Password-Authenticated TLS via OPAQUE and Post-Handshake Authentication

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

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

February 17, 2023

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.

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

Contents

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. See Appendix A for a more detailed discussion.

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 [19, 12]. 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[1] [29] 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 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)* [32] (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 [33].

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 [13] with two authentication flavors: via public-key certificates as the EA protocol [32] 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

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

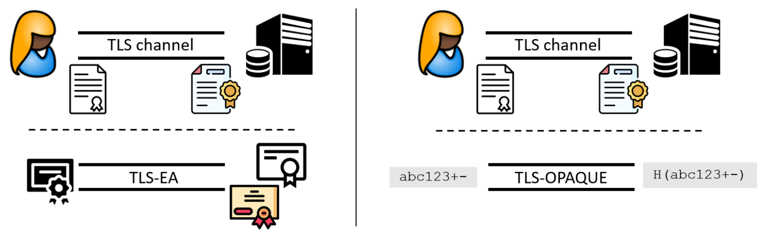


Figure 1: (Post-)Authentication options for TLS channels. **Left:** The Exported Authenticators TLS extention (TLS-EA) allows both channel endpoints to subsequently add more public-key identities to a TLS channel. **Right:** TLS-OPAQUE allows to subsequently add (asymmetric) password identities to a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

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 [28, 30, 31] and triple-handshake attacks [8]. See [9] 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 [14] 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 [4, 9], the source of many of the serious attacks against PHA mechanisms in practice. We formally prove in Theorem 4.1 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[2] by the creators of a TLS channel, and only by those. Therefore formally proving the security of the TLS-EA protocol from [32] reduces to showing that the protocol realizes the PK-based PHA functionality. This is shown in Theorem 5.1 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 [21, 33] that uses the TLS-EA mechanism to transport the OPAQUE messages for providing *password-based* post-handshake authentication to the TLS channel.

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

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, 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 6.1 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.

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 [29, 32, 33] 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 9), and prove in Theorem 4.1 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 11), and present a modular version of TLS-EA (Π_{EA} in Figure 12) that uses secure channels with binders (i.e., $\mathcal{F}_{\mathsf{cbSC}}$) as an abstract building block. Theorem 5.1 proves that this modular version of TLS-EA implements $\mathcal{F}_{\mathsf{PHA}}$. Invoking the UC composition theorem on Theorems 4.1 and 5.1 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 13), and present a modular version of TLS-OPAQUE ($\Pi_{\mathsf{TLS-OPAQUE}}$ in Figure 14) that uses secure channels with post-handshake public-key authentication (i.e., $\mathcal{F}_{\mathsf{PHA}}$) as an abstract building block. Theorem 6.1 proves that this modular version of TLS-OPAQUE implements $\mathcal{F}_{\mathsf{pwPHA}}$. Invoking again the UC composition theorem on Theorems 5.1 and 6.1 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 supplementary material of this paper provides previously known security notions for signatures and MACs, as well as details on Oblivious Pseudorandom Functions (OPRFs) (all in Appendix 3, a detailed walk-through of functionality $\mathcal{F}_{\mathsf{cbSC}}$ (Appendix 4.1), considerations for implementing, deploying, operating, and using TLS-OPAQUE in a variety of use cases (Appendix A), and full proofs and sketches of all our Theorems (Appendix B).

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 [24] 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 [9]. 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 [11]. Extending this work, [18] presents an automated analysis of the Exported Authenticators (TLS-EA) protocol [32] 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 [4, 31, 10, 15, 6].

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 [19, 12] in schematic form in Figure 2 (more details are included in the presentation of TLS-OPAQUE in Fig. 5).

Fig. 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.

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

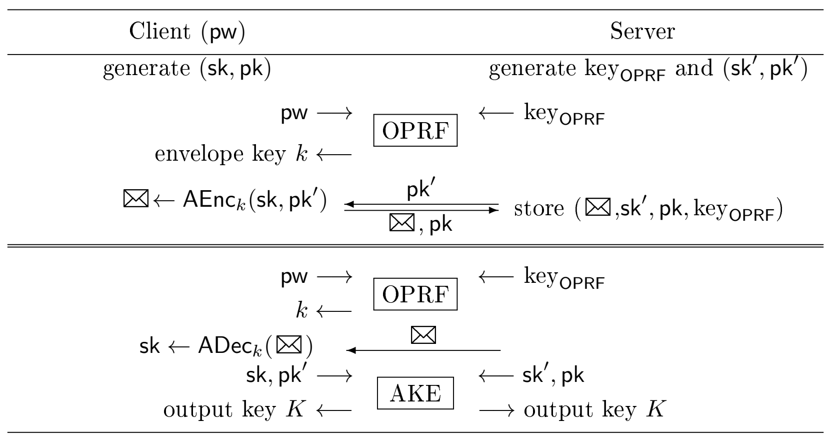


Figure 2: OPAQUE registration (top) and key exchange (bottom).

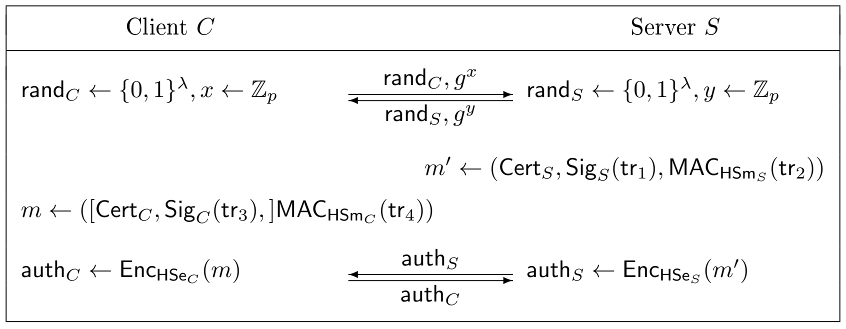

Client C
Server S
$\mathsf{rand}_C \leftarrow \{0,1\}^\lambda, x \leftarrow \mathbb{Z}_p$
rand_C, g^x
rand_S, g^y
$\mathsf{rand}_S \leftarrow \{0,1\}^\lambda, y \leftarrow \mathbb{Z}_p$
$m' \leftarrow (\mathsf{Cert}_S, \mathsf{Sig}_S(\mathsf{tr}_1), \mathsf{MAC}_{\mathsf{HSm}_S}(\mathsf{tr}_2))$
$m \leftarrow ([\mathsf{Cert}_C, \mathsf{Sig}_C(\mathsf{tr}_3),]\mathsf{MAC}_{\mathsf{HSm}_C}(\mathsf{tr}_4))$
$\mathsf{auth}_C \leftarrow \mathsf{Enc}_{\mathsf{HSe}_C}(m)$
auth_S
auth_C
$\mathsf{auth}_S \leftarrow \mathsf{Enc}_{\mathsf{HSe}_S}(m')$

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

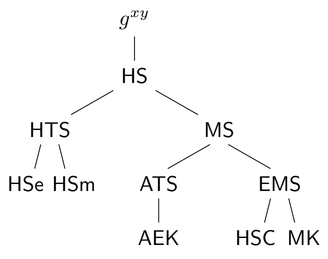
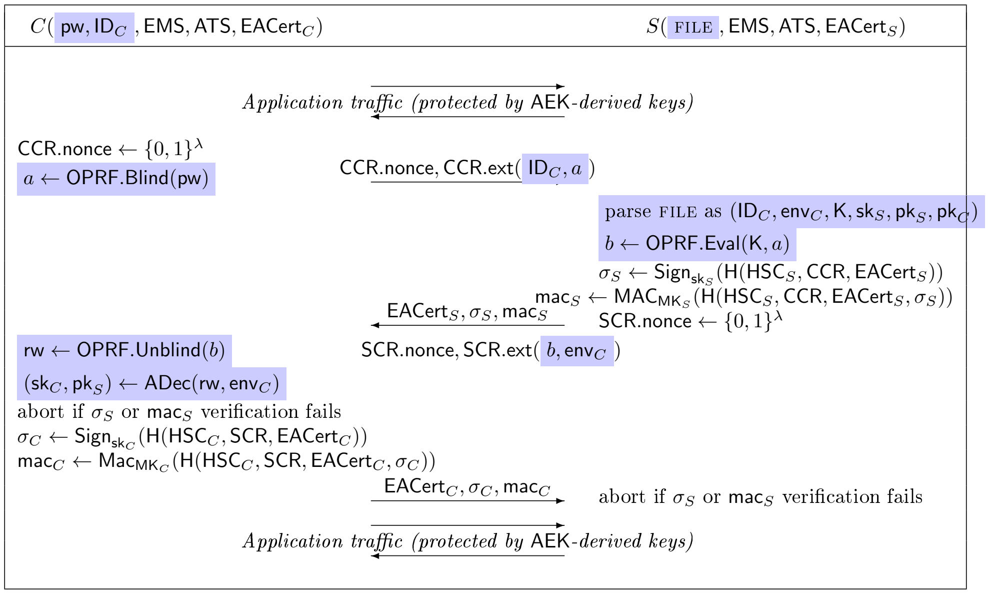


Figure 4: Key Derivation in TLS 1.3



$C(\mathsf{pw}, \mathsf{ID}_C, \mathsf{EMS}, \mathsf{ATS}, \mathsf{EACert}_C)$
$S(\textsc{file}, \mathsf{EMS}, \mathsf{ATS}, \mathsf{EACert}_S)$
Application traffic (protected by AEK*-derived keys)*
$\mathsf{CCR.nonce} \leftarrow \{0,1\}^\lambda$
$a \leftarrow \mathsf{OPRF.Blind}(\mathsf{pw})$
$\mathsf{CCR.nonce}, \mathsf{CCR.ext}(\mathsf{ID}_C, a)$
parse $\textsc{file}$ as $(\mathsf{ID}_C, \mathsf{env}_C, \mathsf{K}, \mathsf{sk}_S, \mathsf{pk}_S, \mathsf{pk}_C)$
$b \leftarrow \mathsf{OPRF.Eval}(\mathsf{K}, a)$
$\sigma_S \leftarrow \mathsf{Sign}_{\mathsf{sk}_S}(\mathsf{H}(\mathsf{HSC}_S, \mathsf{CCR}, \mathsf{EACert}_S))$
$\mathsf{mac}_S \leftarrow \mathsf{MAC}_{\mathsf{MK}_S}(\mathsf{H}(\mathsf{HSC}_S, \mathsf{CCR}, \mathsf{EACert}_S, \sigma_S))$
$\mathsf{SCR.nonce} \leftarrow \{0,1\}^\lambda$
$\mathsf{EACert}_S, \sigma_S, \mathsf{mac}_S$
$\mathsf{SCR.nonce}, \mathsf{SCR.ext}(b, \mathsf{env}_C)$
$\mathsf{rw} \leftarrow \mathsf{OPRF.Unblind}(b)$
$(\mathsf{sk}_C, \mathsf{pk}_S) \leftarrow \mathsf{ADec}(\mathsf{rw}, \mathsf{env}_C)$
abort if σ_S or mac_S verification fails
$\sigma_C \leftarrow \mathsf{Sign}_{\mathsf{sk}_C}(\mathsf{H}(\mathsf{HSC}_C, \mathsf{SCR}, \mathsf{EACert}_C))$
$\mathsf{mac}_C \leftarrow \mathsf{Mac}_{\mathsf{MK}_C}(\mathsf{H}(\mathsf{HSC}_C, \mathsf{SCR}, \mathsf{EACert}_C, \sigma_C))$
$\mathsf{EACert}_C, \sigma_C, \mathsf{mac}_C$
abort if σ_S or mac_S verification fails
Application traffic (protected by AEK*-derived keys)*

Figure 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.

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.[3] 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 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.

Fig. 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 [32], the first one authenticating the

[3] 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.

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) 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 descendent 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 [23]. 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[4]. 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 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

[4]From [32]: "The application MAY use the existing TLS connection to transport the authenticator." The use of MAY makes this protected transport optional.

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.

Definition 3.1 (Digital signature scheme). *A digital signature scheme* $\mathsf{SIG} = (\mathsf{KG}, \mathsf{Sign}, \mathsf{Vfy})$ *for message space* $\mathcal{M}$ *consists of the following three probabilistic polynomial-time algorithms.*

- ***Key generation:*** *Algorithm* KG *takes as input the security parameter* 1^λ. *It outputs a key pair* $(\mathsf{pk}, \mathsf{sk})$.
- ***Signing:*** *Algorithm* Sign *takes as input a secret key* sk *and a message* $m \in \mathcal{M}$ *and outputs a signature* σ. *We use notation* $\sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$.
- ***Signature verification:*** *Algorithm* Vfy *takes as input a verification key* pk, *signature* σ *and message* m *and outputs* 0 *(reject) or* 1 *(accept). We use notation* $\mathsf{Vfy}_{\mathsf{pk}}(\sigma, m)$.

We say that a signature scheme $\mathsf{SIG} = (\mathsf{KG}, \mathsf{Sign}, \mathsf{Vfy})$ for message space $\mathcal{M}$ is *perfectly complete* if for all $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KG}(1^\lambda)$ and for all $m \in \mathcal{M}$ it holds that $\mathsf{Vfy}_{\mathsf{pk}}(\mathsf{Sign}_{\mathsf{sk}}(m), m) = 1$. We sometimes assume existence of an efficiently computable algorithm PKGen which, on input sk outputs pk for all key pairs $(\mathsf{pk}, \mathsf{sk})$ in the range of KG.

Definition 3.2 (EUF-CMA Security, for signature schemes). *The advantage of an adversary against EUF-CMA security (existential unforgeability under adaptive chosen message attack) of a signature scheme* SIG *is defined as*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{euf-cma}}_{\mathcal{A},\mathsf{SIG}}(\lambda) := \Pr[&\mathsf{Vfy}_{\mathsf{pk}}(\sigma^*, m^*) = 1 | \\ &(\sigma^*, m^*) \leftarrow \mathcal{A}^{\mathsf{Sign}_{\mathsf{sk}}(\cdot)}(\mathsf{pk})],\end{aligned}$$

where $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KG}(1^\lambda)$, *and* m^* *is fresh in the sense that it has never been queried to the signing oracle* $\mathsf{Sign}_{\mathsf{sk}}(\cdot)$.

Definition 3.3 (Message authentication code (MAC)). *A message authentication code* $\mathsf{MAC} = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vfy})$ *consists of the following three probabilistic polynomial-time algorithms.*

- ***Key generation:*** *Algorithm* Gen *takes as input the security parameter* 1^λ *and outputs a key* k *with* $|k| \geq n$.
- ***Mac generation:*** *Algorithm* Mac *takes as input a key* k *and a message* $m \in \{0,1\}^*$, *and outputs a mac* mac. *We use notation* $\mathsf{mac} \leftarrow \mathsf{Mac}_k(m)$.
- ***Mac verification:*** *Algorithm* Vfy *takes as inputs key* k, *mac* mac *and message* m, *and outputs* 0 *(reject) or* 1 *(accept). We use notation* $\mathsf{Vfy}_k(\mathsf{mac}, m)$=1.

We say that a mac scheme $\mathsf{MAC} = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vfy})$ is *perfectly complete* if for all $k \leftarrow \mathsf{Gen}(1^\lambda)$ and for all $m \in \{0,1\}^*$ it holds that $\mathsf{Vfy}_k(\mathsf{Mac}(m), m) = 1$. For deterministic MAC schemes, note that $\mathsf{Vfy}_k(\cdot,\cdot)$ can be implemented by recomputing the MAC and checking equality. We next recall the standard security definition for a MAC, augmented with a verification oracle [5].

Definition 3.4 (EUF-CMA Security, for MACs). *The advantage of an adversary against EUF-CMA security (existential unforgeability under adaptive chosen message attack) of a MAC* MAC *is defined as*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{euf-cma}}_{\mathcal{A},\mathsf{MAC}}(\lambda) := \Pr[&\mathsf{Vfy}_k(\mathsf{mac}^*, m^*) = 1 | \\ (\mathsf{mac}^*, m^*) \leftarrow \mathcal{A}^{\mathsf{Mac}_k(\cdot),\mathsf{Vfy}_k(\cdot,\cdot)}(1^\lambda)&],\end{aligned}$$

where $k \leftarrow \mathsf{Gen}(1^\lambda)$, *and* m^* *is fresh in the sense that it has never been queried to the oracle* $\mathsf{Mac}_k(\cdot)$.

We next define authenticated encryption and properties thereof that we use in this paper following [20].

Definition 3.5 (CUF-CCA security)**.** *We say that $\mathcal{E}$ is CUF-CCA secure if it is both CCA-secure and CUF-secure. The notion of CCA-security of encryption is well-known, while advantage of an adversary against CUF security (ciphertext unforgeability) [22] of a symmetric encryption scheme $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is defined as*

$$\mathsf{Adv}^{\mathsf{cuf}}_{\mathcal{A},\mathcal{E}}(\lambda) := \Pr[\mathsf{Dec}_k(c) \neq \perp | c \leftarrow \mathcal{A}^{\mathsf{Enc}_k(\cdot)}(1^\lambda)],$$

where $k \leftarrow \mathsf{Gen}(1^\lambda)$, and c is fresh in the sense that it has never been output by oracle $\mathsf{Enc}_k(\cdot)$.

We next recall a variant of ciphertext unforgeability that was identified to be sufficient for proving security of the OPAQUE protocol [19]. In a nutshell, the property says that an adversary cannot come up with a ciphertext that successfully decrypts under two randomly chosen keys, even when knowing these keys.

Definition 3.6 (RKR security)**.** *The advantage of an adversary against RKR security (random-key robustness) of a symmetric encryption scheme $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is defined as*

$$\mathsf{Adv}^{\mathsf{rkr}}_{\mathcal{A},\mathcal{E}}(\lambda) := \Pr[\mathsf{Dec}_k(c) \neq \perp \wedge \mathsf{Dec}_{k'}(c) \neq \perp | c \leftarrow \mathcal{A}(1^\lambda, k, k')],$$

where $k, k' \leftarrow \mathsf{Gen}(1^\lambda)$.

Lastly, we define encryption equivocability, which requires existence of a simulator which can produce indistinguishable ciphertexts without knowing the message, and is able to equivoke these ciphertexts afterwards: when learning the message, the simulator can come up with a key that makes the ciphertext decrypt to that message. More formally:

Definition 3.7 (Equivocable encryption)**.** *A symmetric encryption scheme $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is called equivocable if there exists an efficient stateful simulator $\mathsf{SIM}_{\mathsf{eq}}$, such that for all efficient stateful adversaries $\mathcal{A}$ the following function is negligible in λ:*

$$\mathsf{Adv}^{\mathsf{eq}}_{\mathcal{A},\mathcal{E}}(\lambda) := \Pr[b = b^* | b^* \leftarrow \mathcal{A}(1^\lambda, c, k)] - \frac{1}{2},$$

where $b \leftarrow \{0, 1\}$, and

$$k \leftarrow \mathsf{Gen}(1^\lambda), c \leftarrow \mathsf{Enc}_k(m) \text{ if } b = 0$$
$$k \leftarrow \mathsf{SIM}_{\mathsf{eq}}(m), c \leftarrow \mathsf{SIM}_{\mathsf{eq}}(|m|) \text{ if } b = 1$$
$$m \leftarrow \mathcal{A}(1^\lambda).$$

We refer the reader to [19] for instantiations of encryption scheme which is CUF-CCA-secure, RKR-secure, *and* equivocable. On example is Encrypt-then-HMAC with a stream cipher encryption, e.g., counter mode, where the the stream cipher is modeled as a random oracle, see [19] for details.

Definition 3.8 (Gap CDH and Gap Square-DH)**.** *Let $\langle g \rangle$ be a cyclic group of order p. Let DDH_g be a DDH oracle, i.e. a function $\mathsf{DDH}_g(g^a, g^b, g^c)$ outputs 1 if $c = a \cdot b \bmod p$ and 0 otherwise. We define an advantage of adversary $\mathcal{A}$ against the Gap CDH and Gap Square-DH assumption on group $\langle g \rangle$ respectively as follows:*

$$\mathsf{Adv}^{\mathsf{G-CDH}}_{\mathcal{A}}(\langle g \rangle) := \Pr[g^{x \cdot y} \leftarrow \mathcal{A}^{\mathsf{DDH}_g}(g, g^x, g^y) \,|\, x \leftarrow \mathbb{Z}_p, y \leftarrow \mathbb{Z}_p]$$

$$\mathsf{Adv}^{\mathsf{G-SqDH}}_{\mathcal{A}}(\langle g \rangle) := \Pr[g^{x^2} \leftarrow \mathcal{A}^{\mathsf{DDH}_g}(g, g^x) \,|\, x \leftarrow \mathbb{Z}_p]$$

We say that Gap CDH (resp. Gap Square-DH) holds in the group if for all efficient attackers Adv the respective advantage above is a negligible function of the group description size $|\langle g \rangle|$. It is well-known that the CDH and Square-DH assumptions are equivalent, and the same holds for their Gap versions.

The functionality $\mathcal{F}_{\mathsf{RO}}$ proceeds as follows, running on security parameter λ, session identifier sid, and talking to arbitrary parties and adversary $\mathcal{A}$:

- $\mathcal{F}_{\mathsf{RO}}$ keeps a list T_{H} (which is initially empty) of pairs of bit strings.
- Upon receiving a value (sid, m) (with $m \in \{0,1\}^*$) from any entity, do:
  - If there is a pair $(m, \tilde{h}) \in T_{\mathsf{H}}$, set $h := \tilde{h}$.
  - If there is no such pair, choose uniformly $h \in \{0,1\}^\lambda$ and store the pair $(m, h) \in T_{\mathsf{H}}$.

  Once h is set, reply to the caller with (sid, h).

Figure 6: Ideal functionality $\mathcal{F}_{\mathsf{RO}}$

3.1 UC functionalities

Random oracle functionality. We recall the Random Oracle (RO) functionality $\mathcal{F}_{\mathsf{RO}}$ in Figure 6. Some of our theorems will be proven modeling some hash functions as functionalities $\mathcal{F}_{\mathsf{RO}}$, which means that they are proven *in the programmable random oracle model.*

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 [19]. We use a UC formalization of OPRFs by Jarecki et al. [19], modified regarding its output of transcripts which we now describe on a high level. The OPRF functionality of [19] 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 [19]. 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 [19]), but also to perform server-side operations in the online protocol instances, via interface $\textsc{SndrComplete}$. Our functionality $\mathcal{F}_{\mathsf{OPRF}}$ is shown in Figure 7

We now argue that a slight variation of the 2HashDH OPRF protocol depicted in Figure 8 UC-emulates our functionality $\mathcal{F}_{\mathsf{OPRF}}$. The protocol is modified from [19] by letting the server output b as postfix, and leaving prefix and postfix transmission up to the application. We specify how to obtain a simulator $\mathcal{S}$ from simulation $\mathcal{S}'$ of [19]. $\mathcal{S}$ is equal to $\mathcal{S}'$ with the following two modifications:

1. Upon $(\textsc{SndrComplete}, \mathsf{ssid}, \mathsf{prfx})$ from $\mathcal{F}_{\mathsf{OPRF}}$, simulator $\mathcal{S}$ sets $\mathsf{pstfx} \leftarrow b$ for $b \leftarrow \mathcal{S}'$ and replies with pstfx.

