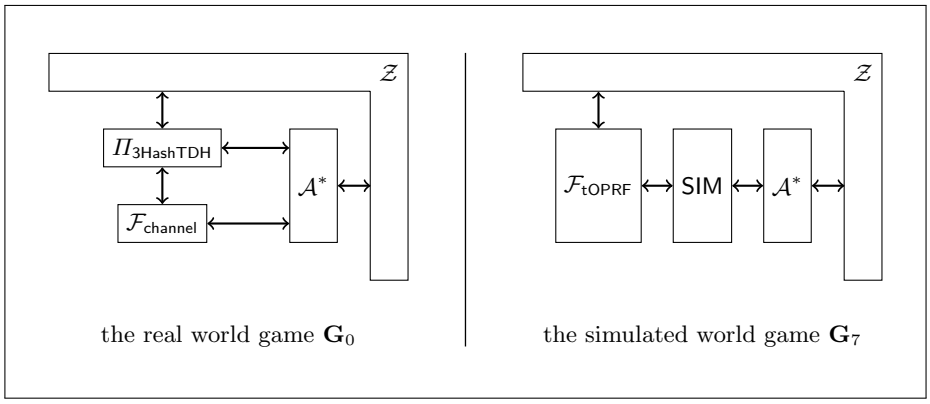

Fig. 12. Diagram of interactions between components in the real and simulated worlds of the 3HashTDH security proof. The proof shows that $\mathcal{Z}$'s views in these two games are indistinguishable.

(GapOMDH) and Decisional Diffie Hellman (DDH) games. By $\mathbf{Adv}^{\mathsf{GapOMDH}}_{\mathcal{R}}$ and $\mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}}$ we denote the advantages of algorithms $\mathcal{R}$ and $\mathcal{Q}$ in GapOMDH and DDH, respectively.

Game $\mathbf{G}_0$: The real world. The distinguisher $\mathcal{Z}$ interacts with Π_{3HashTDH} (Figure 11) in the role of the honest parties and in the role of the adversary. H_1, H_2, and H_3 are all true random oracles.

As a purely conceptual change from the true real world, one can imagine all the computational processes of the honest parties (i.e. Π_{3HashTDH} and $\mathcal{F}_{\mathsf{channel}}$) abstracted into a single monolithic component, which we call the simulator. This component also simulates the code of $\mathcal{F}_{\mathsf{tOPRF}}$ in parallel to its "real world" functions. By the end of the following sequence of game changes, all interactions with the honest parties will exclusively occur through the interface of $\mathcal{F}_{\mathsf{tOPRF}}$.

Game $\mathbf{G}_1$: Key generation never collides. $\mathbf{G}_1$ is $\mathbf{G}_0$ with only one change: no secret key $\alpha_0 = k(0)$ is ever chosen twice during initialization (Figure 13).

Clearly, $\mathbf{G}_0$ and $\mathbf{G}_1$ are identical unless there is a key collision in $\mathbf{G}_0$. The key is sampled from $\mathbb{Z}_m$ uniformly at random, so the probability that any two particular instances pick the same key is $\frac{1}{m}$. Therefore, if initialization is run q_I times, the probability that one or more collisions occur is upper-bounded by $\frac{q_I^2}{m}$.

$$\mathbf{Dist}^{\mathbf{G}_0,\mathbf{G}_1}_{\mathcal{Z}} \leq \frac{q_I^2}{m}$$

Game $\mathbf{G}_2$: H_1 has a trapdoor. $\mathbf{G}_2$ is $\mathbf{G}_1$ with only one change: instead of sampling a random group element directly, $H_1(x)$ samples a random exponent $\tau \leftarrow_{\$} \mathbb{Z}_m$ and returns $h := g^\tau$. A record (x, τ, h) is saved. H_1 now behaves exactly as it will in the fully simulated world (Figure 15).

Clearly, the external view of H_1 is unchanged.

Notation

Initially, $\mathsf{evalset}_{sid}(ssid_{\mathsf{S}}, a) := \emptyset$ for all sid, $ssid_{\mathsf{S}}$, and a. Values t, n, l are parameters.

Honest Initialization

1. On $(\mathsf{toprf.init}, sid, \mathsf{P}_0)$ from $\mathcal{F}_{\mathsf{tOPRF}}$ where $\mathsf{P}_0 \neq \mathcal{A}^*$:
   - pick $\alpha_0, ..., \alpha_t \leftarrow_{\$} \mathbb{Z}_m$ (such that α_0 has never been picked before) and define polynomial $k(x) := \alpha_0 + \alpha_1 x + ... + \alpha_t x^t$
   - pick $\beta_1, ..., \beta_t \leftarrow_{\$} \mathbb{Z}_m$ and define polynomial $z(x) := \beta_1 x + \beta_2 x^2 + ... + \beta_t x^t$
   - save $(\mathsf{toprf.init}, sid, \mathsf{P}_0, g^{k(0)})$
   - for each $i \in [n]$, save $(\mathsf{toprf.share}, sid, i, \mathsf{P}_0, k(i), z(i))$ marked INACTIVE
   - for each $i \in [n]$, send $(\mathsf{channel.send}, [id||i], \mathsf{P}_0, \mathbf{S}_{sid}[i], |(i, k(i), z(i))|)$ to $\mathcal{A}^*$
2. On $(\mathsf{channel.deliver}, [sid||i], \mathsf{P}_0, \mathbf{S}_{sid}[i])$ from $\mathcal{A}^*$ where $\exists$ record $(\mathsf{toprf.share}, sid, i, \mathsf{P}_0, k_i, z_i)$ marked INACTIVE:
   - mark the record ACTIVE (or if $\mathsf{P}_0 = \mathcal{A}^*$ mark it COMPR)
   - send $(\mathsf{toprf.fininit}, sid, i)$ to $\mathcal{F}_{\mathsf{tOPRF}}$

Dishonest Initialization

3. On $(\mathsf{channel.send}, [sid||i], \mathbf{S}_{sid}[i], (i, k_i, z_i))$ from $\mathcal{A}^*$ on behalf of $\mathsf{P}_0 \in \mathbf{Corr}$:
   - if this is the first such message for sid, then send $(\mathsf{toprf.init}, sid)$ to $\mathcal{F}_{\mathsf{tOPRF}}$ on behalf of P_0
   - receive $(\mathsf{toprf.initeval}, sid)$ in response
   - and save $(\mathsf{toprf.share}, sid, i, \mathcal{A}^*, k_i, z_i)$ marked INACTIVE

Corruption

4. On corruption by $\mathcal{A}^*$ of party $\mathcal{P}$ (with permission from $\mathcal{Z}$):
   - send $(\mathsf{toprf.corrupt}, \mathsf{P})$ to $\mathcal{F}_{\mathsf{tOPRF}}$
   - set $\mathbf{Corr} := \mathbf{Corr} \cup \{\mathsf{P}\}$
5. If ever $\exists$ record $(\mathsf{toprf.share}, sid, i, \mathsf{P}_0, k_i, z_i)$ marked ACTIVE where $\mathbf{S}_{sid}[i] \in \mathbf{Corr}$ and $\mathbf{S}_{sid}[i] \neq \mathcal{A}^*$, mark it COMPR and send $(\mathsf{toprf.share}, sid, i, k_i, z_i)$ to $\mathcal{A}^*$
6. Whenever the simulator eventually halts, for each record $(\mathsf{toprf.init}, sid, \mathsf{P}_0, k)$, emit $(\text{CORR}, sid, |\{i : \mathbf{S}_{sid}[i] \in \mathbf{Corr}\}|)$

Fig. 13. Simulator $\mathsf{SIM}_{\mathsf{3HashTDH}}$ for protocol Π_{3HashTDH}, part 1: Notation, Honest Initialization, Corrupt Initialization, and Corruption

Honest Evaluation

7. On $(\mathsf{toprf.eval}, sid, ssid_{\mathsf{U}}, \mathsf{U})$ from $\mathcal{F}_{\mathsf{tOPRF}}$ where $\mathsf{U} \neq \mathcal{A}^*$:
   - pick $r \leftarrow_{\$} \mathbb{Z}_m$ and define $a := g^r$
   - for each $i \in [n]$, send $(sid, i, ssid_{\mathsf{U}}, a)$ to $\mathcal{A}^*$ (addressed from U to $\mathbf{S}_{sid}[i]$)
   - and await responses $(sid, i, ssid_{\mathsf{U}}, ssid_{\mathsf{S}}, b_i)$ from $\mathcal{A}^*$ (addressed from $\mathbf{S}_{sid}[i]$ to U) for all $i \in \mathbf{C}$, for any set $\mathbf{C} \subseteq [n]$ of size $t+1$;
   - then compute $b := \prod_{i \in \mathbf{C}} b_i^{\lambda_i}$ where λ_i is the Lagrange interpolation coefficient for index i and index set $\mathbf{C}$
   - run $\mathsf{FindEvalset}(b^{1/r})$, which returns $(sid^*, ssid^*_{\mathsf{S}}, \mathbf{C}')$ (see routine 9)
   - send $(\mathsf{toprf.rcvcomplete}, sid, ssid_{\mathsf{U}}, sid^*, ssid^*_{\mathsf{S}}, \mathbf{C}', (\mathtt{tr}_1, ..., \mathtt{tr}_n))$ to $\mathcal{F}_{\mathsf{tOPRF}}$ where $\mathtt{tr}_i := (ssid_{\mathsf{S}}, a, b_i)$ for all $i \in \mathbf{C}$ and $\mathtt{tr}_i := \perp$ for all $i \notin \mathbf{C}$
8. On $(\mathsf{toprf.sndrcomplete}, sid, i, ssid_{\mathsf{S}})$ from $\mathcal{F}_{\mathsf{tOPRF}}$ and $(sid, i, ssid'_{\mathsf{U}}, a)$ from $\mathcal{A}^*$ (addressed from U to $\mathbf{S}_{sid}[i]$) where $\mathbf{S}_{sid}[i] \neq \mathcal{A}^*$:
   - set $\mathsf{evalset}_{sid}(ssid_{\mathsf{S}}, a) := \mathsf{evalset}_{sid}(ssid_{\mathsf{S}}, a) \cup \{i\}$
   - retrieve $(\mathsf{toprf.share}, sid, i, \mathsf{P}_0, k_i, z_i)$
   - define $\kappa_{a,i} := a^{k_i}$ and $\zeta_{(ssid_{\mathsf{S}},a),i} := H_2(ssid_{\mathsf{S}}, a)^{z_i}$
   - compute $b_i := \kappa_{a,i} \cdot \zeta_{(ssid_{\mathsf{S}},a),i}$
   - send $(sid, i, ssid'_{\mathsf{U}}, ssid_{\mathsf{S}}, b_i)$ to $\mathcal{A}^*$ (addressed from $\mathbf{S}_{sid}[i]$ to U)
   - send $(\mathsf{toprf.sndrtrans}, sid, i, ssid_{\mathsf{S}}, (ssid_{\mathsf{S}}, a, b_i))$ to $\mathcal{F}_{\mathsf{tOPRF}}$
9. Define subroutine $\mathsf{FindEvalset}(k^*)$:
   - find record $(\mathsf{toprf.init}, sid^*, \mathsf{P}_0, k^*)$
   - if no such record exists, create it as follows:
     - choose an unused sid^* such that $\mathbf{S}_{sid^*} = (\mathcal{A}^*)^n$
     - send $(\mathsf{toprf.init}, sid^*)$ to $\mathcal{F}_{\mathsf{tOPRF}}$
     - receive $(\mathsf{toprf.init}, sid^*, \mathcal{A}^*)$ and $(\mathsf{toprf.initeval}, sid^*)$ in response
     - for each $i \in [n]$, send $(\mathsf{toprf.sinit}, sid^*, i, \mathcal{A}^*)$ and $(\mathsf{toprf.fininit}, sid^*, i)$ to $\mathcal{F}_{\mathsf{tOPRF}}$
     - save $(\mathsf{toprf.init}, sid^*, \mathcal{A}^*, k^*)$
   - if $\mathsf{P}_0 = \mathcal{A}^*$, then define $\mathbf{C}^* := [n]$
   - if $\mathsf{P}_0 \neq \mathcal{A}^*$, then define $\mathbf{C}^* := \{i : \exists$ record $(\mathsf{toprf.share}, sid^*, i, \mathsf{P}_0, k_i, z_i)$ marked COMPR$\}$
   - for each $i \in \mathbf{C}^*$:
     - pick an unused $ssid'$
     - send $(\mathsf{toprf.sndrcomplete}, sid^*, i, ssid')$ to $\mathcal{F}_{\mathsf{tOPRF}}$
     - receive the same message in response (don't run routine 8)
     - and send $(\mathsf{toprf.sndrtrans}, sid^*, i, ssid', \perp)$ to $\mathcal{F}_{\mathsf{tOPRF}}$
   - pick any $(ssid^*_{\mathsf{S}}, a)$ such that $|\mathsf{evalset}_{sid^*}(ssid^*_{\mathsf{S}}, a)| \geq (t+1) - |\mathbf{C}^*|$ (if no such $(ssid^*_{\mathsf{S}}, a)$ exists, then emit (FAIL, sid^*) and halt the entire simulator)
   - pick any evaluation set $\mathbf{C}' \subseteq \mathbf{C}^* \cup \mathsf{evalset}_{sid^*}(ssid^*_{\mathsf{S}}, a)$ of size $|\mathbf{C}'| = t+1$
   - set $\mathsf{evalset}_{sid^*}(ssid^*_{\mathsf{S}}, a) := \emptyset$
   - return $(sid^*, ssid^*_{\mathsf{S}}, \mathbf{C}')$

Fig. 14. Simulator $\mathsf{SIM}_{\mathsf{3HashTDH}}$ for protocol Π_{3HashTDH}, part 2: Honest Evaluation

Dishonest Evaluation

10. On fresh query x to $H_1(\cdot)$:
    - pick $\tau \leftarrow_\$ \mathbb{Z}_m$ and define $h := g^\tau$
    - save $(\mathsf{toprf.h1}, x, \tau, h)$
    - set $H_1(x) := h$ and return it
11. On fresh query $(ssid_\mathsf{S}, a)$ to $H_2(\cdot)$, simply set $H_2(ssid_\mathsf{S}, a) \leftarrow_\$ \mathbb{G}$ and return it
12. On fresh query (x, u) to $H_3(\cdot)$:
    - retrieve $(\mathsf{toprf.h1}, x, \tau, h)$ if it exists (otherwise, simply set $H_3(x, u) \leftarrow_\$ \{0,1\}^l$ and return it)
    - run $\mathsf{FindEvalset}(u^{1/\tau})$, which returns $(sid^*, ssid^*_\mathsf{S}, \mathbf{C}')$ (see routine 9)
    - pick an unused $ssid_\mathsf{U}$ and send $(\mathsf{toprf.eval}, sid^*, ssid_\mathsf{U}, x)$ to $\mathcal{F}_{\mathsf{tOPRF}}$
    - send $(\mathsf{toprf.rcvcomplete}, sid^*, ssid_\mathsf{U}, sid^*, ssid^*_\mathsf{S}, \mathbf{C}', (\perp)^n)$ to $\mathcal{F}_{\mathsf{tOPRF}}$
    - receive $(\mathsf{toprf.eval}, ssid_\mathsf{U}, y, (\perp)^n)$ in response
    - set $H_3(x, u) := y$ and return it

Fig. 15. Simulator $\mathsf{SIM}_{\mathsf{3HashTDH}}$ for protocol Π_{3HashTDH}, part 3: Dishonest Evaluation

$$\mathbf{Dist}_\mathcal{Z}^{\mathbf{G}_1,\mathbf{G}_2} = 0$$

Game $\mathbf{G}_3$: Illegal evaluations cause failure. $\mathbf{G}_3$ is $\mathbf{G}_2$ with the following changes:

- As honest servers perform evaluations (i.e. $\mathsf{toprf.sndrcomplete}$) the $\mathsf{evalset}$ of each $(ssid_\mathsf{S}, a)$ pair is tracked (Figure 14).
- Subroutine $\mathsf{FindEvalset}$ is introduced (Figure 14). Given parameter $k^* \in \mathbb{G}$, this subroutine finds a PRF instance that was initialized with key $k(0)$ such that $k^* = g^{k(0)}$. If no such instance exists, it uses the interface of $\mathcal{F}_{\mathsf{tOPRF}}$ to create one, with $\mathcal{A}^*$ taking the nominal role of all n servers. It then "prints blank tickets" for as many of this instance's servers as it can (all n if the instance was just created); these blank tickets could correspond to any pair $(ssid_\mathsf{S}, a)$. Finally it finds a pair $(ssid^*_\mathsf{S}, a)$ that has been evaluated by a set $\mathbf{C}'$ of at least $t+1$ servers and "uses it up" by resetting its $\mathsf{evalset}$ to empty. If no such $(ssid^*_\mathsf{S}, a)$ can be found, then it is illegal to perform an evaluation on this PRF instance; the simulator immediately emits the event FAIL and halts. Otherwise, the relevant instance identifier sid^*, server subsession identifier $ssid^*_\mathsf{S}$, and evaluation set $\mathbf{C}'$ are returned.
- Whenever the simulator eventually halts (e.g. due to FAIL), it emits an event CORR reporting the number of corrupted servers for each PRF instance (Figure 13).
- H_3 queries by the adversary check for illegal evaluation before completing (Figure 15). In particular, upon fresh query (x, u) where $H_1(x) = g^\tau$, H_3 calls $\mathsf{FindEvalset}(u^{1/\tau})$. If that subroutine succeeds at finding a matching evaluation set, then H_3 uses $\mathcal{F}_{\mathsf{tOPRF}}$'s $\mathsf{toprf.rcvcomplete}$ interface to query the PRF output y for x. However, in this game H_3 does **not** return y. It simply samples a random output and returns it as before.

– The honest clients' PRF evaluation process makes the same check (Figure 14). In particular, before querying $H_3(x, b^{1/r})$ it calls $\mathsf{FindEvalset}((b^{1/r})^{1/\tau})$ (note that, unlike in the eventual fully simulated world, this routine is still using $H_1(x)$ as a). If FindEvalset succeeds, then client evaluation continues via the $\mathcal{F}_{\mathsf{tOPRF}}$'s toprf.rcvcomplete interface. However, in this game the PRF output produced by that interface is **not** the value returned to the client. It simply samples a random output and returns it as before.
– Note that the use of H_3 during initialization remains unchanged. No check to FindEvalset is made. Randomly sampled outputs are returned as before.

Observe that FindEvalset's mechanism for determining the legality of an evaluation is always at least as restrictive as the "ticketing" mechanism used by $\mathcal{F}_{\mathsf{tOPRF}}$ internally. $\mathcal{F}_{\mathsf{tOPRF}}$ allows an evaluation as long as every server in the evaluation set has at least one unused ticket corresponding to $ssid^*_{\mathsf{S}}$. FindEvalset only allows an evaluation if every server in the evaluation set has at least one unused ticket corresponding to $ssid^*_{\mathsf{S}}$ **and** the same common query a. Therefore, the interactions with toprf.rcvcomplete introduced in this game change are guaranteed to succeed (since they are always preceded by successful calls to FindEvalset).

Observe furthermore that the eventual output values of the adversarial H_3 and honest client evaluation routines are unaltered by this game change. Therefore, the only potential change to $\mathcal{Z}$'s view is the newly added possibility of the FAIL event, which causes all execution to immediately halt. $\mathcal{Z}$'s probability of distinguishing between $\mathbf{G}_2$ and $\mathbf{G}_3$ is upper-bounded by the probability that $\mathcal{Z}$ triggers FAIL. This probability can expressed as a summation over every possible number of servers that could be corrupt when that event occurs. Notice that FAIL is impossible when the number of corruptions is $(t+1)$ or greater.

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_2,\mathbf{G}_3} \leq \sum_{sid} \Pr_{\mathcal{Z}\leftrightarrow\mathbf{G}_3}[(\text{FAIL}, sid)] = \sum_{sid}\sum_{t'=0}^{t} \Pr_{\mathcal{Z}\leftrightarrow\mathbf{G}_3}[(\text{FAIL}, sid) \wedge (\text{CORR}, sid, t')]$$

Notice that the FAIL event is only possible for honestly initialized instances; by q_I we denote the number of times honest initialization occurs. From this point forward, we consider only the instance sid^* with the maximum probability of hosting the FAIL event. For succicntness, we will omit its identifier sid^* from the FAIL and CORR events.

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_2,\mathbf{G}_3} \leq q_I \cdot \sum_{t'=0}^{t} \Pr_{\mathcal{Z}\leftrightarrow\mathbf{G}_3}[\text{FAIL} \wedge (\text{CORR}, t')]$$

In order to bound $\Pr_{\mathcal{Z}\leftrightarrow\mathbf{G}_3}[\text{FAIL} \wedge (\text{CORR}, t')]$, we define an auxiliary series of games derived from $\mathbf{G}_3$.

Game H′: The simulator guesses which servers will be corrupted. $\mathbf{H}'$ is $\mathbf{G}_3$ with the following changes:

– When the instance identified by sid^* is initialized, pick a t-element subset of $[n]$. Call it $\mathbf{Guess}_0$.
  - STATIC: If possible, pick $\mathbf{Guess}_0$ such that for every server $\mathbf{S}_{sid^*}[i] \in \mathbf{Corr}$, $i \in \mathbf{Guess}_0$. Otherwise (i.e. if more than t of this instance's servers are corrupted), simply pick $\mathbf{Guess}_0$ uniformly at random.
  - ADAPTIVE: Pick $\mathbf{Guess}_0$ uniformly at random.
– Whenever the simulator eventually halts, report whether the guess was unsuccessful. Specifically, if $\exists$ server $\mathbf{S}_{sid^*}[i] \in \mathbf{Corr}$ for $i \notin \mathbf{Guess}_0$, then emit $(\text{EXPOSED}, sid^*)$.
– For servers not in $\mathbf{Guess}_0$, compute ζ by interpolation from the ζ's held by the servers that are in $\mathbf{Guess}_0$. Specifically, $\forall (ssid_{\mathsf{S}}, a)\ \forall i \notin \mathbf{Guess}_0\ \zeta_{(ssid_{\mathsf{S}},a),i} := \prod_{j \in \mathbf{Guess}_0} \zeta_{(ssid_{\mathsf{S}},a),j}^{\lambda_j}$ (where λ_j is the Lagrange interpolation coefficient for index j, index set $\mathbf{Guess}_0$, and target index i).

Clearly, $\mathcal{Z}$'s views in $\mathbf{G}_3$ and $\mathbf{H}'$ are identical.

Game $\mathbf{H}_0$: Unsuccessful guesses cause immediate failure. $\mathbf{H}_0$ is $\mathbf{H}'$ with only one change: if ever $\exists$ server $\mathbf{S}_{sid^*}[i] \in \mathbf{Corr}$ for $i \notin \mathbf{Guess}_0$, then immediately emit $(\text{EXPOSED}, sid^*)$ and halt the simulator.

Unless $\mathcal{Z}$ manages to trigger EXPOSED by corrupting a server outside of $\mathbf{Guess}_0$, its views in $\mathbf{H}'$ and $\mathbf{H}_0$ are identical.

STATIC: Our choice of $\mathbf{Guess}_0$ makes EXPOSED impossible as long as $t' \leq t$.

$$\Pr_{\mathcal{Z} \leftrightarrow \mathbf{G}_3}[\text{FAIL} \wedge (\text{CORR}, t')] = \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_0}[\text{FAIL} \wedge (\text{CORR}, t')]$$

ADAPTIVE: We can bound $\mathcal{Z}$'s distinguishing advantage combinatorially (for all $0 \leq t' \leq t$).

$$\begin{aligned}
\Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_0}[\text{FAIL} \wedge (\text{CORR}, t')] &= \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_0}[\text{FAIL} \wedge (\text{CORR}, t') \wedge \neg \text{EXPOSED}] && (1)\\
&= \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}'}[\text{FAIL} \wedge (\text{CORR}, t') \wedge \neg \text{EXPOSED}] && (2)\\
&= \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}'}[\text{FAIL} \wedge (\text{CORR}, t')] \\
&\quad \cdot \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}'}[\neg \text{EXPOSED} \mid \text{FAIL} \wedge (\text{CORR}, t')] && (3)\\
&= \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}'}[\text{FAIL} \wedge (\text{CORR}, t')] \\
&\quad \cdot \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}'}[\neg \text{EXPOSED} \mid (\text{CORR}, t')] && (4)\\
&= \Pr_{\mathcal{Z} \leftrightarrow \mathbf{G}_3}[\text{FAIL} \wedge (\text{CORR}, t')] \\
&\quad \cdot \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}'}[\neg \text{EXPOSED} \mid (\text{CORR}, t')] && (5)\\
&= \Pr_{\mathcal{Z} \leftrightarrow \mathbf{G}_3}[\text{FAIL} \wedge (\text{CORR}, t')] \cdot \frac{\binom{t}{t'}}{\binom{n}{t'}} && (6)\\
\Pr_{\mathcal{Z} \leftrightarrow \mathbf{G}_3}[\text{FAIL} \wedge (\text{CORR}, t')] &= \frac{\binom{n}{t'}}{\binom{t}{t'}} \cdot \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_0}[\text{FAIL} \wedge (\text{CORR}, t')] && (7)
\end{aligned}$$

At step (1) observe that the FAIL and EXPOSED events are disjoint in $\mathbf{H}_0$ since both cause the execution to immediately halt. At step (2) observe that as long as the event EXPOSED does not happen, $\mathcal{Z}$'s views in $\mathbf{H}'$ and $\mathbf{H}_0$ are identical. At step (4) observe that FAIL and EXPOSED are independent in $\mathbf{H}'$ when conditioned on the event (CORR, t'). At step (5) observe that $\mathbf{G}_3$ and $\mathbf{H}'$ are identical except for the EXPOSED event.

In order to bound $\Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_0}[\text{FAIL} \wedge (\text{CORR}, t')]$ for all t', we now define additional games $\mathbf{H}_1, \mathbf{H}_2, \ldots \mathbf{H}_t$.

Game $\mathbf{H}_k$: One more server picks its responses randomly. For all $1 \leq k \leq t$, $\mathbf{H}_k$ is recursively defined as $\mathbf{H}_{k-1}$ with the following changes:

- After defining $\mathbf{Guess}_{k-1}$, pick one element i_k from $\mathbf{Guess}_{k-1}$. Define $\mathbf{Guess}_k := \mathbf{Guess}_{k-1} \setminus \{i_k\}$ (so $|\mathbf{Guess}_k| = t - k$).
  - STATIC: If possible, pick i_k such that $\mathbf{S}_{sid^*}[i_k] \notin \mathbf{Corr}$. Otherwise (i.e. if more than $t - k$ of this instance's servers are corrupted), simply pick i_k uniformly at random.
  - ADAPTIVE: Pick i_k uniformly at random.
- If server i_k is ever corrupted, then immediately emit $(\text{EXPOSED}, sid^*)$ and halt the simulator.
- For all a, server i_k sets ζ at random. Specifically, $\forall(ssid_{\mathsf{S}}, a)\ \zeta_{(ssid_{\mathsf{S}},a),i_k} \leftarrow_{\$} \mathbb{G}$.

In each of $\mathbf{H}_0, \mathbf{H}_1, \ldots, \mathbf{H}_t$, $\mathcal{Z}$'s probability of triggering FAIL can be bounded using a GapOMDH reduction.

