Password-Authenticated TLS via OPAQUE and Post-Handshake Authentication

Julia Hesse[1⋆], Stanislaw Jarecki[2], Hugo Krawczyk[3], and Christopher Wood[4]

[1] IBM Research Europe – Zurich
jhs@zurich.ibm.com
[2] UC Irvine
stanislawjarecki@gmail.com
[3] Algorand Foundation
hugokraw@gmail.com
[4] Cloudflare
caw@heapingbits.net

Abstract. OPAQUE is an Asymmetric Password-Authenticated Key Exchange (aPAKE) protocol being standardized by the IETF (Internet Engineering Task Force) as a more secure alternative to the traditional "password-over-TLS" mechanism prevalent in current practice. OPAQUE defends against a variety of vulnerabilities of password-over-TLS by dispensing with reliance on PKI and TLS security, and ensuring that the password is never visible to servers or anyone other than the client machine where the password is entered. In order to facilitate the use of OPAQUE in practice, integration of OPAQUE with TLS is needed. The main proposal for standardizing such integration uses the Exported Authenticators (TLS-EA) mechanism of TLS 1.3 that supports post-handshake authentication and allows for a smooth composition with OPAQUE. We refer to this composition as TLS-OPAQUE and present a detailed security analysis for it in the Universal Composability (UC) framework.

Our treatment is general and includes the formalization of components that are needed in the analysis of TLS-OPAQUE but are of wider applicability as they are used in many protocols in practice. Specifically, we provide formalizations in the UC model of the notions of post-handshake authentication and channel binding. The latter, in particular, has been hard to implement securely in practice, resulting in multiple protocol failures, including major attacks against prior versions of TLS. Ours is the first treatment of these notions in a computational model with composability guarantees.

We complement the theoretical work with a detailed discussion of practical considerations for the use and deployment of TLS-OPAQUE in real-world settings and applications.

Keywords: Transport Layer Security, Passwords, Authentication

⋆ This work was supported by the Swiss National Science Foundation (SNSF) under the AMBIZIONE grant "Cryptographic Protocols for Human Authentication and the IoT"

1 Introduction

For a multitude of reasons, passwords remain a ubiquitous type of authenticator. Despite the existence of tools for improving passwords (password managers) and password-less authentication protocols (e.g., WebAuthn), password-based authentication remains commonplace. Legacy software and lack of support for modern alternatives, integration issues for better tooling to improve password quality , and usability problems in adopting any new form of authenticator have all contributed in one way or another to the prolonged usage of passwords for authentication onn the Internet (and beyond).

As a result, much of the security infrastructure depends to a large extent on passwords. And, yet, the prime mechanism of client-server password authentication in practice has not changed in the last decades and remains the traditional password-over-TLS (more generally, the transport of passwords over channels protected by public key encryption). Weaknesses of this mechanism include, though are not limited to: visibility of plaintext passwords to the application server and to other decrypting intermediaries, accidental storage of passwords in the clear (as several high-profile incidents have shown [1,2]), and ease of password leakage in the event of phishing attacks.

Recently, the IETF (Internet Engineering Task Force) has initiated a process of standardizing a much stronger mechanism, the so-called *Asymmetric Password-Authenticated Key Exchange (aPAKE)* that does not rely on PKI (except, optionally, at user registration time) and ensures that user passwords are never visible outside the client machine. Essentially, aPAKE protocols are as secure as possible, restricting attacks to unavoidable password guesses and offline attacks upon server compromise. The specific protocol chosen for instantiation of the aPAKE standard is OPAQUE [18,9]. In addition to enjoying the aPAKE security (including an enhancement in the form of security against pre-computation attacks), OPAQUE offers the flexibility of working with any authenticated key exchange mechanism. Hence, it is a natural candidate for integration with existing protocols such as TLS 1.3, IKEv2, etc.

Clearly, integration with TLS is desirable for improving the security of password authentication in TLS, but also because while OPAQUE provides authentication and key exchange, it does not offer the secure channels required to protect data; TLS provides such functionality via its record layer. Additionally, integration with TLS allows for protection of user account information during a login protocol.

A natural approach to such integration is to use the *post-handshake authentication (PHA)* mechanism of TLS 1.3[5] [23] that allows clients to authenticate after the TLS handshake (the key establishment component of TLS) has completed, and within the ensuing record-layer session (where data is exchanged under the protection of the keys established by the handshake). For example, a server can serve public webpages to an unauthenticated client but may require client authentication once the client requests access to restricted pages, thus

[5] Except if said otherwise, we use 'TLS' to refer to TLS 1.3.

triggering post-handshake authentication by the client. More general support for PHA is provided in a TLS 1.3 extension standard called *Exported Authenticators (TLS-EA)* [26] (we often shorten TLS-EA to EA). EA extends the post-handshake client authentication component of TLS 1.3 and can support multiple authentications within the same TLS session for both clients and servers. As such, EA is a natural tool for integrating OPAQUE into TLS 1.3 as a way to enable strong password authentication within TLS connections. While EA natively supports certificate-based authentication, its fields can easily be repurposed for transporting OPAQUE's signature-based authentication. This integration of OPAQUE and TLS-EA, referred to here as TLS-OPAQUE, has been proposed for standardization in the TLS Working Group of the IETF [27].

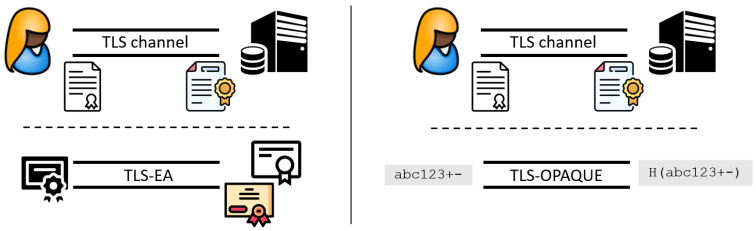

Fig. 1: (Post-)Authentication options for TLS channels. **Left:** The Exported Authenticators TLS extention (TLS-EA) allows both channel endpoints to subsequently add more public-key identities to a TLS channel. **Right:** TLS-OPAQUE allows to subsequently add (asymmetric) password identities to a TLS channel.

In this work we investigate the security of the above schemes: TLS-EA as a general post-handshake mechanism and TLS-OPAQUE for password-authenticated TLS. However, our treatment is more general and independent of any particular protocol instantiation. We formalize the notion of post-handshake authentication in the Universal Composability (UC) setting [11] with two authentication flavors: via public-key certificates as the EA protocol [26] specifies and via passwords as TLS-OPAQUE requires.

While this formalization of PHA serves the analysis of EA and TLS-OPAQUE, post-handshake authentication is a more general notion implemented in practice as extension to multiple protocols, including IPsec, SSH as well as previous versions of TLS. In general terms, the PHA main functionality is to enable multiple authentications (possibly using different credentials and identities) of a previously established channel between two endpoints; *it guarantees that in each of these authentications, the authenticating parties are the same as those that established the channel in the first place.*

Thus, a crucial ingredient in the implementation of any PHA protocol is a mechanism for *binding* the PHA authentications to the original channel. A common design, that we follow in our PHA instantiations, is to define a *channel*

binding value generated at the time of the original channel establishment and passed to PHA for inclusion in all subsequent authentications. This channel binder can take the form of a handshake transcript digest, a cryptographic key, or a combination of both. While the notion itself is simple, its implementation in the real world has been remarkably challenging and has led to serious security failures against multiple protocols, including major attacks against previous versions of TLS such as the notorious renegotiation [22,24,25] and triple-handshake attacks [5]. See [6] for an account of attacks on multiple protocols based on PHA failures due to wrong channel binding designs. It is a main goal and motivation of our work to set an analytical framework and proofs to prevent this type of failures in new designs such as those presented here.

To capture the channel binding requirements, we extend the traditional formalism of secure channel functionalities [12] with a *channel binder* element that is output from the channel generation module (e.g., a key exchange) and used by parties engaging in a PHA as a way to bind their post-handshake authentication to the original channel establishment. Informally, we set two requirements on the channel binder: *being unique among all channels established by an honest party and being pseudorandom.* The latter property enables the use of the binder as a cryptographic key in the process of post-handshake authentication. The uniqueness element is crucial for defeating what is known as channel synchronization attacks [3,6], the source of many of the serious attacks against PHA mechanisms in practice. We formally prove in Theorem 1 in Section 4 that TLS 1.3 with its Exporter Main Secret (EMS) implements a secure channel with such binder qualities.

We frame the security of post-handshake authentication via a UC functionality that enforces that only valid credentials presented by the *original endpoints of the channel* (technically, those that know the binder's cryptographic key) are accepted. Our PHA formalism comes in two flavors: one that supports public keys as the post-handshake authentication means and one that supports password-based authentication. The first flavor captures the essence of the security requirements of TLS-EA, namely, the ability to support any number of PK-based authentications[6] by the creators of a TLS channel, and only by those. Therefore formally proving the security of the TLS-EA protocol from [26] reduces to showing that the protocol realizes the PK-based PHA functionality. This is shown in Theorem 2 in Section 5. In particular, the proof of this theorem validates that the channel binder defined by TLS 1.3 (called EMS, for Exporter Master Secret) has the required properties for the purpose of implementing a secure post-handshake authentication mechanism.

We now consider the TLS-OPAQUE protocol [19,27] that uses the TLS-EA mechanism to transport the OPAQUE messages for providing *password-based* post-handshake authentication to the TLS channel. To prove security of this protocol, we show it realizes our password-based PHA functionality. The latter functionality essentially ensures that any mechanism that realizes the functionality provides authentication guarantees similar to those of an aPAKE. Namely,

[6] In the particular case of TLS-EA, it is signature-based authentication.

the key established upon channel creation (even if anonymous at the time) is authenticated by the client and server; the only way to subvert the protocol is by an online password guessing attack or an offline dictionary attack if the server is compromised. Furthermore, not only does the password-based PHA functionality ensure the correct authentication by the endpoints of the original channel but it also guarantees that no other than these endpoints will succeed in such authentication. By proving that TLS-OPAQUE realizes the password-based PHA functionality (Theorem 3 in Section 6) we get that TLS-OPAQUE enjoys all these aPAKE-like security properties.

On a technical level, our analysis of TLS-OPAQUE builds on the proven guarantees of EA detailed above. In a nutshell, TLS-OPAQUE strips the key exchange part from OPAQUE, and uses only OPAQUE's password authentication mechanism to authenticate the already established TLS key material. This authentication is signature-based and can be outsourced to EA. We detail in Section 2 how exactly TLS-OPAQUE is combined from both EA and (parts of) OPAQUE. A main goal of our analysis is to tame the complexity of TLS-OPAQUE by modularizing the security proof: we first prove the security of EA, and then analyze the security of TLS-OPAQUE assuming that EA is already secure. We refer the reader to the technical roadmap below for a summary of all formal results in the paper, and how they combine with each other.

Altogether, our work delivers the first formal analysis of TLS-EA in the UC framework, and of TLS-OPAQUE overall. Our modular approach yields formal models for widely-used concepts such as channel binders as well as public-key and password-based post-handshake authentication. Our models deepen the understanding of these concepts, and we expect them to be useful for real-world protocol analysis beyond our work.

Finally, we would like to highlight a fundamental element in our treatment: We do not assume the original channel to be authenticated upon creation, only that no one other than the endpoints of the channel can transmit over the channel (as enforced by the encryption and authentication keys created within the channel, e.g., via a plain Diffie-Hellman exchange). Therefore, the security of TLS-OPAQUE depends on the Diffie-Hellman key exchange of TLS 1.3 but not on the server and/or client authentication of this exchange. Thus, TLS-OPAQUE is secure even if the original channel was anonymous. On the other hand, if this channel was originally authenticated, say by the server, that authentication property is additional to the password-based authentication provided by TLS-OPAQUE.

DEPLOYMENT CONSIDERATIONS. TLS-OPAQUE provides real improvements for password-based authentication systems in a variety of environments. Mobile applications, for example, can use TLS-OPAQUE for secure password authentication without any risk of disclosing the password to the server, and without any noticeable change in user experience. Use cases where TLS-OPAQUE is used without fallback to password-over-TLS also mitigate common phishing vectors: even if an attacker can intercept the underlying TLS connection, clients never reveal the plaintext password to the attacker. TLS-OPAQUE also complements

modern authentication technologies such as password managers and multi-factor authentication protocols such as WebAuthn [16].

Using TLS-OPAQUE is not without tradeoffs, however, as TLS-OPAQUE requires changes to applications *and* the underlying TLS implementations. However, such changes are not insurmountable in practice. Additionally, in environments where fallback to password-over-TLS authentication must be supported for backwards compatibility purposes, such as the web, concerns such as phishing remain. Client-side user interface changes may help mitigate such risks, though additional user studies are required to demonstrate feasibility.