2. When $\mathcal{S}'$ receives a message (ssid, a) from $\mathcal{A}$ to S, $\mathcal{S}$ instead obtains this message from $(\textsc{SndrComplete}, \mathsf{ssid}, a, S)$ of $\mathcal{F}_{\mathsf{OPRF}}$. Likewise, whenever $\mathcal{S}'$ receives message (ssid, b) from $\mathcal{A}$ to $\mathcal{P}$, simulator $\mathcal{S}$ obtains the payload of this message from $(\textsc{Finalize}, \mathsf{ssid}, b, \mathcal{P})$.

```
$\mathcal{F}_{\mathsf{OPRF}}$ takes inputs from arbitrary "server" parties $\mathcal{P}_S$, from arbitrary "evaluator" (or "receiver") parties $\mathcal{P}_R$, and from the adversary $\mathcal{A}$.
Public Parameters: PRF output-length $\ell$, polynomial in security parameter $\lambda$. For every $i, x$, value $F_{\mathsf{sid},i}(x)$ is initially undefined, and if undefined value $F_{\mathsf{sid},i}(x)$ is referenced then $\mathcal{F}_{\mathsf{OPRF}}$ assigns $F_{\mathsf{sid},i}(x) \leftarrow \{0,1\}^\ell$.

Initialization and Compromise
On message $(\textsc{Init}, \mathsf{sid})$ from party $\mathcal{P}_S$:
    // A server can initialize at most once.
  [I.1] If this is the first $\textsc{Init}$ message for $\mathsf{sid}$, set $\mathsf{tx} = 0$ and send $(\textsc{Init}, \mathsf{sid}, \mathcal{P}_S)$ to $\mathcal{A}$;
  [I.2] Otherwise drop the query.
From now on use tag "$\mathcal{P}_S$" to denote the unique entity which sent the $\textsc{Init}$ message for the session identifier $\mathsf{sid}$.

On $(\textsc{Compromise}, \mathcal{P}_S)$ from $\mathcal{A}$ (requires permission from $\mathcal{Z}$):
  [C.1] Declare server $\mathcal{P}_S$ as $\texttt{compromised}$.
(If $\mathcal{P}_S$ is corrupted then it is declared $\texttt{compromised}$ from the beginning.)

Offline Evaluation
On $(\textsc{OfflineEval}, \mathsf{sid}, i, x)$ from $\mathcal{P} \in \{\mathcal{P}_S, \mathcal{A}\}$:
  [O.1] Send $(\textsc{OfflineEval}, \mathsf{sid}, i, x, F_{\mathsf{sid},i}(x))$ to $\mathcal{P}$ if any of the following hold:
        [O.1.1] $\mathcal{P}_S$ is corrupted,
        [O.1.2] $\mathcal{P} = \mathcal{P}_S$ and $i = \mathcal{P}_S$,
        [O.1.3] $\mathcal{P} = \mathcal{A}$ and $i \neq \mathcal{P}_S$,
        [O.1.4] $\mathcal{P} = \mathcal{A}$ and $\mathcal{P}_S$ is marked $\texttt{compromised}$.

Online Evaluation
On $(\mathsf{Eval}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}'_S, x)$ from $\mathcal{P} \in \{\mathcal{P}_R, \mathcal{A}\}$:
  [E.1] Send $(\mathsf{Eval}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}, \mathcal{P}'_S)$ to $\mathcal{A}$ and receive back $\mathsf{prfx}$;
  [E.2] If $\mathsf{prfx}$ was used before drop the query;
  [E.3] Record $\langle \mathsf{ssid}, \mathcal{P}, x, \mathsf{prfx}, \bot \rangle$ and send $(\textsc{Tr}, \mathsf{sid}, \mathsf{ssid}, \mathsf{prfx})$ to $\mathcal{P}$.

On $(\textsc{SndrComplete}, \mathsf{sid}, \mathsf{ssid}', \mathsf{prfx})$ from $\mathcal{P}' \in \{\mathcal{P}_S, \mathcal{A}\}$:
  [SC.1] If $\mathcal{P}' = \mathcal{A}$ and $\mathcal{P}_S$ is not marked $\texttt{compromised}$ then drop the query;
  [SC.2] Send $(\textsc{SndrComplete}, \mathsf{sid}, \mathsf{ssid}', \mathsf{prfx}, \mathcal{P}')$ to $\mathcal{A}$ and receive back $\mathsf{pstfx}$;
  [SC.3] If $\mathsf{pstfx}$ was used before drop the query;
  [SC.4] Send $(\textsc{Tr}, \mathsf{sid}, \mathsf{ssid}', \mathsf{pstfx})$ to $\mathcal{P}'$;
  [SC.5] If $\exists$ record $\langle \mathsf{ssid}, \mathcal{P}, x, \mathsf{prfx}, \bot \rangle$ for $\mathcal{P} \neq \mathcal{A}$, change it to $\langle \mathsf{ssid}, \mathcal{P}, x, \texttt{trOK}, \mathsf{pstfx} \rangle$, else set $\mathsf{tx}{+}{+}$.

On $(\textsc{Finalize}, \mathsf{sid}, \mathsf{ssid}, \mathsf{pstfx})$ from $\mathcal{P} \in \{\mathcal{P}_R, \mathcal{A}\}$:
    // Invariant: $\mathsf{pstfx} = \texttt{trOK}$ only if $\mathsf{prfx} = \texttt{trOK}$, i.e. if $\mathcal{P}_R$'s session was passively-connected (HbC) to some $\mathcal{P}_S$'s session.
  [F.1] If $\exists$ record $\langle \mathsf{ssid}, \mathcal{P}, x, \texttt{trOK}, \mathsf{pstfx} \rangle$, change it to $\langle \mathsf{ssid}, \mathcal{P}, x, \texttt{trOK}, \texttt{trOK} \rangle$.
  [F.2] Send $(\textsc{Finalize}, \mathsf{sid}, \mathsf{ssid}, \mathsf{pstfx}, \mathcal{P})$ to $\mathcal{A}$.

On $(\textsc{RcvComplete}, \mathsf{sid}, \mathsf{ssid}, \mathcal{P}, i)$ from $\mathcal{A}$
  [RC.1] If there is no record $\left\langle \mathsf{ssid}, \mathcal{P}, x, \mathsf{prfx}, \mathsf{pstfx} \right\rangle$ ignore the query;
    // Ensure correct (=w.r.t $\mathcal{P}_S$) evaluation if prefixes and postfixes match:
  [RC.2] If ($i \neq \mathcal{P}_S$ and $\mathsf{pstfx} = \texttt{trOK}$) ignore the query;
    // Server's function evaluated only on prefix-matching sessions or if there is a ticket:
  [RC.3] If ($i = \mathcal{P}_S$, $\mathsf{prfx} \neq \texttt{trOK}$, and $\mathsf{tx} = 0$) ignore the query;
    // Server's function evaluation on non-HbC session takes a ticket:
  [RC.4] If ($i = \mathcal{P}_S$ and $\mathsf{prfx} \neq \texttt{trOK}$) set $\mathsf{tx}{-}{-}$;
  [RC.5] Send $(\mathsf{Eval}, \mathsf{sid}, \mathsf{ssid}, F_{\mathsf{sid},i}(x))$ to $\mathcal{P}$;
```

Figure 7: Functionality $\mathcal{F}_{\mathsf{OPRF}}$ with adaptive server compromise, transcript prefix prfx [19] and (newly introduced) transcript postfix pstfx. For notation brevity, we omit the overall session identifier of $\mathcal{F}_{\mathsf{OPRF}}$ and use sid for (long-lived) function identifiers and ssid for (short-lived) identifiers of evaluation instances. We refer the reader to [19] for a full explanation of the non-gray parts of the functionality.

Parameters: H hashes to $\{0,1\}^\lambda$, H' hashes to a group of order q. We omit session identifiers sid from all inputs and outputs.

Evaluator $\mathcal{P}$	Server S
	On input $(\text{Init}, \mathsf{sid})$
On input $(\mathsf{Eval}, \mathsf{ssid}, S, x)$	$k \leftarrow \mathbb{Z}_q$
$r \leftarrow \mathbb{Z}_q$, $a \leftarrow \mathsf{H}'(x)^r$	
output $(\text{Tr}, \mathsf{ssid}, a)$	On input
	$(\text{SndrComplete}, \mathsf{ssid}, a)$
	$b \leftarrow a^k$
On $(\text{Finalize}, \mathsf{ssid}, b)$	output $(\text{Tr}, \mathsf{ssid}, b)$
else output $(\mathsf{Eval}, \mathsf{ssid}, \mathsf{H}(x, b^{1/r}))$	

Figure 8: Protocol 2HashDH which UC-realizes $\mathcal{F}_{\mathsf{OPRF}}$. Since it includes only two messages, (ssid, a) and (ssid, b), prefix and postfix can be used for message transport instead of direct communication.

It is straightforward to extend the argument in [19] to verify that the protocol from Figure 8 UC-emulates $\mathcal{F}_{\mathsf{OPRF}}$ with the above simulator $\mathcal{S}$, under the Gap CDH assumption in ROM, and with respect to static malicious corruptions and adaptive server compromise.

We recommend the reader to review again the technical roadmap at the end of Section 1, which describes the contents of the three upcoming technical sections containing our main results.

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 $\mathcal{P}$ [13]. 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.[5]

In Figure 9 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 9 is a standard unauthenticated secure channel functionality (e.g., [14]), 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 [14]. 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 messages between them. However, if the adversary plays a man-in-the-middle, which is modeled by the Attack interface, it can arbitrarily set

[5]TLS Handshake includes authentication, implemented by messages auth_S and auth_C in Fig. 3. However, as mentioned in footnote 3, 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.

the channel binder the attacked parties output. The contract which $\mathcal{F}_{\mathsf{cbSC}}$ enforces regarding the CB outputs is thus the same as keys output in a key exchange: Two passively connected sessions output the same random CB value unknown to the adversary, while an actively attacked session outputs an adversarially chosen value. However, crucially, there is one aspect in which $\mathcal{F}_{\mathsf{cbSC}}$ strengthens this standard contract: On each actively attacked session its CB output must be chosen subject to the condition that it cannot equal to a CB value output by any other uncorrupted party.

In a standard key exchange notion it is not excluded if two actively attacked parties output the same key known by the attacker. However, we need to prevent such collisions for channel binders because binders must uniquely define a channel, and if parties $\mathcal{P}$ and $\mathcal{P}'$ establish TLS channels with an adversary rather than with one another than we cannot allow the subsequent TLS-EA or TLS-OPAQUE authentication action issued by $\mathcal{P}$ on its channel with the attacker to be usable by the attacker on its channel with $\mathcal{P}'$. 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. 9.

4.1 Modeling Secure Channel with a Binder

We walk through ideal functionality $\mathcal{F}_{\mathsf{cbSC}}$ to make it easier to understand how it models a secure channel (establishment and use) protocol like TLS. Party $\mathcal{P}$ can start a Key Exchange (KE) protocol to establish a secure session using interface $(\textsc{NewSession}, \mathsf{cid}, \mathcal{P}', \mathsf{role})$ [N.1], where cid is the *channel identifier*, i.e. a locally unique handle which $\mathcal{P}$ will use for that channel, e.g. a port number for TLS, $\mathcal{P}'$ is the counterparty with which $\mathcal{P}$ intends to establish a session with, e.g. a domain name or an IP address, and role is a flag equal to either `clt` or `srv`, which helps to break symmetry in a protocol, e.g. in the TLS handshake the role bit determines the order in which party's message is placed in the transcript $\mathsf{tr} = (\mathsf{rand}_C, g^x, \mathsf{rand}_S, g^y)$. Since cid is unique per $\mathcal{P}$, we use $(\mathcal{P}, \mathsf{cid})$ to designate a unique session started by this interface. Note that $\mathcal{F}_{\mathsf{cbSC}}$, like TLS, is a *blind dating* functionality, in the sense that entity $\mathcal{P}'$ specified by $\mathcal{P}$ as an intended counterparty in this handshake, is used only to direct the protocol messages $\mathcal{P}$ sends. However, it is not used otherwise in the protocol processing, hence the network adversary can freely connect party $\mathcal{P}$ to party $\mathcal{P}^* \neq \mathcal{P}'$ even though $\mathcal{P}$ specified $\mathcal{P}'$ as its intended counterparty in the $\textsc{NewSession}$ inputs. All these inputs are considered public hence $\mathcal{F}_{\mathsf{cbSC}}$ passes them to the adversary. Note that $\mathcal{F}_{\mathsf{cbSC}}$ does not record field $\mathcal{P}'$ in the record $(\textsc{session}, \mathcal{P}, \mathsf{cid}, \mathsf{role})$ labeled `wait` it creates for session $(\mathcal{P}, \mathsf{cid})$, because it is not used afterwards.

If the real-world adversary is passive and forwards all KE messages between sessions $(\mathcal{P}, \mathsf{cid})$ and $(\mathcal{P}', \mathsf{cid}')$ then whenever session $(\mathcal{P}, \mathsf{cid})$ receives the last message the ideal-world adversary $\mathcal{A}$ sends $(\textsc{Connect}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}', \mathsf{cid}^*, \mathsf{CB}^*)$ to $\mathcal{F}_{\mathsf{cbSC}}$ for $(\mathsf{cid}^*, \mathsf{CB}^*) = (\mathsf{cid}', \perp)$. If both parties are honest and session $(\mathcal{P}, \mathsf{cid})$ is the first to terminate [C.1.2.1] then $\mathcal{F}_{\mathsf{cbSC}}$ sends $(\textsc{Finalize}, \mathsf{cid}, \mathsf{cid}^*, \mathsf{CB})$ to $\mathcal{P}$ for a random CB, which causes $(\mathcal{P}, \mathsf{cid})$ to output a uniformly random channel binder CB (just like a session key output by a standard KE functionality if the adversary is passive). To account [C.2] for this secure connection $\mathcal{F}_{\mathsf{cbSC}}$ marks $(\mathcal{P}, \mathsf{cid})$ as $\mathtt{conn}(\mathcal{P}', \mathsf{cid}')$ and sets up an initially empty queue on which $(\mathcal{P}, \mathsf{cid})$ can securely communicate with $(\mathcal{P}', \mathsf{cid}')$. If the counterparty $(\mathcal{P}', \mathsf{cid}')$ terminated first on such passively-connected KE instance then $\mathcal{F}_{\mathsf{cbSC}}$ created a random CB' for it and marked $(\mathcal{P}', \mathsf{cid}')$ as $\mathtt{conn}(\mathcal{P}, \mathsf{cid})$, in which case [C.1.1] $\mathcal{F}_{\mathsf{cbSC}}$ sets $\mathsf{CB} \leftarrow \mathsf{CB}'$ which enforces that two passively connected KE sessions establish a secure channel *and* output same random value as the channel binder. Note that every CB created by $\mathcal{F}_{\mathsf{cbSC}}$ is added to set CBset, and with overwhelming probability these values are all distinct (except for equality of CB's on pairs of passively-connected sessions).

The functionality talks to arbitrarily many parties $\mathfrak{P} = \{\mathcal{P}, \mathcal{P}', \ldots\}$ and to the adversary $\mathcal{A}$. It maintains list CBset of all created channel binders.

Channel establishment

On $(\textsc{NewSession}, \mathsf{cid}, \mathcal{P}', \mathsf{role})$ from party $\mathcal{P}$:

[N.1] If $\mathsf{role} \in \{\texttt{clt}, \texttt{srv}\}$ and $\nexists$ record $(\textsc{session}, \mathcal{P}, \mathsf{cid}, *)$ then create record $(\textsc{session}, \mathcal{P}, \mathsf{cid}, \mathsf{role})$ labeled $\texttt{wait}$ and send $(\textsc{NewSession}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{role})$ to $\mathcal{A}$.

On $(\textsc{Attack}, \mathcal{P}, \mathsf{cid}, \mathsf{cid}^*, \mathsf{CB}^*)$ from $\mathcal{A}$:

[A.1] If $\exists$ record $(\textsc{session}, \mathcal{P}, \mathsf{cid}, \mathsf{role})$ labeled $\texttt{wait}$ and $\mathsf{CB}^* \notin \mathsf{CBset}$ then add CB^* to CBset, re-label this record $\texttt{att}$, and send $(\textsc{Finalize}, \mathsf{cid}, \mathsf{cid}^*, \mathsf{role}, \mathsf{CB}^*)$ to $\mathcal{P}$.

On $(\textsc{Connect}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}', \mathsf{cid}^*, \mathsf{CB}^*)$ from $\mathcal{A}$, if $\exists$ record $(\textsc{session}, \mathcal{P}, \mathsf{cid}, \mathsf{role})$ labeled $\texttt{wait}$:

```
[C.1] If $\exists$ rec. $(\textsc{session}, \mathcal{P}', \mathsf{cid}', \mathsf{role}')$ labeled $\texttt{conn}(\mathcal{P}, \mathsf{cid})$ s.t. $\mathsf{role}' \neq \mathsf{role}$
    [C.1.1] then set $\mathsf{CB} \leftarrow \mathsf{CB}'$ for $\mathsf{CB}'$ used in message $(\textsc{Finalize}, \mathsf{cid}', \mathsf{cid}, \mathsf{role}', \mathsf{CB}')$ sent formerly to $\mathcal{P}'$;
    [C.1.2] otherwise:
        [C.1.2.1] If $\mathcal{P}$ honest and $(\mathcal{P}'$ honest $\vee\ \mathcal{P}' = \bot)$ then $\mathsf{CB} \leftarrow \{0,1\}^\lambda$;
        [C.1.2.2] If $\mathcal{P}$ or $\mathcal{P}'$ is corrupted and $\mathsf{CB}^* \notin \mathsf{CBset}$ then $\mathsf{CB} \leftarrow \mathsf{CB}^*$ (if $\mathsf{CB}^* \in \mathsf{CBset}$
            then drop this query);
[C.2] Initialize an empty queue $\textsc{Queue}(\mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}')$, re-label record $(\textsc{session}, \mathcal{P}, \mathsf{cid}, \mathsf{role})$ as $\texttt{conn}(\mathcal{P}', \mathsf{cid}')$, add $\mathsf{CB}$ to $\mathsf{CBset}$, and send $(\textsc{Finalize}, \mathsf{cid}, \mathsf{cid}^*, \mathsf{role}, \mathsf{CB})$ to $\mathcal{P}$.
```

Using the channel

On $(\textsc{Send}, \mathsf{cid}, m)$ from party $\mathcal{P}$, if $\exists$ record $(\textsc{session}, \mathcal{P}, \mathsf{cid}, \mathsf{role})$ marked $\texttt{flag}$ then:

[S.1] If $\texttt{flag} = \texttt{att}$ send $(\textsc{Send}, \mathcal{P}, \mathsf{cid}, m)$ to $\mathcal{A}$;

[S.2] If $\texttt{flag} = \texttt{conn}(\mathcal{P}', \mathsf{cid}')$ add m to the back of queue $\textsc{Queue}(\mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}')$ and send $(\textsc{Send}, \mathcal{P}, \mathsf{cid}, |m|)$ to $\mathcal{A}$;

[S.3] If $\texttt{flag} \in \{\texttt{wait}, \texttt{exp}\}$ ignore this query.

On $(\textsc{Deliver}, \mathcal{P}', \mathsf{cid}', m^*)$ from $\mathcal{A}$, if $\exists$ record $(\textsc{session}, \mathcal{P}', \mathsf{cid}', \mathsf{role}')$ marked $\texttt{flag}$ then:

[D.1] If $\texttt{flag} = \texttt{att}$ send $(\textsc{Received}, \mathsf{cid}', m^*)$ to $\mathcal{P}'$;

[D.2] If $\texttt{flag} = \texttt{conn}(\mathcal{P}, \mathsf{cid})$ remove m from the front of $\textsc{Queue}(\mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}')$ (ignore this query if this queue does not exist or is empty) and send $(\textsc{Received}, \mathsf{cid}', m)$ to $\mathcal{P}'$;

[D.3] If $\texttt{flag} \in \{\texttt{wait}, \texttt{exp}\}$ ignore this query.

On $(\textsc{ExpireSession}, \mathsf{cid})$ from $\mathcal{P}$, if $\exists$ record $(\textsc{session}, \mathcal{P}, \mathsf{cid}, \mathsf{role})$:

[E.1] label it $\texttt{exp}$ and send $(\textsc{ExpireSession}, \mathcal{P}, \mathsf{cid})$ to $\mathcal{A}$.

Figure 9: 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.

Note that the output of session $(\mathcal{P}, \mathsf{cid})$ includes counterparty's channel identifier cid^*. $\mathcal{A}$ can set it as $\mathsf{cid}^* = \mathsf{cid}'$, which models transmission of cid' without interference, but $\mathcal{A}$ can also set it arbitrarily even for passively-connected sessions, which models modifying this field in transmission. Note that in TLS 1.3 Handshake, Fig. 3, session keys are derived from the Diffie-Hellman key g^{xy} and the transcript $\mathsf{tr} = (\mathsf{rand}_C, g^x, \mathsf{rand}_S, g^y)$, which does not include addresses $\mathcal{P}, \mathcal{P}'$ or port numbers $\mathsf{cid}, \mathsf{cid}'$, hence the two parties can finalize TLS 1.3 handshake with matching (and secure) keys but with incompatible notions of each other's $(\mathcal{P}, \mathsf{cid})$ values. (In Section 2 we explain that tr used for key derivation is appended by $\mathsf{auth}_{\mathcal{P}}$ values but that also do not bind to $\mathcal{P}, \mathsf{cid}$.)

If the adversary actively attacks session $(\mathcal{P}, \mathsf{cid})$, which includes man-in-the-middle attacker which interferes in the transmitted messages otherwise than just changing fields $\mathsf{cid}, \mathsf{cid}'$, we model it via adversarial interface $(\textsc{Attack}, \mathcal{P}, \mathsf{cid}, \mathsf{cid}^*, \mathsf{CB}^*)$ [A.1]. In this case $\mathcal{F}_{\mathsf{cbSC}}$ marks session $(\mathcal{P}, \mathsf{cid})$ as $\mathtt{att}$ and sends $\textsc{Finalize}$ to $\mathcal{P}$ with values which make session $(\mathcal{P}, \mathsf{cid})$ outputs adversarially chosen values $(\mathsf{cid}^*, \mathsf{CB}^*)$. Functionality $\mathcal{F}_{\mathsf{cbSC}}$ allows $\mathcal{A}$ an arbitrary choice of CB^* except for choosing any value in CBset. In other words, an actively attacked party establishes an insecure channel (where the adversary has a receiving and sending capability, see below), but the channel binder it outputs on that insecure channel is guaranteed to differ from the channel binders output by any other honest party, on either secure or insecure channels these parties create. Note that $\textsc{Connect}$ also allows $\mathcal{A}$ to set CB to arbitrary $\mathsf{CB}^* \notin \mathsf{CBset}$ [C.1.2.2] but it is used only if connecting a corrupted party.

Finally, interface $\textsc{Send}$ models session $(\mathcal{P}, \mathsf{cid})$ sending message m on a secure channel created above, allowed if the channel is active [S.3]. If $(\mathcal{P}, \mathsf{cid})$ terminated KE by creating a secure channel with some $(\mathcal{P}', \mathsf{cid}')$ then [S.2] message m is placed on the $(\mathcal{P}, \mathsf{cid})$-to-$(\mathcal{P}', \mathsf{cid}')$ queue, but if $(\mathcal{P}, \mathsf{cid})$ was actively attacked in KE, i.e. the channel is hijacked by the adversary, then [S.1] message m is sent to the adversary. Likewise, if the real-world adversary successfully delivers message sent on such channel from $(\mathcal{P}, \mathsf{cid})$ to $(\mathcal{P}', \mathsf{cid}')$ this is modeled by interface $(\textsc{Deliver}, \mathcal{P}', \mathsf{cid}', m^*)$: If $(\mathcal{P}', \mathsf{cid}')$ has a secure channel with $(\mathcal{P}, \mathsf{cid})$ [D.2] then m^* is ignored and $(\mathcal{P}', \mathsf{cid}')$ gets the next message from the $(\mathcal{P}, \mathsf{cid})$-to-$(\mathcal{P}', \mathsf{cid}')$ queue, and if the adversary hijacked this channel in KE [D.1] then $(\mathcal{P}', \mathsf{cid}')$ receives the adversarial message m^*.

4.2 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 10 we specify how TLS 1.3 implements $\mathcal{F}_{\mathsf{cbSC}}$ commands $\textsc{NewSession}$ and $\textsc{Send}$, 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 $\textsc{Finalize}$ which finalizes the handshake, and $\textsc{Received}$ which stands for receiving a message on the channel.

The implementation in Fig. 10 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. 10 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 $(k, x|f)$ [23], and we use both properties in the security analysis. Finally, we emulate TLS message transport by implementing command $(\textsc{Send}, \mathsf{cid}, m)$ 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.)

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

Theorem 4.1 (Security of TLS as unauthenticated secure channel)**.** *TLS 1.3 handshake and message transport protocol specified in Fig. 10 UC-emulates functionality $\mathcal{F}_{\mathsf{cbSC}}$ in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model, with* H *modeled*

```
parameters: group $\langle g \rangle$ of order $p$, sec. par. $\lambda$, hash $\mathsf{H}$ onto $\{0,1\}^\lambda$

Party $\mathcal{P}$ on input $(\textsc{NewSession}, \mathsf{cid}_C, \mathcal{P}', \mathtt{clt})$ [here $\mathcal{P}$ is a Client]:
  • Pick $\mathsf{rand}_C \leftarrow \{0,1\}^\lambda, x \leftarrow \mathbb{Z}_p$, send $(\mathsf{cid}_C, \mathsf{rand}_C, g^x)$ to $\mathcal{P}'$;
  • On receiving network message $(\mathsf{cid}'_S, \mathsf{rand}'_S, Y')$,
      – set $K \leftarrow (Y')^x, \mathsf{HS} \leftarrow \mathsf{H}(K), \mathsf{tr} \leftarrow (\mathsf{rand}_C, g^x, \mathsf{rand}'_S, Y')$,
      – $\mathsf{EMS} \leftarrow \mathsf{KDF}^0(\mathsf{HS}, \mathsf{tr})$, $\mathsf{AEK}_C \leftarrow \mathsf{KDF}^1(\mathsf{HS}, \mathsf{tr})$, $\mathsf{AEK}_S \leftarrow \mathsf{KDF}^2(\mathsf{HS}, \mathsf{tr})$,
      – save $(\mathsf{cid}_C, \mathcal{P}', \mathsf{cid}'_S, \mathsf{AEK}_C, \mathsf{AEK}_S, 0, 0)$,
      – output $(\textsc{Finalize}, \mathsf{cid}_C, \mathsf{cid}'_S, \mathtt{clt}, \mathsf{EMS})$.

Party $\mathcal{P}$ on input $(\textsc{NewSession}, \mathsf{cid}_S, \mathcal{P}', \mathtt{srv})$ [here $\mathcal{P}$ is a Server]:
  • On receiving network message $(\mathsf{cid}'_C, \mathsf{rand}'_C, X')$,
      – pick $\mathsf{rand}_S \leftarrow \{0,1\}^\lambda, y \leftarrow \mathbb{Z}_p$, send $(\mathsf{cid}_S, \mathsf{rand}_S, g^y)$ to $\mathcal{P}'$,
      – set $K \leftarrow (X')^y, \mathsf{HS} \leftarrow \mathsf{H}(K), \mathsf{tr} \leftarrow (\mathsf{rand}'_C, X', \mathsf{rand}_S, g^y)$,
      – $\mathsf{EMS} \leftarrow \mathsf{KDF}^0(\mathsf{HS}, \mathsf{tr})$, $\mathsf{AEK}_C \leftarrow \mathsf{KDF}^1(\mathsf{HS}, \mathsf{tr})$, $\mathsf{AEK}_S \leftarrow \mathsf{KDF}^2(\mathsf{HS}, \mathsf{tr})$,
      – save $(\mathsf{cid}_S, \mathcal{P}', \mathsf{cid}'_C, \mathsf{AEK}_S, \mathsf{AEK}_C, 0, 0)$,
      – output $(\textsc{Finalize}, \mathsf{cid}_S, \mathsf{cid}'_C, \mathtt{srv}, \mathsf{EMS})$.

Party $\mathcal{P}$ on local input $(\textsc{Send}, \mathsf{cid}, m)$:
  • Retrieve $(\mathsf{cid}, \mathcal{P}', \mathsf{cid}', \mathsf{AEK}, \mathsf{AEK}', \mathsf{ctr}, \mathsf{ctr}')$ (abort if it is not found),
      – send $(\mathsf{cid}', \mathsf{AEnc}(\mathsf{AEK}, (\mathsf{ctr}, m)))$ to $\mathcal{P}'$ and increment $\mathsf{ctr}$.

Party $\mathcal{P}$ on network message $(\mathsf{cid}', c)$:
  • Retrieve $(\mathsf{cid}, \mathcal{P}', \mathsf{cid}', \mathsf{AEK}, \mathsf{AEK}', \mathsf{ctr}, \mathsf{ctr}')$ (abort if it is not found),
      – $(\mathsf{ctr}^*, m) \leftarrow \mathsf{ADec}(\mathsf{AEK}', c)$, abort if output doesn't parse as such pair
      – if $\mathsf{ctr}^* = \mathsf{ctr}'$ then output $(\textsc{Received}, \mathsf{cid}, m)$ and increment $\mathsf{ctr}'$.

Party $\mathcal{P}$ on input $(\textsc{ExpireSession}, \mathsf{cid})$:
  • Erase record $(\mathsf{cid}, \mathcal{P}', \mathsf{cid}', \mathsf{AEK}, \mathsf{AEK}', \mathsf{ctr}, \mathsf{ctr}')$.
```

Figure 10: TLS 1.3 as realization of functionality $\mathcal{F}_{\mathsf{cbSC}}$

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 Section 3 for the cryptographic assumptions in this theorem, and to Appendix B.1 for the 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.

The functionality talks to arbitrarily many parties $\mathfrak{P} = \{\mathcal{P}, \mathcal{P}', \ldots\}$ 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 9, 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 $\mathcal{P}'$.

Figure 11: $\mathcal{F}_{\mathsf{PHA}}$ 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.

5.1 Post-Handshake Authentication Model

Figure 11 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 $\mathtt{mode}, \mathsf{ak}, \mathsf{ske}$ 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 $(\text{KeyGen}, \mathsf{kid}, \varepsilon, \varepsilon, \mathtt{std})$, resulting in output $(\text{key}, \mathsf{kid}, \varepsilon, \varepsilon, \varepsilon, \mathsf{pk})$ 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 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 $(\text{AuthSend}, S, \mathsf{cid}_C, \mathsf{ssid}, \mathsf{ctx}, \varepsilon, \varepsilon, \mathsf{pk}, \mathtt{std})$. 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 $(\text{Auth}, C, \mathsf{cid}_C, S, \mathsf{cid}_S, \mathsf{ssid}, \mathsf{ctx}, \mathsf{pk})$, 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 $(\text{KeyGen}, \mathsf{kid}, \mathsf{ak}, \mathsf{aux}, \mathtt{tk})$, 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.