Reduction $\mathcal{R}_k$: GapOMDH. The GapOMDH input is a vector $(y^*, h_1, \ldots, h_q)$ where all h_j are uniformly random group elements and $y^* = g^s$ for some uniformly random secret exponent s. The reduction has access to an oracle $\mathsf{OMDH}(a)$, which returns a^s. It also has access to an oracle $\mathsf{DDH}(y, h, u)$, which returns a bit that is 1 if and only if (y, h, u) is a Diffie-Hellman tuple (i.e. there exist $a, b \in \mathbb{Z}_m$ such that $y = g^a$, $h = g^b$, and $u = g^{ab}$). The reduction wins if it outputs a set W of pairs (j, h_j^s) and $|W|$ is greater than the number of (unique) queries it made to the OMDH oracle.

For all $1 \leq k \leq t$, $\mathcal{R}_k$ is constructed from $\mathcal{Z} \leftrightarrow \mathbf{H}_k$ with the following changes:

- Initially, set $W := \emptyset$.
- Queries to H_1 are answered with the h_j values. Specifically, $H_1(x_j) := h_j$ where j is initialized as 1 and increments after a fresh query is seen. As a consequence of this change, H_3 no longer has $\mathsf{toprf.h1}$ records ("trapdoors") to rely on.
- H_3 uses the DDH oracle in place of the H_1 trapdoors. Specifically, upon fresh query $H_3(x, u)$ where $H_1(x) = h_j$ (if H_1 was never queried on x, then make such a query now on behalf of $\mathcal{Z}$), do the following:
  - retrieve every initialization record $(\mathsf{toprf.init}, sid, \mathcal{A}^*, y_{sid})$ where $sid \neq sid^*$ and query $\mathsf{DDH}(y_{sid}, h_j, u)$ for each one,
  - also query $\mathsf{DDH}(y^*, h_j, u)$,

- if all oracle responses are 0, then simply set $H_3(x,u) \leftarrow_\$ \{0,1\}^l$ and return it,
- but otherwise, if some $\mathsf{DDH}(y, h_j, u) = 1$, then call $\mathsf{FindEvalset}(y)$, and proceed accordingly (if $y = y^*$, then also save $W := W \cup \{(j,u)\}$).

– For all a, for the first k queries to servers not in $\mathbf{Guess}_k$, set κ at random (only for instance sid^*). Specifically, $\forall a \ \forall i \notin \mathbf{Guess}_k \ \kappa_{a,i} \leftarrow_\$ \mathbb{G}$ if $|(\cup_{ssid_\mathsf{S}} \mathsf{evalset}(ssid_\mathsf{S}, a)) \setminus \mathbf{Guess}_k| \leq k$.
– For the $(k+1)$st query and on, use the OMDH oracle and interpolate κ (only for instance sid^*). Specifically, $\forall a \ \forall i \notin \mathbf{Guess}_k$ $\kappa_{a,i} := \mathsf{OMDH}(a)^{\lambda_0} \prod_{j \in \mathbf{E}} \kappa_{a,j}^{\lambda_j}$ if $|(\cup_{ssid_\mathsf{S}} \mathsf{evalset}(ssid_\mathsf{S}, a)) \setminus \mathbf{Guess}_k| > k$ (where $\mathbf{E}$ is any t-element subset of $\cup_{ssid_\mathsf{S}} \mathsf{evalset}(ssid_\mathsf{S}, a) \cup \mathbf{Guess}_k$ and λ_j is the Lagrange interpolation coefficient for index j, index set $\mathbf{E}$, and target index i).
– If a call to $\mathsf{FindEvalset}$ triggers event FAIL followed by $(\text{CORR}, t-k)$, then output W and thereby win the GapOMDH game.

These modifications do not change $\mathcal{Z}$'s view at all. Therefore, this reduction's probability of winning the GapOMDH game is equal to the probability that $\mathcal{Z}$ in interaction with $\mathbf{H}_k$ triggers FAIL with exactly $(t-k)$ corruptions.

$$\forall\ 1 \leq k \leq t \ \ \mathbf{Adv}^{\mathsf{GapOMDH}}_{\mathcal{R}_k} = \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_k}[\text{FAIL} \wedge (\text{CORR}, t-k)]$$

All that remains is to relate $\Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_0}[\text{FAIL} \wedge (\text{CORR}, t')]$ to $\Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_{t-t'}}[\text{FAIL} \wedge (\text{CORR}, t')]$ for all t'. To do so, we use a reduction from DDH to bound $\mathcal{Z}$'s ability to distinguish between $\mathbf{H}_k$ and $\mathbf{H}_{k-1}$ for all k. First we recall the following useful variant of DDH.

Lemma 1. *Let $(\mathbb{G}, \cdot)$ be a cyclic group of prime order m with generator g. For any k polynomial in τ, the* Generalized Decisional Diffie-Hellman problem *on $\mathbb{G}$ is to distinguish the following two distributions: $\{(h_1, h'_1, h_2, h'_2, ..., h_k, h'_k) : \forall i \in \{1,2,..,k\} \ h_i, h'_i \leftarrow_\$ \mathbb{G}\}$ and $\{(h_1, h'_1, h_2, h'_2, ...) : r \leftarrow_\$ \mathbb{Z}_m, \forall i \in \{1,2,..,k\} \ h_i \leftarrow_\$ \mathbb{G}, h'_i = {h_i}^r\}$. The Generalized DDH problem on $\mathbb{G}$ reduces tightly to standard DDH on $\mathbb{G}$. (This reduction is well-known; for example, see [7].)*

Reduction $\mathcal{Q}_{k,t'}$: DDH. The Generalized DDH input is a vector $(h_1, h'_1, \ldots, h_q, h'_q)$ where all h_j are uniformly random group elements. If challenge bit $b = 0$, then there exists a secret exponent s such that every $h'_j = h_j^s$. If challenge bit $b = 1$, then all h'_j are uniformly random. The reduction's advantage is its probability of distinguishing the case where $b = 0$ from the case where $b = 1$.

For all $1 \leq k \leq t$ and $0 \leq t' \leq t$, $\mathcal{Q}_{k,t'}$ is constructed from $\mathcal{Z} \leftrightarrow \mathbf{H}_k$ with the following changes:

– Queries to H_2 are answered with the h_j values. Specifically, $H_2((ssid_\mathsf{S}, a)_j) := h_j$ where j is initialized as 1 and increments after a fresh $(ssid_\mathsf{S}, a)$ is seen.

– For all $(ssid_{\mathsf{S}}, a)_j$, server i_k sets ζ as h'_j. Specifically, $\forall (ssid_{\mathsf{S}}, a)_j \; \zeta_{(ssid_{\mathsf{S}},a)_j, i_k} := h'_j$ where j is as above.
– Ultimately, output 1 iff execution ends with the event FAIL followed by (CORR, t').

If $b = 1$, $\mathcal{Z}$'s view in $\mathcal{Q}_{k,t'}$ is identical to its view in $\mathbf{H}_k$. If $b = 0$, its view is identical to its view in $\mathbf{H}_{k-1}$ unless server i_k is corrupted. Denote the event of this corruption by (COMP, i_k).

$$\mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}_{k,t'}} = \left| \Pr_{\mathcal{Q}_{k,t'}} [\text{FAIL} \wedge (\text{CORR}, t') \,|\, b = 0] - \Pr_{\mathcal{Q}_{k,t'}} [\text{FAIL} \wedge (\text{CORR}, t') \,|b = 1] \right| \quad (1)$$

$$= \left| \Pr_{\mathcal{Q}_{k,t'}} [\text{FAIL} \wedge (\text{CORR}, t') \,|\, b = 0] - \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_k} [\text{FAIL} \wedge (\text{CORR}, t')] \right| \quad (2)$$

$$= \Bigg| \Pr_{\mathcal{Q}_{k,t'}} [\text{FAIL} \wedge (\text{CORR}, t') \wedge \neg (\text{COMP}, i_k) \,|\, b = 0] - \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_k} [\text{FAIL} \wedge (\text{CORR}, t')] \Bigg| \quad (3)$$

$$= \Bigg| \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_{k-1}} [\text{FAIL} \wedge (\text{CORR}, t') \wedge \neg (\text{COMP}, i_k)] - \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_k} [\text{FAIL} \wedge (\text{CORR}, t')] \Bigg| \quad (4)$$

$$= \Bigg| \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_{k-1}} [\text{FAIL} \wedge (\text{CORR}, t')] \cdot \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_{k-1}} [\neg (\text{COMP}, i_k) \,|\, \text{FAIL} \wedge (\text{CORR}, t')] - \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_k} [\text{FAIL} \wedge (\text{CORR}, t')] \Bigg| \quad (5)$$

At step (3) observe that the FAIL and (COMP, i_k) events are disjoint in $\mathcal{Q}_{k,t'}$ since both cause the execution to immediately halt.

In the case of static corruptions, our choice of i_k makes (COMP, i_k) impossible as long as $t' \leq t - k$ (or equivalently, $k \leq t - t'$).

STATIC:

$$\mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}_{k,t'}} = \left| \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_{k-1}} [\text{FAIL} \wedge (\text{CORR}, t')] - \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_k} [\text{FAIL} \wedge (\text{CORR}, t')] \right|$$

$$\Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_{k-1}} [\text{FAIL} \wedge (\text{CORR}, t')] \leq \mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}_{k,t'}} + \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_k} [\text{FAIL} \wedge (\text{CORR}, t')]$$

By solving this recurrence relation with the base case $\Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_{t-t'}} [\text{FAIL} \wedge (\text{CORR}, t')] = \mathbf{Adv}^{\mathsf{GapOMDH}}_{\mathcal{R}_{t-t'}}$ from above, we can set a bound on $\Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_0} [\text{FAIL} \wedge (\text{CORR}, t')]$. Here we use $\mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}_{t'}}$ to denote $\max_{0 \leq k \leq t-t'} \mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}_{k,t'}}$.

$$\text{STATIC: } \Pr_{\mathcal{Z}\leftrightarrow\mathbf{H}_0}[\text{FAIL} \wedge (\text{CORR}, t')] \leq t \cdot \mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}_{t'}} + \mathbf{Adv}^{\mathsf{GapOMDH}}_{\mathcal{R}_{t-t'}}$$

In the case of adaptive corruptions, we can calculate the probability of (COMP, i_k) combinatorially. Notice that the event FAIL implies that only servers in $\mathbf{Guess}_{k-1}$ were corrupted (any other corruption would cause execution to halt with event EXPOSED).

ADAPTIVE:

$$\begin{aligned}
\mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}_{k,t'}} &= \Big| \Pr_{\mathcal{Z}\leftrightarrow\mathbf{H}_{k-1}}[\text{FAIL} \wedge (\text{CORR}, t')] \cdot (1 - \frac{t'}{t-(k-1)}) \\
&\qquad - \Pr_{\mathcal{Z}\leftrightarrow\mathbf{H}_k}[\text{FAIL} \wedge (\text{CORR}, t')] \Big| \\
&= \Big| \Pr_{\mathcal{Z}\leftrightarrow\mathbf{H}_{k-1}}[\text{FAIL} \wedge (\text{CORR}, t')] \cdot (\frac{t-t'-k+1}{t-k+1}) \\
&\qquad - \Pr_{\mathcal{Z}\leftrightarrow\mathbf{H}_k}[\text{FAIL} \wedge (\text{CORR}, t')] \Big| \\
&\geq \Pr_{\mathcal{Z}\leftrightarrow\mathbf{H}_{k-1}}[\text{FAIL} \wedge (\text{CORR}, t')] \cdot (\frac{t-t'-k+1}{t-k+1}) \\
&\qquad - \Pr_{\mathcal{Z}\leftrightarrow\mathbf{H}_k}[\text{FAIL} \wedge (\text{CORR}, t')]
\end{aligned}$$

$$\begin{aligned}
&\Pr_{\mathcal{Z}\leftrightarrow\mathbf{H}_{k-1}}[\text{FAIL} \wedge (\text{CORR}, t')] \\
&\qquad \leq (\frac{t-k+1}{t-t'-k+1}) \cdot (\mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}_{k,t'}} + \Pr_{\mathcal{Z}\leftrightarrow\mathbf{H}_k}[\text{FAIL} \wedge (\text{CORR}, t')])
\end{aligned}$$

By solving this recurrence relation with the base case $\Pr_{\mathcal{Z}\leftrightarrow\mathbf{H}_{t-t'}}[\text{FAIL} \wedge (\text{CORR}, t')] = \mathbf{Adv}^{\mathsf{GapOMDH}}_{\mathcal{R}_{t-t'}}$ from above, we can set a bound on $\Pr_{\mathcal{Z}\leftrightarrow\mathbf{H}_0}[\text{FAIL} \wedge (\text{CORR}, t')]$. Here we use $\mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}_{t'}}$ to denote

$\max_{0 \le k \le t-t'} \mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}_{k,t'}}$.

$$
\begin{aligned}
\Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_0}[\text{FAIL} \wedge (\text{CORR}, t')] &\le \left(\mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}_{t'}} \cdot \sum_{i=1}^{t-t'} \prod_{j=1}^{i} \frac{t+1-j}{t-t'+1-j} \right) \\
&\quad + \left(\mathbf{Adv}^{\mathsf{GapOMDH}}_{\mathcal{R}_{t-t'}} \cdot \prod_{j=1}^{t-t'} \frac{t+1-j}{t-t'+1-j} \right) \\
&\le \left(\mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}_{t'}} \cdot (t-t') \cdot \prod_{j=1}^{t-t'} \frac{t+1-j}{t-t'+1-j} \right) \\
&\quad + \left(\mathbf{Adv}^{\mathsf{GapOMDH}}_{\mathcal{R}_{t-t'}} \cdot \prod_{j=1}^{t-t'} \frac{t+1-j}{t-t'+1-j} \right) \\
&= \left(\prod_{j=1}^{t-t'} \frac{t+1-j}{t-t'+1-j} \right) \\
&\quad \cdot ((t-t') \cdot \mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}_{t'}} + \mathbf{Adv}^{\mathsf{GapOMDH}}_{\mathcal{R}_{t-t'}}) \\
&= \frac{t!}{t'!(t-t')!} \cdot ((t-t') \cdot \mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}_{t'}} + \mathbf{Adv}^{\mathsf{GapOMDH}}_{\mathcal{R}_{t-t'}}) \\
&= \binom{t}{t'} \cdot ((t-t') \cdot \mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}_{t'}} + \mathbf{Adv}^{\mathsf{GapOMDH}}_{\mathcal{R}_{t-t'}}) \\
&\le \binom{t}{t'} \cdot (t \cdot \mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}_{t'}} + \mathbf{Adv}^{\mathsf{GapOMDH}}_{\mathcal{R}_{t-t'}})
\end{aligned}
$$

Finally we can combine our results to reach a bound on $\mathbf{Dist}^{\mathbf{G}_2,\mathbf{G}_3}_{\mathcal{Z}}$. Here we use $\mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}}$ to denote $\max_{0 \le t' \le t} \mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}_{t'}}$ and $\mathbf{Adv}^{\mathsf{GapOMDH}}_{\mathcal{R}}$ to denote $\max_{0 \le t' \le t} \mathbf{Adv}^{\mathsf{GapOMDH}}_{\mathcal{R}_{t-t'}}$.

STATIC:

$$
\begin{aligned}
\mathbf{Dist}^{\mathbf{G}_2,\mathbf{G}_3}_{\mathcal{Z}} &\le q_I \cdot \sum_{t'=0}^{t} \Pr_{\mathcal{Z} \leftrightarrow \mathbf{G}_3}[\text{FAIL} \wedge (\text{CORR}, t')] \\
&= q_I \cdot \sum_{t'=0}^{t} \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_0}[\text{FAIL} \wedge (\text{CORR}, t')] \\
&\le q_I \cdot \sum_{t'=0}^{t} t \cdot \mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}_{t'}} + \mathbf{Adv}^{\mathsf{GapOMDH}}_{\mathcal{R}_{t-t'}} \\
&\le q_I \cdot (t \cdot \mathbf{Adv}^{\mathsf{DDH}}_{\mathcal{Q}} + \mathbf{Adv}^{\mathsf{GapOMDH}}_{\mathcal{R}}) \cdot (t+1)
\end{aligned}
$$

ADAPTIVE:

$$\begin{aligned}
\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_2,\mathbf{G}_3} &\leq q_I \cdot \sum_{t'=0}^{t} \Pr_{\mathcal{Z} \leftrightarrow \mathbf{G}_3}[\text{FAIL} \wedge (\text{CORR}, t')] \\
&= q_I \cdot \sum_{t'=0}^{t} \frac{\binom{n}{t'}}{\binom{t}{t'}} \cdot \Pr_{\mathcal{Z} \leftrightarrow \mathbf{H}_0}[\text{FAIL} \wedge (\text{CORR}, t')] \\
&\leq q_I \cdot \sum_{t'=0}^{t} \frac{\binom{n}{t'}}{\binom{t}{t'}} \cdot \binom{t}{t'} \cdot (t \cdot \mathbf{Adv}_{\mathcal{Q}_{t'}}^{\mathsf{DDH}} + \mathbf{Adv}_{\mathcal{R}_{t-t'}}^{\mathsf{GapOMDH}}) \\
&\leq q_I \cdot (t \cdot \mathbf{Adv}_{\mathcal{Q}}^{\mathsf{DDH}} + \mathbf{Adv}_{\mathcal{R}}^{\mathsf{GapOMDH}}) \cdot \sum_{t'=0}^{t} \binom{n}{t'}
\end{aligned}$$

Game $\mathbf{G}_4$: H_3 returns PRF outputs. $\mathbf{G}_4$ is $\mathbf{G}_3$ with only one change: instead of randomly sampling outputs, all uses of H_3 return the PRF outputs provided by $\mathcal{F}_{\mathsf{tOPRF}}$. In the cases of adversarial queries to H_3 and honest client evaluations, the process for querying these outputs from $\mathcal{F}_{\mathsf{tOPRF}}$ via toprf.rcvcomplete is already in place from $\mathbf{G}_3$ (Figures 15 and 14). In the case of eval-during-init, the honest user interface of $\mathcal{F}_{\mathsf{tOPRF}}$ is simply used (Figure 10).

In the previous game, $H_3(x, u)$ outputs a randomly sampled value corresponding to (x, u). In this game, it outputs a randomly sampled value corresponding to (x, sid), where sid is a PRF instance identifier that maps to (x, u). In the cases of adversarial queries to H_3 and honest client evaluations, this mapping is determined by FindEvalset. In the case of eval-during-init, the mapping is immediate. If there exists a perfect bijection between pairs (x, u) and pairs (x, sid), then $\mathcal{Z}$'s views of $\mathbf{G}_3$ and $\mathbf{G}_4$ are identical. Consider the two directions of this bijection:

1. Suppose there exist x, u, sid, and sid' such that $H_3(x, u)$ maps to both (x, sid) and (x, sid'). The PRF instances identified by sid and sid' use secret keys $k(0)$ and $k'(0)$, respectively.
   In all cases (i.e. eval-during-init, adversarial query to H_3, and honest client evaluation), $H_3(x, u)$ can only map to sid if $u^{1/\tau} = g^{k(0)}$ where $H_1(x) = g^{\tau}$ (and it can only map to sid' if $u^{1/\tau} = g^{k'(0)}$). Therefore, it follows that $g^{k(0)} = g^{k'(0)}$ and further that $k(0) = k'(0)$. Per the game change to $\mathbf{G}_1$, collisions never occur during honest key generation. FindEvalset can also never cause a key collision, since new (adversarially chosen) keys are only recorded if an existing match cannot be found. Thus, it must be the case that $sid = sid'$.
2. Suppose there exist x, u, u', and sid such that $H_3(x, u)$ and $H_3(x, u')$ both map to (x, sid).
   Eval-during-init can never create such a collison (because different instances use different sids). Thus, the only possibility to consider is that two calls $\mathsf{FindEvalset}(u^{1/\tau})$ and $\mathsf{FindEvalset}(u'^{1/\tau})$ both map to sid. In the case of

adversarial query to H_3, this u is provided directly by the adversary; in the case of honest client evaluation, it is calculated as $b^{1/r}$. In both cases, $H_1(x) = g^\tau$. Per the procedure of $\mathsf{FindEvalset}$, it follows that both $u^{1/\tau} = g^{k(0)}$ and $u'^{1/\tau} = g^{k(0)}$, where $k(0)$ is the secret key for the PRF instance identified by *sid*. Therefore, $u = u'$.

Thus, this bijection between pairs (x, u) and pairs (x, sid) does indeed hold.

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_3,\mathbf{G}_4} = 0$$

Game $\mathbf{G}_5$: Honest client evaluation doesn't use $H_1(x)$. $\mathbf{G}_5$ is $\mathbf{G}_4$ with only one change: after picking random exponent r, the honest client evaluation procedure simply sets $a := g^r$ rather than $a := H_1(x)^r$. Consequently, it later calls $\mathsf{FindEvalset}(b^{1/r})$ rather than $\mathsf{FindEvalset}((b^{1/r})^{1/\tau})$ (Figure 14).

$\mathcal{Z}$'s views of $\mathbf{G}_4$ and $\mathbf{G}_5$ are identical. In both games a is a uniformly random group element.

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_4,\mathbf{G}_5} = 0$$

Game $\mathbf{G}_6$: Transcript integrity is enforced. $\mathbf{G}_6$ is $\mathbf{G}_5$ with only one change: the final step of honest client evaluation aborts if all $t+1$ servers in the evaluation set $\mathbf{C}'$ have recorded a PRF instance identifier *sid* different from the identifier sid^* of the instance that actually underlies this evaluation, yet they agree on the communication transcript observed by the client. This abortion does not take place if the initial party P_0 was dishonest during the initialization of the instance identified by *sid* (Figure 10).

Clearly, $\mathcal{Z}$'s views of $\mathbf{G}_5$ and $\mathbf{G}_6$ are identical unless this abortion happens. In order for it to occur, each server $\mathbf{S}_{sid}[i]$ in $\mathbf{C}'$ must have received a message $(sid, i, ssid'_{\mathsf{U}}, a)$ and responded with $ssid_{\mathsf{S}}$ and $b_i := a^{k_i} \cdot H_2(ssid_{\mathsf{S}}, a)^{z_i}$ where k_i and z_i are the secret shares held by $\mathbf{S}_{sid}[i]$ and the subsession identifier $ssid_{\mathsf{S}}$ is common between all servers. Since P_0 was honest during initialization, the values k_i must share the PRF secret key $k(0)$, and the values z_i must share zero. Therefore, the client's computation of $b := \prod_{i \in \mathbf{C}} b_i^{\lambda_i}$ must have yielded $a^{k(0)}$ as expected (λ_i is the Lagrange interpolation coefficient for index i and index set $\mathbf{C}$). Since a is defined as g^r, the client finally determines sid^* by calling $\mathsf{FindEvalset}(g^{k(0)})$. Per the procedure of $\mathsf{FindEvalset}$, the mapping of $g^{k(0)}$ to sid^* implies that $k(0)$ and the secret key $k^*(0)$ of PRF instance sid^* are equal. Since there are guaranteed to be no key collisions, it follows that $sid = sid^*$. Thus, the hypothetical abortion introduced by this game change is impossible.

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_5,\mathbf{G}_6} = 0$$

Game $\mathbf{G}_7$: The simulated world. The change from $\mathbf{G}_6$ to $\mathbf{G}_7$ is purely conceptual. All interactions between the simulator and the honest parties now occur through the interface of $\mathcal{F}_{\mathsf{tOPRF}}$, so one can imagine the monolithic simulator now cleanly split into the two components $\mathcal{F}_{\mathsf{tOPRF}}$ and $\mathsf{SIM}_{\mathsf{3HashTDH}}$.

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_6,\mathbf{G}_7} = 0$$

Summing up the distinguishing advantage bounds for each incremental game change yields an overall bound on $\mathcal{Z}$'s distinguishing advantage between the real and simulated worlds.

$$\text{STATIC:} \quad \mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_0,\mathbf{G}_7} \leq \frac{q_I^2}{m} + q_I \cdot (t \cdot \mathbf{Adv}_{\mathcal{Q}}^{\mathsf{DDH}} + \mathbf{Adv}_{\mathcal{R}}^{\mathsf{GapOMDH}}) \cdot (t+1)$$

$$\text{ADAPTIVE:} \quad \mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_0,\mathbf{G}_7} \leq \frac{q_I^2}{m} + q_I \cdot (t \cdot \mathbf{Adv}_{\mathcal{Q}}^{\mathsf{DDH}} + \mathbf{Adv}_{\mathcal{R}}^{\mathsf{GapOMDH}}) \cdot \sum_{t'=0}^{t} \binom{n}{t'}$$

$m = |\mathbb{G}|$ is an exponential function of the security parameter. If $\mathcal{Z}$ is efficient, then q_I is a polynomial function of the security parameter. Furthermore, $\mathbf{Adv}_{\mathcal{Q}}^{\mathsf{DDH}}$ and $\mathbf{Adv}_{\mathcal{R}}^{\mathsf{GapOMDH}}$ are both negligible functions of the security parameter under the DDH and GapOMDH assumptions, respectively. Thus, in the case of static corruptions, the distinguishing advantage of any efficient $\mathcal{Z}$ between the real and simulated worlds is negligible. Under the additional assumption that $\binom{n}{t'}$ is polynomial for all $0 \leq t' \leq t$, the same is true in the case of adaptive corruptions.

Protocol 3HashTDH realizes $\mathcal{F}_{\mathsf{tOPRF}}$.

□

C Proof of tOPRF-atPAKE Security

In this section we prove Theorem 3 from Section 5, i.e. security of protocol $\Pi_{\mathsf{tOPRF-atPAKE}}$.

Proof. For any adversary $\mathcal{A}^*$, we construct simulator $\mathsf{SIM}_{\mathsf{tOPRF-atPAKE}}$ as shown in Figures 16, 17, and 18. Without loss of generality, we assume that $\mathcal{A}^*$ is a "dummy" adversary that merely passes messages to and from the environment $\mathcal{Z}$.

We now show that, for any efficient (i.e. PPT) $\mathcal{Z}$, the distinguishing advantage of $\mathcal{Z}$ between the real and simulated worlds is negligible. The argument uses only a single game change from the real world $\mathbf{G}_0$ to the simulated world $\mathbf{G}_1$. By $\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_0,\mathbf{G}_1}$ we denote distinguisher $\mathcal{Z}$'s distinguishing advantage between world $\mathbf{G}_0$ and world $\mathbf{G}_1$. Specifically, $\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_0,\mathbf{G}_1} = |\Pr_{\mathcal{Z}\leftrightarrow\mathbf{G}_0}[\mathcal{Z} \text{ outputs } 1] - \Pr_{\mathcal{Z}\leftrightarrow\mathbf{G}_1}[\mathcal{Z} \text{ outputs } 1]|$.

Game $\mathbf{G}_0$: The real world. The distinguisher $\mathcal{Z}$ interacts with $\Pi_{\mathsf{tOPRF-atPAKE}}$ (Figure 8) in the role of the honest parties and in the role of the adversary.

Game $\mathbf{G}_1$: The simulated world. By inspection, $\mathsf{SIM}_{\mathsf{tOPRF-atPAKE}}$ in interaction with $\mathcal{F}_{\mathsf{atPAKE}}$ behaves identically to the real world protocol, except in the case of a rare PRF collision event. Therefore, the probability that $\mathcal{Z}$ can

Notation

Initially $\mathsf{tx}[sid_\mathsf{A}, i, ssid_\mathsf{A}] := 0$ for all sid_A, i, and $ssid_\mathsf{A}$, and $\mathsf{cflag}[sid_\mathsf{T}, j] :=$ UNCOMPROMISED for all sid_T and j. Also, $F_{sid_\mathsf{A}}(x)$ is undefined for all sid_A and x. When first referenced, the functionality assigns $F_{sid_\mathsf{A}}(x) \leftarrow_\$ \{0,1\}^\tau$.