TECHNICAL ROADMAP. The analysis of real-world protocols in abstract complexity-theoretic formalisms like the UC framework typically requires simplifications that ignore many technical aspects of the full specifications. Yet, such analysis serves to validate the core cryptographic design at the basis of the protocols. To be concrete, in Section 2 (Figures 3 and 5), we present the core cryptographic elements extracted from IETF RFCs and Internet Drafts [23,26,27] that we analyze and that we use as the basis for abstract representation of these protocols in subsequent sections.

Our formal treatment includes the following elements. In Section 4 we formalize secure channels exporting pseudorandom and unique channel binders in the UC framework (functionality $\mathcal{F}_{\mathsf{cbSC}}$ in Figure 6), and prove in Theorem 1 that the TLS handshake protocol implements such functionality. We then formalize in Section 5 secure channels with post-handshake public-key authentication (functionality $\mathcal{F}_{\mathsf{PHA}}$ in Figure 8), and present a modular version of TLS-EA (Π_{EA} in Figure 9) that uses secure channels with binders (i.e., $\mathcal{F}_{\mathsf{cbSC}}$) as an abstract building block. Theorem 2 proves that this modular version of TLS-EA implements $\mathcal{F}_{\mathsf{PHA}}$. Invoking the UC composition theorem on Theorems 1 and 2 yields our first main result, namely that "real" EA, which corresponds to Π_{EA} with calls to the handshake part of TLS 1.3 instead of $\mathcal{F}_{\mathsf{cbSC}}$, securely implements $\mathcal{F}_{\mathsf{PHA}}$.

We then turn to analyze TLS-OPAQUE. First, we formalize secure channels with post-handshake password authentication (functionality $\mathcal{F}_{\mathsf{pwPHA}}$ in Figure 10), and present a modular version of TLS-OPAQUE ($\Pi_{\mathsf{TLS-OPAQUE}}$ in Figure 12) that uses secure channels with post-handshake public-key authentication (i.e., $\mathcal{F}_{\mathsf{PHA}}$) as an abstract building block. Theorem 3 proves that this modular version of TLS-OPAQUE implements $\mathcal{F}_{\mathsf{pwPHA}}$. Invoking again the UC composition theorem on Theorems 2 and 3 yields our second main result, namely that "real" TLS-OPAQUE, which corresponds to $\Pi_{\mathsf{TLS-OPAQUE}}$ with calls to TLS-EA (i.e., Π_{EA}) instead of $\mathcal{F}_{\mathsf{PHA}}$, securely implements $\mathcal{F}_{\mathsf{pwPHA}}$.

The full version of this paper [15] provides previously known security notions for signatures and MACs, as well as details on Oblivious Pseudorandom Functions (OPRFs), a detailed walk-through of functionality $\mathcal{F}_{\mathsf{cbSC}}$, considerations for implementing, deploying, operating, and using TLS-OPAQUE in a variety of use case, and full proofs and sketches of all our Theorems.

RELATED WORK. TLS 1.3 is perhaps one of the most carefully analyzed security protocols used on the Internet today. Our work analyzes, in the UC model, the

aspects of TLS 1.3 that are directly relevant to TLS-EA and TLS-OPAQUE, yet it may set a basis for a broader UC analysis of TLS 1.3. Our study of these protocols also fits with the analysis-prior-to-deployment approach that characterized the development of TLS 1.3,

Partial study of post-handshake authentication in a game-based model appears in [21] which focused on post-handshake client authentication as a way of upgrading a unilaterally authenticated key exchange to a mutually authenticated one, but did not consider the server side or multiple authentications. In particular, it did not analyze the security of the TLS-EA mechanism.

Most relevant to the subject of our work is the analysis of channel binding and post-handshake authentication techniques (under the notion of *compound authentication*) presented in [6]. The paper analyzes these techniques in several deployed protocols (but not TLS 1.3), showing a variety of attacks due to shortcomings in the channel binding design. They carry a formal analysis of these mechanisms using the protocol analyzer ProVerif [8]. Extending this work, [17] presents an automated analysis of the Exported Authenticators (TLS-EA) protocol [26] based on a symbolic model of the protocol using the Tamarin Prover. Additional papers relevant to the analysis of channel binding mechanisms in practice (particularly pointing to vulnerabilities) include [3,25,7,13,4]. Finally, we mention [10] who formalize (using game-based definitions) a notion of channel binding but with a different functionality than ours. In their case the binding is between identities and a key in a 3-party key exchange setting. Their mechanisms and formal treatment do not seem to apply to our setting.

2 TLS-OPAQUE Specification

In this section we describe the protocols we study in this work: OPAQUE, TLS 1.3 Handshake, TLS-EA, and TLS-OPAQUE. We start by recalling OPAQUE [18,9] in schematic form in Figure 2 (more details are included in the presentation of TLS-OPAQUE in Fig. 5).

FIGURE 2 (SIMPLIFIED SCHEMATIC OPAQUE PROTOCOL). During registration, the user creates an "envelope" containing a user's private key and a server's public key. The envelope is protected (for secrecy and authentication) by a key computed jointly between user and server using an Oblivious PRF (OPRF) (to which the user inputs its password and the server inputs a secret user-specific OPRF key; neither party learns the other's input). The server stores the envelope as well as the user's public key and the server's own private key. For login, the user receives the envelope from the server and obtains the key to unlock the envelope by running the OPRF with the client using the same password as upon registration. Now, user and server have the keys to run an authenticated key exchange between them (for TLS-OPAQUE, these keys will be signature keys similar to those used in TLS).

Next, we recall the elements from the TLS 1.3 handshake that play a role in this work, and which serve as a basis for TLS-EA and TLS-OPAQUE.

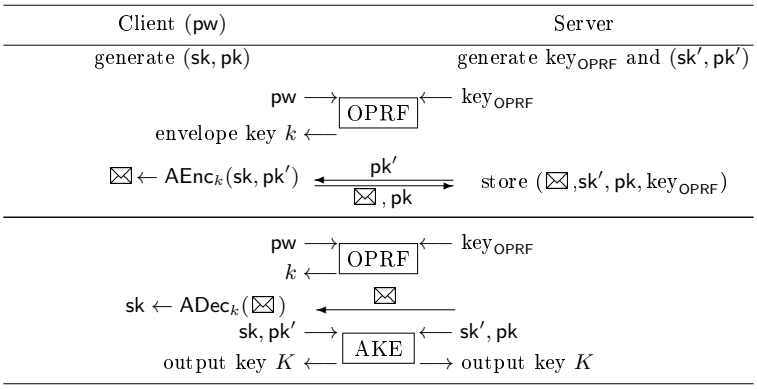

Fig. 2: OPAQUE registration (top) and key exchange (bottom).

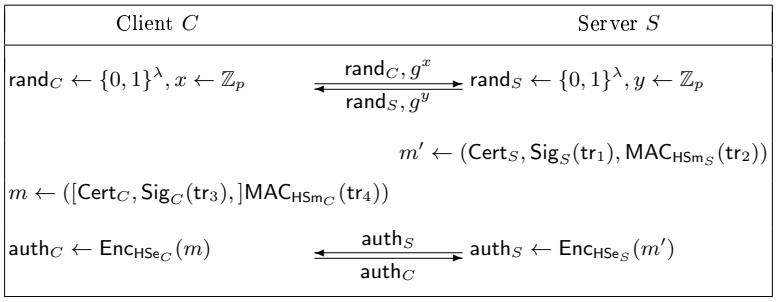

Fig. 3: Schematic representation of TLS 1.3 Handshake (showing subset considered in our analysis).

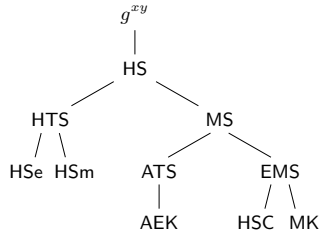

Fig. 4: Key Derivation in TLS 1.3

SIMPLIFIED SCHEMATIC TLS HANDSHAKE (FIGURE 3). The figure shows a schematic representation of a subset of the TLS 1.3 handshake, the key exchange part of TLS. It is intended to show the components that play a role in the protocols studied here. The first two flows show the exchange of nonces ($\mathsf{rand}_C, \mathsf{rand}_S$) and an unauthenticated Diffie-Hellman run between client and server resulting in a key g^{xy} from which a key, HS (for Handshake Secret), is extracted as shown in the key derivation tree in Fig. 4. In the next message, the server authenticates to the client using TLS's sign-and-mac mechanism. The signature (called `CertificateVerify` in TLS) is applied to the handshake transcript and is verified by the client using the server's public key transported in a certificate Cert_S. The MAC part (known as the `Finish` message) uses key HSm_S derived from HS and is applied to the transcript as well. The following message shows client authentication mimicking the server's where the signature part is optional; only the MAC part is mandatory in TLS 1.3. Messages auth_S and auth_C are protected using an authenticated encryption with keys HSe_S and HSe_C also derived from HS. Our analysis in the following sections *proves security of TLS-EA and TLS-OPAQUE even if the handshake DH is unauthenticated,* hence from the point of view of this analysis these authentication messages can be omitted.[7] Each of the transcripts $\mathsf{tr}_1, ..., \mathsf{tr}_4$ cover all previous elements in the handshake until the point of use of the transcript. However, since our analysis does not require the sending of the auth messages, we can set $\mathsf{tr} \leftarrow (\mathsf{rand}_C, g^x, \mathsf{rand}_S, g^y)$.

HANDSHAKE'S KEY DERIVATION (FIGURE 4). The figure shows a key derivation tree used by TLS 1.3. Some of the keys have separate server and client derivations (e.g., $\mathsf{HSC}_S, \mathsf{HSC}_C$) but for simplicity we show them as one key. The root of the tree, g^{xy}, is the product of the handshake's DH exchange. A key HS (for Handshake Secret) is extracted from g^{xy} and from it a tree of keys is derived; we explain their roles. Key HTS (for Handshake Traffic Secret) spawns two keys: HSe for encrypting messages auth_S and auth_C, and HSm used as a MAC key in server and client authentication. Key MS (for Main Secret) has two siblings ATS (Application Traffic Secret) and EMS (Exporter Main Secret). Key AEK, derived from ATS, is used to derive Authenticated Encryption keys for protecting data exchanged in the record layer (that follows the handshake) - it can be thought as the session key in a traditional AKE. EMS spawns HSC and MK which play the critical role (see below) of channel binders in TLS-EA and TLS-OPAQUE. The extraction of HS from g^{xy} and the derivation of MS use $\mathsf{HKDF\text{-}Extract}$ while all other derivations use a PRF implemented via $\mathsf{HKDF\text{-}Expand}$ (because of the particular way that the derivation of MS uses $\mathsf{HKDF\text{-}Extract}$, also this derivation can be seen as produced by a PRF). All derivations use public labels and parts of the transcript to enforce domain separation and (computational) independence of the keys.

[7] Proving our results in the case of an unauthenticated handshake, shows that although TLS handshake is commonly authenticated by the server, TLS-EA's security does not depend on this authentication. On the other hand, when certificate-based server authentication is present during the handshake that precedes a run of TLS-OPAQUE, one gets the benefits of both certificate-based and password-based authentications.

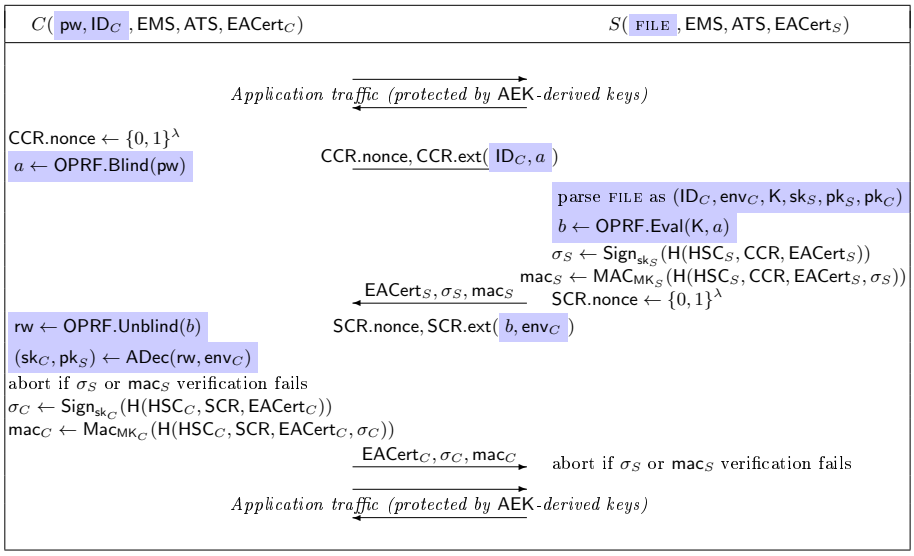

Fig. 5: The TLS-OPAQUE protocol, formed by the subset of the TLS handshake shown in Fig. 3 and the present figure. Omitting blue-colored parts (which correspond to the OPAQUE envelope decryption) one obtains two TLS-EA instances.