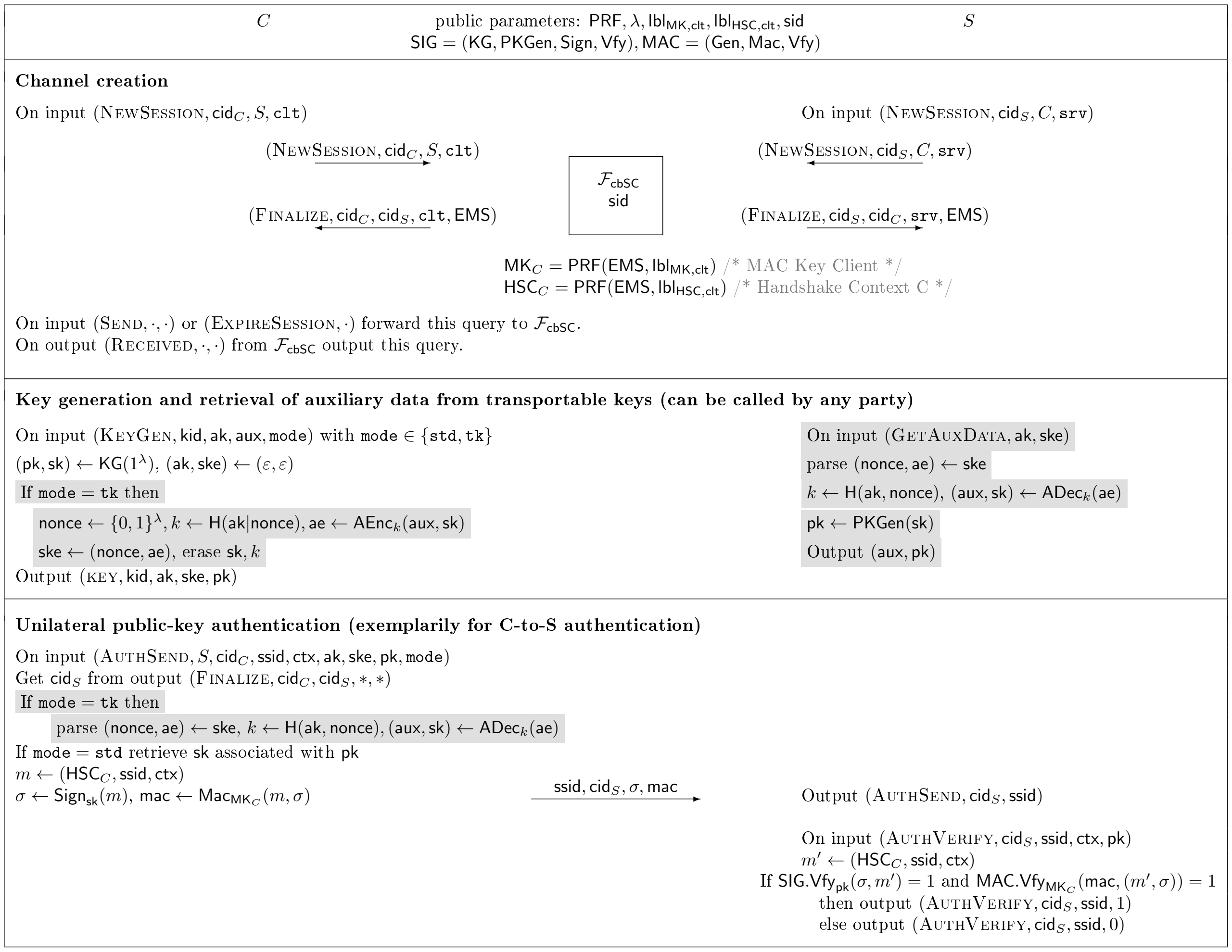
Adversarial interfaces. The adversary $\mathcal{A}$ can register both $\mathtt{std}$ and $\mathtt{tk}$ keys via interface KeyGen. $\mathcal{F}_{\mathsf{PHA}}$ adds such compromised keys to set pkComp. For transportable keys $\mathsf{ak}, \mathsf{ske}$, the adversary can also reveal which public key they "work for", by querying (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 $\mathtt{mode}$ 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 12. 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, (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, and (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 $\mathtt{clt}$, $\mathtt{srv}$ 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.


C public parameters: $\mathsf{PRF}, \lambda, \mathsf{lbl}_{\mathsf{MK,clt}}, \mathsf{lbl}_{\mathsf{HSC,clt}}, \mathsf{sid}$ S

$\mathsf{SIG} = (\mathsf{KG}, \mathsf{PKGen}, \mathsf{Sign}, \mathsf{Vfy}), \mathsf{MAC} = (\mathsf{Gen}, \mathsf{Mac}, \mathsf{Vfy})$

Channel creation

On input $(\textsc{NewSession}, \mathsf{cid}_C, S, \mathtt{clt})$ | On input $(\textsc{NewSession}, \mathsf{cid}_S, C, \mathtt{srv})$

$(\textsc{NewSession}, \mathsf{cid}_C, S, \mathtt{clt}) \longrightarrow \boxed{\mathcal{F}_{\mathsf{cbSC}}\ \mathsf{sid}} \longleftarrow (\textsc{NewSession}, \mathsf{cid}_S, C, \mathtt{srv})$

$(\textsc{Finalize}, \mathsf{cid}_C, \mathsf{cid}_S, \mathtt{clt}, \mathsf{EMS}) \longleftarrow \boxed{\mathcal{F}_{\mathsf{cbSC}}\ \mathsf{sid}} \longrightarrow (\textsc{Finalize}, \mathsf{cid}_S, \mathsf{cid}_C, \mathtt{srv}, \mathsf{EMS})$

$\mathsf{MK}_C = \mathsf{PRF}(\mathsf{EMS}, \mathsf{lbl}_{\mathsf{MK,clt}})$ /* MAC Key Client */
$\mathsf{HSC}_C = \mathsf{PRF}(\mathsf{EMS}, \mathsf{lbl}_{\mathsf{HSC,clt}})$ /* Handshake Context C */

On input $(\textsc{Send}, \cdot, \cdot)$ or $(\textsc{ExpireSession}, \cdot)$ forward this query to $\mathcal{F}_{\mathsf{cbSC}}$.
On output $(\textsc{Received}, \cdot, \cdot)$ from $\mathcal{F}_{\mathsf{cbSC}}$ output this query.

Key generation and retrieval of auxiliary data from transportable keys (can be called by any party)

```
On input $(\textsc{KeyGen}, \mathsf{kid}, \mathsf{ak}, \mathsf{aux}, \mathtt{mode})$ with $\mathtt{mode} \in \{\mathtt{std}, \mathtt{tk}\}$
$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KG}(1^\lambda)$, $(\mathsf{ak}, \mathsf{ske}) \leftarrow (\varepsilon, \varepsilon)$
If $\mathtt{mode} = \mathtt{tk}$ then
    $\mathsf{nonce} \leftarrow \{0,1\}^\lambda, k \leftarrow \mathsf{H}(\mathsf{ak}|\mathsf{nonce}), \mathsf{ae} \leftarrow \mathsf{AEnc}_k(\mathsf{aux}, \mathsf{sk})$
    $\mathsf{ske} \leftarrow (\mathsf{nonce}, \mathsf{ae})$, erase $\mathsf{sk}, k$
Output $(\textsc{key}, \mathsf{kid}, \mathsf{ak}, \mathsf{ske}, \mathsf{pk})$
```

```
On input $(\textsc{GetAuxData}, \mathsf{ak}, \mathsf{ske})$
parse $(\mathsf{nonce}, \mathsf{ae}) \leftarrow \mathsf{ske}$
$k \leftarrow \mathsf{H}(\mathsf{ak}, \mathsf{nonce})$, $(\mathsf{aux}, \mathsf{sk}) \leftarrow \mathsf{ADec}_k(\mathsf{ae})$
$\mathsf{pk} \leftarrow \mathsf{PKGen}(\mathsf{sk})$
Output $(\mathsf{aux}, \mathsf{pk})$
```

Unilateral public-key authentication (exemplarily for C-to-S authentication)

```
On input $(\textsc{AuthSend}, S, \mathsf{cid}_C, \mathsf{ssid}, \mathsf{ctx}, \mathsf{ak}, \mathsf{ske}, \mathsf{pk}, \mathtt{mode})$
Get $\mathsf{cid}_S$ from output $(\textsc{Finalize}, \mathsf{cid}_C, \mathsf{cid}_S, *, *)$
If $\mathtt{mode} = \mathtt{tk}$ then
    parse $(\mathsf{nonce}, \mathsf{ae}) \leftarrow \mathsf{ske}$, $k \leftarrow \mathsf{H}(\mathsf{ak}, \mathsf{nonce})$, $(\mathsf{aux}, \mathsf{sk}) \leftarrow \mathsf{ADec}_k(\mathsf{ae})$
If $\mathtt{mode} = \mathtt{std}$ retrieve $\mathsf{sk}$ associated with $\mathsf{pk}$
$m \leftarrow (\mathsf{HSC}_C, \mathsf{ssid}, \mathsf{ctx})$
$\sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$, $\mathsf{mac} \leftarrow \mathsf{Mac}_{\mathsf{MK}_C}(m, \sigma)$
```

$\xrightarrow{\mathsf{ssid}, \mathsf{cid}_S, \sigma, \mathsf{mac}}$

```
Output $(\textsc{AuthSend}, \mathsf{cid}_S, \mathsf{ssid})$

On input $(\textsc{AuthVerify}, \mathsf{cid}_S, \mathsf{ssid}, \mathsf{ctx}, \mathsf{pk})$
$m' \leftarrow (\mathsf{HSC}_C, \mathsf{ssid}, \mathsf{ctx})$
If $\mathsf{SIG.Vfy}_{\mathsf{pk}}(\sigma, m') = 1$ and $\mathsf{MAC.Vfy}_{\mathsf{MK}_C}(\mathsf{mac}, (m', \sigma)) = 1$
    then output $(\textsc{AuthVerify}, \mathsf{cid}_S, \mathsf{ssid}, 1)$
    else output $(\textsc{AuthVerify}, \mathsf{cid}_S, \mathsf{ssid}, 0)$
```

Figure 12: 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 $\mathtt{std}$ key mode or transportable key mode $\mathtt{tk}$. For brevity we omit the functionality's identifier sid from all queries and messages.

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 12. We start with standard signing keys and for now ignore the gray parts of the figure. Upon input $(\textsc{KeyGen}, \mathsf{kid}, \mathsf{ak}, \mathsf{aux}, \mathtt{std})$, 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[6] identifier cid_S in the $\textsc{Finalize}$ 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 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 $\mathsf{ak}, \mathsf{ske}, \mathsf{pk}$ (cf. gray parts in Figure 12): before signing and mac'ing, C first recovers sk from $\mathsf{ak}, \mathsf{ske}$.

This concludes our description of Π_{EA}, and we are ready to state its security. We refer to Section 3 for formal definitions of the cryptographic assumptions within the Theorem and to Appendix B.2 for the full proof.

Theorem 5.1 (Security of Π_{EA}). *Protocol Π_{EA} depicted in Figure 12 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,* $(\mathsf{KG}, \mathsf{PKGen}, \mathsf{Sign}, \mathsf{Vfy})$ *a perfectly complete and EUF-CMA-secure signature scheme,* MAC *is perfectly complete and EUF-CMA-secure MAC, and* $(\mathsf{AEnc}, \mathsf{ADec})$ *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 [13].

Corollary 5.2. *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 13. On a high level, $\mathcal{F}_{\mathsf{pwPHA}}$ guarantees the following:

[6]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.

- **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 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 $(\text{PWAUTH}, \mathsf{ssid}, C, \mathsf{cid}_C, S, \mathsf{cid}_S, \mathsf{uid}, \mathsf{pw}, \mathtt{init}, 0)$ 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 $\mathtt{match}$, otherwise it is rewritten to $\mathtt{fail}$. 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 $\mathtt{fail}$ or $\mathtt{match}$, 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 $\mathtt{completed}$, 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 $\mathtt{compromised}$, which "unlocks" interfaces 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

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 9, but without gray parts.*

Password Registration, Compromise, and Offline Dictionary Attack

On $(\textsc{StorePwdFile}, \mathsf{ssid}, \mathcal{P}, \mathsf{uid}, \mathsf{pw},)$ from $\mathcal{P}'$:

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

On $(\textsc{StorePwdComplete}, \mathcal{P}^*, \mathsf{ssid})$ from $\mathcal{A}$:

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

On $(\textsc{StealPwdFile}, \mathcal{P}, \mathsf{uid})$ from $\mathcal{A}$ *(requires permission from $\mathcal{Z}$)*:

[S.1] If $\nexists$ record $(\textsc{file}, \mathcal{P}, \mathsf{uid}, \mathsf{pw})$, return "no file" to $\mathcal{A}$;

[S.2] Else mark this record `compromised` and return "file stolen".

On $(\textsc{OfflTestPwd}, \mathcal{P}, \mathsf{uid}, \mathsf{pw}')$ from $\mathcal{A}$ *(requires permission from $\mathcal{Z}$)*:

[O.1] If $\exists$ record $(\textsc{file}, \mathcal{P}, \mathsf{uid}, \mathsf{pw})$ marked `compromised` then return "correct guess" if $\mathsf{pw} = \mathsf{pw}'$ and "wrong guess" if $\mathsf{pw} \neq \mathsf{pw}'$.

Active Attacks

On $(\textsc{ActiveAttack}, \mathsf{ssid}', \mathcal{P}, \mathsf{cid}, \mathsf{uid})$ from $\mathcal{A}$:

[A.1] If $\exists$ record $(\textsc{session}, \mathcal{P}, \mathsf{cid})$ marked `att`, or if $\exists$ record $(\textsc{session}, \mathcal{P}', \mathsf{cid}')$ marked $\mathtt{conn}(\mathcal{P}, \mathsf{cid})$ where $\mathcal{P}'$ is corrupted, then create record $(\textsc{pwAuth}, \mathsf{ssid}', \mathcal{A}, \varepsilon, \mathcal{P}, \mathsf{cid}, \mathsf{uid}, \varepsilon, \mathtt{init}, 0)$

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

On $(\textsc{TestPwd}, \mathcal{P}, \mathsf{uid}, \mathsf{pw}')$ from $\mathcal{A}$:

[T.1] If $\exists$ record $(\textsc{file}, \mathcal{P}, \mathsf{uid}, \mathsf{pw})$ and $\mathsf{ctr}[\mathcal{P}, \mathsf{uid}] > 0$ then do:

- [T.1.1] If $\mathsf{pw} = \mathsf{pw}'$ then return "correct guess" and rewrite `init` to `match` in all records $(\textsc{pwAuth}, *, \mathcal{A}, \varepsilon, \mathcal{P}, *, \mathsf{uid}, \varepsilon, \mathtt{init}, 0)$;
- [T.1.2] If $\mathsf{pw} \neq \mathsf{pw}'$ then return "wrong guess";
- [T.1.2] Set $\mathsf{ctr}[\mathcal{P}, \mathsf{uid}] - -$.

On $(\textsc{Impersonate}, \mathsf{ssid}', \mathcal{P}, \mathsf{uid}, \mathsf{pw}^*)$ from $\mathcal{A}$:

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

[Im.2] If $\mathsf{pw}^* \neq \varepsilon$ and $\exists$ record $(\textsc{pwAuth}, \mathsf{ssid}', *, *, \mathcal{A}, \varepsilon, \mathsf{uid}, \mathsf{pw}, \mathtt{init}, 0)$, if $\mathsf{pw} = \mathsf{pw}^*$ then overwrite `init` with `match` and reply with "correct guess", otherwise overwrite `init` with `fail` and reply with "wrong guess".

Asymmetric Password Authentication

On $(\textsc{pwInit}, \mathcal{P}'', \mathsf{cid}, \mathsf{ssid}', \mathsf{uid}, \mathsf{pw})$ from $\mathcal{P} \in \mathfrak{P}$:

[In.1] Drop the query if it is not the first one for ssid';

[In.2] Send $(\textsc{pwInit}, \mathcal{P}, \mathcal{P}'', \mathsf{cid}, \mathsf{ssid}', \mathsf{uid})$ to $\mathcal{A}$ and receive back $(\textsc{pwInit}, \mathcal{P}, \mathcal{P}'', \mathsf{cid}, \mathsf{ssid}', \mathsf{uid}, \mathsf{ok})$;

[In.3] If $\exists$ record $(\textsc{session}, \mathcal{P}, \mathsf{cid})$ marked `att`, create record $(\textsc{pwAuth}, \mathsf{ssid}', \mathcal{P}, \mathsf{cid}, \mathcal{A}, \varepsilon, \mathsf{uid}, \mathsf{pw}, \mathtt{init}, 0)$, set $\mathcal{P}'' \leftarrow \mathcal{A}$;

[In.4] If $\exists$ record $(\textsc{session}, \mathcal{P}, \mathsf{cid})$ marked $\mathtt{conn}(\mathcal{P}', \mathsf{cid}')$ create record $(\textsc{pwAuth}, \mathsf{ssid}', \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}', \mathsf{uid}, \mathsf{pw}, \mathtt{init}, 0)$, set $\mathcal{P}'' \leftarrow \mathcal{P}'$, $\mathsf{cid} \leftarrow \mathsf{cid}'$;

[In.5] Output $(\textsc{pwInit}, \mathsf{cid}, \mathsf{ssid}', \mathsf{uid})$ to $\mathcal{P}''$

On $(\textsc{pwProceed}, \mathsf{ssid}')$ from $\mathcal{P}$:

[P.1] Send $(\textsc{pwProceed}, \mathcal{P}, \mathsf{cid}, \mathsf{ssid}', \mathsf{uid})$ to $\mathcal{A}$;

[P.2] If $\exists$ record $(\textsc{pwAuth}, \mathsf{ssid}', \mathcal{A}, \varepsilon, \mathcal{P}, *, \mathsf{uid}, \varepsilon, \mathtt{init}, 0)$ then set $\mathsf{ctr}[\mathcal{P}, \mathsf{uid}] + +$;

[P.3] If $\exists$ records $(\textsc{pwAuth}, \mathsf{ssid}', *, *, \mathcal{P}, *, \mathsf{uid}, \mathsf{pw}, \mathtt{init}, 0)$ with $\mathsf{pw} \neq \varepsilon$ and $(\textsc{file}, \mathcal{P}, \mathsf{uid}, \mathsf{pw}')$, then overwrite `init` with `match` if $\mathsf{pw} = \mathsf{pw}'$ and with `fail` otherwise.

On $(\textsc{pwDeliver}, \mathsf{ssid}', \mathcal{P}, b)$ from $\mathcal{A}$

[D.1] If $b = 0$ output $(\textsc{pwDecision}, \mathsf{ssid}', \mathtt{fail})$ to $\mathcal{P}$.

[D.2] Find record $(\textsc{pwAuth}, \mathsf{ssid}', \mathcal{P}', *, \mathcal{P}'', *, *, *, \mathtt{state}, ctr)$ with $\mathcal{P} = \mathcal{P}''$ or $\mathcal{P} = \mathcal{P}''$, otherwise drop query;

[D.3] Rewrite state from `init` to `fail`, then set $\mathtt{result} = \mathtt{state}$ and $ctr + +$; if $ctr = 2$ overwrite `state` with `completed` in the record;

[D.4] Output $(\textsc{pwDecision}, \mathsf{ssid}', \mathtt{result})$ to $\mathcal{P}$.

Figure 13: $\mathcal{F}_{\mathsf{pwPHA}}$ model of *password-based* post-handshake authentication, again omitting session identifier sid.

authentication sessions described above, using ($\textsc{Impersonate}, \mathsf{ssid}, S, \mathsf{uid}, \varepsilon$) [Im.1]. We capture the ability of the attacker of compute its own password files by an optional input pw to $\textsc{Impersonate}$. 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 $\textsc{pwAuth}$ record with $\mathcal{A}$ as server and follows the procedure of honest authentication, except that it expects $\mathcal{A}$ to use an $\textsc{Impersonate}$ query instead of $\textsc{pwProceed}$. 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 $\textsc{OfflTestPwd}$ interface since it enforces **resistance to precomputation attacks** [19]: $\mathcal{F}_{\mathsf{pwPHA}}$ does not allow to pre-register guesses[7] 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 $\textsc{StealPwdFile}$ and $\textsc{OfflTestPwd}$ can only be queried by $\mathcal{A}$ upon being instructed by $\mathcal{Z}$.

$\textsc{ActiveAttack}$ and $\textsc{TestPwd}$ 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 $\textsc{ActiveAttack}$, specifying S, uid. $\mathcal{F}_{\mathsf{pwPHA}}$ initializes the [A.1] corresponding $\textsc{pwAuth}$ 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 $\textsc{ActiveAttack}$ interface with interface $\textsc{TestPwd}$, 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 $\textsc{TestPwd}$ 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 $\textsc{pwAuth}$ records to `match`). To make sure that the number of adversarial $\textsc{TestPwd}$ 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 $\mathsf{ctr}[S, \mathsf{uid}]$ 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 authentication, 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 14, 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 Section 3.1 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 $\mathcal{F}_{\mathsf{PHA}}$.

$\Pi_{\mathsf{TLS-OPAQUE}}$ 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 $\mathsf{ak} = \mathsf{rw}$ and $\mathsf{aux} = \mathsf{pk}_S$,

[7]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 [16] for a “non-strong” aPAKE functionality allowing for such attacks.

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 12. 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 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 Section 3 for the definition of $\mathcal{F}_{\mathsf{OPRF}}$, and to Appendix B.3 for the full proof.

Theorem 6.1 (Security of $\Pi_{\mathsf{TLS-OPAQUE}}$). *Protocol $\Pi_{\mathsf{TLS-OPAQUE}}$ (Figure 14) 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 6.2. *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. 5.1) using the UC composition theorem [13], where in turn $\mathcal{F}_{\mathsf{cbSC}}$ is instantiated with the TLS 1.3 protocol snippet from Figure 10 (Thm. 4.1), and $\mathcal{F}_{\mathsf{OPRF}}$ instantiated with 2HashDH of Figure 8.

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] Akamai. Credential Stuffing: Attacks and Economies, `https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/soti-security-credential-stuffing-attacks-and-economies-report-2019.pdf`, 2019.