Initialization Phase

1. On $(\mathsf{atpake.userinit}, \mathsf{U}, sid_\mathsf{A}, sid_\mathsf{T})$ from $\mathcal{F}_{\mathsf{atPAKE}}$ for honest U:
   - save $(\mathsf{userinit}, \mathsf{U}, sid_\mathsf{A}, sid_\mathsf{T})$
   - send $(\mathsf{toprf.init}, sid_\mathsf{A}, \mathsf{U})$ to $\mathcal{A}^*$
   - for every $j \in \{1, \ldots, |\mathbf{S}_{sid_\mathsf{T}}|\}$, send $(\mathsf{channel.send}, (sid_\mathsf{A}||sid_\mathsf{T}||j), \mathsf{U}, \mathbf{S}_{sid_\mathsf{T}}[j], \tau)$ to $\mathcal{A}^*$
2. On $(\mathsf{toprf.init}, sid_\mathsf{A}, \mathsf{pw}_1^*, \ldots, \mathsf{pw}_k^*)$ from $\mathcal{A}^*$ on behalf of some $\mathsf{U} \in \mathbf{Corr}$:
   - save $(\mathsf{userinit}, \mathsf{U}, sid_\mathsf{A}, \perp)$ marked ADVERSARIAL
   - send $(\mathsf{toprf.init}, sid_\mathsf{A}, \mathsf{U})$ to $\mathcal{A}^*$
   - send $(\mathsf{toprf.initeval}, sid_\mathsf{A}, F_{sid_\mathsf{A}}(\mathsf{pw}_1^*), \ldots, F_{sid_\mathsf{A}}(\mathsf{pw}_k^*))$ to U
   - if $\mathsf{U} \neq \mathcal{A}^*$, choose any sid_T and pw and send $(\mathsf{atpake.userinit}, sid_\mathsf{A}, sid_\mathsf{T}, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{atPAKE}}$ on behalf of U
3. On $(\mathsf{atpake.auxinit}, sid_\mathsf{A}, i, \mathsf{U}')$ from $\mathcal{F}_{\mathsf{atPAKE}}$:
   - save $(\mathsf{auxinit}, sid_\mathsf{A}, i, \mathsf{U}')$ marked PENDING
4. On $(\mathsf{toprf.sinit}, sid_\mathsf{A}, i, \mathsf{U}')$ from $\mathcal{A}^*$ where $\mathbf{S}_{sid_\mathsf{A}}[i] \in \mathbf{Corr}$:
   - if $\mathbf{S}_{sid_\mathsf{A}}[i] \neq \mathcal{A}^*$, send $(\mathsf{atpake.auxinit}, sid_\mathsf{A}, i, \mathsf{U}')$ to $\mathcal{F}_{\mathsf{atPAKE}}$
   - otherwise, save $(\mathsf{auxinit}, sid_\mathsf{A}, i, \mathsf{U})$ marked PENDING
5. On $(\mathsf{atpake.targetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, \mathsf{U}')$ from $\mathcal{F}_{\mathsf{atPAKE}}$:
   - save $(\mathsf{targetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, \mathsf{U}')$ marked PENDING
6. On $(\mathsf{toprf.fininit}, sid_\mathsf{A}, i)$ from $\mathcal{A}^*$:
   - retrieve $\mathsf{urec} = (\mathsf{userinit}, [\mathsf{U}], sid_\mathsf{A}, [sid_\mathsf{T}])$ (abort if not found)
   - retrieve $(\mathsf{auxinit}, sid_\mathsf{A}, i, \mathsf{U})$ marked PENDING and change its mark to COMPLETED (abort if not found)
   - save $(\mathsf{auxiliaryfile}, sid_\mathsf{A}, i)$ and mark it ADVERSARIAL if urec is ADVERSARIAL
   - send $(\mathsf{atpake.finishauxiliaryinit}, sid_\mathsf{A}, i)$ to $\mathcal{F}_{\mathsf{atPAKE}}$
7. On $(\mathsf{channel.deliver}, (sid_\mathsf{A}||sid_\mathsf{T}||j), \mathsf{U}, \mathbf{S}_{sid_\mathsf{T}}[j])$ from $\mathcal{A}^*$:
   - retrieve $\mathsf{trec} = (\mathsf{targetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, \mathsf{U})$ marked PENDING (abort if not found)
   - if $(\mathsf{channel.send}, (sid_\mathsf{A}||sid_\mathsf{T}||j), \mathbf{S}_{sid_\mathsf{T}}[j], [\mathsf{rw}_j^*])$ was previously sent by $\mathcal{A}^*$ on behalf of $\mathsf{U} \in \mathbf{Corr}$:
     - if $\exists\ (sid_\mathsf{A}^*, \mathsf{pw}^*)$ s.t. $\mathsf{rw}_j^* = \mathsf{KDF}(F_{sid_\mathsf{A}^*}(\mathsf{pw}^*), \mathbf{S}_{sid_\mathsf{T}}[i])$ then send $(\mathsf{atpake.finishtargetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, sid_\mathsf{A}^*, \mathsf{pw}^*)$ to $\mathcal{F}_{\mathsf{atPAKE}}$ and save $(\mathsf{targetfile}, sid_\mathsf{A}^*, sid_\mathsf{T}, j, \mathsf{pw}^*, \mathsf{rw}_j^*)$
     - otherwise, send $(\mathsf{atpake.finishtargetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, \perp, \perp)$ to $\mathcal{F}_{\mathsf{atPAKE}}$ and save $(\mathsf{targetfile}, \perp, sid_\mathsf{T}, j, \perp, \mathsf{rw}_j^*)$
   - otherwise, retrieve record $(\mathsf{userinit}, \mathsf{U}, sid_\mathsf{A}, sid_\mathsf{T})$ (abort if not found), send $(\mathsf{atpake.finishtargetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, \perp, \perp)$ to $\mathcal{F}_{\mathsf{atPAKE}}$, and save $(\mathsf{targetfile}, sid_\mathsf{A}, sid_\mathsf{T}, j, \perp, \perp)$
   - mark trec COMPLETED

Fig. 16. Simulator $\mathsf{SIM}_{\mathsf{tOPRF-atPAKE}}$ for protocol $\Pi_{\mathsf{tOPRF-atPAKE}}$, part 1: Notation and Initialization

Party Corruption

8. On $(\mathsf{toprf.corrupt}, \mathsf{P})$ from $\mathcal{A}^*$:
   - set $\mathbf{Corr} := \mathbf{Corr} \cup \{\mathsf{P}\}$
   - send $(\mathsf{atpake.corrupt}, \mathsf{P})$ to $\mathcal{F}_{\mathsf{atPAKE}}$
9. On $(\mathsf{sapake.stealpwdfile}, (sid_{\mathsf{T}}||j))$ from $\mathcal{A}^*$:
   - retrieve $(\mathsf{targetfile}, [sid_{\mathsf{A}}], sid_{\mathsf{T}}, j, [\mathsf{pw}, \mathsf{rw}_j])$ and mark it COMPR (abort if not found)
   - send $(\mathsf{atpake.stealtargetfile}, sid_{\mathsf{T}}, j)$ to $\mathcal{F}_{\mathsf{atPAKE}}$

Authentication Phase (I)

10. On $(\mathsf{atpake.usersession}, \mathsf{U}', sid_{\mathsf{A}}, sid_{\mathsf{T}}, j, ssid)$ from $\mathcal{F}_{\mathsf{atPAKE}}$:
    - send $(\mathsf{toprf.eval}, sid_{\mathsf{A}}, ssid, \mathsf{U}')$ to $\mathcal{A}^*$
    - save $(\mathsf{session}, \mathsf{U}', \mathbf{S}_{sid_{\mathsf{T}}}[j], sid_{\mathsf{A}}, sid_{\mathsf{T}}, j, ssid, \perp, \perp)$ marked PRELIM
11. On $(\mathsf{toprf.eval}, sid_{\mathsf{A}}, ssid, \mathsf{pw}^*)$ from $\mathcal{A}^*$:
    - send $(\mathsf{toprf.eval}, sid_{\mathsf{A}}, ssid, \mathcal{A}^*)$ to $\mathcal{A}^*$
    - save $(\mathsf{eval}, sid_{\mathsf{A}}, ssid, \mathsf{pw}^*)$ marked FRESH
12. On $(\mathsf{atpake.auxsession}, sid_{\mathsf{A}}, i, ssid_{\mathsf{A}})$ from $\mathcal{F}_{\mathsf{atPAKE}}$:
    - send $(\mathsf{toprf.sndrcomplete}, sid_{\mathsf{A}}, i, ssid_{\mathsf{A}})$ to $\mathcal{A}^*$
    - set $\mathsf{tx}[sid_{\mathsf{A}}, i, ssid_{\mathsf{A}}]{+}{+}$
13. On $(\mathsf{toprf.sndrcomplete}, sid_{\mathsf{A}}, i, ssid_{\mathsf{A}})$ from $\mathcal{A}^*$:
    - retrieve $\mathsf{rec} = (\mathsf{auxiliaryfile}, sid_{\mathsf{A}}, i)$ (abort if not found)
    - if rec is marked ADVERSARIAL, send $(\mathsf{toprf.sndrcomplete}, sid_{\mathsf{A}}, i, ssid_{\mathsf{A}})$ to $\mathcal{A}^*$ and set $\mathsf{tx}[sid_{\mathsf{A}}, i, ssid_{\mathsf{A}}]{+}{+}$; otherwise, send $(\mathsf{atpake.auxsession}, sid_{\mathsf{A}}, i, ssid_{\mathsf{A}})$ to $\mathcal{F}_{\mathsf{atPAKE}}$
14. On $(\mathsf{toprf.rcvcomplete}, sid_{\mathsf{A}}, ssid, sid^*_{\mathsf{A}}, ssid^*_{\mathsf{A}}, \mathbf{C})$ from $\mathcal{A}^*$ s.t. $\exists$ record $\mathsf{urec} = (\mathsf{session}, [\mathsf{U}', \mathbf{S}_{sid_{\mathsf{T}}}[j]], sid_{\mathsf{A}}, [sid_{\mathsf{T}}, j], ssid, \perp, \perp)$ marked PRELIM:
    - abort if $\exists_{i \in \mathbf{C}}\ \mathsf{tx}[sid^*_{\mathsf{A}}, i, ssid^*_{\mathsf{A}}] = 0$, else $\forall_{i \in \mathbf{C}}$ set $\mathsf{tx}[sid^*_{\mathsf{A}}, i, ssid^*_{\mathsf{A}}]{-}{-}$
    - change urec's mark to FRESH
    - update field sid_{A} in urec to sid^*_{A}
    - if record $(\mathsf{userinit}, [\mathsf{U}], sid^*_{\mathsf{A}}, [sid'_{\mathsf{T}}])$ is marked ADVERSARIAL, send $(\mathsf{atpake.auxactive}, \mathsf{U}', ssid)$ to $\mathcal{F}_{\mathsf{atPAKE}}$; otherwise, send $(\mathsf{atpake.auxproceed}, \mathsf{U}', ssid, sid^*_{\mathsf{A}}, ssid^*_{\mathsf{A}}, \mathbf{C})$
    - send $(\mathsf{sapake.usrsession}, (sid_{\mathsf{T}}||j), ssid, \mathsf{U}', \mathbf{S}_{sid_{\mathsf{T}}}[j])$ to $\mathcal{A}^*$
15. On $(\mathsf{toprf.rcvcomplete}, sid_{\mathsf{A}}, ssid, sid^*_{\mathsf{A}}, ssid^*_{\mathsf{A}}, \mathbf{C})$ from $\mathcal{A}^*$ s.t. $\exists$ record $\mathsf{urec} = (\mathsf{eval}, sid_{\mathsf{A}}, ssid, [\mathsf{pw}^*])$ marked FRESH:
    - abort if $\exists_{i \in \mathbf{C}}\ \mathsf{tx}[sid^*_{\mathsf{A}}, i, ssid^*_{\mathsf{A}}] = 0$, else $\forall_{i \in \mathbf{C}}\ \mathsf{tx}[sid^*_{\mathsf{A}}, i, ssid^*_{\mathsf{A}}]{-}{-}$
    - change urec's mark to COMPLETED
    - send $(\mathsf{toprf.eval}, sid_{\mathsf{A}}, ssid, F_{sid^*_{\mathsf{A}}}(\mathsf{pw}^*))$ to $\mathcal{A}^*$
    - $\forall_{i \in \mathbf{C}}$ send $(\mathsf{atpake.offlinetestpwd}, sid^*_{\mathsf{A}}, i, ssid^*_{\mathsf{A}}, \perp, \perp, \mathsf{pw}^*)$ to $\mathcal{F}_{\mathsf{atPAKE}}$
    - save $(\mathsf{completedeval}, sid^*_{\mathsf{A}}, \mathsf{pw}^*, ssid^*_{\mathsf{A}}, F_{sid^*_{\mathsf{A}}}(\mathsf{pw}^*))$

Fig. 17. Simulator $\mathsf{SIM}_{\mathsf{tOPRF-atPAKE}}$ for protocol $\Pi_{\mathsf{tOPRF-atPAKE}}$, part 2: Authentication Phase (I)

Authentication Phase (II)

16. On $(\mathsf{atpake.targetsession}, sid_\mathsf{T}, j, \mathsf{U}', ssid)$ from $\mathcal{F}_{\mathsf{atPAKE}}$:
    - retrieve $(\mathsf{targetfile}, [sid_\mathsf{A}], sid_\mathsf{T}, j, [\mathsf{pw}, \mathsf{rw}_j])$
    - send $(\mathsf{sapake.svrsession}, (sid_\mathsf{T}||j), ssid, \mathsf{U}', \mathbf{S}_{sid_\mathsf{T}}[j])$ to $\mathcal{A}^*$
    - save $(\mathsf{session}, \mathbf{S}_{sid_\mathsf{T}}[j], \mathsf{U}', sid_\mathsf{A}, sid_\mathsf{T}, j, ssid, \mathsf{pw}, \mathsf{rw}_j)$ marked FRESH
17. On $(\mathsf{sapake.interrupt}, (sid_\mathsf{T}||j), ssid, \mathbf{S}_{sid_\mathsf{T}}[j])$ from $\mathcal{A}^*$:
    - send $(\mathsf{atpake.interrupt}, sid_\mathsf{T}, j, ssid)$ to $\mathcal{F}_{\mathsf{atPAKE}}$
18. On $(\mathsf{sapake.testpwd}, (sid_\mathsf{T}||j), ssid, \mathsf{P}, \mathsf{rw}_j^*)$ from $\mathcal{A}^*$:
    - retrieve $\mathsf{rec} = (\mathsf{session}, \mathsf{P}, [\mathsf{P}', sid_\mathsf{A}], sid_\mathsf{T}, j, ssid, [\mathsf{pw}', \mathsf{rw}_j'])$ (abort if not found)
    - if $\mathsf{rw}_j' \neq \bot$ and $\mathsf{rw}_j^* = \mathsf{rw}_j'$, send $(\mathsf{atpake.testpwd}, \mathsf{P}, ssid, \mathsf{pw}')$ to $\mathcal{F}_{\mathsf{atPAKE}}$
    - else if $\mathsf{rw}_j' \neq \bot$ but $\mathsf{rw}_j^* \neq \mathsf{rw}_j'$, send $(\mathsf{atpake.testpwd}, \mathsf{P}, ssid, (\mathsf{pw}'||0))$
    - else if $\exists\ \mathsf{pw}^*$ s.t. $\mathsf{rw}_j^* = \mathsf{KDF}(F_{sid_\mathsf{A}}(\mathsf{pw}^*), \mathbf{S}_{sid_\mathsf{T}}[j])$, send $(\mathsf{atpake.testpwd}, \mathsf{P}, ssid, \mathsf{pw}^*)$
    - else, send $(\mathsf{atpake.testpwd}, \mathsf{P}, ssid, \bot)$
    - in any case, forward the response (if any) to $\mathcal{A}^*$
    - if the response is "correct guess", mark rec COMPR and set $\mathsf{cflag}[sid_\mathsf{T}, j] :=$ COMPR
19. On $(\mathsf{sapake.impersonate}, (sid_\mathsf{T}||j), ssid)$ from $\mathcal{A}^*$:
    - send $(\mathsf{atpake.impersonate}, sid_\mathsf{T}, j, ssid)$ to $\mathcal{F}_{\mathsf{atPAKE}}$
    - forward the response (if any) to $\mathcal{A}^*$
    - if the response is "correct guess", mark rec COMPR
20. On $(\mathsf{sapake.newkey}, (sid_\mathsf{T}||j), ssid, \mathsf{P}, K^*)$ from $\mathcal{A}^*$:
    - retrieve $\mathsf{rec} = (\mathsf{session}, \mathsf{P}, [\mathsf{P}', sid_\mathsf{A}], sid_\mathsf{T}, j, ssid, [\mathsf{pw}, \mathsf{rw}_j])$ not marked PRELIM (abort if not found)
    - if rec is not COMPR and $\mathsf{cflag}[sid_\mathsf{T}, j] =$ UNCOMPROMISED, set $K^* \leftarrow_\$ \{0,1\}^\tau$
    - send $(\mathsf{atpake.newkey}, \mathsf{P}, ssid, K^*)$ to $\mathcal{F}_{\mathsf{atPAKE}}$
    - mark rec COMPLETED
21. On $(\mathsf{sapake.testabort}, (sid_\mathsf{T}||j), ssid, \mathsf{U}')$ from $\mathcal{A}^*$:
    - send $(\mathsf{atpake.testabort}, \mathsf{U}', sid_\mathsf{T}, j, ssid)$ to $\mathcal{F}_{\mathsf{atPAKE}}$
    - forward the response (if any) to $\mathcal{A}^*$

Offline Password Tests

22. On $(\mathsf{sapake.offlinetestpwd}, (sid_\mathsf{T}||j), \mathsf{rw}_j^*)$ from $\mathcal{A}^*$:
    - retrieve $(\mathsf{targetfile}, [sid_\mathsf{A}], sid_\mathsf{T}, j, [\mathsf{pw}, \mathsf{rw}_j])$
    - if $\mathsf{rw}_j^* = \mathsf{rw}_j \neq \bot$, send "correct guess" to $\mathcal{A}^*$
    - else if $\exists$ record $(\mathsf{completedeval}, sid_\mathsf{A}, [\mathsf{pw}^*, ssid_\mathsf{A}], \mathsf{rw}^*)$ s.t. $\mathsf{rw}_j^* = \mathsf{KDF}(\mathsf{rw}^*, \mathbf{S}_{sid_\mathsf{T}}[j])$, send $(\mathsf{atpake.offlinetestpwd}, sid_\mathsf{A}, \bot, ssid_\mathsf{A}, sid_\mathsf{T}, j, \mathsf{pw}^*)$ to $\mathcal{F}_{\mathsf{atPAKE}}$ and forward the response to $\mathcal{A}^*$
    - else, send "wrong guess" to $\mathcal{A}^*$
    - in any case, if the response is "correct guess", set $\mathsf{cflag}[sid_\mathsf{T}, j] :=$ COMPR

Fig. 18. Simulator $\mathsf{SIM}_{\mathsf{tOPRF-atPAKE}}$ for protocol $\Pi_{\mathsf{tOPRF-atPAKE}}$, part 3: Authentication Phase (II) and Offline Password Tests

distinguish between the real and simulated worlds is upper-bounded by the probability that such an event occurs.

These collision events can be classified into 3 types:

1. For some target server instance identified by (sid_{T}, j), there exist two non-equal pairs $(sid_{\mathsf{A}}, \mathsf{pw}) \neq (sid'_{\mathsf{A}}, \mathsf{pw}')$ such that $\mathsf{rw}_j = \mathsf{KDF}(F_{sid_{\mathsf{A}}}(\mathsf{pw}), \mathbf{S}_{sid_{\mathsf{T}}}[j]) = \mathsf{rw}'_j = \mathsf{KDF}(F_{sid'_{\mathsf{A}}}(\mathsf{pw}'), \mathbf{S}_{sid_{\mathsf{T}}}[j])$.
   In the real world, $\mathcal{F}_{\mathsf{saPAKE}}$ considers only rw_j equality when evaluating the success of a honest session or adversarial attack. In the simulated world, the equality of $(sid_{\mathsf{A}}, \mathsf{pw})$ pairs is used instead. Therefore, if there exists a session where non-equal $(sid_{\mathsf{A}}, \mathsf{pw})$ pairs actually collide to the same rw, the real world will treat that session a success while the simulated world treats it as a failure. Specifically, this disparity can occur as the result of a testpwd, impersonate, newkey, testabort, or offlinetestpwd message.
   For any (sid_{T}, j), the rw_j mapped to by each $(sid_{\mathsf{A}}, \mathsf{pw})$ is completely random. Therefore, $\mathcal{Z}$'s probability of finding a collision can be bounded:

$$\Pr[\text{collision type \#1}] \leq q_{\mathsf{T}} \cdot \frac{\mathsf{q}_{\mathsf{eval}}^2}{2^\tau}$$

   Here, q_{T} is the number of target server instances, q_{eval} is the number of tOPRF evaluations, and τ is the output length of PRFs F and KDF.
2. For some rw_j^* used in an (online or offline) password-guessing attack against target server instance (sid_{T}, j) initialized with $\mathsf{rw}_j = \mathsf{KDF}(F_{sid_{\mathsf{A}}}(\mathsf{pw}), \mathbf{S}_{sid_{\mathsf{T}}}[j])$, $\mathsf{rw}_j^* = \mathsf{rw}_j$ even though the adversary did not derive rw_j^* from any $(sid_{\mathsf{A}}^*, \mathsf{pw}^*)$ pair.
   In the real world, $\mathcal{F}_{\mathsf{saPAKE}}$ considers only rw_j equality when evaluating the success of a password-guessing attack. In the simulated world, the equality of $(sid_{\mathsf{A}}, \mathsf{pw})$ pairs is used instead, and it is assumed that the adversary never succeeds in a guess using a rw_j^* that is not derived from a $(sid_{\mathsf{A}}, \mathsf{pw})$ pair. Therefore, if the adversary "gets lucky" and guesses the correct rw_j directly, the real world will treat the attack as a success while the simulated world treats it as a failure. Specifically, this disparity can occur as the result of a testpwd or offlinetestpwd message.
   For any target server, the rw_j mapped to by the $(sid_{\mathsf{A}}, \mathsf{pw})$ with which it was initialized is completely random. Therefore $\mathcal{Z}$'s probability of "getting lucky" in this way can be bounded:

$$\Pr[\text{collision type \#2}] \leq \frac{\mathsf{q}_{\mathsf{test}}}{2^\tau}$$

   Here, q_{test} is the number of online and offline password-guessing attacks against $\mathcal{F}_{\mathsf{saPAKE}}$ and τ is the output length of PRFs F and KDF.
3. For some rw_j^* dishonestly sent to an initializing target server identified by (sid_{T}, j), there later is found a pair $(sid_{\mathsf{A}}, \mathsf{pw})$ such that $\mathsf{rw}_j^* = \mathsf{KDF}(F_{sid_{\mathsf{A}}}(\mathsf{pw}), \mathbf{S}_{sid_{\mathsf{T}}}[j])$ even though the adversary did not derive rw_j^* from any $(sid_{\mathsf{A}}^*, \mathsf{pw}^*)$ pair.
   In both the real and simulated worlds, a dishonest user can cause a target server to initialize using some rw_j^* that was not derived from a $(sid_{\mathsf{A}}, \mathsf{pw})$

pair. In the real world, it is possible that an honest (or dishonest) user later "gets lucky" and authenticates to that server using a $(sid_{\mathsf{A}}, \mathsf{pw})$ pair that just happens to map to that rw_j^*. In the simulated world, it is impossible for any honest user to ever authenticate to that target server. Specifically, this disparity can occur on target server instances initialized with TAMPERED $\mathsf{targetfiles}$.

For any (sid_{T}, j), the rw_j mapped to by $(sid_{\mathsf{A}}, \mathsf{pw})$ is completely random. Therefore, $\mathcal{Z}$'s probability of finding a "getting lucky" in this way can be bounded:

$$\Pr[\text{collision type \#3}] \leq \frac{\mathsf{q}_{\mathsf{T}}^* \cdot \mathsf{q}_{\mathsf{eval}}}{2^\tau}$$

Here, q_{T}^* is the number of dishonestly initialized target server instances, q_{eval} is the number of tOPRF evaluations, and τ is the output length of PRFs F and KDF.

Summing up these probabilities yields an overall bound on $\mathcal{Z}$'s distinguishing advantage between the real and simulated worlds.

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_0,\mathbf{G}_1} \leq \frac{q_{\mathsf{T}} \cdot q_{\mathsf{eval}}^2 + q_{\mathsf{test}} + q_{\mathsf{T}}^* \cdot q_{\mathsf{eval}}}{2^\tau}$$

τ is the security parameter. If $\mathcal{Z}$ is efficient, then q_{T}, q_{eval}, q_{test}, and q_{T}^* are all polynomial functions of the security parameter. Thus, the distinguishing advantage of any efficient $\mathcal{Z}$ is negligible.

Protocol $\Pi_{\mathsf{tOPRF-atPAKE}}$ realizes $\mathcal{F}_{\mathsf{atPAKE}}$.

□

D Strong Asymmetric PAKE Functionality

For completeness we include in Figure 19 the *strong asymmetric PAKE* (saPAKE) functionality, $\mathcal{F}_{\mathsf{saPAKE}}$ [44].

We make two edits in this functionality, relative to how it appears in [44], one simplification and one correction. The simplification is in the $\mathsf{testabort}$ interface, which we restrict only to user U, because that's the only side which can abort mid-protocol in OPAQUE, and only to client's sessions marked FRESH. The correction is that the server password file does not keep the identity U because this moniker identifies a machine which can execute the saPAKE protocol on the user's behalf, but this identity is flexible: Indeed the user can authenticate using his/her password from an arbitrary network entity. Instead, the intended communication counterparty can be dynamically decided by the environment, and it is passed to the server in the $\mathsf{serversession}$ query.

E atPAKE Construction from Non-Strong aPAKE

In this section we present our atPAKE variant which uses ordinary asymmetric password-authenticated key exchange (aPAKE) as a building block, rather than

Password Registration

- On $(\mathsf{storepwdfile}, sid, \mathsf{pw})$ from S, if this is the first such message, record $\langle \text{FILE}, sid, \mathsf{S}, \mathsf{pw} \rangle$, mark it UNCOMPROMISED, set $\mathsf{cflag} :=$ UNCOMPROMISED.

Stealing Password Data

- On $(\mathsf{stealpwdfile}, sid)$ from $\mathcal{A}^*$, if there is no record $\langle \text{FILE}, sid, \mathsf{S}, \mathsf{pw} \rangle$, return "no password file" to $\mathcal{A}^*$. Otherwise, if the record is marked UNCOMPROMISED, mark it COMPROMISED; regardless, return "password file stolen" to $\mathcal{A}^*$.
- On $(\mathsf{offlinetestpwd}, sid, \mathsf{pw}^*)$ from $\mathcal{A}^*$, if there is a record $\langle \text{FILE}, sid, \mathsf{S}, \mathsf{pw} \rangle$ marked COMPROMISED, do: if $\mathsf{pw}^* = \mathsf{pw}$, return "correct guess" to $\mathcal{A}^*$ and set $\mathsf{cflag} :=$ COMPROMISED; otherwise return "wrong guess."