FIGURE 5 (TLS-EA AND TLS-OPAQUE PROTOCOLS). We are now ready to explain TLS-EA and TLS-OPAQUE. We show protocol TLS-OPAQUE in Figure 5. However, if one ignores all the blue-colored elements in Fig. 5, one obtains two instances of the TLS-EA protocol [26], the first one authenticating the server to the client, the second one vice versa. This presentation shows how TLS-OPAQUE is built as an extension of TLS-EA, because all the additional cryptography required by OPAQUE is carried using CCR and SCR extension fields of TLS-EA. Note that Fig. 5 shows only the post-handshake authentication parts of TLS-OPAQUE and TLS-EA, while the full protocols are obtained by running the TLS handshake shown in Fig. 3 followed by the post-handshake authentication shown in Fig. 5.

TLS-EA allows an application that established a TLS connection via the handshake to request its peer (client or server) to authenticate at any time after the handshake is completed. For the client to request server authentication, TLS-EA defines a message `ClientCertificateRequest` (abbreviated CCR) that includes a nonce called `certificate_request_context` and which we denote by CCR.nonce. In addition, CCR has an extensions field (we denote it CCR.ext) where an application can carry additional auxiliary information. The analogous message `ServerCertificateRequest` (SCR) (or simply called `CertificateRequest` in the case of the server) is used by the server to request client authentication. The response to such requests is an authentication message by the responder that includes a certificate, a signature and a MAC, implementing the regular sign-and-mac mechanism of TLS with elements σ and mac (i.e., TLS's

`CertificateVerify` and `Finish` messages). The keys for generating and verifying σ correspond to the public keys transported in the certificates (this changes in the case of OPAQUE – see below). The goal of this authentication is not only to validate the identity of the peer but also to tie this peer to the specific connection (or handshake session) on which TLS-EA is executed and to the secure channel (record layer) established by this handshake. A party accepting a set of credentials via TLS-EA is linking these credentials to the party with whom it originally ran the handshake even if that party did not authenticate with these credentials during the handshake, and possibly did not authenticate during the handshake at all.

Linkage of an authentication to the handshake is obtained via a *channel binder* that in TLS-EA (and in our modeling in Section 4) is composed of two elements: a transcript digest (HSC) included under the signature σ and a MAC key (MK) used to produce the value mac. Key MK needs to have properties similar to a session key in a regular key exchange protocol. Informally, it can be seen in the derivation tree of Fig. 4 that MK is a descendant of the original key g^{xy} and is independent (via PRF derivations) from keys used elsewhere by the protocol such as ATS and HTS (exact requirements and proofs are provided in our extensive formal treatment in the following sections). What is less clear is why HSC qualifies as a handshake transcript digest. This property follows from the fact that EMS is computed as an output of a PRF computed on input the handshake's transcript tr, with the PRF instantiated by HKDF [20]. Hence EMS is the product of a chain of hashes computed on tr, and since none of these hash computations is truncated, this ensures that EMS is an output of a collision resistant function computed on tr. The digest property also applies to HSC which is derived from EMS using HKDF, hence as the output of a chain of collision resistant hashes.

The uncolored part of Fig. 5 shows the flows for the case where a client request is followed by a server response and then a server request is followed by a client response. Protocol TLS-OPAQUE adds the colored elements that transport OPAQUE messages inside the extension fields of TLS-EA. This includes OPAQUE's OPRF messages and the user's envelope transmitted from server to client. In this case, signature authentication uses the OPAQUE keys rather than the normal certificate-based keys of TLS. For the client, it uses the private key contained in the envelope and for the server it is the server's signing key stored at the server and whose corresponding public key is included in the envelope. Verification at the server uses the user's public key stored at the server.

Note on the record layer protection of TLS-OPAQUE messages. When the TLS-EA messages are transported over TLS's record layer, all the messages in Figure 5 are protected by the record layer keys (derived from AEK). In our treatment we ignore this protection as TLS-EA does not mandate transmission within the channel[8]. Thus our results establish that TLS-EA and TLS-OPAQUE security does not depend on this protection. On the other hand, the addition of this

[8] From [26]: "The application MAY use the existing TLS connection to transport the authenticator." The use of MAY makes this protected transport optional.

layer of protection does not jeopardize security; this is so since AEK is derived from ATS which is (computationally) independent from any element used in TLS-EA. Indeed, the latter only uses keys derived from EMS which is a sibling of ATS in the derivation from MS, hence independent from AEK (formally, one can simulate the record layer encryption using a random independent ATS). Finally, we note that while not required for TLS-OPAQUE security, running TLS-OPAQUE over a protected record layer can provide privacy to user account information transmitted as part of the protocol.

3 Preliminaries

NOTATION. We denote by $x \leftarrow A$ the assigment of the outcome of A to variable x if A is an algorithm or a function. In case A is a set, $x \leftarrow A$ denotes that x is sampled uniformly at random from A.

OBLIVIOUS PSEUDORANDOM FUNCTIONS. An Oblivious Pseudorandom Function (OPRF) is a 2-party protocol between an evaluator and a server, where the evaluator contributes an input x and the server contributes a PRF key k. The outcome of the protocol is that the evaluator learns $\mathsf{PRF}_k(x)$ but nothing beyond, and the server learns nothing at all. OPRFs have been extensively used in password-based protocols, and they are also the main building block of the OPAQUE protocol [18]. We use a UC formalization of OPRFs by Jarecki et al. [18], modified regarding its output of transcripts which we now describe on a high level. The OPRF functionality of [18] has a (session-wise unique) "transcript prefix" prfx that the adversary contributes. If the view of both parties of this prefix match, the adversary cannot use the honest evaluation session anymore to evaluate the PRF himself (e.g., by modifying the transcript). For the purpose of analyzing TLS-OPAQUE, we introduce two changes to their functionality:

1. We add a "transcript postfix" pstfx, which is also determined by the adversary. prfx and pstfx together constitute the full transcript of the OPRF protocol. In particular, if the view of both parties on prfx and pstfx match, then the OPRF output computed by the evaluator is guaranteed to be correct.
2. We let the evaluator output prfx and require the sender to input prfx. Likewise, the sender outputs pstfx and the evaluator requires it as input to complete the evaluation. These changes are only syntactical since as outputs both prfx and pstfx are adversarially-determined, and as inputs both are leaked to the adversary. However, in TLS-OPAQUE the OPRF transcript is transported over EA messages, hence making it fully visible to the environment enables a modular usage of $\mathcal{F}_{\mathsf{OPRF}}$ in our analysis of TLS-OPAQUE.

Furthermore, our functionality $\mathcal{F}_{\mathsf{OPRF}}$ fixes an important omission in the OPRF functionality as written in [18]. Namely, if the adversary compromises server $\mathcal{P}_S$, the adversary gains the ability not only to offline evaluate the (O)PRF values, via interface Eval (as in [18]), but also to perform server-side operations in

the online protocol instances, via interface SndrComplete. Our functionality $\mathcal{F}_{\mathsf{OPRF}}$ is shown in the full version of this paper [15], where we also present a slight modification of the 2HashDH protocol of [18] which UC-emulates our modified $\mathcal{F}_{\mathsf{OPRF}}$.

Corruption model. In this paper we consider two types of corruption. First, every party can be statically and maliciously corrupted by the adversary using standard "corrupt party $\mathcal{P}$" instructions that can be issued by the adversary in the UC model at the beginning of the protocol, against any party $\mathcal{P}$ [11]. This means that party $\mathcal{P}$ will be corrupted from its first activation on, and can deviate arbitrarily from the protocol code. Second, our functionalities $\mathcal{F}_{\mathsf{OPRF}}, \mathcal{F}_{\mathsf{PHA}}, \mathcal{F}_{\mathsf{pwPHA}}$ have a special type of corruption we call "compromise" (modeled, respectively, by adversarial interfaces Compromise and StealPwdFile). If such corruption happens to a party which stores long-term protocol data, such as in our setting a server storing an OPRF key or password files, the adversary obtains the stored data. However, the server continues to follow the protocol honestly. Formally, a compromise is hence an adaptive but passive corruption.

4 Secure Channels with Binders

In this section we analyze the security of TLS 1.3 Handshake, Fig. 3 as a universally composable *unauthenticated* secure channel establishment protocol. The *Key Exchange* (KE) part of the TLS 1.3 Handshake generates a communication key which is subsequently used to implement a *secure channel*, i.e. the secure message transmission, and a *channel binder*, which can be subsequently used by TLS-EA and TLS-OPAQUE to bind post-execution authentication decisions to this secure channel.[9]

In Figure 6 we show functionality $\mathcal{F}_{\mathsf{cbSC}}$ which models both parts, i.e. an (unauthenticated) secure channel establishment extended by outputting an (exported) channel binder, and a secure communication using this channel. The first part is implemented by interfaces NewSession, Attack, and Connect, the second by interfaces Send, Deliver, and interface ExpireSession allows any party to close the channel. Functionality $\mathcal{F}_{\mathsf{cbSC}}$ in Figure 6 is a standard unauthenticated secure channel functionality (e.g., [12]), extended with a channel binder CB. We mark this extension with gray boxes. The channel binder CB is output to both channel endpoints. The code that determines CB is very similar to the way in which (unauthenticated) key exchange (KE) is modeled in UC [12]. Just like a session key created by KE, CB is a random bitstring if the adversary allows two parties to passively "connect" by transmitting the

[9] TLS Handshake includes authentication, implemented by messages auth_S and auth_C in Fig. 3. However, as mentioned in footnote 7, we treat it as *unauthenticated* key exchange / secure channel establishment, because this allows us to show that the security of TLS-EA and TLS-OPAQUE is *independent* of the security of the initial authentication performed within the TLS 1.3 Handshake.

messages between them. However, if the adversary plays a man-in-the-middle, which is modeled by the ATTACK interface, it can arbitrarily set the channel binder the attacked parties output, subject to it being unique among all channel binders output by honest parties. This is how channel binders differ from session keys: It makes no difference if $\mathcal{P}$ and $\mathcal{P}'$ use the same session key on two attacked sessions, because the adversary can anyway decrypt all messages sent by $\mathcal{P}$ and it can re-encrypt them so they are successfully received by $\mathcal{P}'$. On the contrary, any authentication action, whether via TLS-EA or TLS-OPAQUE done by $\mathcal{P}$ will pertain to its channel binder CB for that session, and because $\mathcal{F}_{\mathsf{cbSC}}$ enforces that the channel binder output CB' of $\mathcal{P}'$ satisfies $\mathsf{CB}' \neq \mathsf{CB}$, the signatures issued in protocols TLS-EA and TLS-OPAQUE protocols by $\mathcal{P}$ (cf. Fig. 5) are useless for creating signatures that can be accepted by $\mathcal{P}'$. In Fig. 5 the channel binder role is played by key EMS, and since value HSC_C is derived from EMS using $\mathsf{HKDF\text{-}Expand}$, which is both a PRF and a collision-resistant hash, if $\mathsf{EMS} \neq \mathsf{EMS}'$ then $\mathsf{HSC}_C \neq \mathsf{HSC}'_C$, and since HSC_C is one of the signed fields, unforgeability of a signature implies that the signature σ_C issued by $\mathcal{P}$ is not useful in authenticating to $\mathcal{P}'$. In our analysis of TLS-HS below we will argue that it realizes functionality $\mathcal{F}_{\mathsf{cbSC}}$ with CB implemented as EMS, see Fig. 6.

We refer the reader to the full version of this paper [15] for an explanation of $\mathcal{F}_{\mathsf{cbSC}}$ interfaces.

4.1 TLS 1.3 as UC secure channel with binder

We analyze TLS 1.3 as a realization of the ideal functionality $\mathcal{F}_{\mathsf{cbSC}}$. In Figure 7 we specify how TLS 1.3 implements $\mathcal{F}_{\mathsf{cbSC}}$ commands NEWSESSION and SEND, used resp. to start a handshake, shown in Fig. 3, and to send a message on a secure channel established by it, and we show how parties form their outputs based on received network messages, resp. in FINALIZE which finalizes the handshake, and RECEIVED which stands for receiving a message on the channel.