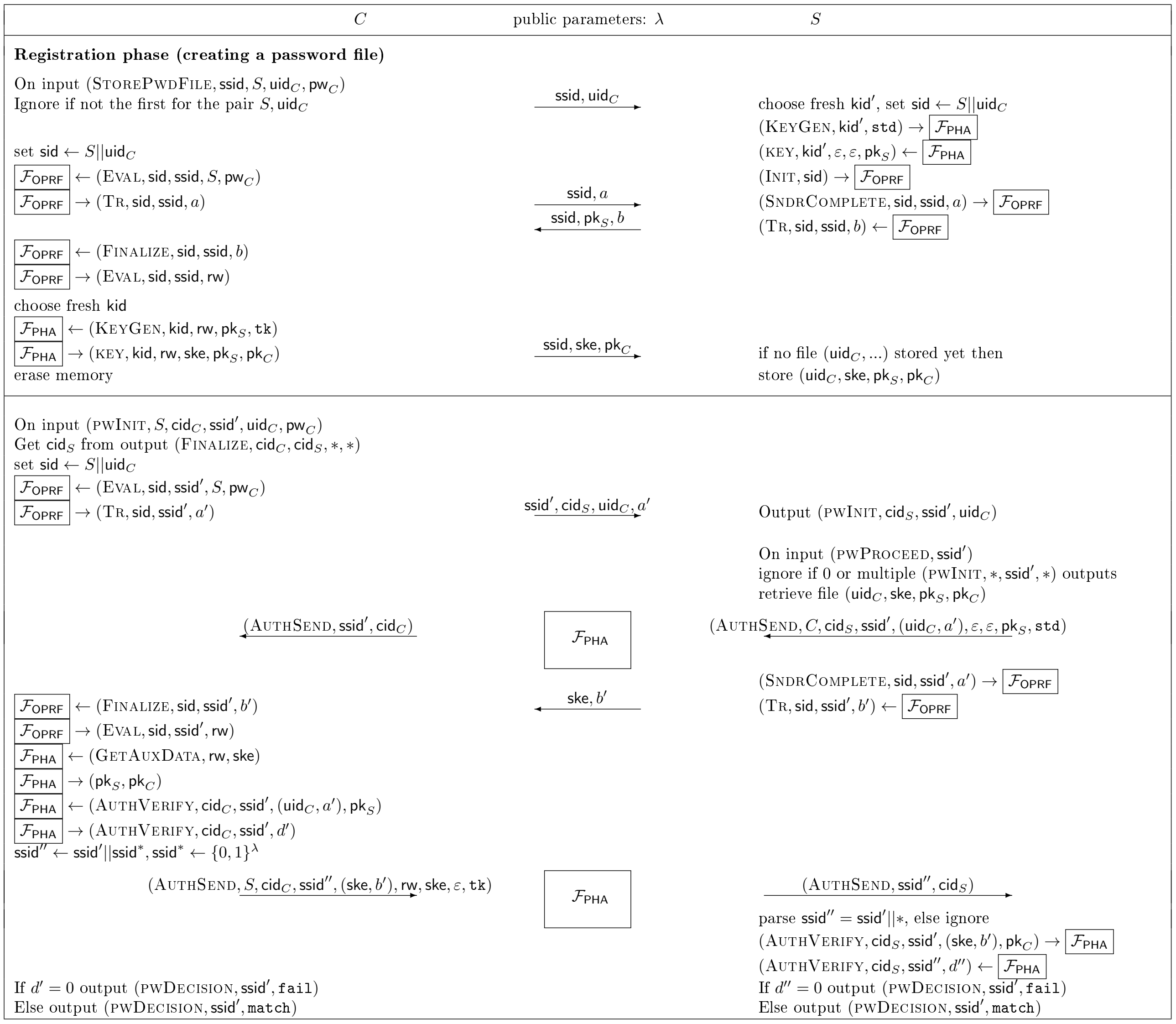

C
public parameters: λ
S
Registration phase (creating a password file)
On input $(\textsc{StorePwdFile}, \mathsf{ssid}, S, \mathsf{uid}_C, \mathsf{pw}_C)$
Ignore if not the first for the pair S, uid_C
$\mathsf{ssid}, \mathsf{uid}_C$
choose fresh kid', set $\mathsf{sid} \leftarrow S||\mathsf{uid}_C$
$(\textsc{KeyGen}, \mathsf{kid}', \mathtt{std}) \rightarrow \boxed{\mathcal{F}_{\mathsf{PHA}}}$
set $\mathsf{sid} \leftarrow S||\mathsf{uid}_C$
$(\textsc{key}, \mathsf{kid}', \varepsilon, \varepsilon, \mathsf{pk}_S) \leftarrow \boxed{\mathcal{F}_{\mathsf{PHA}}}$
$\boxed{\mathcal{F}_{\mathsf{OPRF}}} \leftarrow (\textsc{Eval}, \mathsf{sid}, \mathsf{ssid}, S, \mathsf{pw}_C)$
$(\textsc{Init}, \mathsf{sid}) \rightarrow \boxed{\mathcal{F}_{\mathsf{OPRF}}}$
$\boxed{\mathcal{F}_{\mathsf{OPRF}}} \rightarrow (\textsc{Tr}, \mathsf{sid}, \mathsf{ssid}, a)$
ssid, a
$(\textsc{SndrComplete}, \mathsf{sid}, \mathsf{ssid}, a) \rightarrow \boxed{\mathcal{F}_{\mathsf{OPRF}}}$
$\mathsf{ssid}, \mathsf{pk}_S, b$
$(\textsc{Tr}, \mathsf{sid}, \mathsf{ssid}, b) \leftarrow \boxed{\mathcal{F}_{\mathsf{OPRF}}}$
$\boxed{\mathcal{F}_{\mathsf{OPRF}}} \leftarrow (\textsc{Finalize}, \mathsf{sid}, \mathsf{ssid}, b)$
$\boxed{\mathcal{F}_{\mathsf{OPRF}}} \rightarrow (\textsc{Eval}, \mathsf{sid}, \mathsf{ssid}, \mathsf{rw})$
choose fresh kid
$\boxed{\mathcal{F}_{\mathsf{PHA}}} \leftarrow (\textsc{KeyGen}, \mathsf{kid}, \mathsf{rw}, \mathsf{pk}_S, \mathtt{tk})$
$\boxed{\mathcal{F}_{\mathsf{PHA}}} \rightarrow (\textsc{key}, \mathsf{kid}, \mathsf{rw}, \mathsf{ske}, \mathsf{pk}_S, \mathsf{pk}_C)$
$\mathsf{ssid}, \mathsf{ske}, \mathsf{pk}_C$
if no file $(\mathsf{uid}_C, \ldots)$ stored yet then
erase memory
store $(\mathsf{uid}_C, \mathsf{ske}, \mathsf{pk}_S, \mathsf{pk}_C)$
On input $(\textsc{pwInit}, S, \mathsf{cid}_C, \mathsf{ssid}', \mathsf{uid}_C, \mathsf{pw}_C)$
Get cid_S from output $(\textsc{Finalize}, \mathsf{cid}_C, \mathsf{cid}_S, *, *)$
set $\mathsf{sid} \leftarrow S||\mathsf{uid}_C$
$\boxed{\mathcal{F}_{\mathsf{OPRF}}} \leftarrow (\textsc{Eval}, \mathsf{sid}, \mathsf{ssid}', S, \mathsf{pw}_C)$
$\boxed{\mathcal{F}_{\mathsf{OPRF}}} \rightarrow (\textsc{Tr}, \mathsf{sid}, \mathsf{ssid}', a')$
$\mathsf{ssid}', \mathsf{cid}_S, \mathsf{uid}_C, a'$
Output $(\textsc{pwInit}, \mathsf{cid}_S, \mathsf{ssid}', \mathsf{uid}_C)$
On input $(\textsc{pwProceed}, \mathsf{ssid}')$
ignore if 0 or multiple $(\textsc{pwInit}, *, \mathsf{ssid}', *)$ outputs
retrieve file $(\mathsf{uid}_C, \mathsf{ske}, \mathsf{pk}_S, \mathsf{pk}_C)$
$(\textsc{AuthSend}, \mathsf{ssid}', \mathsf{cid}_C)$
$\mathcal{F}_{\mathsf{PHA}}$
$(\textsc{AuthSend}, C, \mathsf{cid}_S, \mathsf{ssid}', (\mathsf{uid}_C, a'), \varepsilon, \varepsilon, \mathsf{pk}_S, \mathtt{std})$
$(\textsc{SndrComplete}, \mathsf{sid}, \mathsf{ssid}', a') \rightarrow \boxed{\mathcal{F}_{\mathsf{OPRF}}}$
ske, b'
$\boxed{\mathcal{F}_{\mathsf{OPRF}}} \leftarrow (\textsc{Finalize}, \mathsf{sid}, \mathsf{ssid}', b')$
$(\textsc{Tr}, \mathsf{sid}, \mathsf{ssid}', b') \leftarrow \boxed{\mathcal{F}_{\mathsf{OPRF}}}$
$\boxed{\mathcal{F}_{\mathsf{OPRF}}} \rightarrow (\textsc{Eval}, \mathsf{sid}, \mathsf{ssid}', \mathsf{rw})$
$\boxed{\mathcal{F}_{\mathsf{PHA}}} \leftarrow (\textsc{GetAuxData}, \mathsf{rw}, \mathsf{ske})$
$\boxed{\mathcal{F}_{\mathsf{PHA}}} \rightarrow (\mathsf{pk}_S, \mathsf{pk}_C)$
$\boxed{\mathcal{F}_{\mathsf{PHA}}} \leftarrow (\textsc{AuthVerify}, \mathsf{cid}_C, \mathsf{ssid}', (\mathsf{uid}_C, a'), \mathsf{pk}_S)$
$\boxed{\mathcal{F}_{\mathsf{PHA}}} \rightarrow (\textsc{AuthVerify}, \mathsf{cid}_C, \mathsf{ssid}', d')$
$\mathsf{ssid}'' \leftarrow \mathsf{ssid}'||\mathsf{ssid}^*, \mathsf{ssid}^* \leftarrow \{0,1\}^\lambda$
$(\textsc{AuthSend}, S, \mathsf{cid}_C, \mathsf{ssid}'', (\mathsf{ske}, b'), \mathsf{rw}, \mathsf{ske}, \varepsilon, \mathtt{tk})$
$\mathcal{F}_{\mathsf{PHA}}$
$(\textsc{AuthSend}, \mathsf{ssid}'', \mathsf{cid}_S)$
parse $\mathsf{ssid}'' = \mathsf{ssid}'||*$, else ignore
$(\textsc{AuthVerify}, \mathsf{cid}_S, \mathsf{ssid}', (\mathsf{ske}, b'), \mathsf{pk}_C) \rightarrow \boxed{\mathcal{F}_{\mathsf{PHA}}}$
$(\textsc{AuthVerify}, \mathsf{cid}_S, \mathsf{ssid}'', d'') \leftarrow \boxed{\mathcal{F}_{\mathsf{PHA}}}$
If $d' = 0$ output $(\textsc{pwDecision}, \mathsf{ssid}', \mathtt{fail})$
Else output $(\textsc{pwDecision}, \mathsf{ssid}', \mathtt{match})$
If $d'' = 0$ output $(\textsc{pwDecision}, \mathsf{ssid}', \mathtt{fail})$
Else output $(\textsc{pwDecision}, \mathsf{ssid}', \mathtt{match})$

Figure 14: 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 clear-text 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`.

[4] 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.

[5] Mihir Bellare, Oded Goldreich, and Anton Mityagin. The power of verification queries in message authentication and authenticated encryption. *IACR Cryptol. ePrint Arch.*, 2004:309, 2004.

[6] 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.

[7] Karthikeyan Bhargavan, Ioana Boureanu, Pierre-Alain Fouque, Cristina Onete, and Benjamin Richard. Content delivery over tls: a cryptographic analysis of keyless ssl. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 1–16. IEEE, 2017.

[8] 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.

[9] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Alfredo Pironti. Verified contributive channel bindings for compound authentication. In *NDSS*, 2015.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In *EUROCRYPT*, pages 337–351. Springer, 2002.

[15] 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.

[16] 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.

[17] 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.

[18] Jonathan Hoyland. *An analysis of TLS 1.3 and its use in composite protocols.* PhD thesis, RHUL, Egham, UK, 2018.

[19] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In *EUROCRYPT*, pages 456–486. Springer, 2018.

[20] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition.* CRC Press, 2014.

[21] 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.

[22] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In *CRYPTO*, pages 310–331, 2001.

[23] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *CRYPTO*, pages 631–648, 2010.

[24] Hugo Krawczyk. Unilateral-to-mutual authentication compiler for key exchange (with applications to client authentication in tls 1.3). In *ACM CCS 2016*, 2016.

[25] Leona Lassak, Annika Hildebrandt, Maximilian Golla, and Blase Ur. "it's stored, hopefully, on an encrypted server": Mitigating users' misconceptions about fido2 biometric webauthn. In *Proc. USENIX Security*, 2021.

[26] Kentrell Owens, Olabode Anise, Amanda Krauss, and Blase Ur. User perceptions of the usability and security of smartphones as fido2 roaming authenticators. In *SOUPS*, pages 57–76, 2021.

[27] Hirak Ray, Flynn Wolf, Ravi Kuber, and Adam J Aviv. Why older adults (don't) use password managers. In *USENIX*, 2021.

[28] Marsh Ray and S Dispensa. Authentication gap in tls renegotiation, 2009.

[29] E. Rescorla. The transport layer security (TLS) protocol version 1.3, rfc 8446, August 2018. `http://www.rfc-editor.org/rfc/rfc8446.txt`.

[30] Martin Rex. Mitm attack on delayed tls-client auth through renegotiation, November 2009.

[31] Joseph Salowey and Eric Rescorla. Tls renegotiation vulnerability, 2009.

[32] N. Sullivan. Exported Authenticators in TLS, RFC 9261, `https://datatracker.ietf.org/doc/html/rfc9261`, July 2022.

[33] 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.

[34] S. Vinberg and J. Overson. Credential Stuffing Report, `https://www.f5.com/labs/articles/threat-intelligence/2021-credential-stuffing-report`, 2021.

[35] Ke Coby Wang and Michael K Reiter. Detecting stuffing of a user's credentials at her own accounts. In *USENIX*, pages 2201–2218, 2020.

A Real World Considerations

In this section we discuss considerations for implementing, deploying, operating, and using TLS-OPAQUE in a variety of use cases.

A.1 TLS Integration

OPAQUE, as is currently undergoing standardization in the IRTF CFRG [12], encapsulates an AKE and uses it to produce fresh secrets. In contrast, TLS-OPAQUE reuses the output of an existing AKE – TLS – for authentication. In the former variant, the client and server run OPAQUE over any arbitrary channel. This could be a raw, unencrypted and unauthenticated connection, or it could be a secure channel such as TLS. However, the privacy posture of OPAQUE is improved with a secure channel, since OPAQUE necessarily reveals the client identity to the server during the exchange. Therefore, running OPAQUE over a secure channel is superior, and TLS-OPAQUE does this by leveraging TLS record layer encryption. Moreover, since TLS-OPAQUE binds itself to the underlying TLS connection. As a result, the protocol has "best of both worlds" security, whereby security only fails if both the underlying TLS authentication fails *and* OPAQUE fails.

A result of this design is that implementing TLS-OPAQUE requires changes at the application-layer and at the TLS layer. Depending on the use case or deployment model, this may or may not be feasible. For example, TLS-OPAQUE deployed in a web browser without access to the underlying TLS connection state (via the EA protocol) is not currently feasible. However, for applications and services with control over the TLS stack, this integration is feasible.

A.2 Password Hiding

A hallmark of TLS-OPAQUE is that the protocol does not leak any plaintext password to the server. Only the client has knowledge of the password. This has numerous benefits, though the primary benefit is in mitigating misuse of sensitive authentication state. (Servers only store or derive per-client OPRF keys, and compromise of these keys does not immediately lead to account compromise.)

One consideration of this property is that it may require alternative techniques for detection of credential stuffing attacks. A credential stuffing attack is one in which an attacker automates using compromised account credentials, such as username and password tuples, to take over user accounts [34]. These attacks are often performed at scale, meaning that login attempts occur in very large bursts.[8] Common approaches for detecting such attacks rely on access to the plaintext password [35].

A.3 Key Management

TLS-OPAQUE requires servers to store or derive unique keying material for each registered client. In general, this requirement is not unique compared to existing authentication protocols, such as password-over-TLS. However, some authentication protocols such as mutual TLS do not necessarily require such per-client state. For example, if a typical PKI is used for mutual TLS, servers need only store intermediate certificates that attest to client certificates, rather than per-client certificates.

TLS-OPAQUE does not currently support such generalized forms of client authentication, though it can outsource OPRF key derivation to a separate, secure, and isolated service. Specifically, when creating the server's credential response, the server can relay the client identity to a central service which stores a single key generation seed used to derive the per-user OPRF key. One important consequence is that this separates sensitive key material used for multiple clients from the rest of the server-side implementation. In the event of server compromise, the only per-client secrets available to the server are those that were queried for individual users. This design is similar to Keyless SSL [7], which separates processes responsible for TLS functionality from TLS private key management. A consequence of this decoupled design is that loss of the key server state requires each client to re-register with the application.

A.4 Phishing Hardening

Phishing is a well-known attack wherein an attacker attempts to trick users into revealing their authentication information to a malicious server. Mitigating phishing depends on the deployment model. Specifically, if it is

[8]Akamai estimated that 30 billion credential stuffing attacks occurred in 2018 [3].

possible for an attacker to downgrade a client from use of TLS-OPAQUE, which does not reveal the plaintext password to the attacker, to more classical authentication mechanisms, phishing may be applicable. One example setting wherein such downgrades are not possible include app-based deployments of TLS-OPAQUE, since the client simply lacks support for weaker authentication protocols. In these scenarios, an attacker that is able to redirect clients to a malicious server can only recover information that is revealed through TLS-OPAQUE protocol messages. In the context of the web, the situation is different, as clients will necessarily fall back to password-over-TLS for backwards compatibility reasons.

In general, if the attacker can successfully direct the client to its attacker-controlled domain, it can force the client to login with its real credentials without using TLS-OPAQUE. Clients could remember whether or not a given domain supports TLS-OPAQUE and choose not to fall back to a less-secure protocol, though pinning this information raises deployment concerns, especially if the application needs to safely disable OPAQUE without breaking authentication.

Phishing mitigations for this use case therefore require solutions external to TLS-OPAQUE. In particular, they may require client-side changes. Some browsers such as Google Chrome already include user-facing warnings to protect against such phishing attacks. Other options might be to change the client password entry interface from something implemented using server-provided Javascript to something natively built into the browser. This may present downstream usability issues. Any such proposal requires an appropriate user study to determine its efficacy and impact; see A.5.

A.5 User Interface Implications

As an authentication protocol, the user experience and its impact on overall system security is paramount. For example, applications which do not need to support legacy protocols and cannot migrate to password-less alternatives such as WebAuthn [17], adopting TLS-OPAQUE may be trivial. In contrast, much of the deployment challenges in PAKEs stem from their interaction with existing protocols, legacy software, and user interfaces, all of which relate to the user experience.

Passwords will likely remain as an authentication mechanism for legacy and modern clients and applications for some time. This raises an opportunity for authentication mechanisms built on TLS-OPAQUE. Specifically, TLS-OPAQUE mitigates server-side risk inherent in password-based authentication by minimizing information stored about clients. Compromise of a web application that uses TLS-OPAQUE for password authentication, for example, does not immediately put client passwords at risk. (OPAQUE prevents pre-computation attacks, though it does not prevent online attacks.) Adoption of TLS-OPAQUE therefore benefits applications which adopt it.

It is also worth noting that, in many scenarios, it is unlikely that TLS-OPAQUE will be deployed as the only possible authentication system. For example, in the context of the web, many different authentication protocols must necessarily be supported to account for legacy clients and servers. In such environments, understanding the user experience is critical. For example, what is the user experience for a web application that uses both TLS-OPAQUE and legacy password-over-TLS authentication mechanisms? Does the UI flow be similar in both cases to avoid confusion, or can PAKE-based authentication use a different UI, such as a browser-based prompt for passwords?

User studies to assess the feasibility of new user interface mechanisms are needed to ensure that any changes here do not further confuse users or cause security regressions. Such studies must carefully consider complementary technologies, such as password-less authentication, and the effect of offering both as a choice to clients. Moreover, such studies must assess the ease of adopting new authentication flows. Adoption of passwordless authentication protocols like WebAuthn and password managers is still a problem in the industry [26, 27, 25] due to such usability obstacles.

B Proofs

B.1 Proof of Theorem 4.1

We first give a brief sketch of the proof. We will use Π_i and role_i for $i = (\mathcal{P}, \mathsf{cid})$ to refer to session $(\mathcal{P}, \mathsf{cid})$, invoked with role flag role_i. Let us first briefly describe the simulation algorithm, included in Figure 15. To simulate honest session Π_i simulator $\mathcal{S}$ picks $\mathsf{rand}_i \leftarrow \{0,1\}^\lambda, z_i \leftarrow \mathbb{Z}_p$, sets $Z_i \leftarrow g^{z_i}$ and sends out $(\mathsf{cid}, \mathsf{rand}_i, Z_i)$. If $\mathcal{A}$ sends message $(\mathsf{cid}^*, \mathsf{rand}^*_i, Z^*_i)$ to Π_i then $\mathcal{S}$ checks if $(\mathsf{rand}^*_i, Z^*_i) = (\mathsf{rand}_j, Z_j)$ for any Π_j for $j = (\mathcal{P}', \mathsf{cid}')$ and $\mathsf{role}'_j \neq \mathsf{role}_i$. If so then $\mathcal{S}$ marks Π_i as "connected to Π_j", connects these two sessions via $\mathcal{F}_{\mathsf{cbSC}}$, it picks random key AEK_i (the two connected sessions will have independent AEK keys but these correspond to AEK_C and AEK_S on this channel) and then emulates the TLS transport for Π_i as follows: First, given $(\textsc{Send}, \mathcal{P}, \mathsf{cid}, \ell)$ from $\mathcal{F}_{\mathsf{cbSC}}$ it forms a ciphertext as $c = \mathsf{AEnc}(\mathsf{AEK}, (\mathsf{ctr}, 0^\ell))$ for the current counter value ctr and saves $(\Pi_i, \Pi_j, \mathsf{ctr}, c)$. Second, if $\mathcal{A}$ sends (cid, c') to $\mathcal{P}$ then $\mathcal{S}$ delivers the corresponding message via $\textsc{Deliver}$ message to $\mathcal{F}_{\mathsf{cbSC}}$ if and only if it saved tuple $(\Pi_j, \Pi_i, \mathsf{ctr}', c')$ for the correct value ctr' of the incoming message counter of Π_i. In the other case, i.e. if $\mathcal{A}$ does not forward to Π_i a message from some Π_j with an opposing role then $\mathcal{S}$ computes everything as the real-world Π_i would, i.e. it derives $\mathsf{EMS}, \mathsf{AEK}_C, \mathsf{AEK}_S$ via $\mathsf{KDF}(\mathsf{H}((Z^*_i)^{z_i}), (\mathsf{tr}_i|f))$ for $f = 0, 1, 2$ and tr_i formed by ordering (rand_i, Z_i) and $(\mathsf{rand}^*_i, Z^*_i)$ appropriately, sets EMS as $\mathcal{P}$'s channel binder output via the $\textsc{Attack}$ message to $\mathcal{F}_{\mathsf{cbSC}}$, and then it encrypts and decrypts TLS transported messages as the real party would using the above keys, because $\mathcal{F}_{\mathsf{cbSC}}$ will mark this session as $\mathtt{att}$, so it will pass all the messages it sends to $\mathcal{S}$, and it will accept any received message $\mathcal{S}$ processes via the $\textsc{Deliver}$ interface.

To argue that this simulation creates an indistinguishable view from the real execution we first eliminate the negligible probability event that two honest sessions output the same rand or g^z values, which guarantees that two honest sessions have the same transcripts only if they are "connected". Since the channel binders EMS^* are computed for attacked sessions via $\mathsf{KDF}(\mathsf{HS}_i, (\mathsf{tr}_i|0))$ the probability that any of them collide is negligible by CRH property of KDF. These values can also not collide with CB's picked by $\mathcal{F}_{\mathsf{cbSC}}$ on the connected sessions: Finally, the only divergence between the real world and the ideal world is due to connected sessions on which the EMS and AEK values are random in the ideal world, whereas in the real world they are computed via $\mathsf{KDF}(\mathsf{H}(\cdot), \mathsf{tr})$ applied to the Diffie-Hellman key $K = g^{z_i \cdot z_j}$. However, all values $\mathsf{H}(g^{z_i \cdot z_j})$ (some of these can be used by sessions treated as "actively attacked", e.g. because $\mathcal{A}$ exchanges the Z_i, Z_j values between two sessions but not the nonces, or because of mismatch in the roles) can be replaced by random by a reduction to Gap CDH, which embeds DH challenges in all Z_i's and uses the DDH oracle to emulate computing $\mathsf{H}((Z^*_i)^{z_i})$ because for any $\mathcal{A}$'s query $\mathsf{H}(K)$ it can check if (Z^*_i, Z_i, K) is a DH tuple. Finally, the PRF property of KDF and the fact that tr's cannot collide unless on mutually connected sessions, imply the pseudorandomness of their EMS and AEK outputs.

We proceed to the formal proof. Let $\mathcal{Z}$ be an efficient environment and $\mathcal{A}$ the "real-world adversary" interface of $\mathcal{Z}$. Let $\mathsf{win}(\mathbf{G}_i)$ be the event that $\mathcal{Z}$ outputs 1 when interacting with Game $\mathbf{G}_i$.

Game $\mathbf{G}_0$: The real execution. Here $\mathcal{Z}$ interacts with parties running the TLS1.3 handshake and message transport as specified in Figure 10, with all network messages sent to (and received from) $\mathcal{Z}$'s adversary interface $\mathcal{A}$. However, all actions of honest parties are implemented by a single entity, denoted $\mathbf{G}_0$. Since this protocol is mostly symmetric for client and servers (the only symmetry-breaker is in ordering inputs to key derivation hash H) we will use denote the DH contributions of session Π_i party as $Z_i = g^{z_i}$ for $z_i \leftarrow \mathbb{Z}_p$, regardless if $\mathsf{role}_i = \mathtt{clt}$ or $\mathsf{role}_i = \mathtt{srv}$.

Game $\mathbf{G}_1$: Eliminating nonce or DH contribution collisions. This game runs just like $\mathbf{G}_0$, i.e. it follows the protocol on behalf of each honest party, except that the game aborts if for any honest session Π_j the game picks $\mathsf{rand}_j \leftarrow \{0,1\}^\lambda$ and $z_j \leftarrow \mathbb{Z}_p$ s.t. either $\mathsf{rand}_j = \mathsf{rand}_i$ or $z_j = z_i$ for some previously invoked honest session Π_i. It follows that

$$|\Pr[\mathsf{win}(\mathbf{G}_1)] - \Pr[\mathsf{win}(\mathbf{G}_0)]| \leq \frac{(q_{\mathsf{SC}})^2}{2 \cdot p} + \frac{(q_{\mathsf{SC}})^2}{2 \cdot 2^{-\lambda}},$$