Password Authentication

- On $(\mathsf{usersession}, sid, ssid, \mathsf{S}, \mathsf{pw}')$ from U, send $(\mathsf{usersession}, sid, ssid, \mathsf{U}, \mathsf{S})$ to $\mathcal{A}^*$. Also, if this is the first usersession message for $ssid$, record $\langle ssid, \mathsf{U}, \mathsf{S}, \mathsf{pw}' \rangle$ and mark it FRESH.
- On $(\mathsf{serversession}, sid, ssid, \mathsf{U})$ from S, retrieve $\langle \text{FILE}, \mathsf{S}, \mathsf{pw} \rangle$, and send $(\mathsf{serversession}, sid, ssid, \mathsf{S}, \mathsf{U})$ to $\mathcal{A}^*$. Also, if this is the first serversession message for $ssid$, record $\langle ssid, \mathsf{S}, \mathsf{U}, \mathsf{pw} \rangle$ and mark it FRESH.

Active Session Attacks

- On $(\mathsf{interrupt}, sid, ssid, \mathsf{S})$ from $\mathcal{A}^*$, if there is a record $\langle ssid, \mathsf{S}, \mathsf{U}, \mathsf{pw} \rangle$ marked FRESH, mark it INTERRUPTED and set $\mathsf{dPT}[ssid] := 1$.
- On $(\mathsf{testpwd}, sid, ssid, \mathsf{P}, \mathsf{pw}^*)$ from $\mathcal{A}^*$, retrieve record $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}' \rangle$ and:
  - If the record is FRESH then do the following: If $\mathsf{pw}^* = \mathsf{pw}'$ return "correct guess" to $\mathcal{A}^*$ and mark $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}' \rangle$ COMPROMISED, otherwise return "wrong guess" and mark $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}' \rangle$ INTERRUPTED.
  - If $\mathsf{P} = \mathsf{S}$ and $\mathsf{dPT}[ssid] = 1$ then set $\mathsf{dPT}[ssid] := 0$ and if $\mathsf{pw}^* = \mathsf{pw}'$ then return "correct guess" to $\mathcal{A}^*$ else return "wrong guess."

  In either case, if $\mathsf{P} = \mathsf{S}$ and $\mathsf{pw}^* = \mathsf{pw}'$ then set $\mathsf{cflag} :=$ COMPROMISED.
- On $(\mathsf{impersonate}, sid, ssid)$ from $\mathcal{A}^*$, if there is a record $\langle ssid, \mathsf{U}, \mathsf{S}, \mathsf{pw}' \rangle$ marked FRESH, do: If there is a record $\langle \text{FILE}, \mathsf{U}, \mathsf{S}, \mathsf{pw} \rangle$ marked COMPROMISED and $\mathsf{pw}' = \mathsf{pw}$, mark $\langle ssid, \mathsf{U}, \mathsf{S}, \mathsf{pw}' \rangle$ COMPROMISED and return "correct guess" to $\mathcal{A}^*$; otherwise mark it INTERRUPTED and return "wrong guess."

Key Generation and Authentication

- On $(\mathsf{newkey}, sid, ssid, \mathsf{P}, SK^*)$ from $\mathcal{A}^*$ where $|SK^*| = \ell$, if there is a record $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}' \rangle$ not marked COMPLETED, do:
  - If the record is COMPROMISED, or ($\mathsf{P} = \mathsf{S}$, the record is INTERRUPTED and $\mathsf{cflag} =$ COMPROMISED), or either P or P' is corrupted, set $SK := SK^*$.
  - Else, if the record is FRESH and $(sid, ssid, SK')$ was sent to P' at the time there was a record $\langle ssid, \mathsf{P}', \mathsf{P}, \mathsf{pw}' \rangle$ marked FRESH, set $SK := SK'$.
  - Else pick $SK \leftarrow_{\$} \{0,1\}^{\ell}$.

  Finally, mark $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}' \rangle$ COMPLETED and send $(sid, ssid, SK)$ to P.
- On $(\mathsf{testabort}, sid, ssid, \mathsf{U})$ from $\mathcal{A}^*$, if $\exists$ record $\langle ssid, \mathsf{U}, \mathsf{S}, \mathsf{pw}' \rangle$ marked FRESH and record $\langle \text{FILE}, sid, \mathsf{S}, \mathsf{pw} \rangle$ do: If $\mathsf{pw} = \mathsf{pw}'$ send SUCC to $\mathcal{A}^*$, otherwise send FAIL to $\mathcal{A}^*$ and $(\mathsf{abort}, sid, ssid)$ to U and mark $\langle ssid, \mathsf{U}, \mathsf{S}, \mathsf{pw}' \rangle$ COMPLETED.

Fig. 19. Functionality $\mathcal{F}_{\mathsf{saPAKE}}$ with marked relaxations specific to OPAQUE [44].

strong aPAKE (saPAKE). We recall that saPAKE was introduced by [44] as a type of aPAKE that is resistant to precomputation attacks. Consequently, using aPAKE rather than saPAKE opens up an adversarial precomputation avenue in the resulting atPAKE protocol. However, this is a mild weakening of the atPAKE notion, because the adversary can only start precomputing ODA attack after compromising $t+1$ auxiliary servers.

For completeness we include in Figure 20 the aPAKE functionality, $\mathcal{F}_{\mathsf{aPAKE}}$ [32]. Its significant difference from $\mathcal{F}_{\mathsf{saPAKE}}$ (Figure 19) is in the handling of precomputed offline password tests.

We define protocol $\Pi_{\mathsf{aPPSS-aPAKE}}$, an atPAKE construction built using $\mathcal{F}_{\mathsf{tOPRF}}$ and $\mathcal{F}_{\mathsf{aPAKE}}$. This protocol is identical to our main construction $\Pi_{\mathsf{tOPRF-atPAKE}}$ of Figure 8, except that it replaces calls to $\mathcal{F}_{\mathsf{saPAKE}}$ with calls to $\mathcal{F}_{\mathsf{aPAKE}}$. Protocol $\Pi_{\mathsf{aPPSS-aPAKE}}$ realizes an ideal functionality $\mathcal{F}_{\mathsf{atPAKE(medium)}}$, which is defined as $\mathcal{F}_{\mathsf{atPAKE}}$ shown in Figures 4, 5, and 6, but with the following 3 differences:

1. In the offlinetestpwd routine, the adversary can precompute offline password tests against target server files before they are compromised. In particular, rather than aborting if no COMPR targetfile is found, the functionality instead still checks whether the password guess has been evaluated with $t+1$ auxiliary servers (i.e. whether $|\mathsf{TS}[sid_{\mathsf{A}}, \mathsf{pw}^*, ssid_{\mathsf{A}}]| \geq t{+}1$). If so, then a record (OFFLINE, $sid_{\mathsf{T}}, sid_{\mathsf{A}}, \mathsf{pw}^*$) is saved, indicating that the adversary has precomputed an offline password test on pw^*.
2. In the corrupt and stealtargetfile routines, compromise of a target server immediately informs the adversary if one of the precomputed password tests was correct. In particular, after a record (targetfile, $sid_{\mathsf{T}}, j, sid_{\mathsf{A}}, \mathsf{pw}, \mathsf{tflag}$) is marked COMPR, the functionality checks where there exists record (OFFLINE, $sid_{\mathsf{T}}, sid_{\mathsf{A}}, \mathsf{pw}$); if so, it sends $(sid_{\mathsf{A}}, \mathsf{pw})$ to $\mathcal{A}^*$.
3. The testabort routine can be run against both user and server sessions, as is the case in $\mathcal{F}_{\mathsf{aPAKE}}$. (In $\mathcal{F}_{\mathsf{saPAKE}}$, only user sessions can be targeted by testabort.) Furthermore, success requires a matching counter-session, not just a matching server file.
   Separately from changes related to ODA precomputation, this last difference reflects a minor additional way in which $\mathcal{F}_{\mathsf{aPAKE}}$ is weaker than $\mathcal{F}_{\mathsf{saPAKE}}$. See Appendix D for further explanation.

Theorem 5. *Protocol $\Pi_{\mathsf{aPPSS-aPAKE}}$ realizes functionality $\mathcal{F}_{\mathsf{atPAKE(medium)}}$ with parameters t and n in the $(\mathcal{F}_{\mathsf{tOPRF}}, \mathcal{F}_{\mathsf{aPAKE}}, \mathcal{F}_{\mathsf{channel}})$-hybrid model.*

Specifically, for any efficient adversary $\mathcal{A}$ against protocol $\Pi_{\mathsf{aPPSS-aPAKE}}$, there exists a simulator SIM *such that no efficient environment $\mathcal{Z}$ can distinguish the view of $\mathcal{A}$ interacting with the real $\Pi_{\mathsf{aPPSS-aPAKE}}$ protocol and the view of* SIM *interacting with the ideal functionality $\mathcal{F}_{\mathsf{atPAKE(medium)}}$ with advantage better than $(q_{\mathsf{T}} \cdot q_{\mathsf{eval}}^2 + q_{\mathsf{test}} + q_{\mathsf{T}}^* \cdot q_{\mathsf{eval}})/2^{\tau}$ where q_{T} is the number of target server instances, q_{eval} is the number of tOPRF evaluations, q_{test} is the number of online and offline password-guessing attacks against $\mathcal{F}_{\mathsf{aPAKE}}$, q_{T}^* is the number of dishonestly initialized target server instances, and security parameter τ is the tOPRF output length.*

Password Registration

- On $(\mathsf{storepwdfile}, sid, \mathsf{pw})$ from S, if this is the first such message, record $\langle \text{FILE}, sid, \mathsf{S}, \mathsf{pw} \rangle$ and mark it UNCOMPROMISED

Stealing Password Data

- On $(\mathsf{stealpwdfile}, sid)$ from $\mathcal{A}^*$, if there is no record $\langle \text{FILE}, sid, \mathsf{S}, \mathsf{pw} \rangle$, return "no password file" to $\mathcal{A}^*$. Otherwise, if the record is marked UNCOMPROMISED, mark it COMPROMISED; regardless,
  - If there is a record $(\mathsf{offline}, sid, \mathsf{pw})$, send pw to $\mathcal{A}^*$.
  - Else return "password file stolen" to $\mathcal{A}^*$.
- On $(\mathsf{offlinetestpwd}, sid, \mathsf{pw}^*)$ from $\mathcal{A}^*$, do:
  - If there is a record $\langle \text{FILE}, sid, \mathsf{S}, \mathsf{pw} \rangle$ marked COMPROMISED, do: if $\mathsf{pw}^* = \mathsf{pw}$, return "correct guess" to $\mathcal{A}^*$; otherwise return "wrong guess."
  - Else record $(\mathsf{offline}, sid, \mathsf{pw}^*)$.

Password Authentication

- On $(\mathsf{usersession}, sid, ssid, \mathsf{S}, \mathsf{pw}')$ from U, send $(\mathsf{usersession}, sid, ssid, \mathsf{U}, \mathsf{S})$ to $\mathcal{A}^*$. Also, if this is the first usersession message for $ssid$, record $\langle ssid, \mathsf{U}, \mathsf{S}, \mathsf{pw}' \rangle$ and mark it FRESH.
- On $(\mathsf{serversession}, sid, ssid, \mathsf{U})$ from S, retrieve $\langle \text{FILE}, \mathsf{S}, \mathsf{pw} \rangle$, and send $(\mathsf{serversession}, sid, ssid, \mathsf{S}, \mathsf{U})$ to $\mathcal{A}^*$. Also, if this is the first serversession message for $ssid$, record $\langle ssid, \mathsf{S}, \mathsf{U}, \mathsf{pw} \rangle$ and mark it FRESH.

Active Session Attacks

- On $(\mathsf{testpwd}, sid, ssid, \mathsf{P}, \mathsf{pw}^*)$ from $\mathcal{A}^*$, if there is a record $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}' \rangle$ marked FRESH, do: if $\mathsf{pw}^* = \mathsf{pw}'$ return "correct guess" to $\mathcal{A}^*$ and mark $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}' \rangle$ COMPROMISED, otherwise return "wrong guess" and mark $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}' \rangle$ INTERRUPTED.
- On $(\mathsf{impersonate}, sid, ssid)$ from $\mathcal{A}^*$, if there is a record $\langle ssid, \mathsf{U}, \mathsf{S}, \mathsf{pw}' \rangle$ marked FRESH, do: If there is a record $\langle \text{FILE}, \mathsf{U}, \mathsf{S}, \mathsf{pw} \rangle$ marked COMPROMISED and $\mathsf{pw}' = \mathsf{pw}$, mark $\langle ssid, \mathsf{U}, \mathsf{S}, \mathsf{pw}' \rangle$ COMPROMISED and return "correct guess" to $\mathcal{A}^*$; otherwise mark it INTERRUPTED and return "wrong guess."

Key Generation and Authentication

- On $(\mathsf{newkey}, sid, ssid, \mathsf{P}, SK^*)$ from $\mathcal{A}^*$ where $|SK^*| = \ell$, if there is a record $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}' \rangle$ not marked COMPLETED, do:
  - If the record is COMPROMISED, or P or P' is corrupted, set $SK := SK^*$.
  - Else, if the record is FRESH and $(sid, ssid, SK')$ was sent to P' at the time there was a record $\langle ssid, \mathsf{P}', \mathsf{P}, \mathsf{pw}' \rangle$ marked FRESH, set $SK := SK'$.
  - Else pick $SK \leftarrow_{\$} \{0,1\}^{\ell}$.

  Finally, mark $\langle ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}' \rangle$ COMPLETED and send $(sid, ssid, SK)$ to P.
- On $(\mathsf{testabort}, sid, ssid, \mathsf{P})$ from $\mathcal{A}^*$, if there is a record $(ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}')$ not marked COMPLETED, do:
  - If it is FRESH and there is a record $(ssid, \mathsf{P}', \mathsf{P}, \mathsf{pw}')$, send SUCC to $\mathcal{A}^*$.
  - Else send FAIL to $\mathcal{A}^*$ and $(\mathsf{abort}, sid, ssid)$ to P, and mark $(ssid, \mathsf{P}, \mathsf{P}', \mathsf{pw}')$ COMPLETED.

Fig. 20. Functionality $\mathcal{F}_{\mathsf{aPAKE}}$ [32], [44]. The shadowed text is the adversary's precomputation avenue, which does not exist in $\mathcal{F}_{\mathsf{saPAKE}}$ (Figure 19).

Note that, in Theorem 5's security bound, q_{test} is the number of *completed* password-guessing attacks. It does not count precomputed offline guesses that are never completed by the later corruption of the relevant target server.

The proof of Theorem 5 is almost entirely identical to our main construction's proof of security (Appendix C). To avoid redundancy, we list only the few ways in which the simulator $\mathsf{SIM}_{\mathsf{tOPRF-atPAKE}}$ (Figures 16, 17, and 18) must be modified in order to accomodate this new proof. All other parts of the proof are identical. The differences all follow in a straightforward manner from the differences between $\mathcal{F}_{\mathsf{aPAKE}}$ and $\mathcal{F}_{\mathsf{saPAKE}}$.

1. In response to apake.offlinetestpwd messages from $\mathcal{A}^*$ (routine 22), send atpake.offlinetestpwd to $\mathcal{F}_{\mathsf{atPAKE}}$ even if there is no targetfile record.
2. After sending atpake.stealtargetfile to $\mathcal{F}_{\mathsf{atPAKE}}$ (routine 9), forward any response to $\mathcal{A}^*$. In particular, $\mathcal{F}_{\mathsf{atPAKE}}$ will respond with pw if there was a correct precomputed offline password test.
3. Never set any cflag to COMPR. The only effect of this change is in the handling of apake.newkey messages from $\mathcal{A}^*$ (routine 20).
4. Omit handling of apake.interrupt messages from $\mathcal{A}^*$ (routine 17). This command does not exist in the $\mathcal{F}_{\mathsf{aPAKE}}$ interface.

F atPAKE Construction from aPPSS

In this section we discuss our atPAKE variant which constructs atPAKE from augmented password-protected secrete sharing (aPPSS) [27] instead of tOPRF. The aPPSS scheme allows the client holding password pw to decrypt and authenticate an arbitrary secret rw which was secret-shared (and password-protected) among the auxiliary servers during initialization. Compared to tOPRF, an aPPSS scheme can be realized from any (non-threshold) OPRF, which in turn can be realized under weaker assumptions, e.g. GapOMDH [40] or DDH (with more protocols rounds) [17]. The disadvantage of using aPPSS is that it (1) it enables offline password testing attack after compromise of $t + 1$ auxiliary servers, without the compromise of a target server, and (2) it lets the client, not the target server, be the first to learn if an authentication attempt succeeded, a feature which some servers might find problematic.

For completeness, we recall the $\mathcal{F}_{\mathsf{aPPSS}}$ functionality of [27] in Figure 21, and we refer to [27] for an OPRF-based implementation of this functionality. As mentioned in the Introduction, see also footnote 10, we believe the UC aPPSS functionality $\mathcal{F}_{\mathsf{aPPSS}}$ can also be generically realized from UC tOPRF via the compiler of [41].

In Figures 22, 23, and 24 we present $\mathcal{F}_{\mathsf{atPAKE(weak)}}$, a weakened variant of $\mathcal{F}_{\mathsf{atPAKE}}$ that can be realized using aPPSS instead of tOPRF. The most important security relaxation is in regard to offline dictionary attacks. In $\mathcal{F}_{\mathsf{atPAKE(weak)}}$, once $t+1$ auxiliary servers are compromised the adversary can freely perform offline password tests (without any target server compromise or involvement). Honest

Notation: Values t, n, τ are parameters. Functionality initializes $\mathsf{ppss.pwtested}(\mathsf{pw}) := \emptyset$ for all pw, and $\mathsf{tx}^{\mathsf{aux}}_{sid}(\mathsf{P}_i) := 0$ for all P_i.
(The functionality code handles only one instance, tagged by a unique string sid.)

Initialization:

1. On $(\mathsf{ppss.uinit}, sid, \mathsf{pw}, \mathsf{sk}^*)$ from party U s.t. $|\mathbf{P}_{sid}| = n$: Send $(\mathsf{ppss.uinit}, sid, \mathsf{U})$ to $\mathcal{A}^*$. If U is honest then set $\mathsf{sk} \leftarrow_{\$} \{0,1\}^{\tau}$, else set $\mathsf{sk} := \mathsf{sk}^*$. Save $(\mathsf{ppss.uinit}, sid, \mathsf{U}, \mathsf{pw}, \mathsf{sk})$. Ignore future $\mathsf{ppss.uinit}$ calls for same sid.
2. On $(\mathsf{ppss.sinit}, sid, i, \mathsf{U})$ from party S, or $(\mathsf{ppss.sinit}, sid, i, \mathsf{S}, \mathsf{U})$ from $\mathcal{A}^*$ for $\mathsf{S} \in \mathbf{Corr}$, send $(\mathsf{ppss.sinit}, sid, i, \mathsf{S}, \mathsf{U})$ to $\mathcal{A}^*$, save $(\mathsf{ppss.sinit}, sid, \mathsf{U}, \mathsf{S}, i)$.
3. If $\exists$ rec. $(\mathsf{ppss.uinit}, sid, \mathsf{U}, \mathsf{pw}, \mathsf{sk})$ and $(\mathsf{ppss.sinit}, sid, \mathsf{U}, \mathsf{S}, i)$ s.t. $\mathsf{S}{=}\mathbf{P}_{sid}[i]$, mark S as ACTIVE.
4. On $(\mathsf{ppss.fininit}, sid)$ from $\mathcal{A}^*$, if $\exists$ rec. $(\mathsf{ppss.uinit}, sid, \mathsf{U}, \mathsf{pw}, \mathsf{sk})$ and all parties in list $\mathbf{P}_{sid}$ are marked ACTIVE, send $(\mathsf{ppss.fininit}, sid, \mathsf{sk})$ to U.

Server Compromise: (*This query requires permission from the environment.*)

1. On $(\mathsf{ppss.compromise}, sid, \mathsf{P})$ from $\mathcal{A}^*$, set $\mathbf{Corr} := \mathbf{Corr} \cup \{\mathsf{P}\}$.

Reconstruction:

1. On $(\mathsf{ppss.urec}, sid, ssid, \mathbf{S}, \mathsf{pw}')$ from party U' or from $\mathsf{U}' = \mathcal{A}^*$, send $(\mathsf{ppss.urec}, sid, ssid, \mathsf{U}', \mathbf{S})$ to $\mathcal{A}^*$. If $\exists$ record $(\mathsf{ppss.uinit}, sid, \mathsf{U}, \mathsf{pw}, \mathsf{sk})$ then create record $(\mathsf{ppss.urec}, sid, ssid, \mathsf{U}', \mathsf{pw}, \mathsf{pw}', \mathsf{sk})$, else create record $(\mathsf{ppss.urec}, sid, ssid, \mathsf{U}', \perp, \mathsf{pw}', \perp)$. Ignore future $\mathsf{ppss.urec}$ calls for same $ssid$.
2. On $(\mathsf{ppss.srec}, sid, ssid, \mathsf{U}')$ from party S or $(\mathsf{ppss.srec}, sid, ssid, \mathsf{S}, \mathsf{U}')$ from $\mathcal{A}^*$ for $\mathsf{S} \in \mathbf{Corr}$, send $(\mathsf{ppss.srec}, sid, ssid, \mathsf{S}, \mathsf{U}')$ to $\mathcal{A}^*$. If S is marked ACTIVE then increment $\mathsf{tx}^{\mathsf{aux}}_{sid}(\mathsf{S})$ by 1.
3. On $(\mathsf{ppss.finrec}, sid, ssid, \mathbf{C}, \mathsf{atflag}, \mathsf{pw}^*, \mathsf{sk}^*)$ from $\mathcal{A}^*$, if $\exists$ rec. $(\mathsf{ppss.urec}, sid, ssid, \mathsf{U}', \mathsf{pw}, \mathsf{pw}', \mathsf{sk})$ then erase it and send $(\mathsf{ppss.finrec}, sid, ssid, \mathsf{sk}')$ to U' s.t.
   (a) if $\mathsf{atflag} = 1$, $|\mathbf{C}| = t+1$, and $\forall_{\mathsf{S}\in\mathbf{C}}(\mathsf{tx}^{\mathsf{aux}}_{sid}(\mathsf{S}) > 0)$ then set $\mathsf{tx}^{\mathsf{aux}}_{sid}(\mathsf{S})$−− for all $\mathsf{S} \in \mathbf{C}$, and if $\mathsf{pw} = \mathsf{pw}'$ then set $\mathsf{sk}' := \mathsf{sk}$ else set $\mathsf{sk}' := \perp$;
   (b) if $\mathsf{atflag} = 2$ and $\mathsf{pw}^* = \mathsf{pw}'$ then set $\mathsf{sk}' := \mathsf{sk}^*$;
   (c) otherwise set $\mathsf{sk}' := \perp$.

Password Test:

1. On $(\mathsf{ppss.testpw}, sid, \mathsf{S}, \mathsf{pw}^*)$ from $\mathcal{A}^*$, retrieve $(\mathsf{ppss.uinit}, sid, \mathsf{U}, \mathsf{pw}, \mathsf{sk})$. If $\mathsf{tx}^{\mathsf{aux}}_{sid}(\mathsf{S}){>}0$ then add S to set $\mathsf{ppss.pwtested}(\mathsf{pw}^*)$ and set $\mathsf{tx}^{\mathsf{aux}}_{sid}(\mathsf{S})$−−. If $|\mathsf{ppss.pwtested}(\mathsf{pw}^*)| = t{+}1$ then return sk to $\mathcal{A}^*$ if $\mathsf{pw}^* = \mathsf{pw}$, else return $\perp$.

Fig. 21. Augmented PPSS functionality $\mathcal{F}_{\mathsf{aPPSS}}$

Notation
Integer t is a threshold parameter.
Set $\mathsf{tx}[x] := 0$ and $\mathsf{TS}[x] := \{\}$ for all x.

Initialization Phase

1. On $(\mathsf{userinit}, sid_\mathsf{A}, sid_\mathsf{T}, \mathsf{pw})$ from $\mathsf{U} \in \mathcal{P}$:
   - send $(\mathsf{userinit}, \mathsf{U}, sid_\mathsf{A}, sid_\mathsf{T})$ to $\mathcal{A}^*$
   - save $(\mathsf{userinit}, \mathsf{U}, sid_\mathsf{A}, sid_\mathsf{T}, \mathsf{pw})$ and set $\mathsf{cflag}[sid_\mathsf{A}] :=$ UNCOMPROMISED
   - ignore future $\mathsf{userinit}$ calls for same sid_A or sid_T
2. On $(\mathsf{auxinit}, sid_\mathsf{A}, i, \mathsf{U})$ from $\mathsf{S} = \mathbf{S}_{sid_\mathsf{A}}[i]$:
   - send $(\mathsf{auxinit}, sid_\mathsf{A}, i, \mathsf{U})$ to $\mathcal{A}^*$
   - save $(\mathsf{auxinit}, sid_\mathsf{A}, i, \mathsf{U})$ marked PENDING
   - ignore future $\mathsf{auxinit}$ calls for same (sid_A, i)
3. On $(\mathsf{targetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, \mathsf{U})$ from $\mathsf{T} = \mathbf{S}_{sid_\mathsf{T}}[j]$:
   - send $(\mathsf{targetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, \mathsf{U})$ to $\mathcal{A}^*$
   - save $(\mathsf{targetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, \mathsf{U})$ marked PENDING
   - ignore future $\mathsf{targetinit}$ calls for same sid_A or sid_T and j
4. On $(\mathsf{finishauxiliaryinit}, sid_\mathsf{A}, i)$ from $\mathcal{A}^*$:
   - find $(\mathsf{userinit}, [\mathsf{U}], sid_\mathsf{A}, [sid_\mathsf{T}, \mathsf{pw}])$ (abort if missing)
   - find $(\mathsf{auxinit}, sid_\mathsf{A}, i, \mathsf{U})$ marked PENDING (abort if missing) and change its mark to COMPLETED
   - save $(\mathsf{auxiliaryfile}, sid_\mathsf{A}, i, \boxed{\mathsf{pw}})$, and if $\mathbf{S}_{sid_\mathsf{A}}[i] \in \mathbf{Corr}$ then mark it COMPR
   - output $(\mathsf{finishauxiliaryinit}, sid_\mathsf{A})$ to $\mathsf{S} = \mathbf{S}_{sid_\mathsf{A}}[i]$
5. On $(\mathsf{finishtargetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, sid^*_\mathsf{A}, \mathsf{pw}^*)$ from $\mathcal{A}^*$:
   - find $\mathsf{rec} = (\mathsf{targetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, [\mathsf{U}])$ marked PENDING (abort if missing)
   - if $\mathsf{U} \notin \mathbf{Corr}$ then find $(\mathsf{userinit}, \mathsf{U}, sid_\mathsf{A}, sid_\mathsf{T}, [\mathsf{pw}])$ (abort if missing) and save $(\mathsf{targetfile}, sid_\mathsf{T}, j, sid_\mathsf{A}, \mathsf{pw}, \text{UNTAMPERED})$
   - otherwise (i.e. if $\mathsf{U} \in \mathbf{Corr}$) save $(\mathsf{targetfile}, sid_\mathsf{T}, j, sid^*_\mathsf{A}, \mathsf{pw}^*, \text{TAMPERED})$
   - output $(\mathsf{finishtargetinit}, sid_\mathsf{A}, sid_\mathsf{T})$ to $\mathsf{T} = \mathbf{S}_{sid_\mathsf{T}}[j]$, mark rec COMPLETED
   - if $\mathbf{S}_{sid_\mathsf{T}}[j] \in \mathbf{Corr}$ then mark the $\mathsf{targetfile}$ COMPR

Fig. 22. $\mathcal{F}_{\mathsf{atPAKE(weak)}}$: atPAKE functionality (1): Initialization Phase.
Dashed text is included in $\mathcal{F}_{\mathsf{atPAKE(weak)}}$ but omitted from $\mathcal{F}_{\mathsf{atPAKE}}$.

clients also learn whether or not their passwords are correct before interacting with a target server. A second and less significant relaxation is that auxiliary server evaluation tickets are no longer bound to a particular *ssid*.