The implementation in Fig. 7 follows the schematic protocol of Fig. 3 except for adding cid fields to the handshake messages, which model sender TCP port number. Also, in Fig. 7 for brevity we denote function $\mathsf{HKDF\text{-}Extract}$ used to derive the handshake secret HS from the Diffie-Hellman value g^{xy} as H, treated as a Random Oracle in the security analysis, and we shortcut the derivation of the Exporter Main Secret EMS (which is output as *channel binder*) and the traffic-encrypting keys $\mathsf{AEK}_C, \mathsf{AEK}_S$ from HS using key derivation function $\mathsf{KDF}^f(\mathsf{MS}, \mathsf{tr})$ for flags $f \in \{0, 1, 2\}$, where $\mathsf{KDF}^f(k, x)$ stands for $\mathsf{KDF}(k, (x|f))$. The key derivation procedure in Fig. 3 can be rendered by setting each derived key in this way. Since function $\mathsf{HKDF\text{-}Expand}$ used in TLS 1.3 is implemented as HMAC, it implies that KDF is both a secure PRF and a collision-resistant hash on full input $(k, x|f)$ [20], and we use both properties in the security analysis. Finally, we emulate TLS message transport by implementing command $(\text{SEND}, \mathsf{cid}, m)$ of $\mathcal{P}$ as sending (cid', c) where cid' is the presumed counterparty channel identifier for session $(\mathcal{P}, \mathsf{cid})$ and $c = \mathsf{AEnc}(\mathsf{AEK}_{\mathcal{P}}, (\mathsf{ctr}, m))$ where ctr is the current value of the counter for this traffic direction. (Note that each direction, $\mathcal{P}$-to-$\mathcal{P}'$ and $\mathcal{P}'$-to-$\mathcal{P}$, uses a separate key AEK and counter ctr.)

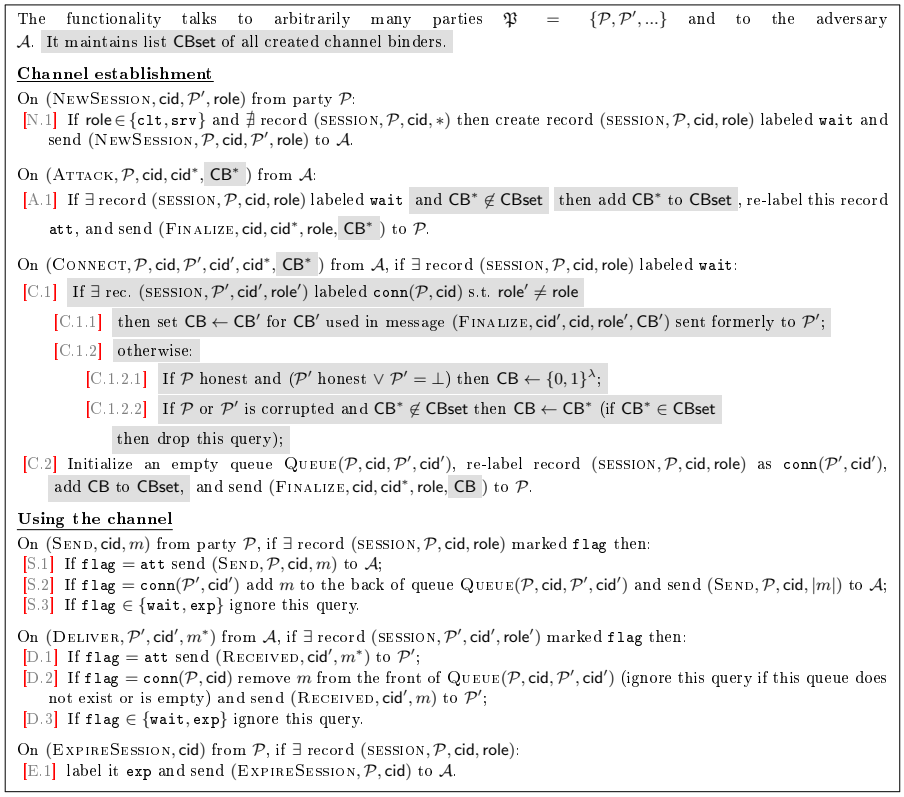

Fig. 6: Secure channel functionality $\mathcal{F}_{\mathsf{cbSC}}$. Without gray parts, the functionality implements secure unauthenticated channels. The gray parts provide both ends of a channel with a high-entropy unique "channel binder" CB that can be used for, e.g., subsequent authentication.

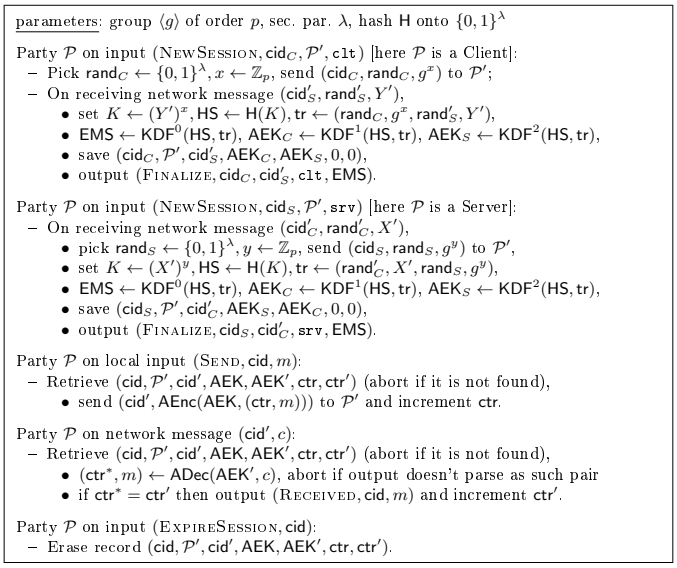

Fig. 7: TLS 1.3 as realization of functionality $\mathcal{F}_{\mathsf{cbSC}}$

The security of TLS handshake and message transport is captured as follows:

Theorem 1 (Security of TLS as unauthenticated secure channel). *TLS 1.3 handshake and message transport protocol specified in Fig. 7 UC-emulates functionality $\mathcal{F}_{\mathsf{cbSC}}$ in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model, with H modeled as random oracle, if function KDF is both a PRF and a CRH, AEnc is CUF-CCA secure, and the* Gap CDH *assumption holds on group $\langle g \rangle$, assuming static malicious corruptions.*

We refer to the full version of this work [15] for the cryptographic assumptions in this theorem, and for its full proof. Sketching it briefly, we exhibit simulator $\mathcal{S}$ which sends $Z_i = g^{z_i}$ for random z_i on behalf of each session $i = (\mathcal{P}, \mathsf{cid})$, hence it can predict its outputs in case of active attacks, but if two honest parties are passively connected $\mathcal{S}$ picks a random key AEK (which it uses to emulate secure channel communication) while $\mathcal{F}_{\mathsf{cbSC}}$ picks channel binder EMS independently at random. Since in the protocol AEK and EMS are derived via KDF from $\mathsf{HS} = \mathsf{H}(K)$ for $K = g^{z_i * z_j}$, computing this value given passively observed values $Z_i = g^{z_i}$ and $Z_j = g^{z_j}$ is related to breaking Diffie-Hellman. By hybridizing over all sessions, and guessing the identity of a passively connected counterparty and the H query which computes the key, it is possible that one could base security on a standard computational DH assumption, albeit with very loose security reduction. Instead, we show a tight reduction to the gap version of the Square DH assumption (which is equivalent to Gap CDH). The reduction embeds a randomization of a single SqDH challenge into all Z_i values, and uses the DDH oracle to detect hash queries $\mathsf{H}(K)$ for $K = \mathsf{DH}(Z'_i, Z'_j)$ into which it can either embed a chosen key HS, if one of Z'_i, Z'_j is adversarial, or

which it can map to the SqDH challenge, if both Z'_i and Z'_j come from honest parties.

5 Post-Handshake Authentication

In this Section we provide a model for post-handshake authentication (PHA), that is, a secure channel that allows for *later* public key authentication of the channel endpoints *after* already establishing the (unautenticated) channel. As a side product, we will prove security of "real-world" TLS-EA. Namely, we demonstrate that Exported Authenticators is a secure post-handshake authentication protocol.

5.1 Post-Handshake Authentication Model

Figure 8 shows a UC model $\mathcal{F}_{\mathsf{PHA}}$ for post-handshake authentication, which allows establishing an *unauthenticated* secure channel between any two parties, and then performing subsequent authentication of that channel with public keys. On a very high level, $\mathcal{F}_{\mathsf{PHA}}$ provides the following guarantees:

- **Unforgeability:** Eve cannot authenticate to Bob under Alice's public key;
- **Channel binding:** Eve cannot authenticate (even with her own keys) on channels that she is not an endpoint of.

An honest walk-through. We exemplarily describe channel establishment and authentication for two parties C and S, with C authenticating to S. We ask the reader to ignore fields $\mathsf{mode}, \mathsf{ak}, \mathsf{ske}$ of $\mathcal{F}_{\mathsf{PHA}}$ for the sake of this walk-through; an explanation of these special fields follows further below.

$\mathcal{F}_{\mathsf{PHA}}$ inherits [C.1] all channel interfaces of $\mathcal{F}_{\mathsf{cbSC}}$, but without channel binder CB, imlementing secure but unauthenticated channels. Both C and S call NewSession of $\mathcal{F}_{\mathsf{PHA}}$ to establish a channel. Let us assume that the adversary decides to connect their requests. Both parties receive a Finalize notification and learn the channel identifier $\mathsf{cid}_C/\mathsf{cid}_S$ under which the other endpoint knows the channel. We note however that neither C nor S learn with whom they actually got connected. The established channel can be used to send messages securely.

To tell his peer on channel cid_C who he is actually connected to, C first generates a key by querying $\mathcal{F}_{\mathsf{PHA}}$ with $(\textsc{KeyGen}, \mathsf{kid}, \varepsilon, \varepsilon, \mathtt{std})$, resulting in output $(\textsc{key}, \mathsf{kid}, \varepsilon, \varepsilon, \varepsilon, \mathsf{pk})$ with [G.1] adversarially-chosen but [G.2] fresh pk. kid denotes a non-secret identifier which helps C managing her public keys. ε denotes an empty string – these fields are only used in a special mode of $\mathcal{F}_{\mathsf{PHA}}$ called *transportable key mode* ($\mathtt{mode} = \mathtt{tk}$, see explanation further below). For this walkthrough, we use standard key generation ($\mathtt{mode} = \mathtt{std}$). $\mathcal{F}_{\mathsf{PHA}}$ adds pk to [G.3] lists pkReg and $\mathsf{pkey}[C]$. pkReg contains all public keys generated through $\mathcal{F}_{\mathsf{PHA}}$ (in any mode). $\mathsf{pkey}[C]$ is a list containing all standard public

The functionality talks to arbitrarily many parties $\mathfrak{P} = \{\mathcal{P}, \mathcal{P}', ...\}$ and to the adversary $\mathcal{A}$. It maintains lists pkReg (all registered public keys), pkComp (all compromised keys), $\mathsf{pkey}[\mathsf{pid}]$ (standard public keys generated by party pid), keReg (all key envelopes) and an array $\mathsf{tkey}[\mathsf{aux}, h]$ associating transportable keys with handle h and auxiliary information aux.

Channel establishment and Use

[C.1] NewSession, Attack, Connect, Send, Deliver, *and* ExpireSession, *as in* $\mathcal{F}_{\mathsf{cbSC}}$, *Figure 6, but without gray parts.*

Key Generation and Corruption

On $(\textsc{KeyGen}, \mathsf{kid}, \mathsf{ak}, \mathsf{aux}, \mathtt{mode})$ from $\mathsf{pid} \in \mathfrak{P} \cup \{\mathcal{A}\}$:
[G.1] Send $(\textsc{KeyGen}, \mathsf{kid}, \mathsf{pid}, \mathsf{aux}, \mathtt{mode})$ to $\mathcal{A}$ and receive back $(\mathsf{kid}, \mathsf{ske}, \mathsf{pk})$.
[G.2] If $(\mathsf{pid} \neq \mathcal{A} \wedge \mathsf{pk} \in \mathsf{pkReg})$, or if $(\mathtt{mode} = \mathtt{tk} \wedge \mathsf{ske} \in \mathsf{keReg})$ then abort;
[G.3] Else,
- If $\mathtt{mode} = \mathtt{std}$ then add pk to $\mathsf{pkey}[\mathsf{pid}]$;
- If $\mathtt{mode} = \mathtt{tk}$ then set $\mathsf{tkey}[\mathsf{ak}, \mathsf{ske}] \leftarrow (\mathsf{aux}, \mathsf{pk})$;
- Add pk to pkReg, and if $\mathsf{pid} = \mathcal{A}$ then also add pk to pkComp;
- Finally, output $(\textsc{key}, \mathsf{kid}, \mathsf{ak}, \mathsf{ske}, \mathsf{aux}, \mathsf{pk})$ to party pid.

On $(\textsc{Compromise}, \mathcal{P})$ from $\mathcal{A}$ *(requires permission from* $\mathcal{Z}$*)*:
– Add $\mathsf{pkey}[\mathcal{P}]$ to pkComp.