where q_{SC} counts $\mathcal{Z}$'s $\textsc{NewSession}$ queries.

```
The simulator talks to $\mathcal{F}_{\mathsf{cbSC}}$ and to the dummy network adversary $\mathcal{A}$, which formally is
an interface of environment $\mathcal{Z}$. $\mathcal{S}$ queries RO function $\mathsf{H}$ which it implements honestly.

On $(\textsc{NewSession}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{role})$ from $\mathcal{F}_{\mathsf{cbSC}}$:
  • set $i \leftarrow (\mathcal{P}, \mathsf{cid})$, pick $\mathsf{rand} \leftarrow \{0,1\}^\lambda$ and $z \leftarrow \mathbb{Z}_p$, set $Z \leftarrow g^z$,
  • save record $(i, \mathsf{role}, \mathsf{rand}, z, Z, \bot, \bot)$ marked $\mathtt{wait}$, send $(\mathsf{cid}, \mathsf{rand}, Z)$ to $\mathcal{A}$.

On receiving $\mathcal{A}$'s message $(\mathsf{cid}^*, \mathsf{rand}^*, Z^*)$ to session $\Pi_i$ for some $i = (\mathcal{P}, \mathsf{cid})$:
  • Retrieve $\mathsf{rec} = (i, \mathsf{role}, \mathsf{rand}, z, Z, \bot, \bot)$ marked $\mathtt{wait}$ (abort if not found);
  • If $\exists$ record $\mathsf{rec}' = (j, \mathsf{role}', \mathsf{rand}', z', Z', \mathsf{AEK}', \bot)$
    (1) $\mathsf{rec}'$ is marked either $\mathtt{wait}$ [in which case $\mathsf{AEK}' = \bot$] or $\mathtt{conn}(i)$,
    (2) furthermore $(\mathsf{rand}', Z') = (\mathsf{rand}^*, Z^*)$ and $\mathsf{role}' \neq \mathsf{role}$;
    then do the following:
    pick $\mathsf{AEK} \leftarrow \{0,1\}^\lambda$;
    replace record $\mathsf{rec}$ with $(i, \mathsf{role}, \mathsf{rand}, z, Z, \mathsf{AEK}, \bot)$ marked $\mathtt{conn}(j, \mathsf{cid}^*)$;
    set $\mathsf{ctrs}(i) \leftarrow (0, 0)$;
    parse $(\mathcal{P}', \mathsf{cid}') \leftarrow j$ and send $(\textsc{Connect}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}', \bot, \bot)$ to $\mathcal{F}_{\mathsf{cbSC}}$;
  • In any other case do the following:
    set $K \leftarrow (Z^*)^z$ and $\mathsf{HS} \leftarrow \mathsf{H}(K)$;
    if $\mathsf{role} = \mathtt{clt}$ then set $\mathsf{tr} \leftarrow (\mathsf{rand}, Z, \mathsf{rand}^*, Z^*)$;
    if $\mathsf{role} = \mathtt{srv}$ then set $\mathsf{tr} \leftarrow (\mathsf{rand}^*, Z^*, \mathsf{rand}, Z)$;
    set $\mathsf{EMS} \leftarrow \mathsf{KDF}^0(\mathsf{HS}, \mathsf{tr})$, $\mathsf{AEK}_C \leftarrow \mathsf{KDF}^1(\mathsf{HS}, \mathsf{tr})$, $\mathsf{AEK}_S \leftarrow \mathsf{KDF}^2(\mathsf{HS}, \mathsf{tr})$;
    if $\mathsf{role} = \mathtt{clt}$ then set $(\mathsf{AEK}, \mathsf{AEK}') \leftarrow (\mathsf{AEK}_C, \mathsf{AEK}_S)$;
    if $\mathsf{role} = \mathtt{srv}$ then set $(\mathsf{AEK}, \mathsf{AEK}') \leftarrow (\mathsf{AEK}_S, \mathsf{AEK}_C)$;
    replace $\mathsf{rec}$ with $(i, \mathsf{rand}, z, Z, \mathsf{AEK}, \mathsf{AEK}')$ marked $\mathtt{att}(\mathsf{cid}^*)$;
    set $\mathsf{ctrs}(i) \leftarrow (0, 0)$;
    send $(\textsc{Attack}, \mathcal{P}, \mathsf{cid}, \mathsf{cid}^*, \mathsf{EMS})$ to $\mathcal{F}_{\mathsf{cbSC}}$.

On receiving $(\textsc{Send}, \mathcal{P}, \mathsf{cid}, t)$ from $\mathcal{F}_{\mathsf{cbSC}}$ for $t \in \mathbb{N}$:
  • Retrieve rec. $(i, \mathsf{rand}, z, Z, \mathsf{AEK}, \bot)$ marked $\mathtt{conn}(j, \mathsf{cid}^*)$ for $i = (\mathcal{P}, \mathsf{cid})$ and:
    retrieve $(\mathsf{ctr}, \mathsf{ctr}') \leftarrow \mathsf{ctrs}(i)$;
    set $c[i, \mathsf{ctr}] \leftarrow \mathsf{AEnc}(\mathsf{AEK}, (0^{|\mathsf{ctr}|+t}))$ and send $(\mathsf{cid}^*, c[i, \mathsf{ctr}])$ to $\mathcal{A}$;
    set $\mathsf{ctrs}(i) \leftarrow (\mathsf{ctr} + 1, \mathsf{ctr}')$.

On receiving $(\textsc{Send}, \mathcal{P}, \mathsf{cid}, m)$ from $\mathcal{F}_{\mathsf{cbSC}}$ for $m \in \{0,1\}^*$:
  • Retrieve rec. $(i, \mathsf{rand}, z, Z, \mathsf{AEK}, \mathsf{AEK}')$ marked $\mathtt{att}(\mathsf{cid}^*)$ for $i = (\mathcal{P}, \mathsf{cid})$ and:
    retrieve $(\mathsf{ctr}, \mathsf{ctr}') \leftarrow \mathsf{ctrs}(i)$;
    send $(\mathsf{cid}^*, \mathsf{AEnc}(\mathsf{AEK}, (\mathsf{ctr}, m)))$ $\mathcal{A}$;
    set $\mathsf{ctrs}(i) \leftarrow (\mathsf{ctr} + 1, \mathsf{ctr}')$.

On receiving $\mathcal{A}$'s message $(\mathsf{cid}^*, c^*)$ to session $\Pi_i$ for some $i = (\mathcal{P}, \mathsf{cid})$:
  • Retrieve record $\mathsf{rec} = (i, \mathsf{rand}, z, Z, \mathsf{AEK}, \mathsf{AEK}')$ and $(\mathsf{ctr}, \mathsf{ctr}') \leftarrow \mathsf{ctrs}(i)$;
  • If record $\mathsf{rec}$ is marked $\mathtt{conn}(j, \mathsf{cid}^*)$ for some $j$ then:
    retrieve record $\mathsf{rec}' = (j, [...])$ [other values in record $\mathsf{rec}'$ are ignored];
    abort if $\mathsf{rec}'$ is not marked $\mathtt{conn}(i)$ or $c^* \neq c[j, \mathsf{ctr}']$;
    else set $\mathsf{ctrs}(i) \leftarrow (\mathsf{ctr}, \mathsf{ctr}' + 1)$ and send $(\textsc{Deliver}, \mathcal{P}, \mathsf{cid}, \bot)$ to $\mathcal{F}_{\mathsf{cbSC}}$.
  • If record $\mathsf{rec}$ is marked $\mathtt{att}(\mathsf{cid}^*)$ then:
    set $(\mathsf{ctr}^*, m^*) \leftarrow \mathsf{ADec}(\mathsf{AEK}', c^*)$;
    abort if output doesn't parse correctly or if $\mathsf{ctr}^* \neq \mathsf{ctr}$
    else set $\mathsf{ctrs}(i) \leftarrow (\mathsf{ctr}, \mathsf{ctr}' + 1)$ and send $(\textsc{Deliver}, \mathcal{P}, \mathsf{cid}, m^*)$ to $\mathcal{F}_{\mathsf{cbSC}}$.
  • If record $\mathsf{rec}$ is marked with some identifier $\mathsf{cid}' \neq \mathsf{cid}^*$ then ignore this messge.

On $(\textsc{ExpireSession}, \mathcal{P}, \mathsf{cid})$:
  • Retrieve record $(i, ...)$ for $i = (\mathcal{P}, \mathsf{cid})$ and erase it.

On $\mathcal{A}$'s query $x$ to hash function $\mathsf{H}$, respond with $\mathsf{H}(x)$.
```

Figure 15: Simulator for Theorem 4.1

Game $\mathbf{G}_2$: Eliminating EMS collisions. We will call sessions Π_i and Π_j *passively matched* (by the adversary) under three conditions: (1) Π_j receives $(\mathsf{cid}^*, \mathsf{rand}^*, Z^*)$ from the network adversary $\mathcal{A}$ s.t. $(\mathsf{rand}^*, Z^*) = (\mathsf{rand}_i, Z_i)$, i.e. Π_j receives the values sent by Π_i (note that value of cid^* received by Π_j is immaterial, e.g. it does not have to match cid sent by Π_i), (2) Π_i receives $(\mathsf{cid}^*, \mathsf{rand}^*, Z^*)$ from $\mathcal{A}$ s.t. $(\mathsf{rand}^*, Z^*) = (\mathsf{rand}_j, Z_j)$, i.e. Π_i receives the values sent by Π_j (value cid^* received by Π_i is immaterial), and (3) $\mathsf{role}_i \neq \mathsf{role}_j$. Note that this condition is equivalent with the condition that $\mathsf{tr}_i = \mathsf{tr}_j$, i.e. that Π_i and Π_j compute the same transcripts, because conditions (1) and (2) enforce that they use the same (rand, Z) tuples, (3) assures that they order them in the same way in inputs to the key derivation hash H, and by the rules of $\mathbf{G}_1$ the correct ordering is necessary because the tuples sent by Π_i and Π_j cannot be the same.

Using this notation, $\mathbf{G}_2$ runs just like $\mathbf{G}_1$ except that it adds an abort if any honest session Π_i computes EMS_i s.t. $\mathsf{EMS}_i = \mathsf{EMS}_j$ for some previously finalized honest session Π_j where Π_i and Π_j are *not* passively matched. In other words $\mathbf{G}_2$ enforces that two honest sessions output the same EMS values only if they are passively matched. Since $\mathsf{EMS}_i = \mathsf{KDF}^0(\mathsf{HS}_i, \mathsf{tr}_i) = \mathsf{KDF}(\mathsf{HS}_i, \mathsf{tr}_i|0)$ and by the above argument failure of the passive-matching condition implies that $\mathsf{tr}_i \neq \mathsf{tr}_j$, a straitghforward reduction to collision resistance of KDF shows that

$$|\Pr[\mathsf{win}(\mathbf{G}_2)] - \Pr[\mathsf{win}(\mathbf{G}_1)]| \leq \mathsf{Adv}^{\mathsf{crh}}_{\mathcal{A}',\mathsf{KDF}}(\lambda),$$

where $\mathsf{Adv}^{\mathsf{crh}}_{\mathcal{A}',\mathsf{KDF}}(\lambda)$ is the bound on adversary's advantage in CRH game against function KDF for $\mathcal{A}'$ which has the computing power of $\mathcal{Z}$ and the reduction, which simply emulates game $\mathbf{G}_2$ and monitors for the H collision.

Note that by the tuple uniqueness constraint imposed by $\mathbf{G}_1$, if Π_i, Π_j are passively matched then Π_i cannot be passively matched to any other party, which implies that in $\mathbf{G}_2$ the same EMS value can occur for at most two sesions.

Game $\mathbf{G}_3$: Hash H as a random table. This game proceeds like $\mathbf{G}_2$ except for emulating RO hash H differently, using an initially empty table T_{H}. Namely, whenever Π_i needs to compute $K_i = \mathsf{H}((Z_i')^{z_i}) = \mathsf{H}(\mathsf{DH}_g(Z_i', g^{z_i}))$, where Z_i' is the DH value Π_i received from $\mathcal{A}$ and DH_g is the Diffie-Hellman function $\mathsf{DH}_g(g^a, g^b) = g^{ab}$, $\mathbf{G}_3$ first checks for any prior entry $((Z_j', z_j), h)$ in T_{H} s.t. $\mathsf{DH}_g(Z_i', g^{z_i}) = \mathsf{DH}_g(Z_j', g^{z_j})$, which holds iff $(Z_i')^{z_i} = (Z_j')^{z_j}$, in which case $\mathbf{G}_3$ returns h. Otherwise it returns a new random λ-bit value h and adds $((Z_i', z_i), h)$ to T_{H}. On $\mathcal{A}$'s direct queries K to H, the game services them the same way as $(Z_i', z_i) = (K, 1)$. Since this is only a syntactic change it follows that

$$\Pr[\mathsf{win}(\mathbf{G}_3)] = \Pr[\mathsf{win}(\mathbf{G}_2)]$$

Game $\mathbf{G}_4$: Random keys on passively-connected sessions. We will call session Π_i *passively connected* (to Π_j) if Π_i receives $(\mathsf{cid}^*, \mathsf{rand}^*, Z^*)$ from $\mathcal{A}$ s.t. $(\mathsf{rand}^*, Z^*) = (\mathsf{rand}_j, Z_j)$ for some honest session Π_j. Using this terminology Π_i, Π_j are passively matched if and only if they are both passively connected to each other and they have different roles.

Using this notation, $\mathbf{G}_4$ proceeds as $\mathbf{G}_3$ except for introducing an abort if $\mathcal{A}$ ever sends a query K to H which matches key K_i computed on some passively connected session Π_i. We argue that distinguishing between $\mathbf{G}_4$ and $\mathbf{G}_3$ implies breaking the Gap CDH assumption on group $\langle g \rangle$. It is well-known that CDH is equivalent to SquareDH, i.e. hardness of computing g^{x^2} on input (g, g^x) for random x, and the presence of DDH oracle which defines the gap version of CDH and SquareDH does not affect that. We will thus show a reduction to Gap SquareDH.

The reduction gets a SquareDH challenge (g, X) where X is a random element of $\langle g \rangle$, and it emulates $\mathbf{G}_4$ except that (1) for each honest session Π_i it sets $Z_i \leftarrow (X)^{t_i}$ for $t_i \leftarrow \mathbb{Z}_p$, i.e. all DH values sent by honest parties are randomizations of the SquareDH challenge, (2) in servicing hash H queries the reduction makes the following adjustments: First, it keeps entries in the form $((Z_i', t_i), h)$ instead of $((Z_i', z_i), h)$ where t_i is the randomization coefficient used on session Π_i, i.e. entry $((Z_i', t_i), h)$ signifies that $\mathsf{H}(\mathsf{DH}_g(Z'_i, Z_i)) = h$ for $Z_i = X^{t_i}$; Second, the direct queries are kept in the same way as before, i.e. in the form $((K, 1), h)$ if $\mathsf{H}(K) = h$; Third, when the reduction needs to check if there if an entry $((Z'_j, t_j), h)$ in T_H defines the same value as used by session Π_i, it checks if $\mathsf{DH}_g(Z'_j, X^{t_j}) = \mathsf{DH}_g(Z'_i, X^{t_i})$, by checking if $(Z'_j)^{t_j} = (Z'_i)^{t_i}$; Fourth, when the reduction needs to check if any pair (Z_i, t_i) [either in prior query or in current one] collides with explicit H query K, i.e. with pair $(K, 1)$, it does so consulting a DDH oracle on tuple (g, Z_i, X^{t_i}, K).

Thus, reduction emulates game $\mathbf{G}_4$ perfectly, except that it monitors for an event that $\mathcal{A}$ queries H on K equal to the key on a passively connected session Π_i. If session Π_i is connected to Π_j, this corresponds to the case that $K = \mathsf{DH}_g(Z_j, Z_i) = \mathsf{DH}_g(X^{t_j}, X^{t_i})$, hence $K^{1/(t_i t_j)} = \mathsf{DH}_g(X, X)$ is a solution to the SquareDH challenge. This implies

$$|\Pr[\mathsf{win}(\mathbf{G}_4)] - \Pr[\mathsf{win}(\mathbf{G}_3)]| \leq \mathsf{Adv}^{\mathsf{G-SqDH}}_{\mathcal{A}'}(\langle g \rangle),$$

where $\mathsf{Adv}^{\mathsf{G-SqDH}}_{\mathcal{A}'}(\langle g \rangle)$ is the bound on adversary's advantage against Gap SquareDH in group $\langle g \rangle$, for $\mathcal{A}'$ which has the computing power of $\mathcal{Z}$ and this reduction.

Note that since in $\mathbf{G}_4$ the adversary cannot access any key $\mathsf{HS}_i = \mathsf{H}(K_i)$ for any passively-connected session via a call to H, the game can just as well pick each such HS_i value at random independently of hash H.

Game $\mathbf{G}_5$: Random outputs on passively-connected sessions. This game emulates $\mathbf{G}_4$ except that for each passively-connected session Π_i the game does not compute values $\mathsf{EMS}_i, \mathsf{AEK}_{C,i}, \mathsf{AEK}_{S,i}$ from HS_i via KDF, but picks each of them directly as random λ-long bitstrings. (For a pair of two passively *matched* sessions, Π_i, Π_j, since their keys K_i, K_j are equal, the game picks these $\mathsf{EMS}, \mathsf{AEK}_C, \mathsf{AEK}_S$ triples as also equal on such two sessions.) Since by game $\mathbf{G}_4$ keys HS_i on passively connected sessions are all independent, an easy reduction to PRF property of KDF shows that each $(\mathsf{EMS}_i, \mathsf{AEK}_{C,i}, \mathsf{AEK}_{S,i})$ tuple can be replaced, one by one, from KDF outputs to independent random values. A hybrid over q_SC of such reductions implies that

$$|\Pr[\mathsf{win}(\mathbf{G}_5)] - \Pr[\mathsf{win}(\mathbf{G}_4)]| \leq q_\mathsf{SC} \cdot \mathsf{Adv}^{\mathsf{PRF}}_{\mathcal{A}',\mathsf{KDF}}(\lambda),$$

where $\mathsf{Adv}^{\mathsf{PRF}}_{\mathcal{A}',\mathsf{KDF}}(\lambda)$ is the bound on adversary's advantage against the PRF property of KDF, for $\mathcal{A}'$ which has the computing power of $\mathcal{Z}$ and the above reduction.

Game $\mathbf{G}_6$: Privacy on passively-connected sessions. In this game each time a passively-connected session Π_i encrypts a (counter,message) pair (ctr, m), the game instead uses Π_i's encryption key AEK_i to encrypt an all-zero string of the same length. Moreover, if the resulting ciphertext c is delivered without change to session Π_j which is *matched* with Π_i, assuming such session exists, the game does not decrypt c, but instead retrieves plaintext (ctr, m) which was supposed to be encrypted in it, and continues processiing on Π_j's side from that point on. This game is indistinguishable from $\mathbf{G}_5$ under CCA security of authenticated encryption AEK. The argument proceeds by a hybrid over each encryption key AEK_i used in the game, i.e. over all passively connected sessions Π_i. Using the standard "single challenge ciphertext" version of CCA security of symmetric encryption, for each Π_i we must also perform a hybrid over all messages sent by Π_i, i.e. over q_Snd queries SEND issued by $\mathcal{Z}$ on session Π_i. For each index $i = 1, ..., q_\mathsf{Snd}$, the reduction, interacting with CCA challenger who holds a random key AEK_i, accesses the encryption oracle using that key to service SEND queries except for the i-th one. On the i-th query the reduction gets ciphertext c as an encryption challenge, either encrypting (ctr, m) pertaining to this query, or an all-zero string. To service decryption on behalf of session Π_j which is passively matched with Π_i (if such session exists), the reduction uses the decryption oracle interface of the CCA challenger. This series of hybrids implies that

$$|\Pr[\mathsf{win}(\mathbf{G}_6)] - \Pr[\mathsf{win}(\mathbf{G}_5)]| \leq q_\mathsf{SC} \cdot q_\mathsf{Snd} \cdot \mathsf{Adv}^{\mathsf{CCA}}_{\mathcal{A}',\mathsf{AEnc}}(\lambda),$$

where $\mathsf{Adv}^{\mathsf{CCA}}_{\mathcal{A}',\mathsf{AEnc}}(\lambda)$ is the bound on adversary's advantage against the CCA security of the authenticated encryption scheme AEnc, for $\mathcal{A}'$ which has the computing power of $\mathcal{Z}$ and the above reduction.

Game $\mathbf{G}_7$: Authenticity on passively-connected sessions. In the previous game the passively-connected sessions encrypt all-zero strings instead of actual messages, and decryption of ciphertexts created in this way is skipped by sessions which are passively matched with such senders. However, passively connected sessions still use their keys AEK' to decrypt all incoming traffic except for the ciphertexts created in this special way by a matching sender. In game $\mathbf{G}_7$ we eliminate the usage of decryption keys AEK'_i for each passively connected session Π_i. Instead, reception of an incoming ciphertext c on any passively connected session Π_i is processed as follows: If c is generated by session Π_j which is matched with Π_i (assuming such session exists), and moreover c was created using (ctr, m) for counter value ctr which matches the current counter used by Π_i for incoming messages, then $\mathbf{G}_7$ passes message m as an output of the party running Π_i. In any other case, i.e. if c is not output on behalf of the (unique) session which matches Π_i (e.g. because such session doesn't exist), or if it was sent by $\mathbf{G}_7$ on behalf of such party, but $\mathcal{A}$ delivers it to Π_i out of order, the game just drops such ciphertext when it is delivered to Π_i.

Distinguishing $\mathbf{G}_6$ and $\mathbf{G}_7$ implies attacking the CUF (ciphertext unforgeability) property of the authenticated encryption scheme AEK. The argument proceeds by a hybrid over each key AEK_i, i.e. over all passively connected sessions Π_i, and a hybrid over all messages an adverasry $\mathcal{A}$ sends to session Π_i. For each index $i = 1, ..., q_{\mathsf{Rcv}}$ where q_{Rcv} is an upper-bound on the number of messages received by Π_i, the reduction queries the encryption oracle of CUF challenger for each ciphertext it needs to send (which is always an encryption of an all-zero), and it outputs the i-th ciphertext which $\mathcal{A}$ delivers to Π_i as a CUF forgery. This series of hybrids implies that

$$|\Pr[\mathsf{win}(\mathbf{G}_7)] - \Pr[\mathsf{win}(\mathbf{G}_6)]| \leq q_{\mathsf{SC}} \cdot q_{\mathsf{Rcv}} \cdot \mathsf{Adv}^{\mathsf{CUF}}_{\mathcal{A}',\mathsf{AEnc}}(\lambda),$$

where $\mathsf{Adv}^{\mathsf{CUF}}_{\mathcal{A}',\mathsf{AEnc}}(\lambda)$ is the bound on adversary's advantage against the CUF property of the authenticated encryption scheme AEnc, for $\mathcal{A}'$ which has the computing power of $\mathcal{Z}$ and the above reduction.

Game $\mathbf{G}_8$: The ideal-world execution. At this point the game is identical to the ideal-world, where $\mathcal{Z}$ interacts with dummy parties interacting with functionality $\mathcal{F}_{\mathsf{cbSC}}$ which interacts with simulator $\mathcal{S}$ described in Figure 15, where $\mathcal{S}$ in turn interacts with $\mathcal{Z}$'s interface $\mathcal{A}$ which models a real-world adversary. Just like game $\mathbf{G}_7$, the ideal-game defined by simulator $\mathcal{S}$ and functionality $\mathcal{F}_{\mathsf{cbSC}}$ creates random tuples $(\mathsf{EMS}_i, \mathsf{AEK}_i, \mathsf{AEK}'_i)$ for each passively connected session Π_i that finalizes, and copies these tuples to a session Π_j that finalizes by being passively matched with Π_i. When any such session then uses the established secure channel, the ideal-world interaction does the same as $\mathbf{G}_7$, i.e. the sender sends encryptions of all-zero strings under encryption key AEK_i, the decryption key AEK'_i is never used, and the only ciphertexts which are not ignored by the receiver are the ones sent by the matching session, and only if they are received in order. Also as in $\mathbf{G}_7$, the ideal-world interaction emulates behavior of real-world sessions which are *not* passively connected, because in both $\mathbf{G}_7$ and $\mathbf{G}_8$ such sessions effectively follow the real-world protocol. We conclude that

$$\Pr[\mathsf{win}(\mathbf{G}_8)] = \Pr[\mathsf{win}(\mathbf{G}_7)]$$

which completes the proof.

B.2 Proof of Theorem 5.1

We first give only some intuitions that readers might find helpful before reading the full proof.

For simulating channel establishment, Figure 17 visualizes the three instances of the channel functionality $\mathcal{F}_{\mathsf{cbSC}}$ within Theorem 5.1: Π_{EA} calls an instance of $\mathcal{F}_{\mathsf{cbSC}}$ *with* channel binder, while in the ideal execution with $\mathcal{F}_{\mathsf{PHA}}$, an instance of $\mathcal{F}_{\mathsf{cbSC}}$ *without* channel binder is run as part of $\mathcal{F}_{\mathsf{PHA}}$. Also, $\mathcal{S}$ maintains a copy of $\mathcal{F}_{\mathsf{cbSC}}$ *with* channel binders, mimicking the one in the real world. See Figure 17 for an illustration. Both worlds would be easily distinguishable if the channels in those three instances of $\mathcal{F}_{\mathsf{cbSC}}$ differ. Our simulation strategy is to run $\mathcal{F}_{\mathsf{cbSC}}$ with inputs obtained from $\mathcal{F}_{\mathsf{PHA}}$ (note that $\mathcal{F}_{\mathsf{PHA}}$ relays all channel-related inputs to the adversary), and it *replicates* adversarial instructions intended for the simulated $\mathcal{F}_{\mathsf{cbSC}}$ instance, removing all CB-related artifacts from them, and feeds them into $\mathcal{F}_{\mathsf{PHA}}$. As an example, if $\mathcal{A}$ decides to connect two parties with channel binder CB, $\mathcal{S}$ relays the request to $\mathcal{F}_{\mathsf{cbSC}}$ unmodified. Then, $\mathcal{S}$ immediately sends a request to connect the two parties to $\mathcal{F}_{\mathsf{PHA}}$.

We next explain simulation of the key generation. Normal key generation is trivial to simulate: $\mathcal{S}$ generates a random key pair and keeps the secret key to simulate signatures of honest parties. Simulation of transportable keys cannot use application keys ak, since $\mathcal{F}_{\mathsf{PHA}}$ keeps them secret: $\mathcal{S}$ chooses a random k for every request (instead of deriving it from a hash of ak) and create the envelope ske with such k. When $\mathcal{Z}$ wants to open ske by hashing $(\mathsf{ak}, \mathsf{nonce})$, $\mathcal{S}$ queries GetAuxData of $\mathcal{F}_{\mathsf{PHA}}$ to find out whether ak "works for" ske. If so, $\mathcal{S}$ answers the hash query with k, resolving all unmet dependencies. For this part of the proof, usage of an idealized assumption to control sk extraction from envelopes performed by $\mathcal{Z}$ seems inherent, and is why Theorem 5.1 relies on the random oracle model. We further rely on the encryption of secret keys to be unforgeable and *random-key robust* [19], i.e., it must be hard to create a ciphertext decrypting successfully under an (unknown) key even when seeing a valid ciphertext, and it must be hard to create a ciphertext that decrypts successfully under two randomly chosen keys. We need these properties to ensure that $\mathcal{F}_{\mathsf{PHA}}$ knows all "working" key pairs $\mathsf{ak}, \mathsf{ske}$. If the first property is void, $\mathcal{Z}$ could compute a forgery ae' containing some secret key, such that $(\mathsf{ak}, (\mathsf{nonce}, \mathsf{ae}'))$ would be a working $\mathtt{tk}$ key pair in the real world but an unknown (and hence, non-functional) one in an ideal world with $\mathcal{F}_{\mathsf{PHA}}$. Similarly, $\mathcal{Z}$ coming up with a ciphertext decrypting successfully under two randomly chosen keys breaks the uniqueness property of ske that $\mathcal{F}_{\mathsf{PHA}}$ enforces.

Simulation of authentication on unattacked channels is trivial: since $\mathcal{S}$ knows secret keys and channel binders, $\mathcal{S}$ holds keys for creating signatures and MACs for honest parties. The main challenge is to argue that $\mathcal{F}_{\mathsf{PHA}}$ outputs are indistinguishable from the real world, since $\mathcal{F}_{\mathsf{PHA}}$ has strict rules regarding adversarial authentication: they can only happen on attacked channels or through corrupt parties in connected channels, and only with respect to compromised or unknown public keys. Our proof relies on the unforgeability of signatures, the above described unforgeability of envelopes & replay security of the SIGMA protocol ($=\mathcal{A}$ cannot authenticate with uncompromised public keys), and secrecy of EMS & unforgeability of MACs ($=\mathcal{A}$ cannot authenticate on honest channels). The reductions are complicated by several dependencies of the simulation. For example, to reduce to the EUF-CMA security of the MAC scheme, we first need to randomize all PRF-derived MAC keys. To reduce to the EUF-CMA security of the signature scheme, $\mathcal{S}$ cannot use any signing keys, e.g., to create envelopes. This can be resolved by first switching to simulated envelopes, exploiting the equivocability of the encryption scheme. This concludes the proof sketch. We provide an overview of the hybrids in Figure 16.

Fix a probabilistic polynomial-time environment $\mathcal{Z}$, and assume that a real-world adversary is a dummy adversary, i.e., it is an interface of algorithm $\mathcal{Z}$. Let $\mathsf{win}(\mathbf{G}_i)$ stand for the event that $\mathcal{Z}$ outputs 1 after an interaction with Game $\mathbf{G}_i$.

Game $\mathbf{G}_0$: The real execution. $\mathcal{Z}$ interacts with parties running protocol Π_{EA} depicted in Figure 12, and with a dummy adversary $\mathcal{A}$.