In Figure 25 we show the atPAKE protocol constructed from aPPSS. Our aPPSS-based atPAKE scheme can be seen as a threshold counterpart to OPAQUE, where the client authenticates the server-supplied data before using it to authenticate to the server, while our tOPRF-based scheme can be seen as a threshold counterpart to OPAQUE', where the (target) server is the first party that can verify an authentication result.

In Figures 26, 27 and 28 we show the simulator for the security proof of the aPPSS-based atPAKE protocol, i.e. for the proof of Theorem 6:

Party Corruption, File Compromise, Offline Password Tests

6. On $(\mathsf{corrupt}, \mathsf{P})$ from $\mathcal{A}^*$ (permitted by $\mathcal{Z}$), set $\mathbf{Corr} := \mathbf{Corr} \cup \{\mathsf{P}\}$
   - if $\exists$ $(\mathsf{auxiliaryfile}, sid_\mathsf{A}, i, [\mathsf{pw}])$ for $\mathbf{S}_{sid_\mathsf{A}}[i] = \mathsf{P}$ mark it COMPR
   - if $\exists$ $(\mathsf{targetfile}, sid_\mathsf{T}, j, sid_\mathsf{A}, [\mathsf{pw}, \mathsf{tflag}])$ for $\mathbf{S}_{sid_\mathsf{T}}[j] = \mathsf{P}$ mark it COMPR
7. On $(\mathsf{stealauxiliaryfile}, sid_\mathsf{A}, i)$ from $\mathcal{A}^*$ (permitted by $\mathcal{Z}$):
   - if $\exists$ $(\mathsf{auxiliaryfile}, sid_\mathsf{A}, i, [\mathsf{pw}])$ mark it COMPR
8. On $(\mathsf{stealtargetfile}, sid_\mathsf{T}, j)$ from $\mathcal{A}^*$ (permitted by $\mathcal{Z}$):
   - if $\exists$ $(\mathsf{targetfile}, sid_\mathsf{T}, j, [sid_\mathsf{A}, \mathsf{pw}, \mathsf{tflag}])$ mark it COMPR
9. On $(\mathsf{offlinetestpwd}, sid_\mathsf{A}, i, \boxed{ssid_\mathsf{A}, sid_\mathsf{T}, j}, \mathsf{pw}^*)$ from $\mathcal{A}^*$:
   - retrieve $(\mathsf{auxiliaryfile}, sid_\mathsf{A}, i, [\mathsf{pw}])$ (abort if not found)
   - if $\mathsf{tx}[sid_\mathsf{A}, i, \boxed{ssid_\mathsf{A}}] > 0$ add i to $\mathsf{TS}[sid_\mathsf{A}, \mathsf{pw}^*, \boxed{ssid_\mathsf{A}}]$, set $\mathsf{tx}[sid_\mathsf{A}, i, \boxed{ssid_\mathsf{A}}]$--
   - $\boxed{\text{retrieve } \mathsf{rec} = (\mathsf{targetfile}, sid_\mathsf{T}, j, sid_\mathsf{A}, [\mathsf{pw}, \mathsf{tflag}]) \text{ (abort if not found)}}$
   - if $|\mathsf{TS}[sid_\mathsf{A}, \mathsf{pw}^*, \boxed{ssid_\mathsf{A}}]| \geq t{+}1$ $\boxed{\text{and } \mathsf{rec} \text{ is marked COMPR}}$ then return "correct guess" to $\mathcal{A}^*$ and set $\mathsf{cflag}[sid_\mathsf{A}] := \text{COMPR}$ if $\mathsf{pw}^* = \mathsf{pw}$, else return "wrong guess" to $\mathcal{A}^*$

Authentication Phase (I): Session Initialization, Passive Transmission

10. On $(\mathsf{usersession}, sid_\mathsf{A}, sid_\mathsf{T}, j, ssid, \mathsf{pw}')$ from $\mathsf{U}' \in \mathcal{P}$:
    - send $(\mathsf{usersession}, \mathsf{U}', sid_\mathsf{A}, sid_\mathsf{T}, j, ssid)$ to $\mathcal{A}^*$
    - save $(\mathsf{session}, \mathsf{U}', \mathbf{S}_{sid_\mathsf{T}}[j], sid_\mathsf{A}, sid_\mathsf{T}, j, ssid, \mathsf{pw}')$ marked PRELIM
    - ignore future usersession calls for same $ssid$
11. On $(\mathsf{auxsession}, sid_\mathsf{A}, i, ssid_\mathsf{A})$ from $\mathsf{S} = \mathbf{S}_{sid_\mathsf{A}}[i]$ or $\mathcal{A}^*$:
    - retrieve $(\mathsf{auxiliaryfile}, sid_\mathsf{A}, i, [\mathsf{pw}])$ (abort if record not found)
    - if sender is $\mathcal{A}^*$ then abort unless the retrieved record is marked COMPR
    - send $(\mathsf{auxsession}, sid_\mathsf{A}, i, ssid_\mathsf{A})$ to $\mathcal{A}^*$
    - set $\mathsf{tx}[sid_\mathsf{A}, i, \boxed{ssid_\mathsf{A}}]{+}{+}$
12. On $(\mathsf{targetsession}, sid_\mathsf{T}, j, \mathsf{U}', ssid)$ from $\mathsf{S} = \mathbf{S}_{sid_\mathsf{T}}[j]$:
    - retrieve $(\mathsf{targetfile}, sid_\mathsf{T}, j, [sid_\mathsf{A}, \mathsf{pw}, \mathsf{tflag}])$ (abort if record not found)
    - send $(\mathsf{targetsession}, sid_\mathsf{T}, j, \mathsf{U}', ssid)$ to $\mathcal{A}^*$
    - save $(\mathsf{session}, \mathsf{S}, \mathsf{U}', sid_\mathsf{A}, sid_\mathsf{T}, j, ssid, \mathsf{pw})$ marked FRESH
    - ignore future targetsession calls for same $ssid$
13. On $(\mathsf{auxproceed}, \mathsf{U}', ssid, \boxed{sid^*_\mathsf{A}, ssid_\mathsf{A}}, \mathbf{C})$ s.t. $|\mathbf{C}| = t + 1$ from $\mathcal{A}^*$:
    - retrieve $\mathsf{rec} = (\mathsf{session}, \mathsf{U}', \mathsf{S}, [sid_\mathsf{A}, sid_\mathsf{T}, j], ssid, [\mathsf{pw}'])$ marked PRELIM (abort if record not found)
    - $\boxed{\text{reset field } sid_\mathsf{A} \text{ in record } \mathsf{rec} \text{ to } sid^*_\mathsf{A}}$
    - abort if $\exists_{i \in \mathbf{C}}\ \mathsf{tx}[sid_\mathsf{A}, i, \boxed{ssid_\mathsf{A}}] = 0$, else $\forall_{i \in \mathbf{C}}$ set $\mathsf{tx}[sid_\mathsf{A}, i, \boxed{ssid_\mathsf{A}}]$--
    - $\boxed{\text{change } \mathsf{rec}\text{'s mark to FRESH}}$
    - retrieve $(\mathsf{auxiliaryfile}, sid_\mathsf{A}, i, [\mathsf{pw}])$ for any $i \in \mathbf{C}$
    - if $\mathsf{pw}' = \mathsf{pw}$ send "success" to $\mathcal{A}^*$ and mark rec as FRESH, else mark rec as COMPLETED, send "fail" to $\mathcal{A}^*$ and output $(\mathsf{abort}, ssid)$ to U'
14. On $(\mathsf{testabort}, \mathsf{U}', sid_\mathsf{T}, j, ssid)$ from $\mathcal{A}^*$:
    - retrieve $\mathsf{rec} = (\mathsf{session}, \mathsf{U}', \mathbf{S}_{sid_\mathsf{T}}[j], [sid'_\mathsf{A}], sid_\mathsf{T}, j, ssid, [\mathsf{pw}'])$ marked FRESH and $(\mathsf{targetfile}, sid_\mathsf{T}, j, [sid_\mathsf{A}, \mathsf{pw}, \mathsf{tflag}])$ (abort if either not found)
    - if $(sid'_\mathsf{A}, \mathsf{pw}') = (sid_\mathsf{A}, \mathsf{pw})$ send "success" to $\mathcal{A}^*$; else mark rec COMPLETED, send "fail" to $\mathcal{A}^*$, and output $(\mathsf{abort}, ssid)$ to U'

Fig. 23. $\mathcal{F}_{\mathsf{atPAKE(weak)}}$: atPAKE functionality (2): Compromises, Authentication (I).
Dashed text is included in $\mathcal{F}_{\mathsf{atPAKE(weak)}}$ but omitted from $\mathcal{F}_{\mathsf{atPAKE}}$.
Boxed text is included in $\mathcal{F}_{\mathsf{atPAKE}}$ but omitted from $\mathcal{F}_{\mathsf{atPAKE(weak)}}$.

Authentication Phase (II): Active Attacks, Session Termination

15. On $(\mathsf{auxactive}, \mathsf{U}', ssid)$ from $\mathcal{A}^*$:
    - retrieve $(\mathsf{session}, \mathsf{U}', \mathsf{S}, [sid_\mathsf{A}, sid_\mathsf{T}, j], ssid, [\mathsf{pw}'])$ marked PRELIM and mark it COUNTERFEIT (abort if record not found)
16. On $(\mathsf{interrupt}, sid_\mathsf{T}, j, ssid)$ from $\mathcal{A}^*$:
    - retrieve $(\mathsf{session}, \mathbf{S}_{sid_\mathsf{T}}[j], [\mathsf{U}', sid_\mathsf{A}], sid_\mathsf{T}, j, ssid, [\mathsf{pw}])$ marked FRESH, mark it INTERRUPTED and set $\mathsf{dPT}[ssid] := 1$.
17. On $(\mathsf{testpwd}, \mathsf{P}, ssid, \mathsf{pw}^*)$ from $\mathcal{A}^*$:
    - retrieve $\mathsf{rec} = (\mathsf{session}, \mathsf{P}, [\mathsf{P}', sid_\mathsf{A}, sid_\mathsf{T}, j], ssid, [\mathsf{pw}])$ (abort if not found)
    - if $\mathsf{dPT}[ssid] = 1$, then set $\mathsf{dPT}[ssid] := 0$; else if rec is not marked FRESH or COUNTERFEIT, abort
    - if $\mathsf{pw}^* = \mathsf{pw}$ and any of the following conditions hold:
      (a) $\boxed{\exists ssid_\mathsf{A} \text{ s.t.}}\ |\mathsf{TS}[sid_\mathsf{A}, \mathsf{pw}^*, \boxed{ssid_\mathsf{A}}]| \geq t{+}1$
      (b) or rec marked COUNTERFEIT
      (c) or $\mathsf{P} = \mathbf{S}_{sid_\mathsf{T}}[j]$ and $\exists$ record $(\mathsf{targetfile}, sid_\mathsf{T}, j, sid_\mathsf{A}, \mathsf{pw}, \text{TAMPERED})$
      then mark rec as COMPR, send "correct guess" to $\mathcal{A}^*$, and (if $\mathsf{P} = \mathbf{S}_{sid_\mathsf{T}}[j]$) set $\mathsf{cflag}[sid_\mathsf{A}] := \text{COMPR}$; else mark rec as INTERRUPTED and send "wrong guess" to $\mathcal{A}^*$
18. On $(\mathsf{impersonate}, sid_\mathsf{T}, j, ssid)$ from $\mathcal{A}^*$:
    - retrieve $\mathsf{rec} = (\mathsf{session}, \mathsf{U}', \mathbf{S}_{sid_\mathsf{T}}[j], [sid_\mathsf{A}], sid_\mathsf{T}, j, ssid, [\mathsf{pw}])$ marked FRESH (abort if not found)
    - if $\exists$ record $(\mathsf{targetfile}, sid_\mathsf{T}, j, sid_\mathsf{A}, \mathsf{pw}, [\mathsf{tflag}'])$ marked COMPR then mark rec as COMPR and send "correct guess" to $\mathcal{A}^*$; else mark rec as INTERRUPTED and send "wrong guess" to $\mathcal{A}^*$
19. On $(\mathsf{newkey}, \mathsf{P}, ssid, K^*)$ from $\mathcal{A}^*$:
    - retrieve $\mathsf{rec} = (\mathsf{session}, \mathsf{P}, [\mathsf{P}', sid_\mathsf{A}, sid_\mathsf{T}, j], ssid, [\mathsf{pw}])$ not marked PRELIM or COMPLETED (abort if record not found) and do:
      - if rec is marked COMPR, then set $K \leftarrow K^*$
      - if $\mathsf{P} = \mathbf{S}_{sid_\mathsf{T}}[j]$, rec is marked INTERRUPTED, and ($\mathsf{cflag}[sid_\mathsf{A}] = \text{COMPR}$ or $\exists$ record $(\mathsf{targetfile}, sid_\mathsf{T}, j, sid_\mathsf{A}, \mathsf{pw}, \text{TAMPERED})$), then set $K \leftarrow K^*$
      - if rec is FRESH and $\exists\ \mathsf{rec}' = (\mathsf{session}, \mathsf{P}', \mathsf{P}, sid_\mathsf{A}, sid_\mathsf{T}, j, ssid, \mathsf{pw})$ s.t. P' received $(\mathsf{newkey}, ssid, K')$ when rec' was FRESH, then set $K \leftarrow K'$
      - else pick $K \leftarrow_\$ \{0,1\}^\tau$
    - finally, mark rec as COMPLETED, output $(\mathsf{newkey}, ssid, K)$ to P

Fig. 24. $\mathcal{F}_{\mathsf{atPAKE(weak)}}$: atPAKE functionality (3): Authentication (II).
Boxed text is included in $\mathcal{F}_{\mathsf{atPAKE}}$ but omitted from $\mathcal{F}_{\mathsf{atPAKE(weak)}}$.

Initialization

1. On input $(\mathsf{atpake.userinit}, sid_\mathsf{A}, sid_\mathsf{T}, \mathsf{pw})$, U does:
   - send $(\mathsf{ppss.uinit}, sid_\mathsf{A}, \mathsf{pw}, \perp)$ to $\mathcal{F}_{\mathsf{aPPSS}}$
   - receive $(\mathsf{ppss.fininit}, sid_\mathsf{A}, \mathsf{rw})$ from $\mathcal{F}_{\mathsf{aPPSS}}$ as response
   - for every $j \in \{1, \ldots, |\mathbf{S}_{sid_\mathsf{T}}|\}$, compute $\mathsf{rw}_j := \mathsf{KDF}(\mathsf{rw}, \mathbf{S}_{sid_\mathsf{T}}[j])$ and send $(\mathsf{channel.send}, (sid_\mathsf{A}||sid_\mathsf{T}||j), \mathbf{S}_{sid_\mathsf{T}}[j], \mathsf{rw}_j)$ to $\mathcal{F}_{\mathsf{channel}}$
2. On input $(\mathsf{atpake.auxinit}, sid_\mathsf{A}, i, \mathsf{U}')$, auxiliary server $\mathsf{S} = \mathbf{S}_{sid_\mathsf{A}}[i]$ does:
   - send $(\mathsf{ppss.sinit}, sid_\mathsf{A}, i, \mathsf{U}')$ to $\mathcal{F}_{\mathsf{aPPSS}}$
   - output $(\mathsf{atpake.finishauxiliaryinit}, sid_\mathsf{A})$
3. On $(\mathsf{atpake.targetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, \mathsf{U}')$, target server $\mathsf{T} = \mathbf{S}_{sid_\mathsf{T}}[j]$ does:
   - await $(\mathsf{channel.deliver}, (sid_\mathsf{A}||sid_\mathsf{T}||j), \mathsf{U}', \mathsf{rw}_j)$ from $\mathcal{F}_{\mathsf{channel}}$
   - send $(\mathsf{sapake.storepwdfile}, (sid_\mathsf{T}||j), \mathsf{U}', \mathsf{rw}_j)$ to $\mathcal{F}_{\mathsf{saPAKE}}$
   - output $(\mathsf{atpake.finishtargetinit}, sid_\mathsf{A}, sid_\mathsf{T})$

Password Authentication

4. On input $(\mathsf{atpake.usersession}, sid_\mathsf{A}, sid_\mathsf{T}, j, ssid, \mathsf{pw}')$, U' does:
   - send $(\mathsf{ppss.urec}, sid_\mathsf{A}, ssid, \mathbf{S}_{sid_\mathsf{A}}, \mathsf{pw}')$ to $\mathcal{F}_{\mathsf{aPPSS}}$
   - await response $(\mathsf{ppss.finrec}, sid_\mathsf{A}, ssid, \mathsf{rw}')$
   - if $\mathsf{rw}' = \perp$ then pick $\mathsf{rw}'_j \leftarrow_\$ \{0,1\}^\tau$. Otherwise compute $\mathsf{rw}'_j := \mathsf{KDF}(\mathsf{rw}', \mathbf{S}_{sid_\mathsf{T}}[j])$
   - send $(\mathsf{sapake.usrsession}, (sid_\mathsf{T}||j), ssid, \mathbf{S}_{sid_\mathsf{T}}[j], \mathsf{rw}'_j)$ to $\mathcal{F}_{\mathsf{saPAKE}}$
   - if receive $(\mathsf{sapake.newkey}, (sid||j), ssid, K)$ from $\mathcal{F}_{\mathsf{saPAKE}}$, then output $(\mathsf{atpake.newkey}, ssid, K)$
   - if instead $(\mathsf{sapake.abort}, (sid_\mathsf{T}||j), ssid)$ is received, then output $(\mathsf{atpake.abort}, ssid)$
5. On input $(\mathsf{atpake.auxsession}, sid_\mathsf{A}, i, ssid_\mathsf{A})$, auxiliary server $\mathsf{S} = \mathbf{S}_{sid_\mathsf{A}}[i]$ sends $(\mathsf{ppss.srec}, sid_\mathsf{A}, ssid_\mathsf{A}, \mathsf{U}')$ to $\mathcal{F}_{\mathsf{aPPSS}}$
6. On input $(\mathsf{atpake.targetsession}, sid_\mathsf{T}, j, \mathsf{U}', ssid)$, target server $\mathsf{T} = \mathbf{S}_{sid_\mathsf{T}}[j]$ does:
   - send $(\mathsf{sapake.svrsession}, (sid_\mathsf{T}||j), ssid)$ to $\mathcal{F}_{\mathsf{saPAKE}}$
   - if response $(\mathsf{atpake.newkey}, (sid_\mathsf{T}||j), ssid, K)$ is received, then output $(\mathsf{atpake.newkey}, ssid, K)$

Fig. 25. Protocol $\Pi_{\mathsf{aPPSS-atPAKE}}$ which realizes $\mathcal{F}_{\mathsf{atPAKE(weak)}}$ in the $(\mathcal{F}_{\mathsf{aPPSS}}, \mathcal{F}_{\mathsf{saPAKE}}, \mathcal{F}_{\mathsf{channel}})$-hybrid world.

Notation

Table r stores $[(sid, \mathsf{pw}), \mathsf{rw}]$ pairs where the initial entries are set to $\perp$. Initially $\mathsf{tx}[sid_\mathsf{A}, i] := 0$ for all sid_A and i.

Initialization Phase

1. On $(\mathsf{atpake.userinit}, \mathsf{U}, sid_\mathsf{A}, sid_\mathsf{T})$ from $\mathcal{F}_{\mathsf{atPAKE}}$ for honest U:
   - save $(\mathsf{userinit}, \mathsf{U}, sid_\mathsf{A}, sid_\mathsf{T})$
   - send $(\mathsf{ppss.uinit}, sid_\mathsf{A}, \mathsf{U})$ to $\mathcal{A}^*$
   - for every $j \in \{1, \ldots, |\mathbf{S}_{sid_\mathsf{T}}|\}$, send $(\mathsf{channel.send}, (sid_\mathsf{A}||sid_\mathsf{T}||j), \mathsf{U}, \mathbf{S}_{sid_\mathsf{T}}[j], \tau)$ to $\mathcal{A}^*$
2. On $(\mathsf{ppss.uinit}, sid_\mathsf{A}, \mathsf{pw}^*, \mathsf{rw}^*)$ from $\mathcal{A}^*$ on behalf of some $\mathsf{U} \in \mathbf{Corr}$:
   - save $(\mathsf{userinit}, \mathsf{U}, sid_\mathsf{A}, \perp)$ marked ADVERSARIAL, save $r[sid_\mathsf{A}, \mathsf{pw}^*] = \mathsf{rw}^*$.
   - send $(\mathsf{ppss.uinit}, sid_\mathsf{A}, \mathsf{U})$ to $\mathcal{A}^*$
   - if $\mathsf{U} \neq \mathcal{A}^*$, choose any sid_T and send $(\mathsf{atpake.userinit}, sid_\mathsf{A}, sid_\mathsf{T}, \mathsf{pw}^*)$ to $\mathcal{F}_{\mathsf{atPAKE}}$ on behalf of U
3. On $(\mathsf{atpake.auxinit}, sid_\mathsf{A}, i, \mathsf{U}')$ from $\mathcal{F}_{\mathsf{atPAKE}}$:
   - save $(\mathsf{auxinit}, sid_\mathsf{A}, i, \mathsf{U}')$ marked PENDING
4. On $(\mathsf{ppss.sinit}, sid_\mathsf{A}, i, \mathsf{U}')$ from $\mathcal{A}^*$ where $\mathbf{S}_{sid_\mathsf{A}}[i] \in \mathbf{Corr}$:
   - if $\mathbf{S}_{sid_\mathsf{A}}[i] \neq \mathcal{A}^*$, send $(\mathsf{atpake.auxinit}, sid_\mathsf{A}, i, \mathsf{U}')$ to $\mathcal{F}_{\mathsf{atPAKE}}$
   - otherwise, save $(\mathsf{auxinit}, sid_\mathsf{A}, i, \mathsf{U})$ marked PENDING
5. On $(\mathsf{atpake.targetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, \mathsf{U}')$ from $\mathcal{F}_{\mathsf{atPAKE}}$:
   - save $(\mathsf{targetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, \mathsf{U}')$ marked PENDING
6. On $(\mathsf{ppss.fininit}, sid_\mathsf{A}, i)$ from $\mathcal{A}^*$:
   - retrieve $\mathsf{urec} = (\mathsf{userinit}, [\mathsf{U}], sid_\mathsf{A}, [sid_\mathsf{T}])$ (abort if not found)
   - retrieve $(\mathsf{auxinit}, sid_\mathsf{A}, i, \mathsf{U})$ marked PENDING and change its mark to COMPLETED (abort if not found)
   - save $(\mathsf{auxiliaryfile}, sid_\mathsf{A}, i)$ and mark it ADVERSARIAL if urec is ADVERSARIAL
   - send $(\mathsf{atpake.finishauxiliaryinit}, sid_\mathsf{A}, i)$ to $\mathcal{F}_{\mathsf{atPAKE}}$
7. On $(\mathsf{channel.deliver}, (sid_\mathsf{A}||sid_\mathsf{T}||j), \mathsf{U}, \mathbf{S}_{sid_\mathsf{T}}[j])$ from $\mathcal{A}^*$:
   - retrieve $(\mathsf{targetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, \mathsf{U})$ marked PENDING and change its mark to COMPLETED (abort if not found)
   - if $(\mathsf{channel.send}, (sid_\mathsf{A}||sid_\mathsf{T}||j), \mathbf{S}_{sid_\mathsf{T}}[j], [\mathsf{rw}_j^*])$ was previously sent by $\mathcal{A}^*$ on behalf of $\mathsf{U} \in \mathbf{Corr}$:
     - if $\exists\ (sid_\mathsf{A}^*, \mathsf{pw}^*)$ s.t. $\mathsf{rw}_j^* = \mathsf{KDF}(\mathsf{rw}^*, \mathbf{S}_{sid_\mathsf{T}}[i])$ where $r[sid_\mathsf{A}^*, \mathsf{pw}^*] = \mathsf{rw}^*$, then send $(\mathsf{atpake.finishtargetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, sid_\mathsf{A}^*, \mathsf{pw}^*)$ to $\mathcal{F}_{\mathsf{atPAKE}}$ and save $(\mathsf{targetfile}, sid_\mathsf{A}^*, sid_\mathsf{T}, j, \mathsf{pw}^*, \mathsf{rw}_j^*)$
     - otherwise, send $(\mathsf{atpake.finishtargetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, \perp, \perp)$ to $\mathcal{F}_{\mathsf{atPAKE}}$ and save $(\mathsf{targetfile}, \perp, sid_\mathsf{T}, j, \perp, \mathsf{rw}_j^*)$
   - otherwise, retrieve record $(\mathsf{userinit}, \mathsf{U}, sid_\mathsf{A}, sid_\mathsf{T})$ (abort if not found), send $(\mathsf{atpake.finishtargetinit}, sid_\mathsf{A}, sid_\mathsf{T}, j, \perp, \perp)$ to $\mathcal{F}_{\mathsf{atPAKE}}$, and save $(\mathsf{targetfile}, sid_\mathsf{A}, sid_\mathsf{T}, j, \perp, \perp)$

Fig. 26. Simulator $\mathsf{SIM}_{\mathsf{aPPSS-atPAKE}}$ for protocol $\Pi_{\mathsf{aPPSS-atPAKE}}$, part 1: Notation and Initialization

Party Corruption

8. On $(\mathsf{ppss.compromise}, \mathsf{P})$ from $\mathcal{A}^*$:
   - set $\mathbf{Corr} := \mathbf{Corr} \cup \{\mathsf{P}\}$
   - send $(\mathsf{atpake.corrupt}, \mathsf{P})$ to $\mathcal{F}_{\mathsf{atPAKE}}$
9. On $(\mathsf{sapake.stealpwdfile}, (sid_\mathsf{T}||j))$ from $\mathcal{A}^*$:
   - retrieve $(\mathsf{targetfile}, [sid_\mathsf{A}], sid_\mathsf{T}, j, [\mathsf{pw}, \mathsf{rw}_j])$ and mark it COMPR (abort if not found)
   - send $(\mathsf{atpake.stealtargetfile}, sid_\mathsf{T}, j)$ to $\mathcal{F}_{\mathsf{atPAKE}}$

Authentication Phase (I)