On $(\textsc{GetAuxData}, \mathsf{ak}, \mathsf{ske})$ from $\mathsf{pid} \in \mathfrak{P} \cup \{\mathcal{A}\}$:
[T.1] If $\mathsf{pid} = \mathcal{A}$ then parse $(*, \mathsf{pk}) \leftarrow \mathsf{tkey}[\mathsf{ak}, \mathsf{ske}]$ and add pk to pkComp;
[T.2] Output $\mathsf{tkey}[\mathsf{ak}, \mathsf{ske}]$ to pid;

Active Attack

On $(\textsc{ActiveAttack}, \mathcal{P}, \mathsf{cid}, \mathsf{ssid}, \mathsf{ctx}^*, \mathsf{pk}^*)$ from $\mathcal{A}$
[A.1] If $\exists$ record $(\textsc{session}, \mathcal{P}, \mathsf{cid}, \mathsf{role})$ marked $\mathtt{att}$, or $\exists$ record $(\textsc{session}, \mathcal{P}, \mathsf{cid}, \mathsf{role})$ marked $\mathtt{conn}(\mathcal{P}', \mathsf{cid}')$ with $\mathcal{P}'$ corrupt, then do:
- If $\mathsf{pk}^* \notin \mathsf{pkReg} \setminus \mathsf{pkComp}$ then record $(\textsc{Auth}, \varepsilon, \varepsilon, \mathcal{P}, \mathsf{cid}, \mathsf{ssid}, \mathsf{ctx}^*, \mathsf{pk}^*)$.

[A.2] Output $(\textsc{AuthSend}, \mathsf{cid}', \mathsf{ssid})$ to $\mathcal{P}'$.

Unilateral Public-Key Authentication

On $(\textsc{AuthSend}, \mathcal{P}', \mathsf{cid}, \mathsf{ssid}, \mathsf{ctx}, \mathsf{ak}, \mathsf{ske}, \mathsf{pk}, \mathtt{mode})$ from $\mathcal{P}$:
[S.1] If $\mathtt{mode} = \mathtt{tk}$ and $\mathsf{tkey}[\mathsf{ak}, \mathsf{ske}]$ is not defined then send $(\mathsf{ssid}, \mathsf{ak}, \mathsf{ske})$ to $\mathcal{A}$ and receive back activation;
[S.2] If $\exists$ record $(\textsc{session}, \mathcal{P}, \mathsf{cid}, \mathsf{role})$ marked $\mathtt{conn}(\mathcal{P}', \mathsf{cid}')$ then initialize $b \leftarrow 0$ and:
- If $\mathtt{mode} = \mathtt{std}$ and [S.2.1] $\mathsf{pk} \in \mathsf{pkey}[\mathcal{P}]$ then set $b \leftarrow 1$;
- If $\mathtt{mode} = \mathtt{tk}$ and $\mathsf{tkey}[\mathsf{ak}, \mathsf{ske}] = (*, \mathsf{pk}')$ then set $\mathsf{pk} \leftarrow \mathsf{pk}'$ and $b \leftarrow 1$.
- If $b = 1$ then record $(\textsc{Auth}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}', \mathsf{ssid}, \mathsf{ctx}, \mathsf{pk})$

[S.3] Send $(\textsc{AuthSend}, \mathcal{P}, \mathcal{P}', \mathsf{cid}, \mathsf{ssid}, \mathsf{ctx}, \mathsf{pk}, \mathtt{mode}, b)$ to $\mathcal{A}$ and receive back activation.
[S.4] Output $(\textsc{AuthSend}, \mathsf{cid}', \mathsf{ssid})$ to $\mathcal{P}'$.

On $(\textsc{AuthVerify}, \mathsf{cid}', \mathsf{ssid}, \mathsf{ctx}, \mathsf{pk})$ from $\mathcal{P}'$
[V.1] Send $(\textsc{AuthVerify}, \mathcal{P}', \mathsf{cid}', \mathsf{ssid}, \mathsf{ctx}, \mathsf{pk})$ to $\mathcal{A}$ and receive back flag f.
[V.2] If $f = 1$ and $\exists$ record $(\textsc{Auth}, *, *, \mathcal{P}', \mathsf{cid}', \mathsf{ssid}, \mathsf{ctx}, \mathsf{pk})$ then $b \leftarrow 1$ else $b \leftarrow 0$;
[V.3] Send $(\textsc{AuthVerify}, \mathsf{cid}', \mathsf{ssid}, b)$ to $\mathcal{P}'$.

Fig. 8: $\mathcal{F}_{\mathsf{PHA}}$ model for post-handshake authentication, which allows for public key authentication on an already existing unauthenticated channel. $\mathcal{F}_{\mathsf{PHA}}$ offers $\mathtt{mode} = \mathtt{std}$ key generation as used by, e.g., EA, as well as *transportable-key* mode $\mathtt{tk}$, which makes $\mathcal{F}_{\mathsf{PHA}}$ a useful modular building block for, e.g., TLS-OPAQUE. For brevity we omit the overall session identifier from all interfaces.

keys that C generated through $\mathcal{F}_{\mathsf{PHA}}$, and which C can use for authenticating on her channels.

Now that C has created pk, C wants to use pk to authenticate to S on channel cid_C. To do so, C queries $(\textsc{AuthSend}, S, \mathsf{cid}_C, \mathsf{ssid}, \mathsf{ctx}, \varepsilon, \varepsilon, \mathsf{pk}, \mathtt{std})$. ctx denotes optional auxiliary public context information that C wants to transmit alongside the authentication request. If [S.2.1] C is allowed to authenticate under pk, $\mathcal{F}_{\mathsf{PHA}}$ records $(\textsc{Auth}, C, \mathsf{cid}_C, S, \mathsf{cid}_S, \mathsf{ssid}, \mathsf{ctx}, \mathsf{pk})$, representing the fact that C successfully performed authentication under pk in this channel. $\mathcal{F}_{\mathsf{PHA}}$ then informs the adversary about the authentication attempt, including all its data and whether authentication was successful (the bit b).

To receive the result of C's authentication, the receiver S has to choose a public key and a context for verification. This data is contributed by S via interface AuthVerify, allowing applications to actively choose under which public key and context verification should be performed. Hence, we assume these public values to be transmitted by the application. In case the verifier wants to perform verification under the same pk and ctx that C performed the authentication with, $\mathcal{F}_{\mathsf{PHA}}$ [V.2] outputs success.

Transportable key mode. $\mathcal{F}_{\mathsf{PHA}}$ as described above binds usage of pk to S, the party who generated pk via interface KeyGen. This is however not realistic in dynamic settings, where, e.g., S transfers her keys to another machine in encrypted form. A concrete example is OPAQUE, where secret keys are encrypted and the resulting envelopes are sent to the server, who then stores them. In order to enable a modular analysis of such "key-handling" protocols, we introduce the notion of *transportable keys* to the UC framework, and to our $\mathcal{F}_{\mathsf{PHA}}$. When generating a transportable key by querying $(\textsc{KeyGen}, \mathsf{kid}, \mathsf{ak}, \mathsf{aux}, \mathtt{tk})$, a party provides a key identifier kid, an *application key* ak and an optional label aux. $\mathcal{F}_{\mathsf{PHA}}$ keeps the application key secret but [G.1] leaks all other values to the adversary. The requesting party then receives back [G.1] adversarially-generated *key envelope* ske and public key pk. One can think of these values as ske being an encryption of sk belonging to pk, encrypted symmetrically with key ak. $\mathcal{F}_{\mathsf{PHA}}$ stores $(\mathsf{aux}, \mathsf{pk})$ in $\mathsf{tkey}[\mathsf{ak}, \mathsf{ske}]$. The semantics of the tkey array are as follows: whoever provides input i, where $\mathsf{tkey}[i] = (\mathsf{aux}, \mathsf{pk})$, can authenticate under pk (see below), and [T.2] retrieve label aux and public key pk via interface GetAuxData. Hence, knowledge of both application key ak and envelope ske will be sufficient to authenticate under pk. Since the requesting party outputs ske, ske can be used by applications which require secret keys to be objects that can be sent around, stored, further encrypted etc.

To authenticate with transportable keys, S calls AuthSend with inputs $\mathsf{ak}, \mathsf{ske}$ and $\mathtt{mode} = \mathtt{tk}$. In case [S.1] $\mathsf{ak}, \mathsf{ske}$ are not known to $\mathcal{F}_{\mathsf{PHA}}$ (i.e., $\mathsf{tkey}[\mathsf{ak}, \mathsf{ske}]$ does not store any pk), no security is guaranteed and the adversary obtains $\mathsf{ak}, \mathsf{ske}$. $\mathcal{F}_{\mathsf{PHA}}$ then [S.2] checks again whether $\mathsf{tkey}[\mathsf{ak}, \mathsf{ske}]$ stores pk, and if so, it grants authentication by creating the corresponding Auth record including pk, and notifies the adversary about the authentication attempt, including all its data and whether authentication was successful (bit b). We note that the double

check of $\mathsf{tkey}[\mathsf{ak}, \mathsf{ske}]$ is necessary since the adversary could have registered $\mathsf{ak}, \mathsf{ske}$ in between both checks.

Adversarial interfaces. The adversary $\mathcal{A}$ can register both `std` and `tk` keys via interface KeyGen. $\mathcal{F}_{\mathsf{PHA}}$ adds such compromised keys to set pkComp. For transportable keys $\mathsf{ak}, \mathsf{ske}$, the adversary can also reveal which public key they "work for", by querying $(\text{GetAuxData}, \mathsf{ak}, \mathsf{ske})$. $\mathcal{F}_{\mathsf{PHA}}$ returns [T.2] $(\mathsf{aux}, \mathsf{pk}) = \mathsf{tkey}[\mathsf{ak}, \mathsf{ske}]$ (or $\perp$ if empty) and [T.1] adds pk to pkComp, accounting for $\mathcal{A}$ now knowing transportable keys for pk. Altogether, in pkComp we find all keys generated in any `mode` by $\mathcal{F}_{\mathsf{PHA}}$ that are compromised: the adversary can authenticate with these keys (as well as unknown keys $\notin \mathsf{pkReg}$) on his channels [A.1] using the ActiveAttack interface. $\mathcal{A}$ can always make authentication fail by [V.1-2] sending $f = 0$ in its AuthVerify query to $\mathcal{F}_{\mathsf{PHA}}$. Regarding leakage, $\mathcal{A}$ learns all inputs of AuthSend except for uncompromised transportable keys $\mathsf{ak}, \mathsf{ske}$, as well as public verification values $\mathsf{pk}, \mathsf{ctx}$. With such a strong adversary, $\mathcal{F}_{\mathsf{PHA}}$ guarantees that an authentication mechanism does not rely on the secrecy of messages.

On usage of party identifiers. Our modeling of PHA, just as our modeling of unauthenticated channels in Section 4, does not provide any initial guarantees about the identity of a peer. Hence, throughout this paper, party identifiers are interpreted only as process identifiers. For example, pid could be a unique combination of IP address and port, and we make only the minimal assumption that it is always the same process sending from this addresses' port. Consequently, party identifiers are used by functionalities only to determine which messages were generated by the same process. In protocol instructions, sending a message requires specification of an intended recipient, and hence we add the intended recipient to input AuthSend of $\mathcal{F}_{\mathsf{PHA}}$. However, since our modeling of unauthenticated channels is weak in the sense that parties are oblivious of which process (i.e., which pid) their channel actually got connected to, the intended recipient might not coincide with the process holding the other end of the channel. By this we capture an authentication-less setting with a network adversary who is freely rerouting/rewriting messages. Consequently, $\mathcal{F}_{\mathsf{PHA}}$ overlooks any mismatch in a party's perception and instead bases authentication decisions for a specific channel and pk solely on whether an endpoint ($=\mathsf{pid}$) is eligible to authenticate under pk.