Game $\mathbf{G}_1$: Introducing simulator and ideal functionality. We group all machines except for $\mathcal{Z}$ into one new machine and call it simulator $\mathcal{S}$. For each party, a dummy party is added between $\mathcal{Z}$ and $\mathcal{S}$. We also add a machine $\mathcal{F}$ between dummy parties and $\mathcal{S}$, and add to it NewSession, Attack, Connect, Send, Deliver, ExpireSession as in Figure 9, omitting all gray parts. $\mathcal{F}$ serves as a relay for all messages that are not send to these interfaces. Looking ahead, we will gradually change $\mathcal{F}$ until it is equal to $\mathcal{F}_{\mathsf{PHA}}$. To account for the changes introduced by $\mathcal{F}$ we detail what $\mathcal{S}$ does whenever $\mathcal{A}$ (hence, $\mathcal{Z}$) sends a message to the simulated $\mathcal{F}_{\mathsf{cbSC}}$, and whenever the new machine $\mathcal{F}$ sends a message to $\mathcal{S}$:

- ($\mathbf{G}_1$) On (Attack, $\mathcal{P}, \mathsf{cid}, \mathsf{cid}^*, \mathsf{CB}^*$) from $\mathcal{A}$, if $\mathsf{CB}^* \in \mathsf{CBset}$ then ignore the query. Otherwise send message (Attack, $\mathcal{P}, \mathsf{cid}, \mathsf{cid}^*, \mathsf{CB}^*$) to $\mathcal{F}_{\mathsf{cbSC}}$ and send message (Attack, $\mathcal{P}, \mathsf{cid}, \mathsf{cid}^*$) to $\mathcal{F}$;
- ($\mathbf{G}_1$) On (Connect, $\mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}', \mathsf{cid}^*, \mathsf{CB}^*$) from $\mathcal{A}$, if there exists a record (session, $\mathcal{P}, \mathsf{cid}, \mathsf{role}$) labeled $\mathtt{wait}$, then do:

Game	Change functionality $\mathcal{F}$	Change simulator $\mathcal{S}$	Ind. argument/assumption
$\mathbf{G}_0$	(starts with empty $\mathcal{F}$)	real execution	
$\mathbf{G}_1$	add channel interfaces	relay queries between simulated $\mathcal{F}_{\mathsf{cbSC}}$, $\mathcal{F}$ and $\mathcal{A}$	syntactical change
$\mathbf{G}_2$	-	abort upon nonce collision during key generation	Birthday bound
$\mathbf{G}_3$	-	abort upon ae collision during key generation	RKR security of $(\mathsf{AEnc}, \mathsf{ADec})$
$\mathbf{G}_4$	-	abort upon pk collision during key generation	EUF-CMA security of SIG
$\mathbf{G}_5$	add KeyGen interface	compute key material for $\mathcal{F}$	syntactical change
$\mathbf{G}_6$	-	abort if $\mathcal{Z}$ generates a working transportable key pair	CUF-CCA security of $(\mathsf{AEnc}, \mathsf{ADec})$
$\mathbf{G}_7$	-	register adversarial keys in $\mathcal{F}$	syntactical change
$\mathbf{G}_8$	add AuthSend interface	use simulated secret key material	syntactical cange
$\mathbf{G}_9$	-	abort upon honest party computing non-verifying signature	completeness of SIG
$\mathbf{G}_{10}$	-	abort upon honest party computing non-verifying MAC	completeness of MAC scheme
$\mathbf{G}_{11}$	-	simulate AEnc ciphertexts	equivocability of $(\mathsf{AEnc}, \mathsf{ADec})$
At this point, $\mathcal{S}$ does not use knowledge of secret keys sk.			
$\mathbf{G}_{12}$	-	abort upon signature forgery	EUF-CMA security of SIG
$\mathbf{G}_{13}$	-	produce real ciphertexts from sk again	equivocability of $(\mathsf{AEnc}, \mathsf{ADec})$
$\mathbf{G}_{14}$	-	randomize PRF outputs	pseudorandomness of PRF
$\mathbf{G}_{15}$	-	abort upon PRF collision	Birthday Bound
$\mathbf{G}_{16}$	-	abort upon receiving a MAC forgery	EUF-CMA sec. of MAC scheme
$\mathbf{G}_{17}$	add AuthVerify for honest channels	acknowledge AuthVerify or signal DoS attack	syntactical change
$\mathbf{G}_{18}$	add AuthVerify for attacked channels	acknowledge AuthVerify or signal DoS attack	syntactical change

Figure 16: Hybrids of the proof of Theorem 5.1.

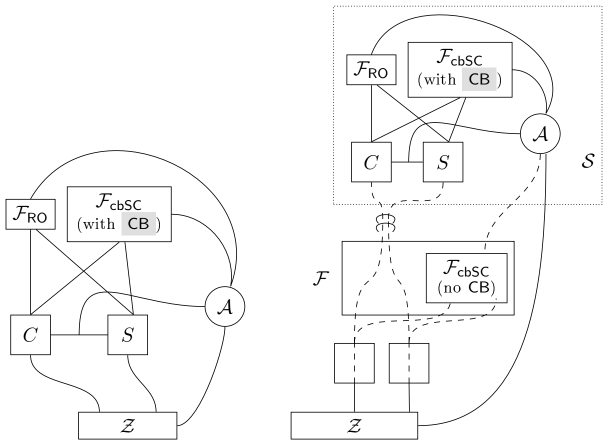
- ($\mathbf{G}_1$) if $\nexists$ record $(\text{SESSION}, \mathcal{P}', \mathsf{cid}', \mathsf{role}')$ labeled $\mathsf{conn}(\mathcal{P}, \mathsf{cid})$ and $\mathsf{CB}^* \in \mathsf{CBset}$, then ignore the query;
- ($\mathbf{G}_1$) otherwise query $\mathcal{F}_{\mathsf{cbSC}}$ with $(\text{Connect}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}', \mathsf{cid}^*, \mathsf{CB}^*)$ and send $(\text{Connect}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}', \mathsf{cid}^*)$ to $\mathcal{F}$.

- ($\mathbf{G}_1$) On $(\text{Deliver}, \mathcal{P}', \mathsf{cid}', m^*)$ from $\mathcal{A}$ forward the query to $\mathcal{F}$;
- ($\mathbf{G}_1$) On $(\text{NewSession}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{role})$ from $\mathcal{F}$, simulate party $\mathcal{P}$ inputting $(\text{NewSession}, \mathsf{cid}, \mathcal{P}', \mathsf{role})$ to $\mathcal{F}_{\mathsf{cbSC}}$;
- ($\mathbf{G}_1$) On $(\text{Send}, \cdot, \cdot)$ from $\mathcal{F}$ forward the query to $\mathcal{Z}$;
- ($\mathbf{G}_1$) On $(\text{ExpireSession}, \mathcal{P}, \mathsf{cid})$ from $\mathcal{F}$ send $(\text{ExpireSession}, \mathsf{cid})$ to $\mathcal{F}_{\mathsf{cbSC}}$.

The transition between this and the previous game is visualized in Figure 17.

We will first argue that the set of channels, i.e., the set of records $(\text{SESSION}, \dots)$, is the same in all three instances of $\mathcal{F}_{\mathsf{cbSC}}$ that appear in Figure 17: the hybrid one in game $\mathbf{G}_0$, the simulated one in game $\mathbf{G}_1$ and the one within $\mathcal{F}$ in game $\mathbf{G}_1$. First of all, CB values, which only appear in interfaces Attack, Connect, do not influence session records unless the adversary provides non-fresh CB values in Attack or in Connect and when case 2 happens. In these cases, $\mathcal{F}_{\mathsf{cbSC}}$ in $\mathbf{G}_0$ will not create a record. The simulation in $\mathbf{G}_1$ will drop the corresponding queries, hence also $\mathcal{F}$ in $\mathbf{G}_1$ will not establish any session record. Equality of SESSION records hence follows from the fact that all other information relevant to SESSION record establishment is exchanged between the $\mathcal{F}_{\mathsf{cbSC}}$ instances in $\mathbf{G}_1$: NewSession, ExpireSession inputs are forwarded as they are by $\mathcal{F}$ to simulated parties via $\mathcal{S}$, and $\mathcal{S}$ forwards all relevant adversarial queries Attack, Connect to $\mathcal{F}$. In particular, protocol Π_{EA} never lets parties create or expire channels without receiving explicit instructions (i.e., inputs) by $\mathcal{Z}$.

Since Π_{EA} relays all channel-related inputs (namely NewSession, Send and ExpireSession) to $\mathcal{F}_{\mathsf{cbSC}}$ and $\mathcal{S}$ in $\mathbf{G}_1$ forwards Deliver queries to $\mathcal{F}$, it follows from matching session records that the view of $\mathcal{Z}$ for these interfaces is equally distributed in both games. Further, the output distribution of other interfaces (KeyGen, AuthSend, AuthVerify) accessible by $\mathcal{Z}$ is equal in both games since all these



$\mathcal{F}_{\mathsf{RO}}$
$\mathcal{F}_{\mathsf{cbSC}}$ (with CB)
C
S
$\mathcal{A}$
$\mathcal{Z}$
$\mathcal{S}$
$\mathcal{F}$
$\mathcal{F}_{\mathsf{cbSC}}$ (no CB)

Figure 17: Transition from real execution (game $\mathbf{G}_0$, left) to game $\mathbf{G}_1$ (right), showing a setting with two honest parties C and S, (dummy) adversary $\mathcal{A}$ and hybrid functionalities $\mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{cbSC}}$. In game $\mathbf{G}_1$, the real execution is grouped into one machine called simulator $\mathcal{S}$. Further, the hybrid functionality $\mathcal{F}_{\mathsf{cbSC}}$ is replicated in the newly introduced machine $\mathcal{F}$ except for its CB-related parts, which remain inaccessible for the environment $\mathcal{Z}$. Empty boxes denote dummy parties, which simply relay all messages.

interfaces *only read* SESSION records but never manipulate them. This concludes our indistinguishability argument and hence we have that

$$\Pr[\mathsf{win}(\mathbf{G}_1)] = \Pr[\mathsf{win}(\mathbf{G}_0)].$$

Looking at Figure 17 we are now at a point in $\mathbf{G}_1$ where two instances of $\mathcal{F}_{\mathsf{cbSC}}$ exist: one instance in $\mathcal{F}$ which can be used by $\mathcal{Z}$ for establishing channels and sending messages over them, and one instance maintained by $\mathcal{S}$ which replicates all channels that exist in the other instance. It is crucial for our proof that $\mathcal{S}$ has consistent views on all channels despite the fact that $\mathcal{F}_{\mathsf{PHA}}$ handles them.

Game $\mathbf{G}_2$: Rule out nonce collisions in KeyGen. We change $\mathcal{S}$ as follows. Upon (KeyGen, $\mathsf{kid}, \mathsf{ak}, \mathsf{aux}, \mathtt{tk}$), $\mathcal{S}$ aborts if it picks a nonce that was already picked before or that was input by $\mathcal{Z}$ via AuthSend value $\mathsf{ske} = (\mathsf{nonce}, *)$. By the Birthday Bound we have

$$|\Pr[\mathsf{win}(\mathbf{G}_2)] - \Pr[\mathsf{win}(\mathbf{G}_1)]| \leq \frac{q(q-1)}{2 \cdot 2^{-\lambda}},$$

with $q = q_{\mathsf{KG}} + q_{\mathsf{AS}}$ being the number of KeyGen and AuthSend queries issued by $\mathcal{Z}$.

Game $\mathbf{G}_3$: Handling ske collisions. We let $\mathcal{S}$ abort whenever a simulated party generates a non-fresh ae. Since nonces are fresh as of the previous game, this can only occur if $\mathsf{AEnc}()$ produces a ciphertext that decrypts faithfully under two different, randomly chosen keys, which is ruled out by the RKR security of the encryption scheme.

Game $\mathbf{G}_4$: Handling public key collisions. We let $\mathcal{S}$ abort whenever a simulated party, upon input KeyGen, generates a key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KG}(1^\lambda)$ where pk has been part of a KEY output or AuthSend input already. We call this event $\mathsf{BAD}_{\mathsf{pk}}$ and show that it happens only with negligible probability if $\mathsf{SIG} = (\mathsf{KG}, \mathsf{Sign}, \mathsf{Vfy})$ is a CMA-secure signature scheme. Consider the following EUF-CMA attacker $\mathcal{B}$ against SIG. The EUF-CMA challenger picks $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KG}(1^\lambda)$ and hands pk to $\mathcal{B}$. $\mathcal{B}$ picks $(\mathsf{sk}', \mathsf{pk}') \leftarrow \mathsf{KG}(1^\lambda)$ and produces his forgery with signing key sk'. If $\mathsf{BAD}_{\mathsf{pk}}$ occurs we have $\mathsf{pk} = \mathsf{pk}'$ and the forgery verifies under pk. Hence,

$$|\Pr[\mathsf{win}(\mathbf{G}_4)] - \Pr[\mathsf{win}(\mathbf{G}_3)]| \leq \Pr[\mathsf{BAD}_{\mathsf{pk}}] \leq \mathsf{Adv}^{\mathsf{euf-cma}}_{\mathcal{B},\mathsf{SIG}}(1^\lambda).$$

Game $\mathbf{G}_5$: Adding KeyGen interfaces to $\mathcal{F}$. We add the interfaces KeyGen, GetAuxData and Compromise of $\mathcal{F}_{\mathsf{PHA}}$ to $\mathcal{F}$. We change $\mathcal{S}$ as follows: upon receiving $(\textsc{key}, \mathsf{kid}, \mathcal{P}, \mathsf{aux}, \mathtt{std})$ from $\mathcal{F}$, we let $\mathcal{S}$ provide the simulated $\mathcal{P}$ with input $(\textsc{KeyGen}, \mathsf{kid}, \varepsilon, \mathsf{aux}, \mathtt{std})$; when $\mathcal{P}$ computes key pair $(\mathsf{pk}, \mathsf{sk})$ and outputs $(\textsc{KeyGen}, \mathsf{kid}, \varepsilon, \varepsilon, \mathsf{pk})$, we let $\mathcal{S}$ send query $(\mathsf{kid}, \varepsilon, \mathsf{pk})$ to $\mathcal{F}$.

Upon receiving $(\textsc{KeyGen}, \mathsf{kid}, \mathsf{pid}, \mathsf{aux}, \mathtt{tk})$ from $\mathcal{F}$ for any pid, including $\mathcal{A}$, simulator $\mathcal{S}$ picks $k \leftarrow \{0,1\}^\lambda$, computes $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KG}(1^\lambda)$, $\mathsf{ae} \leftarrow \mathsf{AEnc}_k(\mathsf{aux}, \mathsf{sk})$, $\mathsf{ske} \leftarrow (\mathsf{nonce}, \mathsf{ae})$ and stores all elements of this KeyGen request (except for the unknown ak) in a record $(k, \mathsf{sk}, \mathsf{ske}, \mathsf{aux}, \mathsf{pk})$. $\mathcal{S}$ sends $(\mathsf{kid}, \mathsf{ske}, \mathsf{pk})$ to $\mathcal{F}$. Upon $(\textsc{Compromise}, \mathcal{P})$ from $\mathcal{Z}$ to $\mathcal{A}$, $\mathcal{S}$ queries $\mathcal{F}$ with $(\textsc{Compromise}, \mathcal{P})$.

On $\mathcal{F}_{\mathsf{RO}}$ query $\mathsf{H}(\mathsf{ak}, \mathsf{nonce})$, if there is no corresponding record in T_{H}, simulator $\mathcal{S}$ additionally looks for a record $(k, *, \mathsf{ske}, \mathsf{aux}, \mathsf{pk})$ with $\mathsf{ske} = (\mathsf{nonce}, *)$. For every such record, $\mathcal{S}$ sends $\textsc{GetAuxData}(\mathsf{ak}, \mathsf{ske})$ to $\mathcal{F}$. Upon reply $(\mathsf{aux}', \mathsf{pk}')$, if $\mathsf{pk} = \mathsf{pk}'$ and $\mathsf{aux} = \mathsf{aux}'$, then $\mathcal{S}$ adds $((\mathsf{ak}, \mathsf{nonce}), k)$ to T_{H} and replies with k. If no such match is found among all records with nonce, $\mathcal{S}$ draws a fresh k, adds $(\mathsf{ak}, \mathsf{nonce}), k)$ to T_{H} and replies with k.

Due to uniqueness of nonces, k is distributed as before (every new KeyGen request has a randomly sampled $k \leftarrow \{0,1\}^\lambda$). In $\mathbf{G}_5$, $\mathcal{F}$ aborts upon receiving a non-fresh pk or ske from $\mathcal{S}$, which does not make a difference since $\mathcal{S}$ aborted already as of $\mathbf{G}_4$ upon event $\mathsf{BAD}_{\mathsf{pk}}$, and upon ske collision in $\mathbf{G}_3$. Apart from pkReg, the newly introduced records in $\mathcal{F}$ only have an effect when honest parties query $(\textsc{GetAuxData}, \mathsf{ak}, \mathsf{ske})$. Since in $\mathbf{G}_4$, such party outputs $(\mathsf{aux}, \mathsf{pk})$ which is stored in $\mathsf{tkey}[\mathsf{ak}, \mathsf{ske}]$ by $\mathcal{F}$ in $\mathbf{G}_5$, the outputs of the GetAuxData interface of honest parties is equally distributed in both games. Hence, the changes are only syntactical and we have

$$\Pr[\mathsf{win}(\mathbf{G}_5)] = \Pr[\mathsf{win}(\mathbf{G}_4)].$$

Game $\mathbf{G}_6$: Dealing with ciphertext authenticity breaks. We change $\mathcal{S}$ upon input AuthSend relayed by $\mathcal{F}$ with values $\mathsf{ak}, \mathsf{ske}, \mathsf{pk}$, and where $\mathsf{ske} = (*, \mathsf{ae})$ and there is a key record $(k, *, (\mathsf{nonce}', \mathsf{ae}'), *, \mathsf{pk})$ with $\mathsf{ae} \neq \mathsf{ae}'$: in case $(\mathsf{ak}, \mathsf{nonce}') \notin T_{\mathsf{H}}$ (i.e., environment $\mathcal{Z}$ does not know k, sk) and $\mathsf{ADec}_k(\mathsf{ae}) \neq \perp$ then $\mathcal{S}$ aborts. We call this event $\mathsf{BAD}_{\mathsf{authEnc}}$ and show that it happens only with negligible probability if the encryption scheme is CUF-CCA secure.

Since we only let $\mathcal{S}$ inspect values and abort if indicated, this and the previous game only differ if event $\mathsf{BAD}_{\mathsf{authEnc}}$ occurs. We construct a CUF-CCA adversary $\mathcal{B}$ against the authenticated encryption scheme $(\mathsf{AEnc}, \mathsf{ADec})$ as follows. $\mathcal{B}$ implements $\mathcal{F}$ and $\mathcal{S}$ of $\mathbf{G}_6$ for a distinguisher $\mathcal{Z}$, and picks a KeyGen query of $\mathcal{Z}$ at random. Let $\mathsf{ak}, \mathsf{ske}' = (\mathsf{nonce}', \mathsf{ae}'), \mathsf{sk}, \mathsf{aux}, \mathsf{pk}$ denote values corresponding to this special KeyGen query. For this query, $\mathcal{B}$ does not choose a key k but uses his encryption oracle to produce $\mathsf{ae}' \leftarrow \mathcal{O}_{\mathsf{AEnc}_k(\cdot)}(\mathsf{aux}, \mathsf{sk})$. $\mathcal{B}$ leaves the key field (value k) in the corresponding record empty. In case $\mathcal{Z}$ wants to see the key by querying $\mathsf{H}(\mathsf{ak}, \mathsf{nonce})$ then $\mathcal{B}$ aborts. $\mathcal{B}$ waits for $\mathcal{Z}$ issuing an AuthSend query that triggers event $\mathsf{BAD}_{\mathsf{authEnc}}$ as defined above, i.e., sends $(\textsc{AuthSend}, *, *, , *, *, \mathsf{ak}, \mathsf{ske}, \mathsf{pk}, *, \mathtt{tk})$, where $\mathsf{ske} = (\mathsf{nonce}, \mathsf{ae})$ and there is a key record $(\varepsilon, *, (\mathsf{nonce}', \mathsf{ae}'), *, \mathsf{pk})$ with $\mathsf{ae} \neq \mathsf{ae}'$, and $(\mathsf{ak}, \mathsf{nonce}') \notin T_{\mathsf{H}}$, and $\mathcal{B}$'s decryption oracle returns a value other that $\perp$ on input ae. If this happens, then $\mathcal{B}$ sends ae from that query as forgery to the CUF-CCA challenger.

In case of $\mathsf{BAD}_{\mathsf{authEnc}}$, $\mathsf{ADec}_k(\mathsf{ae}) \neq \perp$ for the same k that is implemented by the oracle $\mathcal{O}_{\mathsf{AEnc}_k(\cdot)}$. Due to the random choice of KeyGen request, the probability that $\mathcal{B}$ does not abort is at least $1/q$, where q_{KG} is the number of KeyGen queries issued by $\mathcal{Z}$. Overall, we have

$$|\Pr[\mathsf{win}(\mathbf{G}_6)] - \Pr[\mathsf{win}(\mathbf{G}_5)]| \leq q_{\mathsf{KG}} \mathsf{Adv}^{\mathsf{cuf-cca}}.$$

We note that $\mathbf{G}_6$ excludes that $\mathcal{Z}$ transforms a previously received ae' into some ae that decrypts to something meaningful under the same key k. With this, we can now be sure that $\mathcal{Z}$ cannot create a working key handle for an uncompromised pk. The next game will deal with $\mathcal{Z}$ creating working key handles for compromised pk.

Game $\mathbf{G}_7$: Registering adversarial keys in $\mathcal{F}$. We change $\mathcal{S}$ upon getting relayed input AUTHSEND with with $\mathtt{mode} = \mathtt{tk}$ and value $\mathsf{ske}, \mathsf{ak}$. $\mathcal{S}$ parses $(\mathsf{nonce}, \mathsf{ae}) \leftarrow \mathsf{ske}$. In case of $(\mathsf{ak}, \mathsf{nonce}) \notin T_\mathsf{H}$ then $\mathcal{S}$ does nothing. Otherwise, $\mathcal{S}$ finds $((\mathsf{ak}, \mathsf{nonce}), k) \in T_\mathsf{H}$ and computes $(\mathsf{aux}, \mathsf{sk}) \leftarrow \mathsf{ADec}_k(\mathsf{ae})$, and sk from pk. If $(\mathsf{sk}, \mathsf{pk})$ is a valid output of $\mathsf{SIG.KG}$, $\mathcal{S}$ stores key record $(k, \mathsf{sk}, (\mathsf{nonce}, \mathsf{ae}), \mathsf{aux}, \mathsf{pk})$, chooses a random kid and queries $\mathcal{F}$ with $(\text{KEYGEN}, \mathsf{kid}, \mathsf{ak}, \mathsf{aux}, \mathtt{tk})$ and $(\mathsf{kid}, \mathsf{ske}, \mathsf{pk})$.

Since key records in $\mathcal{S}$ and $\mathsf{tkey}[], \mathsf{pkComp}$ lists in $\mathcal{F}$ do not have any effect yet, the changes in this game are only syntactical and hence $\Pr[\mathsf{win}(\mathbf{G}_7)] = \Pr[\mathsf{win}(\mathbf{G}_6)]$.

We note that public keys registered in this game will automatically be added to pkComp by $\mathcal{F}$. Further, as of this game, key records $(*, *, *, *, \mathsf{pk})$ are no longer unique w.r.t pk or nonces. The same holds for entries of the list $\mathsf{tkey}[]$ of $\mathcal{F}$. Indeed, for a compromised pk (i.e., either a registered one where $\mathcal{Z}$ hashed to obtain k, to subsequently decrypt ae to sk, or one that $\mathcal{Z}$ created himself), $\mathcal{Z}$ can create many valid $\mathtt{tk}$ key handles $\mathsf{ak}, \mathsf{ske}$ by computing encryption keys through hashing arbitrary pairs $(\mathsf{ak}, \mathsf{nonce})$. With this game, we ensured that $\mathcal{F}$ is aware of all these valid key handles. Lastly, we note that $\mathcal{S}$ still maintains unique key records for uncompromised pk.

Game $\mathbf{G}_8$: Add the AUTHSEND interface to $\mathcal{F}$. We add interface AUTHSEND of $\mathcal{F}_\mathsf{PHA}$ to $\mathcal{F}$. To account for this change, we adapt $\mathcal{S}$ as follows. First, $\mathcal{S}$ runs the code newly introduced in $\mathbf{G}_7$ to register a maliciously generated transportable key not upon getting relayed AUTHSEND, but upon $(\mathsf{ak}, \mathsf{ske}, \varepsilon)$ from $\mathcal{F}$. Second, upon receiving $(\text{AUTHSEND}, \mathcal{P}, \mathcal{P}', \mathsf{cid}, \mathsf{ssid}, \mathsf{ctx}, \mathsf{pk}, \mathtt{mode}, b)$ from $\mathcal{F}$, if $\mathtt{mode} = \mathtt{std}$, $\mathcal{S}$ sends input $(\text{AUTHSEND}, \mathcal{P}', \mathsf{cid}, \mathsf{ssid}, \mathsf{ctx}, \varepsilon, \varepsilon, \mathsf{pk})$ to simulated $\mathcal{P}$. Otherwise we have $\mathtt{mode} = \mathtt{tk}$ and $\mathcal{S}$ already received values $\mathsf{ak}, \mathsf{ske}$ before. If $\mathcal{S}$ finds a record $(k, \mathsf{sk}, \mathsf{ske}, \mathsf{aux}, \mathsf{pk})$, then $\mathcal{S}$ skips the first part of KEYGEN in the simulation of $\mathcal{P}$ and jumps right to signature generation, and using sk. Otherwise, no such record exists. In this case, $\mathcal{S}$ sends input $(\text{AUTHSEND}, \mathcal{P}', \mathsf{cid}, \mathsf{ssid}, \mathsf{ctx}, \mathsf{ak}, \mathsf{ske}, \varepsilon)$ to simulated $\mathcal{P}$.

Regarding indistinguishability, AUTH records do not have any effect yet in $\mathcal{F}$ and thus we only have to analyze whether (a) the information kept from $\mathcal{S}$ by AUTHSEND causes a difference in the output distribution, and (b) the code of $\mathbf{G}_7$ only has an effect in case $\mathcal{F}$ sends $(\mathsf{ak}, \mathsf{ske}, \mathsf{pk})$.

Regarding (a), in case of a malicious transportable key (the "otherwise" case above), $\mathcal{F}$ separately informs $\mathcal{S}$ about $(\mathsf{ak}, \mathsf{ske})$ and hence no information is kept from $\mathcal{S}$. The same holds in the $\mathtt{mode} = \mathtt{std}$ case, since $\mathcal{P}$ does not access $\mathsf{ak}, \mathsf{ske}$ inputs in this mode. Hence, the only interesting case is $\mathtt{mode} = \mathtt{tk}$ where pk is a public key that was output by a KEYGEN query in $\mathtt{tk}$ mode. In this case, $\mathcal{F}$ keeps $\mathsf{ak}, \mathsf{ske}$ private. However, since pk was registered through KEYGEN, $\mathcal{S}$ finds the (not necessarily unique) record $(*, \mathsf{sk}, *, *, \mathsf{pk})$ with a secret key sk that signs correctly for pk. Regarding (b), we distinguish two cases upon $\mathcal{S}$ storing key record $(k, \mathsf{sk}, (\mathsf{nonce}, \mathsf{ae}), \mathsf{aux}, \mathsf{pk})$ in $\mathbf{G}_7$: (1) the exact same record already existed, (2) either a different record or no record for pk existed. Case (1) corresponds to $\mathcal{Z}$ using formerly generated $\mathtt{tk}$ key material, and hence the KEYGEN query of $\mathcal{S}$ did not have effect in $\mathbf{G}_7$ since pk was already in pkReg due to KEYGEN and in pkComp due to $\mathcal{S}$ issueing GETAUXDATA upon query $\mathsf{H}(\mathsf{ak}, \mathsf{nonce})$. Case (2) corresponds to $\mathcal{Z}$ using $\mathtt{tk}$ key material that $\mathcal{Z}$ computed locally. Hence, $\mathsf{tkey}[\mathsf{ak}, \mathsf{ske}]$ is not set yet and thus $\mathcal{F}$ sends $(\mathsf{ak}, \mathsf{ske}, \varepsilon)$ to $\mathcal{S}$.