10. On $(\mathsf{atpake.usersession}, \mathsf{U}', sid_\mathsf{A}, sid_\mathsf{T}, j, ssid)$ from $\mathcal{F}_{\mathsf{atPAKE}}$:
    - send $(\mathsf{ppss.urec}, sid_\mathsf{A}, ssid, \mathsf{U}')$ to $\mathcal{A}^*$
    - save $(\mathsf{session}, \mathsf{U}', \mathbf{S}_{sid_\mathsf{T}}[j], sid_\mathsf{A}, sid_\mathsf{T}, j, ssid, \perp, \perp)$ marked PRELIM
11. On $(\mathsf{ppss.urec}, sid_\mathsf{A}, ssid, \mathsf{pw}^*)$ from $\mathcal{A}^*$:
    - send $(\mathsf{ppss.urec}, sid_\mathsf{A}, ssid, \mathcal{A}^*)$ to $\mathcal{A}^*$
    - save $(\mathsf{urec}, sid_\mathsf{A}, ssid, \mathsf{pw}^*, r[sid_\mathsf{A}, \mathsf{pw}^*])$ marked FRESH
12. On $(\mathsf{atpake.auxsession}, sid_\mathsf{A}, i, ssid_\mathsf{A})$ from $\mathcal{F}_{\mathsf{atPAKE}}$:
    - send $(\mathsf{ppss.srec}, sid_\mathsf{A}, i, ssid_\mathsf{A})$ to $\mathcal{A}^*$
    - set $\mathsf{tx}[sid_\mathsf{A}, i]{+}{+}$
13. On $(\mathsf{ppss.srec}, sid_\mathsf{A}, i, ssid_\mathsf{A})$ from $\mathcal{A}^*$:
    - retrieve $\mathsf{rec} = (\mathsf{auxiliaryfile}, sid_\mathsf{A}, i)$ (abort if not found)
    - set $\mathsf{tx}[sid_\mathsf{A}, i]{+}{+}$ if rec is marked ADVERSARIAL; otherwise send $(\mathsf{atpake.auxproceed}, \mathsf{U}', ssid, \mathbf{C})$ to $\mathcal{F}_{\mathsf{atPAKE}}$
14. On $(\mathsf{ppss.finrec}, sid_\mathsf{A}, ssid, \mathbf{C}, \perp, \perp, \perp)$ from $\mathcal{A}^*$ s.t. $\exists$ record $\mathsf{urec} = (\mathsf{session}, [\mathsf{U}', \mathbf{S}_{sid_\mathsf{T}}[j]], sid_\mathsf{A}, [sid_\mathsf{T}, j], ssid, \perp, \perp)$ marked PRELIM:
    - abort if $\exists_{i \in \mathbf{C}}\ \mathsf{tx}[sid_\mathsf{A}, i] = 0$, else $\forall_{i \in \mathbf{C}}$ set $\mathsf{tx}[sid_\mathsf{A}, i]{-}{-}$
    - change urec's mark to FRESH
    - if record $(\mathsf{userinit}, [\mathsf{U}], sid_\mathsf{A}, [sid'_\mathsf{T}])$ is marked ADVERSARIAL, send $(\mathsf{atpake.auxactive}, \mathsf{U}', ssid)$ to $\mathcal{F}_{\mathsf{atPAKE}}$; otherwise, send $(\mathsf{atpake.auxproceed}, \mathsf{U}', ssid, \mathbf{C})$
    - send $(\mathsf{sapake.usrsession}, (sid_\mathsf{T}||j), ssid, \mathsf{U}', \mathbf{S}_{sid_\mathsf{T}}[j])$ to $\mathcal{A}^*$
15. On $(\mathsf{ppss.finrec}, sid_\mathsf{A}, ssid, \mathbf{C}, \perp, \mathsf{pw}^*, \mathsf{rw}^*)$ from $\mathcal{A}^*$ s.t. $\exists$ record $\mathsf{urec} = (\mathsf{urec}, sid_\mathsf{A}, ssid, [\mathsf{pw}^*], [\mathsf{rw}^*])$ marked FRESH:
    - abort if $\exists_{i \in \mathbf{C}}\ \mathsf{tx}[sid_\mathsf{A}, i] = 0$, else $\forall_{i \in \mathbf{C}}\ \mathsf{tx}[sid_\mathsf{A}, i]{-}{-}$
    - change urec's mark to COMPLETED
    - send $(\mathsf{ppss.urec}, sid_\mathsf{A}, ssid, \mathsf{rw}^*)$ to $\mathcal{A}^*$
    - $\forall_{i \in \mathbf{C}}$ send $(\mathsf{atpake.offlinetestpwd}, sid_\mathsf{A}, i, \mathsf{pw}^*)$ to $\mathcal{F}_{\mathsf{atPAKE}}$
    - save $(\mathsf{completedurec}, sid_\mathsf{A}, \mathsf{pw}^*, ssid_\mathsf{A})$

Fig. 27. Simulator $\mathsf{SIM}_{\mathsf{aPPSS-atPAKE}}$ for protocol $\Pi_{\mathsf{aPPSS-atPAKE}}$, part 2: Authentication Phase (I)

Authentication Phase (II)

16. On $(\mathsf{atpake.targetsession}, sid_\mathsf{T}, j, \mathsf{U}', ssid)$ from $\mathcal{F}_{\mathsf{atPAKE}}$:
    - retrieve $(\mathsf{targetfile}, [sid_\mathsf{A}], sid_\mathsf{T}, j, [\mathsf{pw}, \mathsf{rw}_j])$
    - send $(\mathsf{sapake.svrsession}, (sid_\mathsf{T}||j), ssid, \mathsf{U}', \mathbf{S}_{sid_\mathsf{T}}[j])$ to $\mathcal{A}^*$
    - save $(\mathsf{session}, \mathbf{S}_{sid_\mathsf{T}}[j], \mathsf{U}', sid_\mathsf{A}, sid_\mathsf{T}, j, ssid, \mathsf{pw}, \mathsf{rw}_j)$ marked FRESH
17. On $(\mathsf{sapake.interrupt}, (sid_\mathsf{T}||j), ssid, \mathbf{S}_{sid_\mathsf{T}}[j])$ from $\mathcal{A}^*$:
    - send $(\mathsf{atpake.interrupt}, sid_\mathsf{T}, j, ssid)$ to $\mathcal{F}_{\mathsf{atPAKE}}$
18. On $(\mathsf{sapake.testpwd}, (sid_\mathsf{T}||j), ssid, \mathsf{P}, \mathsf{rw}_j^*)$ from $\mathcal{A}^*$:
    - retrieve $\mathsf{rec} = (\mathsf{session}, \mathsf{P}, [\mathsf{P}', sid_\mathsf{A}], sid_\mathsf{T}, j, ssid, [\mathsf{pw}', \mathsf{rw}_j'])$ (abort if not found)
    - if $\mathsf{rw}_j' \neq \perp$ and $\mathsf{rw}_j^* = \mathsf{rw}_j'$, send $(\mathsf{atpake.testpwd}, \mathsf{P}, ssid, \mathsf{pw}')$ to $\mathcal{F}_{\mathsf{atPAKE}}$
    - else if $\mathsf{rw}_j' \neq \perp$ but $\mathsf{rw}_j^* \neq \mathsf{rw}_j'$, send $(\mathsf{atpake.testpwd}, \mathsf{P}, ssid, (\mathsf{pw}'||0))$
    - else if $\exists\ \mathsf{pw}^*$ s.t. $\mathsf{rw}_j^* = \mathsf{KDF}(\mathsf{rw}^*), \mathbf{S}_{sid_\mathsf{T}}[j])$ where $r[sid_\mathsf{A}, \mathsf{pw}^*] = \mathsf{rw}^*$, send $(\mathsf{atpake.testpwd}, \mathsf{P}, ssid, \mathsf{pw}^*)$
    - else, send $(\mathsf{atpake.testpwd}, \mathsf{P}, ssid, \perp)$
    - in any case, forward the response (if any) to $\mathcal{A}^*$
    - if the response is "correct guess", mark rec COMPR and set $\mathsf{cflag}[sid_\mathsf{T}, j] :=$ COMPR
19. On $(\mathsf{sapake.impersonate}, (sid_\mathsf{T}||j), ssid)$ from $\mathcal{A}^*$:
    - send $(\mathsf{atpake.impersonate}, sid_\mathsf{T}, j, ssid)$ to $\mathcal{F}_{\mathsf{atPAKE}}$
    - forward the response (if any) to $\mathcal{A}^*$
    - if the response is "correct guess", mark rec COMPR
20. On $(\mathsf{sapake.newkey}, (sid_\mathsf{T}||j), ssid, \mathsf{P}, K^*)$ from $\mathcal{A}^*$:
    - retrieve $\mathsf{rec} = (\mathsf{session}, \mathsf{P}, [\mathsf{P}', sid_\mathsf{A}], sid_\mathsf{T}, j, ssid, [\mathsf{pw}, \mathsf{rw}_j])$ not marked PRELIM (abort if not found)
    - if rec is not COMPR and $\mathsf{cflag}[sid_\mathsf{T}, j] =$ UNCOMPROMISED, set $K^* \leftarrow_\$ \{0,1\}^\tau$
    - send $(\mathsf{atpake.newkey}, \mathsf{P}, ssid, K^*)$ to $\mathcal{F}_{\mathsf{atPAKE}}$
    - mark rec COMPLETED
21. On $(\mathsf{sapake.testabort}, (sid_\mathsf{T}||j), ssid, \mathsf{U}')$ from $\mathcal{A}^*$:
    - send $(\mathsf{atpake.testabort}, \mathsf{U}', sid_\mathsf{T}, j, ssid)$ to $\mathcal{F}_{\mathsf{atPAKE}}$
    - forward the response (if any) to $\mathcal{A}^*$

Offline Password Tests

22. On $(\mathsf{sapake.offlinetestpwd}, (sid_\mathsf{T}||j), \mathsf{rw}_j^*)$ from $\mathcal{A}^*$:
    - retrieve $(\mathsf{targetfile}, [sid_\mathsf{A}], sid_\mathsf{T}, j, [\mathsf{pw}, \mathsf{rw}_j])$
    - if $\mathsf{rw}_j^* = \mathsf{rw}_j \neq \perp$, send "correct guess" to $\mathcal{A}^*$
    - else if $\exists$ record $(\mathsf{completedurec}, sid_\mathsf{A}, [\mathsf{pw}^*])$ s.t. $\mathsf{rw}_j^* = \mathsf{KDF}(\mathsf{rw}^*, \mathbf{S}_{sid_\mathsf{T}}[j])$, where $r[sid_\mathsf{A}, \mathsf{pw}^*] = \mathsf{rw}^*$, send $(\mathsf{atpake.offlinetestpwd}, sid_\mathsf{A}, \perp, j, \mathsf{pw}^*)$ to $\mathcal{F}_{\mathsf{atPAKE}}$ and forward the response to $\mathcal{A}^*$
    - else, send "wrong guess" to $\mathcal{A}^*$
    - in any case, if the response is "correct guess", set $\mathsf{cflag}[sid_\mathsf{T}, j] :=$ COMPR

Fig. 28. Simulator $\mathsf{SIM}_{\mathsf{aPPSS-atPAKE}}$ for protocol $\Pi_{\mathsf{aPPSS-atPAKE}}$, part 3: Authentication Phase (II) and Offline Password Tests

Theorem 6. *Protocol $\Pi_{\mathsf{aPPSS-atPAKE}}$ realizes functionality $\mathcal{F}_{\mathsf{atPAKE(weak)}}$ with parameters t and n in the $(\mathcal{F}_{\mathsf{aPPSS}}, \mathcal{F}_{\mathsf{saPAKE}}, \mathcal{F}_{\mathsf{channel}})$-hybrid model, i.e. for any efficient adversary $\mathcal{A}$ against protocol $\Pi_{\mathsf{aPPSS-atPAKE}}$, there exists a simulator SIM s.t. environment $\mathcal{Z}$ can only distinguish the view of $\mathcal{A}$ interacting with the real $\Pi_{\mathsf{aPPSS-atPAKE}}$ protocol and the view of SIM interacting with the ideal functionality $\mathcal{F}_{\mathsf{atPAKE(weak)}}$ with negligible advantage.*

G Partially Oblivious 3HashTDH

In this section we provide an ideal functionality for Partially Oblivious Pseudorandom Functions (POPRFs) and a protocol that we prove to realize that functionality (Section 3.4 provides a high-level overview of these contributions).

G.1 Security Model

Figure 29 shows $\mathcal{F}_{\mathsf{tPOPRF}}$, a generalization of our $\mathcal{F}_{\mathsf{tOPRF}}$ functionality (Figure 10) to support partially oblivious inputs. As with $\mathcal{F}_{\mathsf{tOPRF}}$, the functionality has two "modes" corresponding to a static and an adaptive corruption model.

Like the full version of $\mathcal{F}_{\mathsf{tOPRF}}$ (Figure 10), $\mathcal{F}_{\mathsf{tPOPRF}}$ incorporates a transcript integrity feature. As explained in Appendix B, this property provides a guarantee that evaluation behaves as expected whenever the man-in-the-middle adversary is passive. If the honest parties have a means of comparing their transcripts, they can then verify after the fact that an evaluation was correct. Looking ahead, our tPOPRF-based atPAKE construction will perform exactly this verification during its initialization phase (during which it is assumed that the client and servers have secure and authenticated channels over which they can send the transcripts) after evaluating the tPOPRF on the client's password. In contrast, our tOPRF-based construction initializes a whole new tOPRF for each new atPAKE initialization, so there is no evaluation to be verified and no need for transcripts.

G.2 The P3HashTDH Protocol

Figure 30 is $\Pi_{\mathsf{P3HashTDH}}$, the partially oblivious 3HashTDH protocol. Like Π_{3HashTDH} (Figure 3), it uses $\mathcal{F}_{\mathsf{channel}}$ (Figure 1) as a building block.

P3HashTDH uses prime-order groups $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ of size m with a bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ and (contrary to its name) *four* hash functions (since the two parts of the input are hashed separately), $H_1 : \{0,1\}^* \to \mathbb{G}_1$, $H_1' : \{0,1\}^* \to \mathbb{G}_2$, $H_2 : \{0,1\}^* \to \mathbb{G}_T$, and $H_3 : \{0,1\}^* \to \{0,1\}^l$. The PRF it ultimately computes is $F_k(x_{\mathsf{priv}}, x_{\mathsf{pub}}) = H_3(x_{\mathsf{priv}}, x_{\mathsf{pub}}, e(H_1(x_{\mathsf{priv}}), H_1'(x_{\mathsf{pub}}))^k)$, which is very similar to the "Pythia" POPRF shown in [28] (Pythia lacks the outer hash H_3, which it does not need because it does not use a simulation-based security model).

Notation

Initially $\mathsf{tx}[sid, ssid_{\mathsf{S}}, x_{\mathsf{pub}}, i] := 0$ for all sid, $ssid_{\mathsf{S}}$, x_{pub}, and i. Also, $F_{sid}(x_{\mathsf{priv}}, x_{\mathsf{pub}})$ is undefined for all sid, x_{pub}, and x_{priv}. When first referenced, the functionality assigns $F_{sid}(x_{\mathsf{priv}}, x_{\mathsf{pub}}) \leftarrow_{\$} \{0,1\}^l$.

Initialization

1. On $(\mathsf{tpoprf.init}, sid, (x_{\mathsf{priv},1}, x_{\mathsf{pub},1}), \ldots, (x_{\mathsf{priv},k}, x_{\mathsf{pub},k}))$ from $\mathsf{P}_0 \in \mathcal{P}^*$, if this is the first such call for sid:
   - send $(\mathsf{tpoprf.init}, sid, \mathsf{P}_0)$ to $\mathcal{A}^*$
   - send $(\mathsf{tpoprf.initeval}, sid, F_{sid}(x_{\mathsf{priv},1}, x_{\mathsf{pub},1}), \ldots, F_{sid}(x_{\mathsf{priv},k}, x_{\mathsf{pub},k}))$ to P_0
   - save $(\mathsf{tpoprf.init}, sid, \mathsf{P}_0)$ and mark it TAMPERED if $\mathsf{P}_0 \in \mathbf{Corr}$
2. On $(\mathsf{tpoprf.sinit}, sid, i, \mathsf{P}_0)$ from S where $\mathsf{S} = \mathbf{S}_{sid}[i]$ or ($\mathsf{S} = \mathcal{A}^*$ and $\mathbf{S}_{sid}[i] \in \mathbf{Corr}$), save record $(\mathsf{tpoprf.sinit}, sid, i, \mathsf{P}_0)$ marked INACTIVE
3. On $(\mathsf{tpoprf.fininit}, sid, i)$ from $\mathcal{A}^*$ where $\exists$ record $\mathsf{urec} = (\mathsf{tpoprf.init}, sid, \mathsf{P}_0)$ and record $\mathsf{srec} = (\mathsf{tpoprf.sinit}, sid, i, \mathsf{P}_0)$ marked INACTIVE:
   - send $(\mathsf{tpoprf.fininit}, sid, i)$ to $\mathbf{S}_{sid}[i]$
   - if urec is TAMPERED then mark srec TAMPERED
   - else if $\mathbf{S}_{sid}[i] \in \mathbf{Corr}$ then mark srec COMPR
   - else (i.e. urec is not TAMPERED and $\mathbf{S}_{sid}[i] \notin \mathbf{Corr}$) mark srec ACTIVE

Corruption

4. On $(\mathsf{tpoprf.corrupt}, \mathsf{P})$ from $\mathcal{A}^*$ (with permission from $\mathcal{Z}$):
   - set $\mathbf{Corr} := \mathbf{Corr} \cup \{\mathsf{P}\}$
   - mark every ACTIVE record $(\mathsf{tpoprf.sinit}, sid, i, \mathsf{P}_0)$ COMPR where $\mathsf{P} = \mathbf{S}_{sid}[i]$

Evaluation

5. On $(\mathsf{tpoprf.eval}, sid, ssid_{\mathsf{U}}, x_{\mathsf{priv}}, x_{\mathsf{pub}})$ from $\mathsf{U} \in \mathcal{P}^*$, if this is the first call from U for sid and $ssid_{\mathsf{U}}$:
   - send $(\mathsf{tpoprf.eval}, sid, ssid_{\mathsf{U}}, \mathsf{U}, x_{\mathsf{pub}})$ to $\mathcal{A}^*$,
   - and save $(\mathsf{tpoprf.eval}, sid, ssid_{\mathsf{U}}, \mathsf{U}, x_{\mathsf{priv}}, x_{\mathsf{pub}})$ marked FRESH.
6. On $(\mathsf{tpoprf.sndrcomplete}, sid, i, ssid_{\mathsf{S}}, x_{\mathsf{pub}})$ from S where $\exists$ record $\mathsf{srec} = (\mathsf{tpoprf.sinit}, sid, i, \mathsf{P}_0)$ not marked INACTIVE and ($\mathsf{S} = \mathbf{S}_{sid}[i]$ or ($\mathsf{S} = \mathcal{A}^*$ and srec is marked COMPR or TAMPERED)):
   - send $(\mathsf{tpoprf.sndrcomplete}, sid, i, ssid_{\mathsf{S}}, x_{\mathsf{pub}})$ to $\mathcal{A}^*$
   - and await $(\mathsf{tpoprf.sndrtrans}, sid, i, ssid_{\mathsf{S}}, \mathtt{tr}_i)$ from $\mathcal{A}^*$;
   - then send $(\mathsf{tpoprf.sndrtrans}, sid, i, ssid_{\mathsf{S}}, \mathtt{tr}_i)$ to S
   - if srec is not TAMPERED, then save $(\mathsf{tpoprf.sndrtrans}, sid, i, \mathtt{tr}_i)$
   - (regardless of the above) set $\mathsf{tx}[sid, ssid_{\mathsf{S}}, x_{\mathsf{pub}}, i]{+}{+}$
7. On $(\mathsf{tpoprf.rcvcomplete}, sid, ssid_{\mathsf{U}}, sid^*, ssid^*_{\mathsf{S}}, \mathbf{C}, \mathtt{tr}_{\mathsf{U}})$ from $\mathcal{A}^*$ where $|\mathbf{C}| = t+1$ and $\exists$ record $(\mathsf{tpoprf.eval}, sid, ssid_{\mathsf{U}}, \mathsf{U}, x_{\mathsf{priv}}, x_{\mathsf{pub}})$ marked FRESH:
   - if (i) $\exists j \in \mathbf{C}$ such that $\mathsf{tx}[sid^*, ssid^*_{\mathsf{S}}, x_{\mathsf{pub}}, j] = 0$, or (ii) $\exists$ a set of records $\{(\mathsf{tpoprf.sndrtrans}, sid', j, \mathtt{tr}_{\mathsf{U}}[j])\}_{j \in \mathbf{C}}$ such that $sid' \neq sid^*$, then abort
   - otherwise mark the record COMPLETED, set $\mathsf{tx}[sid^*, ssid^*_{\mathsf{S}}, x_{\mathsf{pub}}, j]{-}{-}$ for all $j \in \mathbf{C}$, and send $(\mathsf{tpoprf.eval}, sid, ssid_{\mathsf{U}}, F_{sid^*}(x_{\mathsf{priv}}, x_{\mathsf{pub}}), \mathtt{tr}_{\mathsf{U}})$ to U

Fig. 29. $\mathcal{F}_{\mathsf{tPOPRF}}$: threshold partially oblivious PRF functionality, parameterized by threshold t, number of servers n, and output length l. For the static corruption model, routine 4 is omitted. Shadowed text can be ignored if transcript integrity is unneeded.

Notation

$\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ are cyclic groups of prime order m with generators g_1, g_2, and g_T, respectively. $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a bilinear pairing.
$H_1 : \{0,1\}^* \to \mathbb{G}_1$, $H_1' : \{0,1\} \to \mathbb{G}_2$, $H_2 : \{0,1\}^* \to \mathbb{G}_T$, and $H_3 : \{0,1\}^* \to \{0,1\}^l$ are hash functions.

Initialization

1. On input $(\mathsf{tpoprf.init}, sid, (x_{\mathsf{priv},1}, x_{\mathsf{pub},1}), \ldots, (x_{\mathsf{priv},k}, x_{\mathsf{pub},k}))$, initializer P_0 does:
   - pick $\alpha_0, ..., \alpha_t \leftarrow_\$ \mathbb{Z}_m$ and define polynomial $k(x) := \alpha_0 + \alpha_1 x + ... + \alpha_t x^t$
   - pick $\beta_1, ..., \beta_t \leftarrow_\$ \mathbb{Z}_m$ and define polynomial $z(x) := \beta_1 x + \beta_2 x^2 + ... + \beta_t x^t$
   - for each $i \in [n]$, send $(\mathsf{channel.send}, [sid||i], \mathbf{S}_{sid}[i], (i, k(i), z(i)))$ to $\mathcal{F}_{\mathsf{channel}}$
   - and output $(\mathsf{tpoprf.initeval}, sid, y_1, \ldots, y_k)$ where $y_j := H_3(x_{\mathsf{priv},j}, x_{\mathsf{pub},j}, e(H_1(x_{\mathsf{priv},j}), H_1'(x_{\mathsf{pub},j}))^{\alpha_0})$ for every $j \in [k]$
2. On input $(\mathsf{tpoprf.sinit}, sid, i, \mathsf{P}_0)$, server $\mathbf{S}_{sid}[i]$ does:
   - await $(\mathsf{channel.deliver}, [sid||i], \mathsf{P}_0, (i, k_i, z_i))$ from $\mathcal{F}_{\mathsf{channel}}$;
   - then save record $(\mathsf{tpoprf.share}, sid, i, k_i, z_i)$
   - output $(\mathsf{tpoprf.fininit}, sid, i)$

Evaluation

3. On input $(\mathsf{tpoprf.eval}, sid, ssid_{\mathsf{U}}, x_{\mathsf{priv}}, x_{\mathsf{pub}})$, evaluator U does:
   - pick $r \leftarrow_\$ \mathbb{Z}_m$ and compute $a := H_1(x_{\mathsf{priv}})^r$
   - for each $i \in [n]$, send $(sid, i, ssid_{\mathsf{U}}, a, x_{\mathsf{pub}})$ to $\mathbf{S}_{sid}[i]$
   - await responses $(sid, i, ssid_{\mathsf{U}}, ssid_{\mathsf{S}}, b_i)$ from $\mathbf{S}_{sid}[i]$ for all $i \in \mathbf{C}$, for any set $\mathbf{C} \subseteq [n]$ of size $t+1$;
   - then compute $b := \prod_{i \in \mathbf{C}} b_i^{\lambda_i}$ where λ_i is the Lagrange interpolation coefficient for index i and index set $\mathbf{C}$
   - output $(\mathsf{tpoprf.eval}, ssid_{\mathsf{U}}, H_3(x_{\mathsf{priv}}, x_{\mathsf{pub}}, b^{1/r}), (\mathtt{tr}_1, ..., \mathtt{tr}_n))$ where $\mathtt{tr}_i := (ssid_{\mathsf{S}}, a, x_{\mathsf{pub}}, b_i)$ for all $i \in \mathbf{C}$ and $\mathtt{tr}_i := \perp$ for all $i \notin \mathbf{C}$
4. On input $(\mathsf{tpoprf.sndrcomplete}, sid, i, ssid_{\mathsf{S}}, x_{\mathsf{pub}})$, server $\mathbf{S}_{sid}[i]$ does:
   - retrieve record $(\mathsf{tpoprf.share}, sid, i, k_i, z_i)$ (abort if not found)
   - await $(sid, i, ssid'_{\mathsf{U}}, a, x_{\mathsf{pub}})$ from any U (if it hasn't already been received);
   - then compute $b_i := e(a, H_1'(x_{\mathsf{pub}}))^{k_i} \cdot H_2(ssid_{\mathsf{S}}, a)^{z_i}$
   - output $(\mathsf{tpoprf.sndrtrans}, sid, i, ssid_{\mathsf{S}}, (ssid_{\mathsf{S}}, a, x_{\mathsf{pub}}, b_i))$
   - and send response $(sid, i, ssid'_{\mathsf{U}}, ssid_{\mathsf{S}}, b_i)$ to U

Fig. 30. Protocol $\Pi_{\mathsf{P3HashTDH}}$ which realizes $\mathcal{F}_{\mathsf{tPOPRF}}$ in the $\mathcal{F}_{\mathsf{channel}}$-hybrid world. Shadowed text can be ignored if transcript integrity is unneeded.

P3HashTDH is essentially a combination of our 3HashTDH tOPRF protocol with the pairing-based POPRF protocol of Pythia. As in 3HashTDH, each server S_i holds a Shamir secret share k_i of the PRF key k, as well as a Shamir secret share z_i of zero. The evaluator first picks $r \leftarrow_\$ \mathbb{Z}_m$ and sends the blinded private input $a := H_1(x_{\mathsf{priv}})^r$ to $t+1$ servers. Server i responds with $b_i := e(a, H_1'(x_{\mathsf{pub}}))^{k_i} \cdot H_2(ssid_{\mathsf{S}}, a)^{z_i}$. To combine these responses, the

evaluator uses polynomial interpolation in the exponent to compute $b := \prod_i b_i^{\lambda_i}$, where the λs are Lagrange interpolation coefficients. As long as all servers used the same $(a, ssid_{\mathsf{S}})$, the evaluator will correctly compute $b = \prod_i e(a, H_1'(x_{\mathsf{pub}}))^{k_i \lambda_i} \cdot \prod_i H_2(ssid_{\mathsf{S}}, a)^{z_i \lambda_i}$, the second part of which interpolates to $H_2(ssid_{\mathsf{S}}, a)^0 = 1$ and disappears. Finally, the bilinear pairing property enables the evaluator to correctly remove the blinding exponent r and compute the final output $H_3(x_{\mathsf{priv}}, x_{\mathsf{pub}}, b^{1/r})$.

G.3 Security Analysis

Under the Gap One-More *Bilinear* Diffie Hellman (GapOMBDH) assumption (replacing the non-bilinear GapOMDH assumption), the P3HashTDH protocol in Figure 30 is as secure as 3HashTDH.