5.2 The Exported Authenticators Protocol

The EA protocol that we consider for our analysis is depicted in Figure 9. It generalizes Exported Authenticators as specified in 2 in several aspects:

1. Π_{EA} abstracts from the channel establishment and can provide post-handshake authentication for any "handshake" protocol that securely instantiates $\mathcal{F}_{\mathsf{cbSC}}$
2. Π_{EA} works with standard signature keys and *transportable* keys (see below), which enable Π_{EA} to use key material provided by an application
3. Π_{EA} does not hash messages before signing/mac'ing

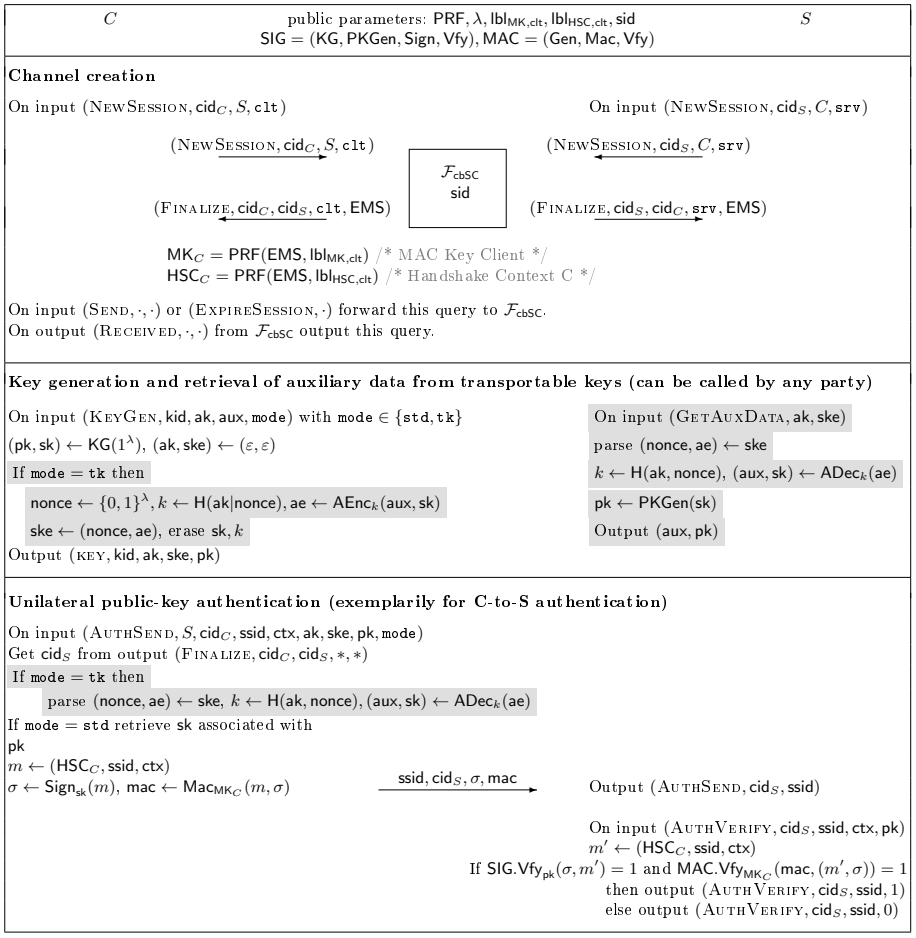

Fig. 9: Protocol Π_{EA} is a unidirectional post-handshake authentication of channel binder EMS provided by hybrid functionality $\mathcal{F}_{\mathsf{cbSC}}$. We depict a C-to-S authentication flow with either `std` key mode or transportable key mode `tk`. For brevity we omit the functionality's identifier sid from all queries and messages.

4. Π_{EA} sends messages in the clear instead of sending them over the channel-to-authenticate
5. Public key and context are verification is provided by the application instead of being sent by the authenticator
6. Fields EACert and extensions ext are subsumed in the ctx object, about which no further assumptions are made

In Π_{EA}, parties can establish channels by calling $\mathcal{F}_{\mathsf{cbSC}}$. If the channel is finalized, the endpoints share a unique channel binder EMS (cf. Section 4 for details). The endpoints, let's call them C and S, then derive transcript digest and MAC keys $\mathsf{MK}_C, \mathsf{MK}_S, \mathsf{HSC}_C, \mathsf{HSC}_S$ from EMS. We note that this is the only place in this paper where the roles `clt`, `srv` have an effect: these are roles that parties have in some application, such as TLS, and they help us here to derive different digest and MAC key for C and S from public labels $\mathsf{lbl}_{\mathsf{MK,clt}}, \mathsf{lbl}_{\mathsf{MK,srv}}, \mathsf{lbl}_{\mathsf{HSC,clt}}, \mathsf{lbl}_{\mathsf{HSC,srv}}$ that reflect these roles.

Π_{EA} is a multi-party protocol that allows arbitrary parties to establish channels with each other, allows unlimited generation of keys and unlimited numbers of unilateral authentication sessions per channel. We exemplarily describe such an authentication performed by C for a channel with S as depicted in Figure 9. We start with standard signing keys and for now ignore the gray parts of the figure. Upon input (KeyGen, $\mathsf{kid}, \mathsf{ak}, \mathsf{aux}, \mathtt{std}$), C generates a key pair $(\mathsf{sk}, \mathsf{pk})$ by running the key generation of the signature scheme (values $\mathsf{ak}, \mathsf{aux}$ are ignored in normal mode), and outputs pk to the application. When C wants to authenticate on her channel cid_C, she looks up[10] identifier cid_S in the Finalize output of $\mathcal{F}_{\mathsf{cbSC}}$, and signs message $(\mathsf{HSC}_C, \mathsf{ssid}, \mathsf{ctx})$ with sk, where ssid is the EA nonce and ctx is the EACert field (containing identity information such as, e.g., a certificate) that C wants to convey. Then C macs the message together with the signature under mac key MK_C. C then sends all values to S, who accepts or rejects depending on whether signature and mac verify for $\mathsf{HSC}_C, \mathsf{MK}_C$ that S computes from channel binder EMS for her channel cid_S.

Instantiating transportable keys. A transportable key is a protected secret key, also called *envelope* throughout the paper. One can think of an envelope as, e.g., an encryption of the secret key. Π_{EA} allows parties to export such envelopes to the application. Since this way envelopes can "travel" to other parties who can then attempt to extract the secret key from them, transportable keys can be used by any party who possesses both the envelope and whatever is required to unlock the secret key from it. A transportable key requires a signature key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KG}(1^\lambda)$. Then, an encryption key k is generated as $k \leftarrow \mathsf{H}(\mathsf{ak}, \mathsf{nonce})$, where ak is an *application key*, and H hashes to the key space of a symmetric cipher. The envelope is then $\mathsf{ske} \leftarrow (\mathsf{nonce}, \mathsf{ae})$, where ae is an encryption of $\mathsf{aux}, \mathsf{sk}$ under k, for auxiliar information (e.g., a label) aux. Obviously, the application

[10] We assume C to learn this information as otherwise, when sending messages over plain connections, we would have no mean of informing S which channel the authentication is intended for. This can be avoided by instead sending messages *over* the secure channel.

key ak is enough to decrypt sk from envelope ske. Hence, the authentication step in Π_{EA} can alternatively be conducted by an authenticator C running on inputs ak, ske, pk (cf. gray parts in Figure 9): before signing and mac'ing, C first recovers sk from ak, ske.

This concludes our description of Π_{EA}, and we are ready to state its security. We refer to the full version [15] for the formal definitions of the cryptographic assumptions within the Theorem and for the full proof.

Theorem 2 (Security of Π_{EA}). *Protocol Π_{EA} depicted in Figure 9 UC-emulates functionality $\mathcal{F}_{\mathsf{PHA}}$ in the $(\mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{cbSC}})$-hybrid model,* PRF *is both a secure PRF and a collision-resistant hash function, with* H *modeled as random oracle,* (KG, PKGen, Sign, Vfy) *a perfectly complete and EUF-CMA-secure signature scheme,* MAC *is perfectly complete and EUF-CMA-secure MAC, and* (AEnc, ADec) *a CUF-CCA- and RKR-secure encryption scheme that is equivocable, and restriction to static malicious corruptions and adaptive server compromise.*

There are 6 discrepancies described above between Π_{EA} and EA as described in Section 2. As already argued in Section 1, (4) does not void security, and neither does hashing (3). (1),(2),(5),(6) are strict generalizations of the Exported Authenticators protocol. Hence, the security of TLS-EA follows from the security of Π_{EA}, with $\mathcal{F}_{\mathsf{cbSC}}$ instantiated with the TLS-HS through the standard UC composition theorem [11].

Corollary 1. *Protocol TLS-EA specified in Section 2 securely realizes $\mathcal{F}_{\mathsf{PHA}}$.*

6 Security of TLS-OPAQUE

6.1 Password-based post-handshake authentication

We give a model $\mathcal{F}_{\mathsf{pwPHA}}$ for password-based post-handshake authentication in Figure 10. On a high level, $\mathcal{F}_{\mathsf{pwPHA}}$ guarantees the following:

- **Limitation to one guess per online attack:** Each run of the protocol reveals at most one bit of information about the opponent's password to each participant;
- **Resistance to offline attacks:** Dictionary attacks on passwords are prevented unless a server is compromised;
- **Resistance to precomputation attack:** An attacker cannot speed up dictionary attacks through computation performed prior to server compromise;
- **Enable rate limiting:** Servers can map login attempts to registered user accounts;
- **Channel binding:** One cannot authenticate (even with correct password) on channels one is not an endpoint of;

We explain how the functionality can be used by a client C and server S to first establish an unauthenticated channel, and then subsequently authenticate to each other using a password (client) and password file (server). We

The functionality talks to arbitrarily many parties $\mathfrak{P} = \{\mathcal{P}, \mathcal{P}', ...\}$ and to the adversary $\mathcal{A}$. It maintains counters $\mathsf{ctr}[\mathcal{P}, \mathsf{uid}]$ initially set to 0.

Channel establishment and Use

NewSession, Attack, Connect, Send, Deliver, *and* ExpireSession, *as in* $\mathcal{F}_{\mathsf{cbSC}}$, *Figure 6, but without gray parts.*

Password Registration, Compromise, and Offline Dictionary Attack

On (StorePwdFile, ssid, $\mathcal{P}$, uid, pw,) from $\mathcal{P}'$:
[F.1] If $\nexists$ record (Store, $\mathcal{P}$, ssid, $\cdot$, $\cdot$), record (Store, $\mathcal{P}$, ssid, uid, pw) and send (Store, $\mathcal{P}'$, ssid, $\mathcal{P}$, uid) to $\mathcal{A}$.

On (StorePwdComplete, $\mathcal{P}^*$, ssid) from $\mathcal{A}$:
[C.1] If $\exists$ record (Store, $*$, ssid, uid, pw) but $\nexists$ record (file, $\mathcal{P}^*$, uid, $\cdot$) then record (file, $\mathcal{P}^*$, uid, pw) and mark it `uncompromised`.

On (StealPwdFile, $\mathcal{P}$, uid) from $\mathcal{A}$ *(requires permission from* $\mathcal{Z}$*)*:
[S.1] If $\nexists$ record (file, $\mathcal{P}$, uid, pw), return "no file" to $\mathcal{A}$;
[S.2] Else mark this record `compromised` and return "file stolen".

On (OfflTestPwd, $\mathcal{P}$, uid, pw$'$) from $\mathcal{A}$ *(requires permission from* $\mathcal{Z}$*)*:
[O.1] If $\exists$ record (file, $\mathcal{P}$, uid, pw) marked `compromised` then return "correct guess" if pw $=$ pw$'$ and "wrong guess" if pw $\neq$ pw$'$.

Active Attacks

On (ActiveAttack, ssid$'$, $\mathcal{P}$, cid, uid) from $\mathcal{A}$:
[A.1] If $\exists$ record (session, $\mathcal{P}$, cid) marked `att`, or if $\exists$ record (session, $\mathcal{P}'$, cid$'$) marked `conn`($\mathcal{P}$, cid) where $\mathcal{P}'$ is corrupted, then create record (pwAuth, ssid$'$, $\mathcal{A}$, ε, $\mathcal{P}$, cid, uid, ε, `init`, 0)
[A.2] Output (pwInit, ssid$'$, cid, uid) to $\mathcal{P}$.

On (TestPwd, $\mathcal{P}$, uid, pw$'$) from $\mathcal{A}$:
[T.1] If $\exists$ record (file, $\mathcal{P}$, uid, pw) and $\mathsf{ctr}[\mathcal{P}, \mathsf{uid}] > 0$ then do:
  [T.1.1] If pw $=$ pw$'$ then return "correct guess" and rewrite `init` to `match` in all records (pwAuth, $*$, $\mathcal{A}$, ε, $\mathcal{P}$, $*$, uid, ε, `init`, 0);
  [T.1.2] If pw $\neq$ pw$'$ then return "wrong guess";
  [T.1.2] Set $\mathsf{ctr}[\mathcal{P}, \mathsf{uid}] - -$.

On (Impersonate, ssid$'$, $\mathcal{P}$, uid, pw*) from $\mathcal{A}$:
[Im.1] If pw* $= \varepsilon$ and $\exists$ record (file, $\mathcal{P}$, uid, pw$'$) marked `compromised` and record (pwAuth, ssid$'$, $*$, $*$, $\mathcal{A}$, ε, uid, pw, `init`, 0), if pw $=$ pw$'$ overwrite `init` with `match` and reply with "correct guess", otherwise overwrite `init` with `fail` and reply with "wrong guess";
[Im.2] If pw$^* \neq \varepsilon$ and $\exists$ record (pwAuth, ssid$'$, $*$, $*$, $\mathcal{A}$, ε, uid, pw, `init`, 0), if pw $=$ pw* then overwrite `init` with `match` and reply with "correct guess", otherwise overwrite `init` with `fail` and reply with "wrong guess".