Altogether, this shows that the changes in $\mathbf{G}_8$ are only syntactical and we have

$$\Pr[\mathsf{win}(\mathbf{G}_8)] = \Pr[\mathsf{win}(\mathbf{G}_7)].$$

Game $\mathbf{G}_9$: Completeness of the signature scheme. We change $\mathcal{S}$ as follows: in case $(\text{AUTHSEND}, \mathcal{P}, \mathcal{P}''$, $\mathsf{cid}, \mathsf{ssid}, \mathsf{ctx}, \mathsf{pk}, \mathtt{mode}, 1)$ is received from $\mathcal{F}$ and cid connects $\mathcal{P}, \mathcal{P}'$, if the simulated $\mathcal{P}$, in session ssid, computes a signature that is not valid for message $(\mathsf{PRF}(\mathsf{EMS}, \mathsf{lbl}_{\mathsf{HSC,clt}}), \mathsf{ssid}, \mathsf{ctx}, \mathsf{pk})$ under pk, with EMS output by $\mathcal{F}_\mathsf{cbSC}$ to $\mathcal{P}, \mathcal{P}'$ for the corresponding channel, then $\mathcal{S}$ aborts.

Outputs in $\mathbf{G}_9$ and $\mathbf{G}_8$ are equally distributed unless the simulator aborts in $\mathbf{G}_9$. Since in both modes, $\mathcal{P}$ uses sk with $(\mathsf{sk}, \mathsf{pk})$ output by KeyGen, the abort is ruled out by the perfect completeness of the signature scheme, and we have

$$\Pr[\mathsf{win}(\mathbf{G}_9)] = \Pr[\mathsf{win}(\mathbf{G}_8)].$$

Game $\mathbf{G}_{10}$: Completeness of MAC scheme. Similar to $\mathbf{G}_9$, if $\mathcal{S}$ receives message (AuthSend, $\mathcal{P}, \mathcal{P}''$, cid, $\mathsf{ssid}, \mathsf{ctx}, \mathsf{pk}, \mathtt{mode}, 1$) from $\mathcal{F}$, where cid connects $\mathcal{P}, \mathcal{P}'$, if the simulated $\mathcal{P}$, in session ssid, computes a MAC for message $(\mathsf{PRF}(\mathsf{EMS}, \mathsf{lbl}_{\mathsf{HSC,clt}}), \mathsf{ssid}, \mathsf{ctx}, \mathsf{pk}, \sigma)$ that does not verify under MAC key $\mathsf{PRF}(\mathsf{EMS}, \mathsf{lbl}_{\mathsf{MK,clt}})$, then $\mathcal{S}$ aborts.

Distributions are equal unless $\mathcal{S}$ aborts, which is ruled out by perfect completeness of the MAC scheme and hence

$$\Pr[\mathsf{win}(\mathbf{G}_{10})] = \Pr[\mathsf{win}(\mathbf{G}_9)].$$

From here on we can be sure that if an Auth record was created by $\mathcal{F}$ while processing AuthSend, *and* if $\mathcal{A}$ delivers the last protocol message faithfully, then the verifying party will output success.

Game $\mathbf{G}_{11}$: Simulate ciphertexts. We define a set of hybrid games $\mathbf{G}_{10} = \mathbf{G}_{11.0}, \mathbf{G}_{11.1}, \ldots, \mathbf{G}_{11.q} = \mathbf{G}_{11}$, where q is an upper bound on the number of KeyGen requests issued by $\mathcal{Z}$. For $i = 0, \ldots, q$, in game $\mathbf{G}_{11.i}$ we change KeyGen as follows. If this is the i-th KeyGen query, we do not let $\mathcal{S}$ pick an encryption key but use the simulator $\mathsf{SIM}_{\mathsf{eq}}$ to compute $\mathsf{ae} \leftarrow \mathsf{SIM}_{\mathsf{eq}}(|\mathsf{sk}|)$ and leave the encryption key (value k) field empty in the record. If $\mathcal{Z}$ wants to see the encryption key via hashing, we use the simulator to find $k \leftarrow \mathsf{SIM}_{\mathsf{eq}}(\mathsf{sk})$ to ensure that $\mathsf{ADec}_k(\mathsf{ae}) = \mathsf{sk}$. We store those values in T_{H} and reply with them. For clarity, the simulation of KeyGen and hash queries is depicted in fig. 19.

Replacement of k by $\perp$ in records has no effect as this part of the record is never accessed from the rest of the code of $\mathcal{S}$ in $\mathbf{G}_{10}$. To see this, we depict the full simulator's code of $\mathbf{G}_{10}$ in fig. 18. The full code of the simulator in $\mathbf{G}_{11}$ is obtained by replacing KeyGen and hash interface in fig. 18 with the ones in fig. 19. For $\mathcal{S}$'s code in both this and the previous game, we can easily see from the code that, besides KeyGen and hash, the only interface which accesses the key record is AuthSend. In AuthSend, only pk, sk parts of the record are looked at, and hence the output distribution of $\mathcal{S}$ is not affected by storing $k \leftarrow \perp$ in the key record. Therefore, a distinguishing environment $\mathcal{Z}$ between $\mathbf{G}_{11.i}$ and $\mathbf{G}_{11.i+1}$ can efficiently distinguish real from simulated ciphertexts and hence yields an efficient attacker breaking equivocability of $(\mathsf{AEnc}, \mathsf{ADec})$. We thus have

$$|\Pr[\mathsf{win}(\mathbf{G}_{11})] - \Pr[\mathsf{win}(\mathbf{G}_{10})]| \leq q\mathsf{Adv}^{\mathsf{eq}}.$$

Game $\mathbf{G}_{12}$: Dealing with signature forgeries. We change the simulation upon receiving message $(\mathsf{ssid}, \mathsf{cid}, \sigma, *)$ from $\mathcal{A}$ as a message to an honest $\mathcal{P}$, where $(\mathsf{ssid}, \mathsf{cid}, *, *)$ was never sent by another honest party to $\mathcal{P}$, and $\mathcal{S}$ either has key record $(\perp, *, *, *, \mathsf{pk})$ (as of $\mathbf{G}_{11}$, $k = \perp$ means sk is unknown to $\mathcal{Z}$), or pk was output of a (KeyGen, $*, *, *, \mathtt{std}$) request to some honest party. If $\mathsf{Vfy}_{\mathsf{pk}}(\sigma, (\mathsf{HSC}, \mathsf{ssid}, \mathsf{ctx})) = 1$, where

- $\mathsf{HSC} = \mathsf{PRF}(\mathsf{EMS}, \mathsf{lbl}_{\mathsf{HSC,role}})$, where EMS is taken from a former output (Finalize, $\mathsf{cid}, \mathsf{cid}', \mathsf{role}', \mathsf{EMS}$) of $\mathcal{F}_{\mathsf{cbSC}}$ to $\mathcal{P}$,
- $\mathsf{role} \neq \mathsf{role}'$,

then $\mathcal{S}$ aborts.

The output distribution is only affected if $\mathcal{S}$ aborts. We call this event $\mathsf{BAD}_{\mathsf{sforgery}}$ and show that it happens only with negligible probability if the signature scheme is EUF-CMA secure.

Let $\mathcal{B}$ denote an attacker in a CMA security game. $\mathcal{B}$ emulates game $\mathbf{G}_{12}$ as follows: $\mathcal{B}$ obtains pk from the CMA challenger and randomly picks a query (KeyGen, $\mathsf{kid}, \mathsf{ak}, \mathsf{aux}, \mathtt{mode}$) of $\mathcal{Z}$. If $\mathtt{mode} = \mathtt{std}$, $\mathcal{B}$

The simulator talks to $\mathcal{F}$ and $\mathcal{Z}$ as only external entities. Communication with $\mathcal{Z}$ goes via simulated dummy adversary $\mathcal{A}$. Hence, in the simulation below, a message by $\mathcal{A}$ comes directly from $\mathcal{Z}$, and messages sent to $\mathcal{A}$ are assumed to immediately be forwarded to $\mathcal{Z}$. See fig. 17, right-hand side, for a picture of the setting. $\mathcal{S}$ initializes empty sets $\mathsf{nonceFresh}, \mathsf{keReg}$.

Channels

$\mathcal{S}$ maintains an instance of $\mathcal{F}_{\mathsf{cbSC}}$ as in fig. 9.

- ($\mathbf{G}_1$) On $(\textsc{Attack}, \mathcal{P}, \mathsf{cid}, \mathsf{cid}^*, \mathsf{CB}^*)$ from $\mathcal{A}$, if $\mathsf{CB}^* \in \mathsf{CBset}$ ignore the query. Otherwise send message $(\textsc{Attack}, \mathcal{P}, \mathsf{cid}, \mathsf{cid}^*, \mathsf{CB}^*)$ to $\mathcal{F}_{\mathsf{cbSC}}$ and message $(\textsc{Attack}, \mathcal{P}, \mathsf{cid}, \mathsf{cid}^*)$ to $\mathcal{F}$;
- ($\mathbf{G}_1$) On $(\textsc{Connect}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}', \mathsf{CB}^*)$ from $\mathcal{A}$, if $\exists$ record $(\textsc{session}, \mathcal{P}, \mathsf{cid}, \mathsf{role})$ labeled $\mathtt{wait}$, then do:
  - ($\mathbf{G}_1$) if $\nexists$ record $(\textsc{session}, \mathcal{P}', \mathsf{cid}', \mathsf{role}')$ labeled $\mathtt{conn}(\mathcal{P}, \mathsf{cid})$ and $\mathsf{CB}^* \in \mathsf{CBset}$, then ignore the query;
  - ($\mathbf{G}_1$) otherwise, send message $(\textsc{Connect}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}', \mathsf{CB}^*)$ to $\mathcal{F}_{\mathsf{cbSC}}$ and message $(\textsc{Connect}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}')$ to $\mathcal{F}$.
- ($\mathbf{G}_1$) On $(\textsc{Deliver}, \mathcal{P}', \mathsf{cid}', m^*)$ from $\mathcal{A}$ forward the query to $\mathcal{F}$;
- ($\mathbf{G}_1$) On $(\textsc{NewSession}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{role})$ from $\mathcal{F}$, send input $(\textsc{NewSession}, \mathsf{cid}, \mathcal{P}', \mathsf{role})$ to $\mathcal{F}_{\mathsf{cbSC}}$;
- ($\mathbf{G}_1$) On $(\textsc{Send}, \cdot, \cdot)$ from $\mathcal{F}$ forward the query to $\mathcal{A}$;
- ($\mathbf{G}_1$) On $(\textsc{ExpireSession}, \mathcal{P}, \mathsf{cid})$ from $\mathcal{F}$ send input $(\textsc{ExpireSession}, \mathsf{cid})$ to $\mathcal{F}_{\mathsf{cbSC}}$.

Hash queries:

($\mathbf{G}_1$) On input m to $\mathcal{F}_{\mathsf{RO}}$, parse $m = (\mathsf{ak}, \mathsf{nonce})$.

- ($\mathbf{G}_1$) If there is a pair $((\mathsf{ak}, \mathsf{nonce}), \tilde{h}) \in T_{\mathsf{H}}$ then reply with $\tilde{h}$;
- ($\mathbf{G}_5$) For every key record $(k, *, (\mathsf{nonce}, \mathsf{ae}), \mathsf{aux}, \mathsf{pk})$, send $(\textsc{GetAuxData}, \mathsf{ak}, (\mathsf{nonce}, \mathsf{ae}))$ to $\mathcal{F}$. Upon reply $(\mathsf{aux}', \mathsf{pk}')$, if $\mathsf{pk}' = \mathsf{pk}$ and $\mathsf{aux} = \mathsf{aux}'$, then add $((\mathsf{ak}, \mathsf{nonce}), k)$ to T_{H} and reply with k; else, set $h \leftarrow \{0,1\}^\lambda$, store $(m, h) \in T_{\mathsf{H}}$ and reply with h.
- ($\mathbf{G}_1$) If none of the above cases apply, then choose uniformly $h \leftarrow \{0,1\}^\lambda$, store $(m, h) \in T_{\mathsf{H}}$ and reply with h.

Key generation and authentication:

On input $(\textsc{KeyGen}, \mathsf{kid}, \mathsf{ak}, \mathsf{aux}, \mathtt{mode})$

- ($\mathbf{G}_1$) compute $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KG}(1^\lambda)$ ($\mathbf{G}_4$) and abort if pk not fresh.
- ($\mathbf{G}_1$) If $\mathtt{mode} = \mathtt{tk}$ then set $\mathsf{nonce} \leftarrow \{0,1\}^\lambda$ and ($\mathbf{G}_2$) abort if $\mathsf{nonce} \in \mathsf{nonceFresh}$. Add nonce to $\mathsf{nonceFresh}$.
- ($\mathbf{G}_5$) Set $k \leftarrow \{0,1\}^\lambda$, ($\mathbf{G}_1$) $\mathsf{ae} \leftarrow \mathsf{AEnc}_k(\mathsf{aux}, \mathsf{sk})$ and ($\mathbf{G}_3$) abort if $\mathsf{ae} \in \mathsf{keReg}$. Compute $\mathsf{ske} \leftarrow (\mathsf{nonce}, \mathsf{ae})$, ($\mathbf{G}_5$) store $(k, \mathsf{sk}, \mathsf{ske}, \mathsf{aux}, \mathsf{pk})$ and send $(\mathsf{kid}, \mathsf{ske}, \mathsf{pk})$ to $\mathcal{F}$.
- ($\mathbf{G}_5$) If $\mathtt{mode} = \mathtt{std}$ send $(\mathsf{kid}, \varepsilon, \mathsf{pk})$ to $\mathcal{F}$.

On input $(\textsc{Compromise}, \mathcal{P})$ from $\mathcal{Z}$ to $\mathcal{A}$:

- ($\mathbf{G}_5$) Send $(\textsc{Compromise}, \mathcal{P})$ to $\mathcal{F}$.

On $(\mathsf{ssid}, \mathsf{ak}, \mathsf{ske})$ from $\mathcal{F}$:

- ($\mathbf{G}_7$) Parse $(\mathsf{nonce}, \mathsf{ae}) \leftarrow \mathsf{ske}$. If $((\mathsf{ak}, \mathsf{nonce}), k) \in T_{\mathsf{H}}$ for some k, set $(\mathsf{aux}, \mathsf{sk}) \leftarrow \mathsf{ADec}_k(\mathsf{ae})$ and compute pk from sk. If $(\mathsf{sk}, \mathsf{pk})$ is a valid output of $\mathsf{KG}(1^\lambda)$, then store $(k, \mathsf{sk}, (\mathsf{nonce}, \mathsf{ae}), \mathsf{pk})$, choose fresh kid and send $(\textsc{KeyGen}, \mathsf{kid}, \mathsf{ak}, \mathsf{aux}, \mathtt{tk})$ to $\mathcal{F}$. Upon $(\textsc{KeyGen}, \mathsf{kid}, \mathcal{A}, \mathsf{aux}, \mathtt{tk})$ from $\mathcal{F}$, reply with $(\mathsf{kid}, \mathsf{ske}, \mathsf{pk})$.

On $(\textsc{AuthSend}, \mathcal{P}, \mathcal{P}'', \mathsf{cid}, \mathsf{ssid}, \mathsf{ctx}, \mathsf{pk}, \mathtt{mode}, b)$ from $\mathcal{F}$:

- If $\mathtt{mode} = \mathtt{std}$ then do:
  - ($\mathbf{G}_8$) Resume simulation of $\mathcal{P}$ with input $(\textsc{AuthSend}, \mathcal{P}'', \mathsf{cid}, \mathsf{ssid}, \mathsf{ctx}, \varepsilon, \varepsilon, \mathsf{pk})$ as in Π_{EA};
  - ($\mathbf{G}_9$) if $b = 1$ and the signature created by $\mathcal{P}$ for some m does not verify under pk, then abort;
  - ($\mathbf{G}_{10}$) if $b = 1$ and the MAC computed by $\mathcal{P}$ does not verify under key $\mathsf{PRF}(\mathsf{EMS}, \mathsf{lbl}_{\mathsf{MK,clt}})$, then abort.
- If $\mathtt{mode} = \mathtt{tk}$, then do:
  - If $\exists$ record $(*, \mathsf{sk}, *, \mathsf{pk})$, then
    - ($\mathbf{G}_8$) Resume simulation of $\mathcal{P}$ at signature generation using $(\mathsf{sk}, \mathsf{pk})$ and message $m \leftarrow (\mathsf{PRF}(\mathsf{EMS}, \mathsf{lbl}_{\mathsf{HSC,role}}), \mathsf{ssid}, \mathsf{ctx})$, where EMS corresponds to cid;
    - ($\mathbf{G}_9$) If $b = 1$ and the signature for m does not verify under pk, then abort;
    - ($\mathbf{G}_{10}$) If $b = 1$ and the MAC computed by $\mathcal{P}$ does not verify under key $\mathsf{PRF}(\mathsf{EMS}, \mathsf{lbl}_{\mathsf{MK,clt}})$, then abort.
  - ($\mathbf{G}_8$) Otherwise, retrieve the corresponding message $(\mathsf{ssid}, \mathsf{ak}, \mathsf{ske})$ from $\mathcal{F}$, set $\mathsf{ske} = (\mathsf{nonce}, *)$, add nonce to $\mathsf{nonceFresh}$ and resume simulation of $\mathcal{P}$ with input $(\textsc{AuthSend}, \mathcal{P}'', \mathsf{cid}, \mathsf{ssid}, \mathsf{ctx}, \mathsf{ak}, \mathsf{ske}, \mathsf{pk})$ as in Π_{EA}.

Figure 18: Simulator in $\mathbf{G}_{10}$, omitting session identifier sid of Π_{EA} in all inputs, outputs and messages.

```
On $j$-th input ($\textsc{KeyGen}, \mathsf{kid}, \mathsf{ak}, \mathtt{mode}$)
  • If $j \leq i$, then do:
    – ($\mathbf{G}_1$) compute $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KG}(1^\lambda)$ ($\mathbf{G}_4$) and abort if $\mathsf{pk}$ not fresh.
    – ($\mathbf{G}_1$) If $\mathtt{mode} = \mathtt{tk}$ then set $\mathsf{nonce} \leftarrow \{0,1\}^\lambda$ and ($\mathbf{G}_2$) abort if $\mathsf{nonce}$ not fresh.
    – ($\mathbf{G}_{11}$) Set $\mathsf{ae} \leftarrow \mathsf{SIM}_{\mathsf{eq}}(|\mathsf{sk}|)$ and ($\mathbf{G}_3$) abort if $\mathsf{ae}$ not fresh. Compute ($\mathbf{G}_5$) $\mathsf{ske} \leftarrow (\mathsf{nonce}, \mathsf{ae})$, ($\mathbf{G}_{11}$) store
      $(\perp, \mathsf{sk}, \mathsf{ske}, \mathsf{pk})$ and send $(\mathsf{kid}, \mathsf{ske}, \mathsf{pk})$ to $\mathcal{F}$.
    – ($\mathbf{G}_5$) If $\mathtt{mode} = \mathtt{std}$ send $(\mathsf{kid}, \varepsilon, \mathsf{pk})$ to $\mathcal{F}$.
  • If $j > i$, then do:
    – ($\mathbf{G}_1$) compute $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KG}(1^\lambda)$ ($\mathbf{G}_4$) and abort if $\mathsf{pk}$ not fresh.
    – ($\mathbf{G}_1$) If $\mathtt{mode} = \mathtt{tk}$ then set $\mathsf{nonce} \leftarrow \{0,1\}^\lambda$ and ($\mathbf{G}_2$) abort if $\mathsf{nonce} \in \mathsf{nonceFresh}$. Add $\mathsf{nonce}$ to $\mathsf{nonceFresh}$.
    – ($\mathbf{G}_5$) Set $k \leftarrow \{0,1\}^\lambda$, ($\mathbf{G}_1$) $\mathsf{ae} \leftarrow \mathsf{AEnc}_k(\mathsf{sk}), \mathsf{ske} \leftarrow (\mathsf{nonce}, \mathsf{ae})$, ($\mathbf{G}_5$) store $(k, \mathsf{sk}, \mathsf{ske}, \mathsf{pk})$ and send $(\mathsf{kid}, \mathsf{ske}, \mathsf{pk})$ to $\mathcal{F}$.
    – ($\mathbf{G}_5$) If $\mathtt{mode} = \mathtt{std}$ send $(\mathsf{kid}, \varepsilon, \mathsf{pk})$ to $\mathcal{F}$.

($\mathbf{G}_1$) On input $m$ to $\mathcal{F}_{\mathsf{RO}}$, parse $m = (\mathsf{ak}, \mathsf{nonce})$.
  • ($\mathbf{G}_1$) If there is a pair $((\mathsf{ak}, \mathsf{nonce}), \tilde{h}) \in T_{\mathsf{H}}$ then reply with $\tilde{h}$;
  • ($\mathbf{G}_5$) For every key record $(k, *, (\mathsf{nonce}, \mathsf{ae}), \mathsf{pk})$, send ($\textsc{GetAuxData}, \mathsf{ak}, (\mathsf{nonce}, \mathsf{ae})$) to $\mathcal{F}$. Upon reply $\mathsf{pk}'$, if $\mathsf{pk}' = \mathsf{pk}$ then
    ($\mathbf{G}_{11}$) if $k = \perp$ set $k \leftarrow \mathsf{SIM}_{\mathsf{eq}}(\mathsf{sk})$, ($\mathbf{G}_5$) add $((\mathsf{ak}, \mathsf{nonce}), k)$ to $T_{\mathsf{H}}$ and reply with $k$; If none of the $\textsc{GetAuxData}$ replies
    equal $\mathsf{pk}$, then set $h \leftarrow \{0,1\}^\lambda$, store $(m, h) \in T_{\mathsf{H}}$ and reply with $h$.
  • ($\mathbf{G}_1$) If none of the above cases apply, then choose uniformly $h \leftarrow \{0,1\}^\lambda$, store $(m, h) \in T_{\mathsf{H}}$ and reply with $h$.
```

Figure 19: Simulator's KeyGen and hash interfaces in Hybrid $\mathbf{G}_{11.i}$. Changes from $\mathbf{G}_{10}$ are marked gray. Namely, for the first i KeyGen requests, instead of drawing a random encryption key k, ciphertexts ae are simulated using $\mathsf{SIM}_{\mathsf{eq}}$ of the equivocable encryption scheme. Whenever the corresponding key is revealed via a hash query, a suitable key can be produced by the simulator $\mathsf{SIM}_{\mathsf{eq}}$.

replies with ($\textsc{key}, \varepsilon, \varepsilon, \mathsf{pk}$). If $\mathtt{mode} = \mathtt{tk}$, CMA attacker $\mathcal{B}$ computes $\mathsf{ae} \leftarrow \mathsf{SIM}_{\mathsf{eq}}(\lambda)$, $\mathsf{ske} \leftarrow (\mathsf{nonce}, \mathsf{ae})$ and stores key record $(\perp, \perp, \mathsf{ske}, \mathsf{aux}, \mathsf{pk})$. In any case, if later a signature needs to be created w.r.t pk by an honest party while processing AuthSend, $\mathcal{B}$ obtains the signature from his signing oracle $\mathcal{O}_{\mathsf{Sign}_{\mathsf{sk}}(\cdot)}()$. In case of $\mathcal{B}$ reaching event $\mathsf{BAD}_{\mathsf{sforgery}}$ upon signature σ for message $m \leftarrow (\mathsf{HSC}, \mathsf{ssid}, \mathsf{ctx})$ under pk, instead of aborting $\mathcal{B}$ submits (m, σ) as forgery to the CMA challenger.

Since one condition of event $\mathsf{BAD}_{\mathsf{sforgery}}$ is that the corresponding key record $(\perp, \perp, \mathsf{ske}, *, \mathsf{pk})$ does not contain any encryption key k, if $\mathcal{S}$ of this game aborts wrt pk, then $\mathcal{B}$ never needs to compute k. Since computation of k and signing are the only two parts in the code of $\mathbf{G}_{12}$ which depend on sk, $\mathcal{B}$'s output distribution towards $\mathcal{Z}$ is equal to $\mathbf{G}_{12}$. Because σ is a verifying signature for a message that was never signed before, $\mathcal{B}$ wins the CMA game whenever $\mathsf{BAD}_{\mathsf{sforgery}}$ happens w.r.t pk in $\mathbf{G}_{12}$. Thus we have that

$$
\begin{aligned}
&|\Pr[\mathsf{win}(\mathbf{G}_{12})] - \Pr[\mathsf{win}(\mathbf{G}_{11})]| \\
&\qquad \leq \Pr[\mathsf{BAD}_{\mathsf{sforgery}}] \leq q\mathsf{Adv}^{\mathsf{euf-cma}}_{\mathcal{B},\mathsf{SIG}}(1^\lambda),
\end{aligned}
$$

where q is an upper bound on the number of KeyGen requests by $\mathcal{Z}$.

Game $\mathbf{G}_{13}$: Back to real ciphertexts. Similar to game $\mathbf{G}_{11}$, we define a series of hybrids to switch the simulation of KeyGen queries back to ciphertexts produced as $\mathsf{ae} \leftarrow \mathsf{AEnc}_k(\mathsf{sk})$. Again, with q an upper bound on the number of KeyGen queries issued by $\mathcal{Z}$, by equivocability of the encryption scheme we have

$$|\Pr[\mathsf{win}(\mathbf{G}_{13})] - \Pr[\mathsf{win}(\mathbf{G}_{12})]| \leq q\mathsf{Adv}^{\mathsf{eq}}.$$

Note that $\mathcal{S}$ now draws encryption keys $k \leftarrow \{0,1\}^\lambda$ again and hence we have these keys k back in the key records.

Game $\mathbf{G}_{14}$: Randomizing all PRF values. We change the simulation for connected channels, i.e., where $\mathcal{A}$ already sent ($\textsc{Connect}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}', \mathsf{CB}^*$) to $\mathcal{F}_{\mathsf{cbSC}}$ and $\mathcal{P}, \mathcal{P}'$ are both honest. $\mathcal{S}$ draws PRF outputs $\mathsf{PRF}(\mathsf{EMS}, \mathsf{lbl}_{\mathsf{HSC},\mathsf{role}})$, $\mathsf{PRF}(\mathsf{EMS}, \mathsf{lbl}_{\mathsf{MK},\mathsf{role}})$ for $\mathcal{P}$ and $\mathcal{P}$ uniformly at random.