Theorem 7. *Protocol P3HashTDH realizes functionality $\mathcal{F}_{\mathsf{tPOPRF}}$ with parameters t and n in the $\mathcal{F}_{\mathsf{channel}}$-hybrid model, assuming static corruptions, hash functions H_1, H_1', H_2, and H_3 modeled as random oracles, the GapOMBDH assumption on pairing e, and the DDH assumption on group $\mathbb{G}_T$.*

Specifically, for any efficient adversary $\mathcal{A}$ against protocol P3HashTDH, there exists a simulator SIM such that no efficient environment $\mathcal{Z}$ can distinguish the view of $\mathcal{A}$ interacting with the real P3HashTDH protocol and the view of SIM interacting with the ideal functionality $\mathcal{F}_{\mathsf{tPOPRF}}$ with advantage better than $q_I^2/m + q_I \cdot (t \cdot \mathbf{Adv}_{\mathcal{Q}}^{\mathsf{DDH}} + \mathbf{Adv}_{\mathcal{R}}^{\mathsf{GapOMBDH}}) \cdot (t+1)$ where q_I is the number of tPOPRF instances, $m = |\mathbb{G}_T|$, and $\mathbf{Adv}_{\mathcal{R}}^{\mathsf{GapOMBDH}}$ and $\mathbf{Adv}_{\mathcal{Q}}^{\mathsf{DDH}}$ are bounds on the probability that any efficient algorithm violates the GapOMBDH and DDH assumptions, respectively.

Theorem 8. *In the case of adaptive corruptions, the statement from Theorem 7 still holds under the additional assumption that $\binom{n}{t'}$ is a polynomial function of the security parameter for all $0 \le t' \le t$.*

Specifically, no efficient adversary $\mathcal{A}$ against P3HashTDH has distinguishing advantage better than $q_I^2/m + q_I \cdot (t \cdot \mathbf{Adv}_{\mathcal{Q}}^{\mathsf{DDH}} + \mathbf{Adv}_{\mathcal{R}}^{\mathsf{GapOMBDH}}) \cdot \sum_{t'=0}^{t} \binom{n}{t'}$.

The proof of Theorems 7 and 8 is extremely similar to the security proof for non-partial 3HashTDH (Appendix B). To avoid redundancy, we provide only an overview of the important differences.

Proof. For any adversary $\mathcal{A}^*$, we construct simulator $\mathsf{SIM}_{\mathsf{P3HashTDH}}$, which is extremely similar to simulator $\mathsf{SIM}_{\mathsf{3HashTDH}}$ shown in Figures 13, 14, and 15. We reuse the notation, conventions, and series of game changes from the proof in Appendix B. With the alterations detailed below, these game changes relate the real world $\mathbf{G}_0$ to the simulated world $\mathbf{G}_7$.

Game $\mathbf{G}_2$: H_1' also has a trapdoor. Like $H_1(x_{\mathsf{priv}})$, $H_1'(x_{\mathsf{pub}})$ samples a random exponent $\tau' \leftarrow_{\$} \mathbb{Z}_m$ and returns $h' = (g_2)^{\tau'}$. A record $(x_{\mathsf{pub}}, \tau', h')$ is saved.

Game $\mathbf{G}_3$: The public input x_{pub} is also used to track evaluation legality.

– As honest servers perform evaluations (i.e. toprf.sndrcomplete) the evalset of each $(ssid_{\mathsf{S}}, a, x_{\mathsf{pub}})$ triple is tracked.
– Subroutine FindEvalset (Figure 14) is modified to interpret its first parameter as $\tilde{k}$ and to take a second parameter x_{pub}.
  • Before anything else, it retrieves the trapdoor τ' such that $H_1'(x_{\mathsf{pub}}) = (g_2)^{\tau'}$ (if H_1' was never queried on x_{pub}, then make such a query now on behalf of $\mathcal{Z}$). It then computes k^* as $\tilde{k}^{1/\tau'}$ and goes on to use it as in the other proof (i.e. by finding or creating a PRF instance initialized with key $k(0)$ such that $k^* = g_T^{k(0)}$).
  • It also must include x_{pub} in its toprf.sndrcomplete messages to $\mathcal{F}_{\mathsf{tPOPRF}}$ when it prints new tickets on behalf of corrupted servers.
  • It ultimately finds a pair $(ssid_{\mathsf{S}}^*, a)$ that has been evaluated by $t+1$ servers specifically for the public input x_{pub}. (This is accomplished using evalset, which tracks $(ssid_{\mathsf{S}}, a, x_{\mathsf{pub}})$ triples during toprf.sndrcomplete as explained above.)
– H_3 queries by the adversary make use of the expanded FindEvalset interface to check for illegal evaluations. In particular, upon fresh query $(x_{\mathsf{priv}}, x_{\mathsf{pub}}, u)$ where $H_1(x_{\mathsf{priv}}) = g_1^{\tau}$, H_3 calls FindEvalset$(u^{1/\tau}, x_{\mathsf{pub}})$. If that subroutine succeeds at finding a matching evaluation set, then H_3 uses $\mathcal{F}_{\mathsf{tOPRF}}$'s toprf.rcvcomplete interface to query the PRF output y for $(x_{\mathsf{priv}}, x_{\mathsf{pub}})$. (Just as in the other proof, H_3 does not yet return y in this game.)
– The honest clients' PRF evaluation process does accordingly.

As in the other proof, this simulation strategy guarantees that a successful FindEvalset check always precedes a toprf.rcvcomplete command to $\mathcal{F}_{\mathsf{tPOPRF}}$ and that this evalset-based check is always at least as restrictive as the functionality's internal "ticketing" mechanism. The same auxiliary series of game changes is used to bound the probability of the FAIL event.

Reduction $\mathcal{R}_k$: GapOMBDH. The GapOMBDH input is a vector $(y^*, h_1, \ldots, h_q, h_1', \ldots, h_q')$ where all h_j are uniformly random group elements in $\mathbb{G}_1$, all $h_{j'}'$ are uniformly random group elements in $\mathbb{G}_2$, and $y^* = g_T^s$ for some uniformly random secret exponent s. The reduction has access to an oracle OMDH(a), which returns a^s given $a \in \mathbb{G}_T$. It also has access to an oracle DDH(y, h, u), which returns a bit that is 1 if and only if (y, h, u) is a Diffie Hellman tuple in $\mathbb{G}_T$ (i.e. there exist $a, b \in \mathbb{Z}_m$ such that $y = g_T^a$, $h = g_T^b$, and $u = g_T^{ab}$). The reduction wins if it outputs a set W of triples $(j, j', e(h_j, h_{j'}')^s)$ and $|W|$ is greater than the number of (unique) queries it made to the OMDH oracle.

For all $1 \le k \le t$, $\mathcal{R}_k$ is constructed from $\mathcal{Z} \leftrightarrow \mathbf{H}_k$ with the following changes:

– Initially, set $W := \emptyset$.
– Queries to H_1 and H_1' are answered with the h_j and $h_{j'}'$ values. Specifically, $H_1(x_j) := h_j$ where j is initialized as 1 and increments after a fresh x is seen; $H_1'(x_{j'}) := h_{j'}'$ where j' is also initialized as 1 and increments after a fresh x is seen. As a consequence of this change, H_3 no longer has trapdoors to rely on.

- H_3 uses the DDH oracle in place of the H_1 and H_1' trapdoors. Specifically, upon fresh query $H_3(x_{\mathsf{priv}}, x_{\mathsf{pub}}, u)$ where $H_1(x_{\mathsf{priv}}) = h_j$ and $H_1'(x_{\mathsf{pub}}) = h'_{j'}$ (if H_1 or H_1' were never queried on x_{priv} or x_{pub}, respectively, then make such queries now on behalf of $\mathcal{Z}$), do the following:
  - retrieve every initialization record $(\mathsf{toprf.init}, sid, \mathcal{A}^*, y_{sid})$ where $sid \neq sid^*$ and query $\mathsf{DDH}(y_{sid}, e(h_j, h'_{j'}), u)$ for each one,
  - also query $\mathsf{DDH}(y^*, e(h_j, h'_{j'}), u)$,
  - if all oracle responses are 0, then simply set $H_3(x_{\mathsf{priv}}, x_{\mathsf{pub}}, u) \leftarrow_{\$} \{0,1\}^l$ and return it,
  - but otherwise, if some $\mathsf{DDH}(y, e(h_j, h'_{j'}), u) = 1$, then call $\mathsf{FindEvalset}(y, x_{\mathsf{pub}})$, and proceed accordingly (if $y = y^*$, then also save $W := W \cup \{(j, j', u)\}$).
- $\mathsf{FindEvalset}$ can now interpret its first parameter as k^* (as in the other proof). In other words, it does not first use an H_1' trapdoor to unblind $\tilde{k}$ into k^* like it does in the other games of this proof.
- For all a and x_{pub}, for the first k queries to servers not in $\mathbf{Guess}_k$, set κ at random (only for instance sid^*). Specifically, $\forall a \ \forall x_{\mathsf{pub}} \ \forall i \notin \mathbf{Guess}_k \ \kappa_{a,x_{\mathsf{pub}},i} \leftarrow_{\$} \mathbb{G}_T$ if $|(\cup_{ssid_{\mathsf{S}}} \mathsf{evalset}(ssid_{\mathsf{S}}, a, x_{\mathsf{pub}})) - \mathbf{Guess}_k| \leq k$.
- For the $(k+1)$st query and on, use the OMDH oracle and interpolate κ (only for instance sid^*). Specifically, $\forall a \ \forall x_{\mathsf{pub}} \ \forall i \notin \mathbf{Guess}_k$ $\kappa_{a,x_{\mathsf{pub}},i} := \mathsf{OMDH}(e(a, H_1'(x_{\mathsf{pub}})))^{\lambda_0} \prod_{j \in \mathbf{E}} \kappa_{a,x_{\mathsf{pub}},j}^{\lambda_j}$ if $|(\cup_{ssid_{\mathsf{S}}} \mathsf{evalset}(ssid_{\mathsf{S}}, a, x_{\mathsf{pub}})) - \mathbf{Guess}_k| > k$ (where $\mathbf{E}$ is any t element subset of $\cup_{ssid_{\mathsf{S}}} \mathsf{evalset}(ssid_{\mathsf{S}}, a, x_{\mathsf{pub}}) \cup \mathbf{Guess}_k$ and λ_j is the Lagrange interpolation coefficient for index j, index set $\mathbf{E}$, and target index i).
- If a call to $\mathsf{FindEvalset}$ triggers event FAIL followed by $(\text{CORR}, t-k)$, then output W and thereby win the GapOMBDH game.

As in the other proof, these modifications do not change $\mathcal{Z}$'s view at all.

The remainder of the proof proceeds without significant alterations. Apart from the swap of $\mathbf{Adv}_{\mathcal{R}}^{\mathsf{GapOMDH}}$ for $\mathbf{Adv}_{\mathcal{R}}^{\mathsf{GapOMBDH}}$, the distinguishing advantage bounds between each pair of games are identical, and therefore their eventual sums are identical (in the both the static and adaptive corruption models).

□

H atPAKE Construction from tPOPRF

Figure 31 is $\Pi_{\mathsf{tPOPRF-atPAKE}}$, an atPAKE construction built using $\mathcal{F}_{\mathsf{tPOPRF}}$ and $\mathcal{F}_{\mathsf{saPAKE}}$ (Figures 29 and 19). It additionally uses $\mathcal{F}_{\mathsf{channel}}$ (Figure 1) as a building block; that functionality models secure authenticated communication, which is only used by our atPAKE protocol during initialization.

The tPOPRF-atPAKE protocol is similar to our main tOPRF-atPAKE construction from above (Figure 8). User authentication follows the same two steps: auxiliary server interaction followed by an saPAKE session with the target server. The only difference is that a single tPOPRF instance identified

Initialization

1. On input $(\mathsf{atpake.userinit}, sid_{\mathsf{A}}, sid_{\mathsf{T}}, \mathsf{pw})$, user U does:
   - for every $i \in [n]$, send $(\mathsf{channel.send}, (sid_{\mathsf{A}}||i||1), \mathbf{S}_{sid_{\mathsf{A}}}[i], \text{INIT})$ to $\mathcal{F}_{\mathsf{channel}}$
   - send $(\mathsf{tpoprf.eval}, \mathbf{S}_{sid_{\mathsf{A}}}, (sid_{\mathsf{A}}||\text{INIT}), \mathsf{pw}, sid_{\mathsf{A}})$ to $\mathcal{F}_{\mathsf{tPOPRF}}$
   - await $(\mathsf{channel.deliver}, (sid_{\mathsf{A}}||i||2), \mathbf{S}_{sid_{\mathsf{A}}}[i], \mathtt{tr}_i)$ from $\mathcal{F}_{\mathsf{channel}}$ $\forall i \in [n]$
   - await $(\mathsf{tpoprf.eval}, \mathbf{S}_{sid_{\mathsf{A}}}, (sid_{\mathsf{A}}||\text{INIT}), \mathsf{rw}, \mathtt{tr}_{\mathsf{U}})$ from $\mathcal{F}_{\mathsf{tPOPRF}}$
   - abort if $\mathtt{tr}_{\mathsf{U}} \neq (\mathtt{tr}_1, \ldots, \mathtt{tr}_n)$
   - for every $j \in \{1, \ldots, |\mathbf{S}_{sid_{\mathsf{T}}}|\}$, compute $\mathsf{rw}_j := \mathsf{KDF}(\mathsf{rw}, \mathbf{S}_{sid_{\mathsf{T}}}[j])$ and send $(\mathsf{channel.send}, (sid_{\mathsf{A}}||sid_{\mathsf{T}}||j), \mathbf{S}_{sid_{\mathsf{T}}}[j], \mathsf{rw}_j)$ to $\mathcal{F}_{\mathsf{channel}}$
2. On input $(\mathsf{atpake.auxinit}, sid_{\mathsf{A}}, i, \mathsf{U}')$, auxiliary server $\mathbf{S}_{sid_{\mathsf{A}}}[i]$ does:
   - await $(\mathsf{channel.deliver}, (sid_{\mathsf{A}}||i||1), \mathsf{U}', \text{INIT})$ from $\mathcal{F}_{\mathsf{channel}}$
   - if there is a PENDING record $(\mathsf{tpoprf}, \mathbf{S}_{sid_{\mathsf{A}}})$, wait for it to be COMPLETED
   - else if there is no record $(\mathsf{tpoprf}, \mathbf{S}_{sid_{\mathsf{A}}})$ at all, create it marked PENDING and do the following before marking it COMPLETED:
     - if $i = 1$, send $(\mathsf{tpoprf.init}, \mathbf{S}_{sid_{\mathsf{A}}})$ to $\mathcal{F}_{\mathsf{tPOPRF}}$
     - (regardless of the above) send $(\mathsf{tpoprf.sinit}, \mathbf{S}_{sid_{\mathsf{A}}}, i, \mathbf{S}_{sid_{\mathsf{A}}}[1])$ to $\mathcal{F}_{\mathsf{tPOPRF}}$
     - await $(\mathsf{tpoprf.fininit}, \mathbf{S}_{sid_{\mathsf{A}}}, i)$ in response
   - then (in any case) output $(\mathsf{atpake.finishauxiliaryinit}, sid_{\mathsf{A}})$
   - if $i \leq t + 1$, do the following:
     - send $(\mathsf{tpoprf.sndrcomplete}, \mathbf{S}_{sid_{\mathsf{A}}}, i, \text{INIT}, sid_{\mathsf{A}})$ to $\mathcal{F}_{\mathsf{tPOPRF}}$
     - await $(\mathsf{tpoprf.sndrtrans}, \mathbf{S}_{sid_{\mathsf{A}}}, i, \text{INIT}, \mathtt{tr}_i)$ in response
     - send $(\mathsf{channel.send}, (sid_{\mathsf{A}}||i||2), \mathsf{U}', \mathtt{tr}_i)$ to $\mathcal{F}_{\mathsf{channel}}$
   - otherwise, send $(\mathsf{channel.send}, (sid_{\mathsf{A}}||i||2), \mathsf{U}', \perp)$ to $\mathcal{F}_{\mathsf{channel}}$
3. On input $(\mathsf{atpake.targetinit}, sid_{\mathsf{A}}, sid_{\mathsf{T}}, j, \mathsf{U}')$, target server $\mathbf{S}_{sid_{\mathsf{T}}}[j]$ does:
   - await $(\mathsf{channel.deliver}, (sid_{\mathsf{A}}||sid_{\mathsf{T}}||j), \mathsf{U}', \mathsf{rw}_j)$ from $\mathcal{F}_{\mathsf{channel}}$;
   - then send $(\mathsf{sapake.storepwdfile}, (sid_{\mathsf{T}}||j), \mathsf{U}', \mathsf{rw}_j)$ to $\mathcal{F}_{\mathsf{saPAKE}}$
   - output $(\mathsf{atpake.finishtargetinit}, sid_{\mathsf{A}}, sid_{\mathsf{T}})$

Authentication

4. On input $(\mathsf{atpake.usersession}, sid_{\mathsf{A}}, sid_{\mathsf{T}}, j, ssid, \mathsf{pw}')$, user U' does:
   - send $(\mathsf{tpoprf.eval}, \mathbf{S}_{sid_{\mathsf{A}}}, ssid, \mathsf{pw}', sid_{\mathsf{A}})$ to $\mathcal{F}_{\mathsf{tPOPRF}}$
   - await response $(\mathsf{tpoprf.eval}, \mathbf{S}_{sid_{\mathsf{A}}}, ssid, \mathsf{rw}', \mathtt{tr}_{\mathsf{U}})$;
   - then compute $\mathsf{rw}'_j := \mathsf{KDF}(\mathsf{rw}', \mathbf{S}_{sid_{\mathsf{T}}}[j])$ and send $(\mathsf{sapake.usrsession}, (sid_{\mathsf{T}}||j), ssid, \mathbf{S}_{sid_{\mathsf{T}}}[j], \mathsf{rw}'_j)$ to $\mathcal{F}_{\mathsf{saPAKE}}$
   - upon response $(\mathsf{sapake.newkey}, (sid_{\mathsf{T}}||j), ssid, K)$, output $(\mathsf{atpake.newkey}, ssid, K)$
   - upon response $(\mathsf{sapake.abort}, (sid_{\mathsf{T}}||j), ssid)$, output $(\mathsf{atpake.abort}, ssid)$
5. On input $(\mathsf{atpake.auxsession}, sid_{\mathsf{A}}, i, ssid_{\mathsf{A}})$, auxiliary server $\mathbf{S}_{sid_{\mathsf{A}}}[i]$ sends $(\mathsf{tpoprf.sndrcomplete}, \mathbf{S}_{sid_{\mathsf{A}}}, i, ssid_{\mathsf{A}}, sid_{\mathsf{A}})$ to $\mathcal{F}_{\mathsf{tPOPRF}}$
6. On input $(\mathsf{atpake.targetsession}, sid_{\mathsf{T}}, j, \mathsf{U}', ssid)$, target server $\mathbf{S}_{sid_{\mathsf{T}}}[j]$ does:
   - send $(\mathsf{sapake.svrsession}, (sid_{\mathsf{T}}||j), ssid)$ to $\mathcal{F}_{\mathsf{saPAKE}}$
   - upon response $(\mathsf{atpake.newkey}, (sid_{\mathsf{T}}||j), ssid, K)$, output $(\mathsf{atpake.newkey}, ssid, K)$

Fig. 31. Protocol $\Pi_{\mathsf{tPOPRF-atPAKE}}$ which realizes $\mathcal{F}_{\mathsf{atPAKE}}$ in the $(\mathcal{F}_{\mathsf{tPOPRF}}, \mathcal{F}_{\mathsf{saPAKE}}, \mathcal{F}_{\mathsf{channel}})$-hybrid world.

by the auxiliary server list $\mathbf{S}_{sid_\mathsf{A}}$ is common to all users that use the same list $\mathbf{S}_{sid_\mathsf{A}}$. Since tPOPRF instances are shared, the user-specific sid_A is used as the public input to the tPOPRF in order to effectively partition the OPRF by user. Target server saPAKE passwords, then, are computed as $\mathsf{rw}_j = \mathsf{KDF}(F_k(\mathsf{pw}, sid_\mathsf{A}), \mathsf{T})$ (using as an example target server T indexed as j in the target server list $\mathbf{S}_{sid_\mathsf{T}}$).

The initialization of these auxiliary servers is somewhat more complex than in tOPRF-atPAKE. The first time that any particular auxiliary server list is used, the first server in that list plays the role of tPOPRF initializer and establishes a new tPOPRF instance with the servers in the list. After that first time, auxiliary server state does not change as the result of a new user registration. Instead, the auxiliary servers participate in a (transcript-verified) tPOPRF evaluation with the new user, who learns $F_k(\mathsf{pw}, sid_\mathsf{A})$ and can then proceed to derive target server-specific passwords and send them out as in the former protocol.

The security of tPOPRF-atPAKE is proven for ideal functionality $\mathcal{F}_{\mathsf{atPAKE}'}$, which we define to be a variation of $\mathcal{F}_{\mathsf{atPAKE}}$ (Figures 4, 5, and 6) with the following 2 differences:

1. In the finishtargetinit routine (Figure 4), the adversary is given a stronger power to meddle in target file creation. If the initializing user U is corrupt, then $\mathcal{A}^*$ can fully overwrite the sid_A and pw fields of the created targetfile as before. However, even if U is not corrupt, if at least $t+1$ of the auxiliary servers in sid_A are corrupt, then the adversary can overwrite sid_A (but not pw).
   This relaxation is necessary to model adversarial attack on the tPOPRF evaluation that takes place during our protocol's initialization phase. In the tOPRF-based protocol, there is no online evaluation during initialization (the user evaluates locally, instead) and therefore this vulnerability does not exist. Still, due to our assumption of secure and authenticated channels during initialization, this attack is only possible if $t+1$ of the auxiliary servers are corrupted. Our intuitive notion that the protocol is secure as long as initialization is honest still holds.
2. In the auxsession routine (Figure 5), execution does not necessarily abort when there is no auxiliaryfile with a matching sid_A. Instead, execution aborts if there is no auxiliaryfile with any sid'_A such that $\mathbf{S}_{sid_\mathsf{A}} = \mathbf{S}_{sid'_\mathsf{A}}$.
   This change is not a security relaxation; on the contrary, it actually models a valuable feature of our tPOPRF-based construction. Auxiliary servers (whether honest or dishonest) can choose to participate in a session tagged by sid_A even if they were never initialized using sid_A (as long as they did at some point initialize on a sid'_A with the same auxiliary server list). In practice, this means that servers can behave in a way that does not reveal whether or not particular users have registered.

Theorem 9. *Protocol $\Pi_{\mathsf{tPOPRF-atPAKE}}$ realizes functionality $\mathcal{F}_{\mathsf{atPAKE}'}$ with parameters t and n in the $(\mathcal{F}_{\mathsf{tPOPRF}}, \mathcal{F}_{\mathsf{saPAKE}}, \mathcal{F}_{\mathsf{channel}})$-hybrid model.*

Specifically, for any efficient adversary $\mathcal{A}$ against protocol $\Pi_{\mathsf{tPOPRF-atPAKE}}$, there exists a simulator SIM such that no efficient environment $\mathcal{Z}$ can distinguish

the view of $\mathcal{A}$ interacting with the real $\Pi_{\mathsf{tPOPRF-atPAKE}}$ protocol and the view of SIM *interacting with the ideal functionality $\mathcal{F}_{\mathsf{atPAKE}'}$ with advantage better than $(q_{\mathsf{T}} \cdot q_{\mathsf{eval}}^2 + q_{\mathsf{test}} + q_{\mathsf{T}}^* \cdot q_{\mathsf{eval}})/2^\tau$ where q_{T} is the number of target server instances, q_{eval} is the number of tPOPRF evaluations, q_{test} is the number of online and offline password-guessing attacks against $\mathcal{F}_{\mathsf{saPAKE}}$, q_{T}^* is the number of dishonestly initialized target server instances, and security parameter τ is the tPOPRF output length.*

The proof of Theorem 9 is similar to the proof of tOPRF-atPAKE security in Appendix C; to avoid redundancy, we list only the differences between the two.

Proof. For any adversary $\mathcal{A}^*$ we construct simulator $\mathsf{SIM}_{\mathsf{tPOPRF-atPAKE}}$ in Figures 32, 33, 34, and 35. Without loss of generality we assume that $\mathcal{A}^*$ is a "dummy" adversary that merely passes messages to and from the environment $\mathcal{Z}$.

We now show that, for any efficient (i.e. PPT) $\mathcal{Z}$, the distinguishing advantage of $\mathcal{Z}$ between the real and simulated worlds is negligible. The argument uses only a single game change from the real world $\mathbf{G}_0$ to the simulated world $\mathbf{G}_1$. By $\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_0,\mathbf{G}_1}$ we denote distinguisher $\mathcal{Z}$'s distinguishing advantage between world $\mathbf{G}_0$ and world $\mathbf{G}_1$. Specifically, $\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_0,\mathbf{G}_1} = |\Pr_{\mathcal{Z}\leftrightarrow\mathbf{G}_0}[\mathcal{Z} \text{ outputs } 1] - \Pr_{\mathcal{Z}\leftrightarrow\mathbf{G}_1}[\mathcal{Z} \text{ outputs } 1]|$.

Game $\mathbf{G}_0$: The real world. The distinguisher $\mathcal{Z}$ interacts with $\Pi_{\mathsf{tPOPRF-atPAKE}}$ (Figure 31) in the role of the honest parties and in the role of the adversary.

Game $\mathbf{G}_1$: The simulated world. By inspection, $\mathsf{SIM}_{\mathsf{tPOPRF-atPAKE}}$ in interaction with $\mathcal{F}_{\mathsf{atPAKE}'}$ behaves identically to the real world protocol, except in the case of a rare PRF collision event. Therefore, the probability that $\mathcal{Z}$ can distinguish between the real and simulated worlds is upper-bounded by the probability that such an event occurs.

These collision events can be classified into 3 types, which are directly analogous to the PRF collision types described in the $\Pi_{\mathsf{tOPRF-atPAKE}}$ proof of security (Appendix C). The only difference is that, in this proof, rw_j values are a function of $(sid, sid_{\mathsf{A}}, \mathsf{pw})$ triples rather than $(sid_{\mathsf{A}}, \mathsf{pw})$ pairs. In honest evaluations, $sid = \mathbf{S}_{sid_{\mathsf{A}}}$, but in cases of adversarial action this is not guaranteed. Nonetheless, the 3 PRF collision types have precisely the same probability bounds as in the other proof; summing up these probabilities yields an overall bound on $\mathcal{Z}$'s distinguishing advantage between the real and simulated worlds.

$$\mathbf{Dist}_{\mathcal{Z}}^{\mathbf{G}_0,\mathbf{G}_1} \leq \frac{q_{\mathsf{T}} \cdot q_{\mathsf{eval}}^2 + q_{\mathsf{test}} + q_{\mathsf{T}}^* \cdot q_{\mathsf{eval}}}{2^\tau}$$

τ is the security parameter. If $\mathcal{Z}$ is efficient, then q_{T}, q_{eval}, q_{test}, and q_{T}^* are all polynomial functions of the security parameter. Thus, the distinguishing advantage of any efficient $\mathcal{Z}$ is negligible.

□

Notation

Initially $\mathsf{tx}[sid, sid_\mathsf{A}, i, ssid_\mathsf{A}] := 0$ for all sid, sid_A, i, and $ssid_\mathsf{A}$, and $\mathsf{cflag}[sid_\mathsf{T}, j] :=$ UNCOMPROMISED for all sid_T and j. Also, $F_{sid}(x_\mathsf{priv}, x_\mathsf{pub})$ is undefined for all sid, x_priv, and x_pub. When first referenced, the functionality assigns $F_{sid}(x_\mathsf{priv}, x_\mathsf{pub}) \leftarrow_\$ \{0,1\}^\tau$.