Asymmetric Password Authentication

On (pwInit, $\mathcal{P}''$, cid, ssid$'$, uid, pw) from $\mathcal{P} \in \mathfrak{P}$:
[In.1] Drop the query if it is not the first one for ssid$'$;
[In.2] Send (pwInit, $\mathcal{P}$, $\mathcal{P}''$, cid, ssid$'$, uid) to $\mathcal{A}$ and receive back (pwInit, $\mathcal{P}$, $\mathcal{P}''$, cid, ssid$'$, uid, `ok`);
[In.3] If $\exists$ record (session, $\mathcal{P}$, cid) marked `att`, create record (pwAuth, ssid$'$, $\mathcal{P}$, cid, $\mathcal{A}$, ε, uid, pw, `init`, 0), set $\mathcal{P}'' \leftarrow \mathcal{A}$;
[In.4] If $\exists$ record (session, $\mathcal{P}$, cid) marked `conn`($\mathcal{P}'$, cid$'$) create record (pwAuth, ssid$'$, $\mathcal{P}$, cid, $\mathcal{P}'$, cid$'$, uid, pw, `init`, 0), set $\mathcal{P}'' \leftarrow \mathcal{P}'$, cid $\leftarrow$ cid$'$;
[In.5] Output (pwInit, cid, ssid$'$, uid) to $\mathcal{P}''$

On (pwProceed, ssid$'$) from $\mathcal{P}$:
[P.1] Send (pwProceed, $\mathcal{P}$, cid, ssid$'$, uid) to $\mathcal{A}$;
[P.2] If $\exists$ record (pwAuth, ssid$'$, $\mathcal{A}$, ε, $\mathcal{P}$, $*$, uid, ε, `init`, 0) then set $\mathsf{ctr}[\mathcal{P}, \mathsf{uid}] + +$;
[P.3] If $\exists$ records (pwAuth, ssid$'$, $*$, $*$, $\mathcal{P}$, $*$, uid, pw, `init`, 0) with pw $\neq \varepsilon$ and (file, $\mathcal{P}$, uid, pw$'$), then overwrite `init` with `match` if pw $=$ pw$'$ and with `fail` otherwise.

On (pwDeliver, ssid$'$, $\mathcal{P}$, b) from $\mathcal{A}$
[D.1] If $b = 0$ output (pwDecision, ssid$'$, `fail`) to $\mathcal{P}$.
[D.2] Find record (pwAuth, ssid$'$, $\mathcal{P}'$, $*$, $\mathcal{P}''$, $*$, $*$, $*$, `state`, ctr) with $\mathcal{P} = \mathcal{P}''$ or $\mathcal{P} = \mathcal{P}''$, otherwise drop query;
[D.3] Rewrite state from `init` to `fail`, then set `result` $=$ `state` and $ctr + +$; if $ctr = 2$ overwrite `state` with `completed` in the record;
[D.4] Output (pwDecision, ssid$'$, `result`) to $\mathcal{P}$.

Fig. 10: $\mathcal{F}_{\mathsf{pwPHA}}$ model of *password-based* post-handshake authentication, again omitting session identifier sid.

emphasize how $\mathcal{F}_{\mathsf{pwPHA}}$ enforces the above guarantees alongside our explanation. To start, the client registers [F.1] with the server by storing some password-dependent information (called *password file*), under some user name uid. This results in $\mathcal{F}_{\mathsf{pwPHA}}$ registering that a file with $\mathsf{uid}, \mathsf{pw}$ was stored at S, by [C.1] installing record FILE. This process can be stopped by the adversary by not sending StorePwdComplete, allowing analysis of protocols with interactive registration phase and without guaranteed delivery of messages.

Parties C and S can establish an unauthenticated channel by calling $\mathcal{F}_{\mathsf{pwPHA}}$'s NewSession interface. See Sec. 4 or description of $\mathcal{F}_{\mathsf{PHA}}$ in Sec. 5.1 for more details. We note that registration and channel establishment do not rely on each other and can thus be performed in arbitrary order.

Having concluded registration and channel establishment, parties connected via a channel can now authenticate to each other using password (client) and file (server). Password authentication is always initialized by the client calling pwInit with credential $\mathsf{uid}, \mathsf{pw}$. The client also specifies the channel to authenticate, cid_C, and intended recipient S. Similar to our modeling of EA, $\mathcal{F}_{\mathsf{pwPHA}}$ ignores intended recipients and instead [In.4] refers to $(\text{SESSION}, *, *)$ records to figure out who the end points of a channel are. Assuming that C's channel cid_C is with S, $\mathcal{F}_{\mathsf{pwPHA}}$ [In.4] stores a record $(\text{pwAuth}, \mathsf{ssid}, C, \mathsf{cid}_C, S, \mathsf{cid}_S, \mathsf{uid}, \mathsf{pw}, \texttt{init}, 0)$ and [In.5] notifies S of the authentication session, the channel and the uid, where disclosure of the uid **enables rate-limiting**. This record reflects initiator and responder roles by order of mention. Having been notified, S can now either accept or decline to participate by calling pwProceed for said session. An application can hence apply rate-limiting policies, such as "at most 5 authentication attempts for uid per minutes" by calling pwProceed in a policy-conforming way. pwProceed will only move authentication forward [P.3] if there is a file stored for S and uid: if the password in that file matches pw, then the state of the pwAuth record is rewritten to `match`, otherwise it is rewritten to `fail`. It is instructive to see that $\mathcal{F}_{\mathsf{pwPHA}}$ bases this decision on password data *held by corresponding channel endpoints* [In.4], ensuring that authentication can only be successful for parties sharing a channel (**channel binding**). Finally, $\mathcal{F}_{\mathsf{pwPHA}}$ creates adversarially-scheduled (via interface pwDeliver) outputs [D.4] reflecting the state of the pwAuth record [D.2-3], namely `fail` or `match`, towards both C and S, notifying them about the outcome of authentication. As soon as two outputs are delivered, $\mathcal{F}_{\mathsf{pwPHA}}$ [D.3] marks a record as `completed`, which concludes the authentication flow.

Adversarial interfaces. $\mathcal{F}_{\mathsf{pwPHA}}$ has a very simple leakage pattern - all inputs are public except for passwords (see messages to $\mathcal{A}$ in [F.1],[In.2] and [P.1]). To account for interactive protocols, we let adversary $\mathcal{A}$ acknowledge all honest inputs ([C.1],[In.1] and [P.1]), modeling Denial-of-Service attacks at different stages of the execution, and we let $\mathcal{A}$ make any authentication session fail [D.1]. StealPwdFile, OfflTestPwd and Impersonate model adaptive compromise of server's password files. If the attacker wants to compromise a file, say, for uid stored at server S, it informs $\mathcal{F}_{\mathsf{pwPHA}}$ by sending $(\text{StealPwdFile}, S, \mathsf{uid})$. $\mathcal{F}_{\mathsf{pwPHA}}$ [S.2] marks the corresponding file as `compromised`, which "unlocks" in-

terfaces OfflTestPwd and Impersonate (**resistance to offline attacks**): $\mathcal{A}$ can now make unlimited password guesses against the file via OfflTestPwd [O.1], and it can use the file to actively play the role of the honest server S in authentication sessions described above, using (Impersonate, ssid, S, uid, ε) [Im.1]. We capture the ability of the attacker of compute its own password files by an optional input pw to Impersonate. If this input [Im.2] is set, $\mathcal{A}$ is specifying which password it wants to use for file creation. $\mathcal{A}$ can only mount such attacks *on an attacked channel*, which is enforced by $\mathcal{F}_{\mathsf{pwPHA}}$ by [In.3] checking whether the attacked client a channel that is flagged `att`. If so, $\mathcal{F}_{\mathsf{pwPHA}}$ creates [In.3] a pwAuth record with $\mathcal{A}$ as server and follows the procedure of honest authentication, except that it expects $\mathcal{A}$ to use an Impersonate query instead of pwProceed. This concludes already the description of active attacks that $\mathcal{F}_{\mathsf{pwPHA}}$ allows to mount against a client. We further note that $\mathcal{F}_{\mathsf{pwPHA}}$ features a strong OfflTestPwd interface since it enforces **resistance to precomputation attacks** [18]: $\mathcal{F}_{\mathsf{pwPHA}}$ does not allow to pre-register guesses[11] and obtain a batched reply upon file compromise. Further, as common for asymmetric password authentication models, server compromise constitutes a form of corruption, which requires permission from the environment $\mathcal{Z}$, and hence StealPwdFile and OfflTestPwd can only be queried by $\mathcal{A}$ upon being instructed by $\mathcal{Z}$.

ActiveAttack and TestPwd interfaces allow an adversary to actively attack server S, guessing which password was used to generate a file. $\mathcal{A}$ initializes such attack by calling ActiveAttack, specifying S, uid. $\mathcal{F}_{\mathsf{pwPHA}}$ initializes the [A.1] corresponding pwAuth record with $\mathcal{A}$ in client role and [A.2] notifies the server. $\mathcal{A}$ can postpone the password guess, allowing analysis of protocols such as TLS-OPAQUE, where the attacker is not committed to a password guess from the very beginning of the attack. We complement the ActiveAttack interface with interface TestPwd, with inputs S, uid and password guess pw. Since cracking a password file of S, uid results in the insecurity of all ongoing and future authentication session with S, uid, interface TestPwd is not session-based but file-based, and a successful guess results in all ongoing active attacks against this file being successful (i.e., $\mathcal{F}_{\mathsf{pwPHA}}$ [T.1.1] rewrites the corresponding pwAuth records to `match`). To make sure that the number of adversarial TestPwd queries does not exceed the number of active attacks against a specific file, i.e., to ensure **limitation to one guess per online attack**, we let $\mathcal{F}_{\mathsf{pwPHA}}$ maintain a counter ctr$[S, \mathsf{uid}]$ for every file ([P.2],[T.1] and [T.1.2]), indicating the remaining password guesses that $\mathcal{A}$ can issue against the file for uid.

On registration and authentication. Typically, user registration will assume some form of authenticated channels for the user and servers to identify each other. This authentication can take many forms from PKI to physical rendezvous. However, we do not force authentication into the model so it can also support, for example, anonymous settings where no authentication, or one-way authentica-

[11] As a real-world example of an attack that is excluded by $\mathcal{F}_{\mathsf{pwPHA}}$, imagine an adversary preparing a list of hashed password guesses and, upon compromise, searching this list for a match. See [14] for a "non-strong" aPAKE functionality allowing for such attacks.

tion, is deemed sufficient. We stress that besides optional authentication during registration, our modeling (and TLS-OPAQUE in particular) is "password-only" where the user is not assumed to carry any information other than the password.

6.2 A UC version of TLS-OPAQUE

We now give a modular representation of TLS-OPAQUE in Figure 12, called $\Pi_{\mathsf{TLS-OPAQUE}}$, which allows for asymmetric password authentication on an unauthenticated channel. $\Pi_{\mathsf{TLS-OPAQUE}}$ is a UC protocol where parties issue calls to one instance of each functionality $\mathcal{F}_{\mathsf{PHA}}$ for PHA, and $\mathcal{F}_{\mathsf{OPRF}}$ for an oblivious pseudorandom function (OPRF), (see the full version [15] for details on OPRFs).

In a nutshell, parties use the OPRF to turn their passwords into an application key rw. During registration, rw is used to generate a key pair at $\mathcal{F}_{\mathsf{PHA}}$, of which the server stores the public key. To authenticate, a client then recomputes rw from pw, then uses rw to recover $(\mathsf{pk}, \mathsf{sk})$ from $\mathcal{F}_{\mathsf{PHA}}$, and subsequently authenticates to the server using sk and the public-key authentication interface of $\mathcal{F}_{\mathsf{PHA}}$. The flow is depicted in Figure 11. It is important to note that, due to our modular modeling, the client never actually sees the key pair $(\mathsf{pk}, \mathsf{sk})$, because $\mathcal{F}_{\mathsf{PHA}}$ never gives out actual keys. However, $\mathcal{F}_{\mathsf{PHA}}$ has the option to bootstrap key generation from any string, in our case the PRF value rw. Since the client can recover rw from only a password, it can, during authentication, re-claim the key pair at $\mathcal{F}_{\mathsf{PHA}}$ from only pw. This concept of *transportable keys* in $\mathcal{F}_{\mathsf{PHA}}$ was explained in more detail in the previous section, and it is the central tool that allows us to abstract the public-key building block of TLS-OPAQUE, i.e., write TLS-OPAQUE modularly with calls to $\mathcal{F}_{\mathsf{PHA}}$.

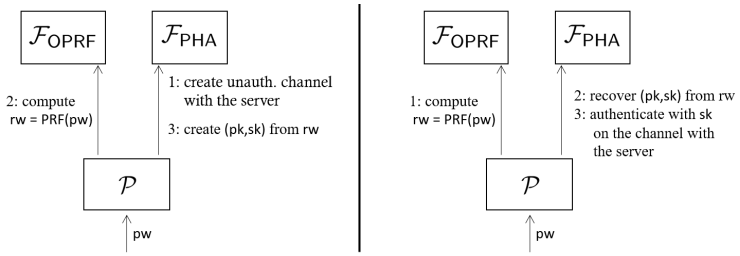

Fig. 11: Flows of TLS-OPAQUE using calls to hybrid functionalities $\mathcal{F}_{\mathsf{OPRF}}$ and $\mathcal{F}_{\mathsf{PHA}}$. **Left:** registration and channel establishment. **Right:** password authentication of the unautenticated channel. Note that the key pair $(\mathsf{pk}, \mathsf{sk})$ is implicit in $\mathcal{F}_{\mathsf{PHA}}$ and is never seen by $\mathcal{P}$. It is generated and can be re-claimed from the application key rw.