Initialization Phase (I)

1. On $(\mathsf{atpake.userinit}, \mathsf{U}, sid_\mathsf{A}, sid_\mathsf{T})$ from $\mathcal{F}_\mathsf{atPAKE}$ for honest U:
   - save $\mathsf{rec} = (\mathsf{userinit}, \mathsf{U}, sid_\mathsf{A}, sid_\mathsf{T})$ marked PENDING
   - for every $i \in [n]$, send $(\mathsf{channel.send}, (sid_\mathsf{A}||i||1), \mathsf{U}, \mathbf{S}_{sid_\mathsf{A}}[i], |\text{INIT}|)$ to $\mathcal{A}^*$
   - send $(\mathsf{tpoprf.eval}, \mathbf{S}_{sid_\mathsf{A}}, (sid_\mathsf{A}||\text{INIT}), \mathsf{U}, sid_\mathsf{A})$ to $\mathcal{A}^*$
   - await $(\mathsf{tpoprf.rcvcomplete}, \mathbf{S}_{sid_\mathsf{A}}, (sid_\mathsf{A}||\text{INIT}), sid^*, ssid^*_\mathsf{A}, \mathbf{C}, \mathtt{tr}_\mathsf{U})$ from $\mathcal{A}^*$ s.t. $\forall_{i \in \mathbf{C}}\ \mathsf{tx}[sid^*, sid_\mathsf{A}, i, ssid^*_\mathsf{A}] > 0$ and there does not exist a set of records $\{(\mathsf{sndrtrans}, sid', i, \mathtt{tr}_\mathsf{U}[i])\}_{i \in \mathbf{C}}$ s.t. $sid' \neq sid^*$
   - $\forall_{i \in \mathbf{C}}$ set $\mathsf{tx}[sid^*, sid_\mathsf{A}, i, ssid^*_\mathsf{A}]{-}{-}$
   - if not $\forall_{i \in [n]}$ there exists record $(\mathsf{inittrans}, sid_\mathsf{A}, i, \mathtt{tr}_\mathsf{U}[i])$ marked COMPLETED, then abort
   - for every $j \in \{1, \ldots, |\mathbf{S}_{sid_\mathsf{T}}|\}$, send $(\mathsf{channel.send}, (sid_\mathsf{A}||sid_\mathsf{T}||j), \mathsf{U}, \mathbf{S}_{sid_\mathsf{T}}[j], \tau)$ to $\mathcal{A}^*$
   - mark rec COMPLETED
2. On $(\mathsf{channel.send}, (sid_\mathsf{A}||i||1), \mathbf{S}_{sid_\mathsf{A}}[i], \text{INIT})$ from $\mathcal{A}^*$ on behalf of some $\mathsf{U} \in \mathbf{Corr}$:
   - save $(\mathsf{userinit}, \mathsf{U}, sid_\mathsf{A}, \perp)$
3. On $(\mathsf{tpoprf.init}, sid, (\mathsf{pw}^*_1, sid_{\mathsf{A},1}), \ldots, (\mathsf{pw}^*_k, sid_{\mathsf{A},k}))$ from $\mathcal{A}^*$ on behalf of some $\mathsf{U} \in \mathbf{Corr}$:
   - send $(\mathsf{tpoprf.init}, sid, \mathsf{U})$ to $\mathcal{A}^*$
   - send $(\mathsf{toprf.initeval}, sid, F_{sid}(\mathsf{pw}^*_1, sid_{\mathsf{A},1}), \ldots, F_{sid}(\mathsf{pw}^*_k, sid_{\mathsf{A},k}))$ to U
   - if $\mathsf{U} = \mathbf{S}_{sid}[1]$, then save $(\mathsf{tpoprf}, \text{INIT}, sid, \mathsf{U})$ marked ADVERSARIAL
4. On $(\mathsf{atpake.auxinit}, sid_\mathsf{A}, i, \mathsf{U}')$ from $\mathcal{F}_\mathsf{atPAKE}$ for honest $\mathbf{S}_{sid_\mathsf{A}}[i]$:
   - await $(\mathsf{channel.deliver}, (sid_\mathsf{A}||i||1), \mathsf{U}', \mathbf{S}_{sid_\mathsf{A}})$ from $\mathcal{A}^*$
   - if there is no record $(\mathsf{userinit}, \mathsf{U}', sid_\mathsf{A}, [sid_\mathsf{T}])$, then abort
   - if there is no record $(\mathsf{tpoprf}, i, \mathbf{S}_{sid_\mathsf{A}}, \mathbf{S}_{sid_\mathsf{A}}[1])$, then create it marked PENDING, and, if $i = 1$, send $(\mathsf{tpoprf.init}, \mathbf{S}_{sid_\mathsf{A}}, \mathbf{S}_{sid_\mathsf{A}}[1])$ to $\mathcal{A}^*$ and save $(\mathsf{tpoprf}, \text{INIT}, \mathbf{S}_{sid_\mathsf{A}}, \mathbf{S}_{sid_\mathsf{A}}[1])$
   - wait until $(\mathsf{tpoprf}, i, \mathbf{S}_{sid_\mathsf{A}}, \mathbf{S}_{sid_\mathsf{A}}[1])$ is COMPLETED (if it isn't already)
   - send $(\mathsf{atpake.finishauxiliaryinit}, sid_\mathsf{A}, i)$ to $\mathcal{F}_\mathsf{atPAKE}$
   - if $i \leq t + 1$, do the following:
     - set $\mathsf{tx}[\mathbf{S}_{sid_\mathsf{A}}, sid_\mathsf{A}, i, \text{INIT}]{+}{+}$
     - send $(\mathsf{tpoprf.sndrcomplete}, \mathbf{S}_{sid_\mathsf{A}}, i, \text{INIT}, sid_\mathsf{A})$ to $\mathcal{A}^*$
     - await $(\mathsf{tpoprf.sndrtrans}, \mathbf{S}_{sid_\mathsf{A}}, i, \text{INIT}, \mathtt{tr}_i)$ from $\mathcal{A}^*$
     - send $(\mathsf{channel.send}, (sid_\mathsf{A}||i||2), \mathbf{S}_{sid_\mathsf{A}}[i], \mathsf{U}', |\mathtt{tr}_i|)$ to $\mathcal{A}^*$
     - save $(\mathsf{inittrans}, sid_\mathsf{A}, i, \mathtt{tr}_i)$ marked PENDING
   - otherwise, send $(\mathsf{channel.send}, (sid_\mathsf{A}||i||2), \mathbf{S}_{sid_\mathsf{A}}[i], \mathsf{U}, |\perp|)$ to $\mathcal{A}^*$ and save $(\mathsf{inittrans}, sid_\mathsf{A}, i, \perp)$ marked PENDING

Fig. 32. Simulator $\mathsf{SIM}_{\mathsf{tPOPRF-atPAKE}}$ for protocol $\Pi_{\mathsf{tPOPRF-atPAKE}}$, part 1: Notation and Initialization Phase (I)

Initialization Phase (II)

5. On $(\mathsf{tpoprf.sinit}, sid, i, \mathsf{U}')$ from $\mathcal{A}^*$ where $\mathbf{S}_{sid}[i] \in \mathbf{Corr}$:
   - if $\mathbf{S}_{sid}[i] \neq \mathcal{A}^*$, send $(\mathsf{atpake.auxinit}, sid, i, \mathsf{U}')$ to $\mathcal{F}_{\mathsf{atPAKE}}$
   - save $(\mathsf{tpoprf}, i, sid_{\mathsf{A}}, \mathsf{U}')$ marked PENDING
6. On $(\mathsf{tpoprf.fininit}, sid, i)$ from $\mathcal{A}^*$:
   - retrieve $(\mathsf{tpoprf}, i, sid, [\mathsf{P}_0])$ marked PENDING and $\mathsf{rec} = (\mathsf{tpoprf}, \text{INIT}, sid, \mathsf{P}_0)$ (abort if either not found) and mark the former COMPLETED
   - save $(\mathsf{auxiliaryfile}, sid, i)$ and mark it ADVERSARIAL if rec is ADVERSARIAL
7. On $(\mathsf{channel.send}, (sid_{\mathsf{A}}||i||2), \mathsf{U}, \mathtt{tr}_i)$ from $\mathcal{A}^*$ on behalf of $\mathbf{S}_{sid_{\mathsf{A}}}[i] \in \mathbf{Corr}$:
   - save $(\mathsf{inittrans}, sid_{\mathsf{A}}, i, \mathtt{tr}_i)$ marked PENDING
8. On $(\mathsf{channel.deliver}, (sid_{\mathsf{A}}||i||2), \mathbf{S}_{sid_{\mathsf{A}}}[i], \mathsf{U})$ from $\mathcal{A}^*$:
   - if there exists record $(\mathsf{userinit}, \mathsf{U}, sid_{\mathsf{A}}, [sid_{\mathsf{T}}])$, retrieve record $(\mathsf{inittrans}, sid_{\mathsf{A}}, i, \mathtt{tr}_i)$ marked PENDING and mark it COMPLETED
9. On $(\mathsf{atpake.targetinit}, sid_{\mathsf{A}}, sid_{\mathsf{T}}, j, \mathsf{U}')$ from $\mathcal{F}_{\mathsf{atPAKE}}$:
   - save $(\mathsf{targetinit}, sid_{\mathsf{A}}, sid_{\mathsf{T}}, j, \mathsf{U}')$ marked PENDING
10. On $(\mathsf{channel.deliver}, (sid_{\mathsf{A}}||sid_{\mathsf{T}}||j), \mathsf{U}, \mathbf{S}_{sid_{\mathsf{T}}}[j])$ from $\mathcal{A}^*$:
    - retrieve $\mathsf{trec} = (\mathsf{targetinit}, sid_{\mathsf{A}}, sid_{\mathsf{T}}, j, \mathsf{U})$ marked PENDING (abort if not found)
    - if $(\mathsf{channel.send}, (sid_{\mathsf{A}}||sid_{\mathsf{T}}||j), \mathbf{S}_{sid_{\mathsf{T}}}[j], [\mathsf{rw}_j^*])$ was previously sent by $\mathcal{A}^*$ on behalf of $\mathsf{U} \in \mathbf{Corr}$:
      - if $\exists\ (sid^*, sid_{\mathsf{A}}^*, \mathsf{pw}^*)$ s.t. $\mathsf{rw}_j^* = \mathsf{KDF}(F_{sid^*}(\mathsf{pw}^*, sid_{\mathsf{A}}^*), \mathbf{S}_{sid_{\mathsf{T}}}[i])$ then send $(\mathsf{atpake.finishtargetinit}, sid_{\mathsf{A}}, sid_{\mathsf{T}}, j, sid'_{\mathsf{A}}, \mathsf{pw}^*)$ to $\mathcal{F}_{\mathsf{atPAKE}}$ where $sid'_{\mathsf{A}} := (sid^*||sid_{\mathsf{A}}^*)$ if $sid^* \neq \mathbf{S}_{sid_{\mathsf{A}}^*}$ and $sid'_{\mathsf{A}} := sid_{\mathsf{A}}^*$ otherwise, and save $(\mathsf{targetfile}, sid'_{\mathsf{A}}, sid_{\mathsf{T}}, j, \mathsf{pw}^*, \mathsf{rw}_j^*)$
      - otherwise, send $(\mathsf{atpake.finishtargetinit}, sid_{\mathsf{A}}, sid_{\mathsf{T}}, j, \perp, \perp)$ to $\mathcal{F}_{\mathsf{atPAKE}}$ and save $(\mathsf{targetfile}, \perp, sid_{\mathsf{T}}, j, \perp, \mathsf{rw}_j^*)$
    - otherwise, retrieve record $(\mathsf{userinit}, \mathsf{U}, sid_{\mathsf{A}}, sid_{\mathsf{T}})$ marked COMPLETED (abort if not found) and do the following:
      - recall $(\mathsf{tpoprf.rcvcomplete}, \mathbf{S}_{sid_{\mathsf{A}}}, (sid_{\mathsf{A}}||\text{INIT}), sid^*, ssid_{\mathsf{A}}^*, \mathbf{C}, \mathtt{tr}_{\mathsf{U}})$ previously sent by $\mathcal{A}^*$ that caused the $\mathsf{userinit}$ record to become COMPLETED
      - if $sid^* = \mathbf{S}_{sid_{\mathsf{A}}}$, define $sid_{\mathsf{A}}^* := sid_{\mathsf{A}}$; else define $sid_{\mathsf{A}}^* := (sid^*||sid_{\mathsf{A}})$
      - send $(\mathsf{atpake.finishtargetinit}, sid_{\mathsf{A}}, sid_{\mathsf{T}}, j, sid^*, \perp)$ to $\mathcal{F}_{\mathsf{atPAKE}}$, and save $(\mathsf{targetfile}, sid^*, sid_{\mathsf{T}}, j, \perp, \perp)$
    - mark trec COMPLETED

Fig. 33. Simulator $\mathsf{SIM}_{\mathsf{tPOPRF-atPAKE}}$ for protocol $\Pi_{\mathsf{tPOPRF-atPAKE}}$, part 2: Initialization Phase (II)

Party Corruption

11. On $(\mathsf{tpoprf.corrupt}, \mathsf{P})$ from $\mathcal{A}^*$:
    - set $\mathbf{Corr} := \mathbf{Corr} \cup \{\mathsf{P}\}$
    - send $(\mathsf{atpake.corrupt}, \mathsf{P})$ to $\mathcal{F}_{\mathsf{atPAKE}}$
12. On $(\mathsf{sapake.stealpwdfile}, (sid_{\mathsf{T}}||j))$ from $\mathcal{A}^*$:
    - retrieve $(\mathsf{targetfile}, [sid_{\mathsf{A}}], sid_{\mathsf{T}}, j, [\mathsf{pw}, \mathsf{rw}_j])$ and mark it COMPR (abort if not found)
    - send $(\mathsf{atpake.stealtargetfile}, sid_{\mathsf{T}}, j)$ to $\mathcal{F}_{\mathsf{atPAKE}}$

Authentication Phase (I)

13. On $(\mathsf{atpake.usersession}, \mathsf{U}', sid_{\mathsf{A}}, sid_{\mathsf{T}}, j, ssid)$ from $\mathcal{F}_{\mathsf{atPAKE}}$:
    - send $(\mathsf{tpoprf.eval}, \mathbf{S}_{sid_{\mathsf{A}}}, ssid, \mathsf{U}', sid_{\mathsf{A}})$ to $\mathcal{A}^*$
    - save $(\mathsf{session}, \mathsf{U}', \mathbf{S}_{sid_{\mathsf{T}}}[j], sid_{\mathsf{A}}, sid_{\mathsf{T}}, j, ssid, \perp, \perp)$ marked PRELIM
14. On $(\mathsf{tpoprf.eval}, sid, ssid, \mathsf{pw}^*, sid_{\mathsf{A}})$ from $\mathcal{A}^*$:
    - send $(\mathsf{tpoprf.eval}, sid, ssid, \mathcal{A}^*, sid_{\mathsf{A}})$ to $\mathcal{A}^*$
    - save $(\mathsf{eval}, sid, ssid, \mathsf{pw}^*, sid_{\mathsf{A}})$ marked FRESH
15. On $(\mathsf{atpake.auxsession}, sid_{\mathsf{A}}, i, ssid_{\mathsf{A}})$ from $\mathcal{F}_{\mathsf{atPAKE}}$ for honest $\mathbf{S}_{sid_{\mathsf{A}}}[i]$:
    - send $(\mathsf{tpoprf.sndrcomplete}, \mathbf{S}_{sid_{\mathsf{A}}}, i, ssid_{\mathsf{A}}, sid_{\mathsf{A}})$ to $\mathcal{A}^*$
    - set $\mathsf{tx}[\mathbf{S}_{sid_{\mathsf{A}}}, sid_{\mathsf{A}}, i, ssid_{\mathsf{A}}]{+}{+}$
16. On $(\mathsf{tpoprf.sndrcomplete}, sid, i, ssid_{\mathsf{A}}, sid_{\mathsf{A}})$ from $\mathcal{A}^*$:
    - retrieve $\mathsf{rec} = (\mathsf{auxiliaryfile}, sid, i)$ (abort if not found)
    - if rec is not ADVERSARIAL, then send $(\mathsf{atpake.auxsession}, sid^*_{\mathsf{A}}, i, ssid_{\mathsf{A}})$ to $\mathcal{F}_{\mathsf{atPAKE}}$ where $sid^*_{\mathsf{A}} = (sid||sid_{\mathsf{A}})$ if $sid \neq \mathbf{S}_{sid_{\mathsf{A}}}$ and $sid^*_{\mathsf{A}} = sid_{\mathsf{A}}$ otherwise (abort if response $(\mathsf{atpake.auxsession}, sid^*_{\mathsf{A}}, i, ssid_{\mathsf{A}})$ is not received)
    - send $(\mathsf{tpoprf.sndrcomplete}, sid, i, ssid_{\mathsf{A}}, sid_{\mathsf{A}})$ to $\mathcal{A}^*$
    - set $\mathsf{tx}[sid, sid_{\mathsf{A}}, i, ssid_{\mathsf{A}}]{+}{+}$
17. On $(\mathsf{tpoprf.rcvcomplete}, \mathbf{S}_{sid_{\mathsf{A}}}, ssid, sid^*, ssid^*_{\mathsf{A}}, \mathbf{C}, \mathtt{tr}_{\mathsf{U}})$ from $\mathcal{A}^*$ s.t. $\exists$ record $\mathsf{urec} = (\mathsf{session}, [\mathsf{U}', \mathbf{S}_{sid_{\mathsf{T}}}[j]], sid_{\mathsf{A}}, [sid_{\mathsf{T}}, j], ssid, \perp, \perp)$ marked PRELIM:
    - abort if $\exists_{i \in \mathbf{C}}\ \mathsf{tx}[sid^*, sid_{\mathsf{A}}, i, ssid^*_{\mathsf{A}}] = 0$, else $\forall_{i \in \mathbf{C}}$ set $\mathsf{tx}[sid^*, sid_{\mathsf{A}}, i, ssid^*_{\mathsf{A}}]{-}{-}$
    - if $sid^* = \mathbf{S}_{sid_{\mathsf{A}}}$, define $sid^*_{\mathsf{A}} := sid_{\mathsf{A}}$; else define $sid^*_{\mathsf{A}} := (sid^*||sid_{\mathsf{A}})$
    - update field sid_{A} in urec to sid^*_{A}, and change urec's mark to FRESH
    - if record $(\mathsf{userinit}, [\mathsf{U}], sid^*, [sid'_{\mathsf{T}}])$ is marked ADVERSARIAL, send $(\mathsf{atpake.auxactive}, \mathsf{U}', ssid)$ to $\mathcal{F}_{\mathsf{atPAKE}}$; otherwise, send $(\mathsf{atpake.auxproceed}, \mathsf{U}', ssid, sid^*_{\mathsf{A}}, ssid^*_{\mathsf{A}}, \mathbf{C})$
    - send $(\mathsf{sapake.usrsession}, (sid_{\mathsf{T}}||j), ssid, \mathsf{U}', \mathbf{S}_{sid_{\mathsf{T}}}[j])$ to $\mathcal{A}^*$
18. On $(\mathsf{tpoprf.rcvcomplete}, sid, ssid, sid^*, ssid^*_{\mathsf{A}}, \mathbf{C}, \mathtt{tr}_{\mathsf{U}})$ from $\mathcal{A}^*$ s.t. $\exists$ record $\mathsf{urec} = (\mathsf{eval}, sid, ssid, [\mathsf{pw}^*, sid_{\mathsf{A}}])$ marked FRESH:
    - abort if $\exists_{i \in \mathbf{C}}\ \mathsf{tx}[sid^*, sid_{\mathsf{A}}, i, ssid^*_{\mathsf{A}}] = 0$, else $\forall_{i \in \mathbf{C}}\ \mathsf{tx}[sid^*, sid_{\mathsf{A}}, i, ssid^*_{\mathsf{A}}]{-}{-}$
    - change urec's mark to COMPLETED
    - send $(\mathsf{tpoprf.eval}, sid, ssid, F_{sid^*}(\mathsf{pw}^*, sid_{\mathsf{A}}))$ to $\mathcal{A}^*$
    - if $sid^* = \mathbf{S}_{sid_{\mathsf{A}}}$, define $sid^*_{\mathsf{A}} := sid_{\mathsf{A}}$; else define $sid^*_{\mathsf{A}} := (sid^*||sid_{\mathsf{A}})$
    - $\forall_{i \in \mathbf{C}}$ send $(\mathsf{atpake.offlinetestpwd}, sid^*_{\mathsf{A}}, i, ssid^*_{\mathsf{A}}, \perp, \perp, \mathsf{pw}^*)$ to $\mathcal{F}_{\mathsf{atPAKE}}$
    - save $(\mathsf{completedeval}, sid^*_{\mathsf{A}}, \mathsf{pw}^*, ssid^*_{\mathsf{A}}, F_{sid^*}(\mathsf{pw}^*, sid_{\mathsf{A}}))$

Fig. 34. Simulator $\mathsf{SIM}_{\mathsf{tPOPRF-atPAKE}}$ for protocol $\Pi_{\mathsf{tPOPRF-atPAKE}}$, part 3: Authentication Phase (I)

Authentication Phase (II)

19. On $(\mathsf{atpake.targetsession}, sid_\mathsf{T}, j, \mathsf{U}', ssid)$ from $\mathcal{F}_{\mathsf{atPAKE}}$:
    - retrieve $(\mathsf{targetfile}, [sid_\mathsf{A}], sid_\mathsf{T}, j, [\mathsf{pw}, \mathsf{rw}_j])$
    - send $(\mathsf{sapake.svrsession}, (sid_\mathsf{T}||j), ssid, \mathsf{U}', \mathbf{S}_{sid_\mathsf{T}}[j])$ to $\mathcal{A}^*$
    - save $(\mathsf{session}, \mathbf{S}_{sid_\mathsf{T}}[j], \mathsf{U}', sid_\mathsf{A}, sid_\mathsf{T}, j, ssid, \mathsf{pw}, \mathsf{rw}_j)$ marked FRESH
20. On $(\mathsf{sapake.interrupt}, (sid_\mathsf{T}||j), ssid, \mathbf{S}_{sid_\mathsf{T}}[j])$ from $\mathcal{A}^*$:
    - send $(\mathsf{atpake.interrupt}, sid_\mathsf{T}, j, ssid)$ to $\mathcal{F}_{\mathsf{atPAKE}}$
21. On $(\mathsf{sapake.testpwd}, (sid_\mathsf{T}||j), ssid, \mathsf{P}, \mathsf{rw}^*_j)$ from $\mathcal{A}^*$:
    - retrieve $\mathsf{rec} = (\mathsf{session}, \mathsf{P}, [\mathsf{P}', sid_\mathsf{A}], sid_\mathsf{T}, j, ssid, [\mathsf{pw}', \mathsf{rw}'_j])$ (abort if not found)
    - if $\mathsf{rw}'_j \neq \perp$ and $\mathsf{rw}^*_j = \mathsf{rw}'_j$, send $(\mathsf{atpake.testpwd}, \mathsf{P}, ssid, \mathsf{pw}')$ to $\mathcal{F}_{\mathsf{atPAKE}}$
    - else if $\mathsf{rw}'_j \neq \perp$ but $\mathsf{rw}^*_j \neq \mathsf{rw}'_j$, send $(\mathsf{atpake.testpwd}, \mathsf{P}, ssid, (\mathsf{pw}'||0))$
    - else if $\exists\ \mathsf{pw}^*$ s.t. $\mathsf{rw}^*_j = \mathsf{KDF}(F_{sid}(\mathsf{pw}^*, sid'_\mathsf{A}), \mathbf{S}_{sid_\mathsf{T}}[j])$ where sid_A is parsed as $(sid||sid'_\mathsf{A})$ if possible and $(sid, sid'_\mathsf{A}) := (\mathbf{S}_{sid_\mathsf{A}}, sid_\mathsf{A})$ otherwise, send $(\mathsf{atpake.testpwd}, \mathsf{P}, sid, \mathsf{pw}^*)$
    - else, send $(\mathsf{atpake.testpwd}, \mathsf{P}, ssid, \perp)$
    - in any case, forward the response (if any) to $\mathcal{A}^*$
    - if the response is "correct guess", mark rec COMPR and set $\mathsf{cflag}[sid_\mathsf{T}, j] :=$ COMPR
22. On $(\mathsf{sapake.impersonate}, (sid_\mathsf{T}||j), ssid)$ from $\mathcal{A}^*$:
    - send $(\mathsf{atpake.impersonate}, sid_\mathsf{T}, j, ssid)$ to $\mathcal{F}_{\mathsf{atPAKE}}$
    - forward the response (if any) to $\mathcal{A}^*$
    - if the response is "correct guess", mark rec COMPR
23. On $(\mathsf{sapake.newkey}, (sid_\mathsf{T}||j), ssid, \mathsf{P}, K^*)$ from $\mathcal{A}^*$:
    - retrieve $\mathsf{rec} = (\mathsf{session}, \mathsf{P}, [\mathsf{P}', sid_\mathsf{A}], sid_\mathsf{T}, j, ssid, [\mathsf{pw}, \mathsf{rw}_j])$ not marked PRELIM (abort if not found)
    - if rec is not COMPR and $\mathsf{cflag}[sid_\mathsf{T}, j] =$ UNCOMPROMISED, set $K^* \leftarrow_\$ \{0,1\}^\tau$
    - send $(\mathsf{atpake.newkey}, \mathsf{P}, ssid, K^*)$ to $\mathcal{F}_{\mathsf{atPAKE}}$
    - mark rec COMPLETED
24. On $(\mathsf{sapake.testabort}, (sid_\mathsf{T}||j), ssid, \mathsf{U}')$ from $\mathcal{A}^*$:
    - send $(\mathsf{atpake.testabort}, \mathsf{U}', sid_\mathsf{T}, j, ssid)$ to $\mathcal{F}_{\mathsf{atPAKE}}$
    - forward the response (if any) to $\mathcal{A}^*$

Offline Password Tests

25. On $(\mathsf{sapake.offlinetestpwd}, (sid_\mathsf{T}||j), \mathsf{rw}^*_j)$ from $\mathcal{A}^*$:
    - retrieve $(\mathsf{targetfile}, [sid_\mathsf{A}], sid_\mathsf{T}, j, [\mathsf{pw}, \mathsf{rw}_j])$
    - if $\mathsf{rw}^*_j = \mathsf{rw}_j \neq \perp$, send "correct guess" to $\mathcal{A}^*$
    - else if $\exists$ record $(\mathsf{completedeval}, sid_\mathsf{A}, [\mathsf{pw}^*, ssid_\mathsf{A}], \mathsf{rw}^*)$ s.t. $\mathsf{rw}^*_j = \mathsf{KDF}(\mathsf{rw}^*, \mathbf{S}_{sid_\mathsf{T}}[j])$, send $(\mathsf{atpake.offlinetestpwd}, sid_\mathsf{A}, \perp, ssid_\mathsf{A}, sid_\mathsf{T}, j, \mathsf{pw}^*)$ to $\mathcal{F}_{\mathsf{atPAKE}}$ and forward the response to $\mathcal{A}^*$
    - else, send "wrong guess" to $\mathcal{A}^*$
    - in any case, if the response is "correct guess", set $\mathsf{cflag}[sid_\mathsf{T}, j] :=$ COMPR

Fig. 35. Simulator $\mathsf{SIM}_{\mathsf{tPOPRF-atPAKE}}$ for protocol $\Pi_{\mathsf{tPOPRF-atPAKE}}$, part 4: Authentication Phase (II) and Offline Password Tests