$\Pi_{\mathsf{TLS-OPAQUE}}$ consists of three phases: registration, channel establishment and asymmetric password authentication.

Registration: If client C with username uid_C and password pw_C wants to register with server S, then C initiates by sending uid_C to S. S then creates a (normal) public key pk_S at $\mathcal{F}_{\mathsf{PHA}}$ and sends it to C. Both parties engage in an OPRF protocol, where S plays the server role on random key K and C evaluates $\mathsf{rw} = PRF_{K,S||\mathsf{uid}}(\mathsf{pw}_C)$. Finally C then generates a transportable key at $\mathcal{F}_{\mathsf{PHA}}$ with application key $\mathsf{ak} = \mathsf{rw}$ and $\mathsf{aux} = \mathsf{pk}_S$, receiving back $\mathsf{ske}, \mathsf{pk}_C$, C sends $\mathsf{ske}, \mathsf{pk}_C$ to S and erases her memory, and S stores $(\mathsf{uid}_C, \mathsf{ske}, \mathsf{pk}_S, \mathsf{pk}_C)$ as the password file.

Channel Establishment: C and S establish an unauthenticated channel as in Figure 9. If establishment goes unattacked, the channel is established between C and S, but both parties are oblivious of whether they actually got connected to the intended process. From their point of view, they might be connected to the adversary, or to a different honest process.

Password Authentication: In order to establish some knowledge about the counterparty in their channel, a party can initiate a password authentication. In our example, C initiates such authentication on his channel cid_C, with username uid_C and password pw_C. On a high level, both C and S will now each perform one public-key authentication, where S uses pk_S stored in the password file, and C uses key material contained in the envelope ske that S piggy-backs to his own authentication flow using the ctx field of $\mathcal{F}_{\mathsf{PHA}}$'s interface AUTHSEND. To authenticate with public keys, both parties invoke $\mathcal{F}_{\mathsf{PHA}}$. S, using "normal" public key pk_S to authenticate, invokes it in `std` mode. C, who receives ske from S, recovers application key $\mathsf{rw} = \mathsf{PRF}_{K,S||\mathsf{uid}}(\mathsf{pw}_C)$ by engaging with S in one PRF evaluation of $\mathcal{F}_{\mathsf{OPRF}}$ with session identifier $\mathsf{sid} = S||\mathsf{uid}$. C then starts an authentication using transportable keys $\mathsf{rw}, \mathsf{ske}$. Both parties piggy-back the OPRF transcript values a', b' to their authentication flows using ctx fields of $\mathcal{F}_{\mathsf{PHA}}$. If C sees a successful authentication under public key pk_S, which C retrieves as auxiliary data from ske using $\mathcal{F}_{\mathsf{PHA}}$ interface GETAUXDATA, then C outputs success, else it outputs failure. If S sees a successful authentication under pk_C from the password file, C outputs success, else C outputs failure. Due to the guarantees of $\mathcal{F}_{\mathsf{PHA}}$, both parties can only output success if they are connected to each other, and if S has a password file that corresponds to pw_C entered by C.

Π_{EA} generalizes TLS-OPAQUE as specified in Section 2 in several aspects:

1. $\Pi_{\mathsf{TLS-OPAQUE}}$ abstracts from the exact secure channel with post-handshake public-key authentication and can provide post-handshake password authentication based on any protocol that securely instantiates $\mathcal{F}_{\mathsf{PHA}}$
2. $\Pi_{\mathsf{TLS-OPAQUE}}$ sends messages in the clear instead of sending them over the channel-to-authenticate
3. $\Pi_{\mathsf{TLS-OPAQUE}}$ abstracts from the underlying OPRF protocol and can be instantiated with any OPRF that securely realizes $\mathcal{F}_{\mathsf{OPRF}}$.

We are now ready to state the security of TLS-OPAQUE. We refer to the full version [15] for the definition of $\mathcal{F}_{\mathsf{OPRF}}$ and the full proof.

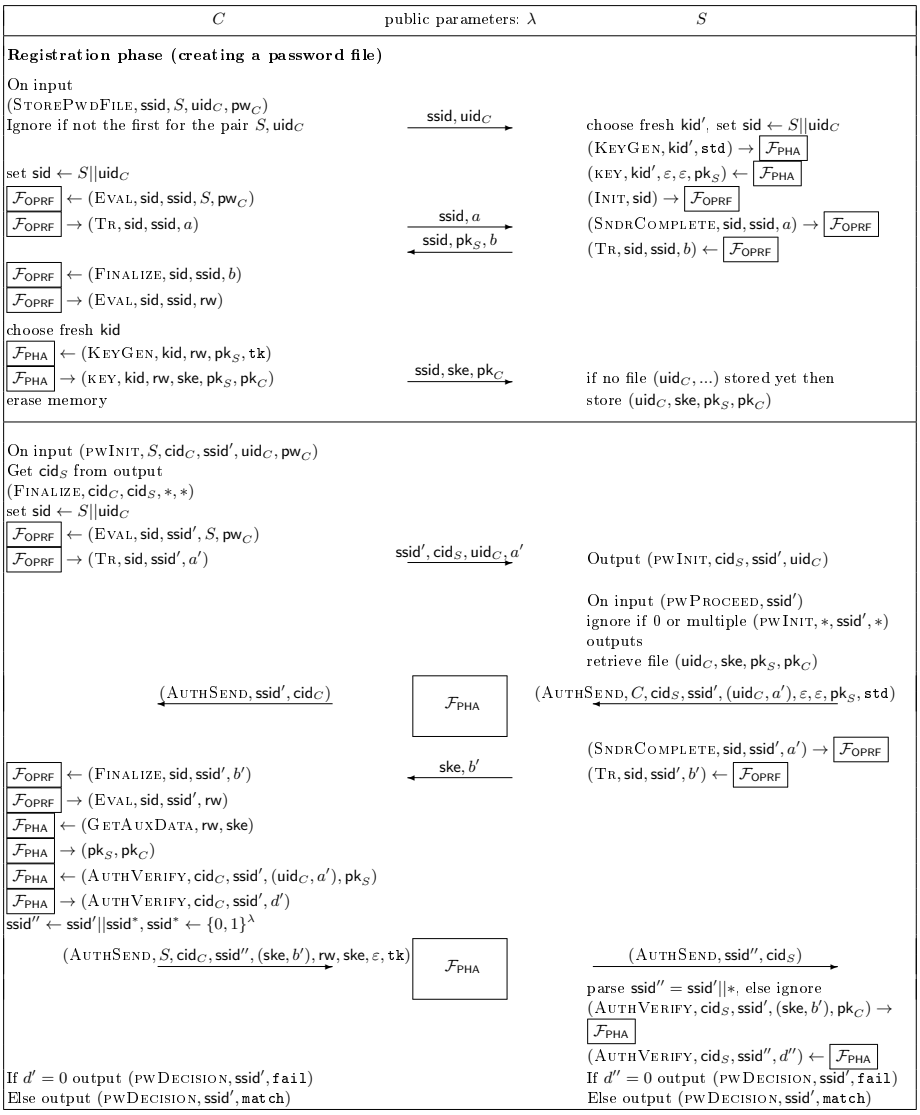

Fig. 12: Protocol $\Pi_{\mathsf{TLS-OPAQUE}}$, using channel and public-key authentication facilities provided by $\mathcal{F}_{\mathsf{PHA}}$. We exemplarily show C registering a password with S, and subsequent authentication of a channel between C, providing a cleartext password, and S, using the data stored at registration. ε denotes the empty string. For brevity we omit handling of NewSession, Send and ExpireSession inputs, which are simply relayed to $\mathcal{F}_{\mathsf{PHA}}$. We also omit the identifiers with which $\mathcal{F}_{\mathsf{PHA}}$ and $\mathcal{F}_{\mathsf{OPRF}}$ are called. An application can simply set those to be, e.g., `tls-opaque_pha` and `tls-opaque_oprf`.

Theorem 3 (Security of $\Pi_{\mathsf{TLS-OPAQUE}}$). *Protocol $\Pi_{\mathsf{TLS-OPAQUE}}$ (Figure 12) UC-emulates functionality $\mathcal{F}_{\mathsf{pwPHA}}$ in the $(\mathcal{F}_{\mathsf{PHA}}, \mathcal{F}_{\mathsf{OPRF}})$-hybrid model with respect to static malicious corruptions and adaptive server compromise.*

Corollary 2. *Protocol TLS-OPAQUE specified in Section 2 securely realizes $\mathcal{F}_{\mathsf{pwPHA}}$.*

The corollary follows from instantiating $\mathcal{F}_{\mathsf{PHA}}$ with Π_{EA} (Thm. 2) using the UC composition theorem [11], where in turn $\mathcal{F}_{\mathsf{cbSC}}$ is instantiated with the TLS 1.3 protocol snippet from Figure 7 (Thm. 1), and $\mathcal{F}_{\mathsf{OPRF}}$ instantiated with 2HashDH [15]

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. N. Asokan, Valtteri Niemi, and Kaisa Nyberg. Man-in-the-middle in tunnelled authentication protocols. In *Security Protocols, 11th International Workshop, Cambridge, UK*, volume 3364 of *LNCS*, pages 28–41. Springer, 2003.
4. Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE Symposium on Security and Privacy*, pages 483–502. IEEE Computer Society, 2017.
5. Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE Symposium on Security and Privacy*, pages 98–113. IEEE Computer Society, 2014.
6. Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Alfredo Pironti. Verified contributive channel bindings for compound authentication. In *NDSS*, 2015.
7. Karthikeyan Bhargavan and Gaëtan Leurent. Transcript collision attacks: Breaking authentication in tls, ike and ssh. In *23rd Annual Network and Distributed System Security Symposium, NDSS*, 2016.
8. Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *IEEE Computer Security Foundations Workshop CSFW-14*, pages 82–96. IEEE Computer Society, 2001.
9. 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.
10. Chris Brzuska and Håkon Jacobsen. A Modular Security Analysis of EAP and IEEE 802.11. In *PKC'2017. Cryptology ePrint Archive, Paper 2017/253*, 2017.
11. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *IEEE Symposium on Foundations of Computer Science – FOCS 2001*, pages 136–145. IEEE, 2001.
12. Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In *EUROCRYPT*, pages 337–351. Springer, 2002.

13. Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. Automated analysis and verification of TLS 1.3: 0-rtt, resumption and delayed authentication. In *IEEE Symposium on Security and Privacy*, pages 470–485. IEEE Computer Society, 2016.
14. Craig Gentry, Philip MacKenzie, and Zulfikar Ramzan. A method for making password-based key exchange resilient to server compromise. In *Advances in Cryptology – CRYPTO 2006*, pages 142–159. Springer, 2006.
15. Julia Hesse, Stanislaw Jarecki, and Hugo Krawczyk. Password-authenticated tls via opaque and post-handshake authentication. Cryptology ePrint Archive, Report 2023/220, 2023. `https://ia.cr/2023/220`.
16. J. Hodges, J. C. Jones, M. B. Jones, A. Kumar, and E. Lundberg. Web Authentication: An API for accessing Public Key Credentials Level 2, `https://www.w3.org/TR/webauthn-2/`, August 2021.
17. Jonathan Hoyland. *An analysis of TLS 1.3 and its use in composite protocols*. PhD thesis, RHUL, Egham, UK, 2018.
18. Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In *EUROCRYPT*, pages 456–486. Springer, 2018.
19. H. Krawczyk. The OPAQUE Asymmetric PAKE Protocol, draft-krawczyk-cfrg-opaque-06, `https://www.ietf.org/archive/id/draft-krawczyk-cfrg-opaque-06.txt`, June 2020.
20. Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *CRYPTO*, pages 631–648, 2010.
21. Hugo Krawczyk. Unilateral-to-mutual authentication compiler for key exchange (with applications to client authentication in tls 1.3). In *ACM CCS 2016*, 2016.
22. Marsh Ray and S Dispensa. Authentication gap in tls renegotiation, 2009.
23. E. Rescorla. The transport layer security (TLS) protocol version 1.3, rfc 8446, August 2018. `http://www.rfc-editor.org/rfc/rfc8446.txt`.
24. Martin Rex. Mitm attack on delayed tls-client auth through renegotiation, November 2009.
25. Joseph Salowey and Eric Rescorla. Tls renegotiation vulnerability, 2009.
26. N. Sullivan. Exported Authenticators in TLS, RFC 9261, `https://datatracker.ietf.org/doc/html/rfc9261`, July 2022.
27. 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.