The execution of $\mathbf{G}_{14}$ is independent of values EMS output by $\mathcal{F}_{\mathsf{cbSC}}$, since in particular we only consider static corruptions. We can thus construct a PRF adversary which emulates $\mathbf{G}_{13}$ if talking to the oracle implementing the PRF, and which emulates $\mathbf{G}_{14}$ if talking to an oracle implementing a truly random function. Hence, we have

$$|\Pr[\mathsf{win}(\mathbf{G}_{14})] - \Pr[\mathsf{win}(\mathbf{G}_{13})]| \leq \mathsf{Adv}^{\mathsf{prf}}.$$

Game $\mathbf{G}_{15}$: Excluding CB and MK collisions. $\mathcal{S}$ aborts whenever a collision upon PRF output sampling occurs. Due to the Birthday bound, this and the previous game are computationally indistinguishable.

Game $\mathbf{G}_{16}$: Dealing with MAC forgeries. We change $\mathcal{S}$ upon receiving adversarially-generated message $(\mathsf{ssid}, \mathsf{cid}, \mathsf{ctx}, \sigma, \mathsf{mac})$ from $\mathcal{A}$ as message from $\mathcal{P}'$ to simulated $\mathcal{P}$, where $(\mathsf{ssid}, \mathsf{cid}, \mathsf{ctx}, \sigma, *, *)$ was never sent by $\mathcal{P}'$. Let $(\textsc{Finalize}, \mathsf{cid}, \mathsf{cid}', \mathsf{role}', \mathsf{EMS})$ denote a former $\mathcal{F}_{\mathsf{cbSC}}$ output towards $\mathcal{P}$. If $\mathcal{F}_{\mathsf{cbSC}}$ maintains a record $(\textsc{session}, \mathcal{P}, \mathsf{cid}, \mathsf{role})$ marked $\mathtt{conn}(\mathcal{P}', \mathsf{cid}')$ and a record $(\textsc{session}, \mathcal{P}', \mathsf{cid}', \mathsf{role})$ marked $\mathtt{conn}(\mathcal{P}, \mathsf{cid})$, $\mathcal{P}, \mathcal{P}'$ are both honest and MAC verification succeeds, i.e., $\mathsf{MAC.Vfy}_k(\mathsf{mac}, m) = 1$ for the message $m = (\mathsf{HSC}, \mathsf{ssid}, \mathsf{ctx}, \sigma)$ and k, HSC the uniformly random MAC key and channel binding secret (cf. $\mathbf{G}_{14}$) for this channel, then $\mathcal{S}$ aborts.

The output distribution is only affected if $\mathcal{S}$ aborts. We call this event $\mathsf{BAD}_{\mathsf{mforgery}}$ and show that it happens only with negligible probability if the MAC scheme is EUF-CMA secure.

Let $\mathcal{B}$ denote an attacker against MAC in an EUF-CMA security game. $\mathcal{B}$ emulates game $\mathbf{G}_{16}$ as follows: $\mathcal{B}$ implements $\mathbf{G}_{16}$ but randomly picks a $\textsc{NewSession}$ query by $\mathcal{Z}$. $\mathcal{B}$ aborts if the channel corresponding to this query is never finalized, or if it is finalized but flagged $\mathtt{att}$, or if any of the two corresponding parties is corrupted. Otherwise, $\mathcal{B}$ simulates both parties on this channel not with sampling a uniform MAC key, but by using his oracle $\mathcal{O}_{\mathsf{Mac}_k(\cdot)}()$ to compute any message authentication code for authentication requests on this channel, and using oracle $\mathcal{O}_{\mathsf{Vfy}_k(\cdot,\cdot)}()$ for verifying a mac/message pair. In case of reaching event $\mathsf{BAD}_{\mathsf{mforgery}}$ upon message $M = (\mathsf{ssid}, \mathsf{ctx}, \sigma, \mathsf{mac})$ from $\mathcal{A}$, which can be detected by $\mathcal{B}$ by using his verification oracle $\mathcal{O}_{\mathsf{Vfy}_k(\cdot,\cdot)}()$, $\mathcal{B}$ submits (mac, M) as forgery to its challenger.

Since we restrict to static corruptions, honest parties never reveal their MAC keys corresponding to non-attacked channels, and these keys are never used anywhere except in interfaces $\textsc{AuthSend}$ and $\textsc{AuthVerify}$, which are now implemented with the oracles. We thus only need to verify that, in case event $\mathsf{BAD}_{\mathsf{mforgery}}$ happens for a message M, M was never submitted to oracle $\mathcal{O}_{\mathsf{Mac}_k(\cdot)}()$. But this is easy to verify: since $\mathsf{BAD}_{\mathsf{mforgery}}$ is conditioned on $\mathcal{P}'$ never sending M, consequently $\mathcal{B}$ never submitted M to oracle $\mathcal{O}_{\mathsf{Mac}_k(\cdot)}()$.

$$\begin{aligned}|\Pr[\mathsf{win}(\mathbf{G}_{16})] - \Pr[\mathsf{win}(\mathbf{G}_{15})]| \\ \leq \Pr[\mathsf{BAD}_{\mathsf{mforgery}}] \leq q\mathsf{Adv}^{\mathsf{euf-cma}}_{\mathcal{B},\mathsf{MAC}}(1^{\lambda}),\end{aligned}$$

where q is an upper bound on the number of $\textsc{KeyGen}$ requests by $\mathcal{Z}$.

Game $\mathbf{G}_{17}$: $\mathcal{F}$ decides authentications on honest channels. We add a partial version of the $\textsc{AuthVerify}$ interface to $\mathcal{F}$, namely one that can only be used by honest parties $\mathcal{P}'$, and only on channels cid' that are connected to some other honest party. This way, we let $\mathcal{F}$ compute the result of the authentication only on honest connected channels.

We change $\mathcal{S}$ as follows: upon receiving $(\textsc{AuthVerify}, \mathcal{P}', \mathsf{cid}', \mathsf{ssid}, \mathsf{ctx}, \mathsf{pk})$ from $\mathcal{F}$, if $\mathcal{F}_{\mathsf{cbSC}}$ has record $(\textsc{session}, \mathcal{P}, \mathsf{cid}, \mathsf{role}')$ marked $\mathtt{conn}(\mathcal{P}', \mathsf{cid}')$, and record $(\textsc{session}, \mathcal{P}', \mathsf{cid}', \mathsf{role})$ marked $\mathtt{conn}(\mathcal{P}, \mathsf{cid})$, and $\mathcal{P}, \mathcal{P}'$ are both honest, and $\mathcal{A}$ delivered message $\mathsf{ssid}, \mathsf{cid}', \sigma, \mathsf{mac}$ from honest $\mathcal{P}$ to $\mathcal{P}'$ unchanged, then $\mathcal{S}$ replies with flag 1. If $\mathcal{A}$ changed any of σ, mac, then $\mathcal{S}$ replies with flag 0.

For indistinguishability, we argue separately for the cases where $\mathcal{A}$ delivers message $\mathsf{ssid}, \mathsf{cid}', \sigma, \mathsf{mac}$ honestly or not. In the honest case, in $\mathbf{G}_{16}$ $\mathcal{P}'$ outputs 1 iff both σ and mac verify wrt. $\mathsf{ctx}, \mathsf{pk}$ from the AuthVerify input to $\mathcal{P}'$, and if the sending party $\mathcal{P}$ found either pk that she knows sk for in the AuthSend input, or a transportable key pair from which she can retrieve sk. The latter two conditions translate to either $\mathsf{pkey}[\mathcal{P}]$ or $\mathsf{tkey}[\mathsf{ak}, \mathsf{ske}] = (*, \mathsf{pk})$ (as of game $\mathbf{G}_6$, if the honest $\mathcal{P}$ reconstructs sk from $\mathsf{ak}, \mathsf{ske}$, it follows that $\mathsf{tkey}[\mathsf{ak}, \mathsf{ske}]$, i.e., $\mathcal{Z}$ cannot manufacture "working" transportable key pairs without the help of the KeyGen interface of $\mathcal{F}$). In this case $\mathcal{F}$ of $\mathbf{G}_{17}$ creates a corresponding Auth record containing pk, and hence $\mathcal{P}'$ will output 1 in $\mathbf{G}_{17}$ if both $\mathsf{ctx}, \mathsf{pk}$ from AuthVerify correspond to the inputs of AuthSend to $\mathcal{P}$, and 0 otherwise. It is important to note that this argument is simplified by the fact that an honest party will never produce a non-verifying mac or signature (cf. games $\mathbf{G}_9$ and $\mathbf{G}_{10}$); it will at most abort in case it does not find honestly produced key material.

For the case of an adversarially-generated σ or mac reaching $\mathcal{P}'$, as of games $\mathbf{G}_{16}$ and $\mathbf{G}_{12}$ we know that $\mathcal{P}'$ will not verify both of these successfully and hence outputs 0. In this game, $\mathcal{F}$ will output 0 to $\mathcal{P}'$ because $\mathcal{S}$ sends $f = 0$.

Hence, the output distribution does not change and we have $\Pr[\mathsf{win}(\mathbf{G}_{17})] = \Pr[\mathsf{win}(\mathbf{G}_{16})]$.

Game $\mathbf{G}_{18}$: $\mathcal{F}$ decides authentication for attacked parties. We add the ActiveAttack interface to $\mathcal{F}$, as well as the full AuthVerify interface, i.e., $\mathcal{F}$ does not only use AuthVerify for honest connections, but now also for connected channels where one party is corrupt, and for `att` channels.

We change $\mathcal{S}$ as follows. Whenever an honest party $\mathcal{P}$ receives an adversarially-generated message $\mathsf{ssid}, \mathsf{sid}, \sigma^*, \mathsf{mac}^*$, $\mathcal{S}$ chooses a random $\mathsf{ctx}^*, \mathsf{pk}^*$ and sends $(\text{ActiveAttack}, \mathcal{P}, \mathsf{cid}, \mathsf{ssid}, \mathsf{ctx}^*, \mathsf{pk}^*)$ to $\mathcal{F}$. Subsequently, upon $\mathcal{F}$ sending message $(\text{AuthVerify}, \mathcal{P}, \mathsf{cid}, \mathsf{ssid}, \mathsf{ctx}, \mathsf{pk})$ to $\mathcal{S}$, if σ^* and mac^* both verify wrt. message $(\mathsf{HSC}, \mathsf{ssid}, \mathsf{ctx})$, where HSC is produced from binder EMS of channel cid, and using verification keys pk for the signature and MK computed from EMS for the MAC, then the simulator queries $\mathcal{F}$ with $(\text{ActiveAttack}, \mathcal{P}, \mathsf{cid}, \mathsf{ssid}, \mathsf{ctx}, \mathsf{pk})$.

AuthSend outputs of honest parties are distributed as in the previous game, since $\mathcal{F}$ outputs AuthSend to an attacked honest party whenever that honest party would have produced such output in game $\mathbf{G}_{17}$, namely upon receiving an $\mathsf{ssid}, \mathsf{cid}, ...$ message.

Regarding AuthVerify outputs, we first show: $\mathcal{P}$ outputs 1 in $\mathbf{G}_{17} \Rightarrow \mathcal{P}$ outputs 1 in $\mathbf{G}_{18}$. In the previous game, $\mathcal{P}$ outputs 1 iff it finds a signature and mac verifying for the channel and the key pk and context ctx from the AuthVerify input. The simulator of $\mathbf{G}_{18}$ sends an ActiveAttack message with results in creation of a corresponding $(\text{Auth}, \ldots, \mathsf{ctx}, \mathsf{pk})$ record. However, $\mathcal{F}$ only installs such record if $\mathsf{pk} \notin \mathsf{pkReg} \setminus \mathsf{pkComp}$. As of game $\mathbf{G}_{12}$ we know that $\mathcal{F}$ has neither record $(\perp, *, *, *, \mathsf{pk})$ nor was pk ever output of a $(\text{KeyGen}, *, *, *, \mathtt{std})$ request. Hence, $\mathsf{pkReg} \setminus \mathsf{pkComp}$, $\mathcal{F}$ creates record $(\text{Auth}, \ldots, \mathsf{ctx}, \mathsf{pk})$ and $\mathcal{P}$ outputs 1 in $\mathbf{G}_{18}$ as well.

$\mathcal{P}$ outputs 0 in $\mathbf{G}_{17} \Rightarrow \mathcal{P}$ outputs 0 in $\mathbf{G}_{18}$: $\mathcal{P}$ outputs 0 in the previous game if either the mac or the signature does not verify. The simulator computes the exact same check on behalf of the simulated $\mathcal{P}$, and indicates failure by sending $f = 0$ to $\mathcal{F}$. $\mathcal{F}$ then outputs 0 as decision to $\mathcal{P}$, and hence the output is the same as in $\mathbf{G}_{17}$.

Altogether, this and the previous game produce equally distributed outputs.

It now holds that $\mathcal{F} = \mathcal{F}_{\mathsf{PHA}}$. This can be seem from the proof, where we subsequently added all the interfaces of $\mathcal{F}_{\mathsf{PHA}}$ to $\mathcal{F}$. Hence, it follows from the negligible distinguishing advantages between subsequent games that

$$|\Pr[\mathsf{win}(\mathbf{G}_{18})] - \Pr[\mathsf{win}(\mathbf{G}_0)]| \leq \eta,$$

where η is a function that is negligible in the security parameter λ, $\mathbf{G}_0$ is the real execution of Π_{EA} with dummy adversary $\mathcal{A}$ and $\mathbf{G}_{18}$ is the ideal execution with dummy parties, $\mathcal{F}_{\mathsf{PHA}}$ and a simulator $\mathcal{S}$ depicted in Figure 20.

The simulator talks to $\mathcal{F}_{\mathsf{PHA}}$ and $\mathcal{Z}$ as only external entities. Communication with $\mathcal{Z}$ goes via simulated dummy adversary $\mathcal{A}$. Hence, in the simulation below, a message by $\mathcal{A}$ comes directly from $\mathcal{Z}$, and messages sent to $\mathcal{A}$ are assumed to immediately be forwarded to $\mathcal{Z}$. See fig. 17, right-hand side, for a picture of the setting. $\mathcal{S}$ initializes empty sets $\mathsf{nonceFresh}, \mathsf{keReg}$.

Channels

$\mathcal{S}$ maintains an instance of $\mathcal{F}_{\mathsf{cbSC}}$ as in fig. 9.

- $(\mathbf{G}_1)$On $(\textsc{Attack}, \mathcal{P}, \mathsf{cid}, \mathsf{cid}^*, \mathsf{CB}^*)$ from $\mathcal{A}$, if $\mathsf{CB}^* \in \mathsf{CBset}$ ignore the query. Otherwise,
  - if $\exists$ record $(\textsc{session}, \mathcal{P}, \mathsf{cid}, \mathsf{role})$ labeled `wait` then
    - $(\mathbf{G}_{14})$store record $(\mathcal{P}, \mathsf{cid}, \mathsf{cid}^*, \mathcal{A}, \mathsf{PRF}(\mathsf{CB}^*, \mathsf{lbl}_{\mathsf{HSC,clt}}), \mathsf{PRF}(\mathsf{CB}^*, \mathsf{lbl}_{\mathsf{MK,clt}}), \mathsf{PRF}(\mathsf{CB}^*, \mathsf{lbl}_{\mathsf{HSC,srv}}), \mathsf{PRF}(\mathsf{CB}^*, \mathsf{lbl}_{\mathsf{MK,srv}}))$;
    - send message $(\textsc{Attack}, \mathcal{P}, \mathsf{cid}, \mathsf{cid}^*, \mathsf{CB}^*)$ to $\mathcal{F}_{\mathsf{cbSC}}$ and message $(\textsc{Attack}, \mathcal{P}, \mathsf{cid}, \mathsf{cid}^*)$ to $\mathcal{F}_{\mathsf{PHA}}$;
- $(\mathbf{G}_1)$On $(\textsc{Connect}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}', \mathsf{CB}^*)$ from $\mathcal{A}$, if $\exists$ record $(\textsc{session}, \mathcal{P}, \mathsf{cid}, \mathsf{role})$ labeled `wait`, then do:
  - if $\nexists$ record $(\textsc{session}, \mathcal{P}', \mathsf{cid}', \mathsf{role}')$ labeled $\mathtt{conn}(\mathcal{P}, \mathsf{cid})$ and $\mathsf{CB}^* \in \mathsf{CBset}$, then ignore the query;
  - $(\mathbf{G}_{14})$if there is no record $(\mathcal{P}, \mathsf{cid}, \mathsf{cid}', \mathcal{P}, ...)$ or $(\mathcal{P}', \mathsf{cid}', \mathsf{cid}, \mathcal{P}, ...)$ yet then do:
    - if $\mathcal{P}, \mathcal{P}'$ are both honest, choose $\mathsf{HSC}_{\mathsf{role}} \leftarrow \{0,1\}^\lambda, \mathsf{MK}_{\mathsf{role}} \leftarrow \{0,1\}^\lambda$, $\mathsf{role} \in \{\mathtt{clt}, \mathtt{srv}\}$ and store record $(\mathcal{P}, \mathsf{cid}, \mathsf{cid}', \mathcal{P}', \mathsf{HSC}_{\mathtt{clt}}, \mathsf{MK}_{\mathtt{clt}}, \mathsf{HSC}_{\mathtt{srv}}, \mathsf{MK}_{\mathtt{srv}})$
    - otherwise store record $(\mathcal{P}, \mathsf{cid}, \mathsf{cid}', \mathcal{P}', \mathsf{PRF}(\mathsf{CB}^*, \mathsf{lbl}_{\mathsf{HSC,clt}}), \mathsf{PRF}(\mathsf{CB}^*, \mathsf{lbl}_{\mathsf{MK,clt}}), \mathsf{PRF}(\mathsf{CB}^*, \mathsf{lbl}_{\mathsf{HSC,srv}}), \mathsf{PRF}(\mathsf{CB}^*, \mathsf{lbl}_{\mathsf{MK,srv}}))$;
  - send message $(\textsc{Connect}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}', \mathsf{CB}^*)$ to $\mathcal{F}_{\mathsf{cbSC}}$ and message $(\textsc{Connect}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}')$ to $\mathcal{F}_{\mathsf{PHA}}$.
- $(\mathbf{G}_1)$On $(\textsc{Deliver}, \mathcal{P}', \mathsf{cid}', m^*)$ from $\mathcal{A}$ forward the query to $\mathcal{F}_{\mathsf{PHA}}$;
- $(\mathbf{G}_1)$On $(\textsc{NewSession}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{role})$ from $\mathcal{F}_{\mathsf{PHA}}$, send input $(\textsc{NewSession}, \mathsf{cid}, \mathcal{P}', \mathsf{role})$ to $\mathcal{F}_{\mathsf{cbSC}}$;
- $(\mathbf{G}_1)$On $(\textsc{Send}, \cdot, \cdot)$ from $\mathcal{F}_{\mathsf{PHA}}$ forward the query to $\mathcal{A}$;
- $(\mathbf{G}_1)$On $(\textsc{ExpireSession}, \mathcal{P}, \mathsf{cid})$ from $\mathcal{F}_{\mathsf{PHA}}$ send input $(\textsc{ExpireSession}, \mathsf{cid})$ to $\mathcal{F}_{\mathsf{cbSC}}$.

Hash queries:

$(\mathbf{G}_1)$On input m to $\mathcal{F}_{\mathsf{RO}}$, parse $m = (\mathsf{ak}, \mathsf{nonce})$.

- $(\mathbf{G}_1)$If there is a pair $((\mathsf{ak}, \mathsf{nonce}), \tilde{h}) \in T_{\mathsf{H}}$ then reply with $\tilde{h}$;
- $(\mathbf{G}_5)$For every key record $(k, *, (\mathsf{nonce}, \mathsf{ae}), \mathsf{aux}, \mathsf{pk})$, send $(\textsc{GetAuxData}, \mathsf{ak}, (\mathsf{nonce}, \mathsf{ae}))$ to $\mathcal{F}_{\mathsf{PHA}}$. Upon reply $(\mathsf{aux}', \mathsf{pk}')$, if $\mathsf{pk}' = \mathsf{pk}$ then add $((\mathsf{ak}, \mathsf{nonce}), k)$ to T_{H} and reply with k; If none of the $\textsc{GetAuxData}$ replies contains pk, then set $h \leftarrow \{0,1\}^\lambda$, store $(m, h) \in T_{\mathsf{H}}$ and reply with h.
- $(\mathbf{G}_1)$If none of the above cases apply, then choose uniformly $h \leftarrow \{0,1\}^\lambda$, store $(m, h) \in T_{\mathsf{H}}$ and reply with h.

Key generation:

On input $(\textsc{KeyGen}, \mathsf{kid}, \mathsf{ak}, \mathsf{aux}, \mathtt{mode})$

- $(\mathbf{G}_1)$compute $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KG}(1^\lambda)$ $(\mathbf{G}_4)$and abort if pk not fresh.
- $(\mathbf{G}_1)$If $\mathtt{mode} = \mathtt{tk}$ then set $\mathsf{nonce} \leftarrow \{0,1\}^\lambda$ and $(\mathbf{G}_2)$abort if $\mathsf{nonce} \in \mathsf{nonceFresh}$. Add nonce to $\mathsf{nonceFresh}$.
- $(\mathbf{G}_5)$Set $k \leftarrow \{0,1\}^\lambda$, $(\mathbf{G}_1, \mathbf{G}_{13})\mathsf{ae} \leftarrow \mathsf{AEnc}_k(\mathsf{aux}, \mathsf{sk})$ and $(\mathbf{G}_3)$abort if $\mathsf{ae} \in \mathsf{keReg}$. Compute $(\mathbf{G}_5)\mathsf{ske} \leftarrow (\mathsf{nonce}, \mathsf{ae})$, store $(k, \mathsf{sk}, \mathsf{ske}, \mathsf{aux}, \mathsf{pk})$ and send $(\mathsf{kid}, \mathsf{ske}, \mathsf{pk})$ to $\mathcal{F}_{\mathsf{PHA}}$.
- $(\mathbf{G}_5)$If $\mathtt{mode} = \mathtt{std}$ send $(\mathsf{kid}, \varepsilon, \mathsf{pk})$ to $\mathcal{F}_{\mathsf{PHA}}$.

On input $(\textsc{Compromise}, \mathcal{P})$ from $\mathcal{Z}$ to $\mathcal{A}$:

- $(\mathbf{G}_5)$Send $(\textsc{Compromise}, \mathcal{P})$ to $\mathcal{F}_{\mathsf{PHA}}$.

Authentication:

On $(\textsc{AuthSend}, \mathcal{P}, \mathcal{P}^*, \mathsf{cid}, \mathsf{ssid}, \mathsf{ctx}, \mathsf{pk}, \mathtt{mode}, b)$ from $\mathcal{F}_{\mathsf{PHA}}$:

- $(\mathbf{G}_{14})$Retrieve record $(\mathcal{P}', \mathsf{cid}', \mathsf{cid}'', \mathcal{P}'', \mathsf{HSC}_{\mathtt{clt}}, \mathsf{MK}_{\mathtt{clt}}, \mathsf{HSC}_{\mathtt{srv}}, \mathsf{MK}_{\mathtt{srv}})$ with $(\mathcal{P}', \mathsf{cid}') = (\mathcal{P}, \mathsf{cid})$ or $(\mathcal{P}'', \mathsf{cid}'') = (\mathcal{P}, \mathsf{cid})$; Let role denote $\mathcal{P}$'s role in channel cid;
- If $\mathtt{mode} = \mathtt{std}$ then do:
  - $(\mathbf{G}_8)$Resume simulation of $\mathcal{P}$ with input $(\textsc{AuthSend}, \mathcal{P}^*, \mathsf{cid}, \mathsf{ssid}, \mathsf{ctx}, \varepsilon, \varepsilon, \mathsf{pk})$ as in Π_{EA}, $(\mathbf{G}_{14})$but with $\mathsf{HSC}_{\mathsf{role}}, \mathsf{MK}_{\mathsf{role}}$ from record.
  - $(\mathbf{G}_9)$if $b = 1$ and the signature created by $\mathcal{P}$ for some m does not verify under pk, then abort;
  - $(\mathbf{G}_{10})$if $b = 1$ and the MAC computed by $\mathcal{P}$ does not verify under key $\mathsf{MK}_{\mathsf{role}}$ from record, then abort.
- If $\mathtt{mode} = \mathtt{tk}$, then do:
  - If $\exists$ record $(*, \mathsf{sk}, *, *, \mathsf{pk})$, then
    - $(\mathbf{G}_8)$Resume simulation of $\mathcal{P}$ at signature generation using $(\mathsf{sk}, \mathsf{pk})$, message $m \leftarrow (\mathsf{HSC}_{\mathsf{role}}, \mathsf{ssid}, \mathsf{ctx})$, with $(\mathbf{G}_{14})\mathsf{HSC}_{\mathsf{role}}, \mathsf{MK}_{\mathsf{role}}$ from the channel record;
    - $(\mathbf{G}_9)$If $b = 1$ and the signature for m does not verify under pk, then abort;
    - $(\mathbf{G}_{10})$If $b = 1$ and the MAC computed by $\mathcal{P}$ does not verify under key $\mathsf{MK}_{\mathsf{role}}$, then abort.
  - $(\mathbf{G}_8)$Otherwise, parse $\mathsf{leakedKeys} = (\mathsf{ak}, \mathsf{ske}), \mathsf{ske} = (\mathsf{nonce}, *)$, add nonce to $\mathsf{nonceFresh}$ and resume simulation of $\mathcal{P}$ with input $(\textsc{AuthSend}, \mathcal{P}^*, \mathsf{cid}, \mathsf{ssid}, \mathsf{ctx}, \mathsf{ak}, \mathsf{ske}, \mathsf{pk})$ as in Π_{EA}, $(\mathbf{G}_{14})$but with $\mathsf{HSC}_{\mathsf{role}}, \mathsf{MK}_{\mathsf{role}}$ from record.

On $(\mathsf{ssid}, \mathsf{cid}, \sigma, \mathsf{mac})$ from $\mathcal{A}$ as message to honest $\mathcal{P}$:

- $(\mathbf{G}_{14})$Retrieve record $(\mathcal{P}', \mathsf{cid}', \mathsf{cid}'', \mathcal{P}'', \mathsf{HSC}_{\mathtt{clt}}, \mathsf{MK}_{\mathtt{clt}}, \mathsf{HSC}_{\mathtt{srv}}, \mathsf{MK}_{\mathtt{srv}})$ with $(\mathcal{P}', \mathsf{cid}') = (\mathcal{P}, \mathsf{cid})$ or $(\mathcal{P}'', \mathsf{cid}'') = (\mathcal{P}, \mathsf{cid})$; Let role denote the role of $\mathcal{P}$ in channel cid;
- $(\mathbf{G}_{12})$Abort if all of the following conditions hold:
  - $(\mathbf{G}_{12})(\mathsf{ssid}, \mathsf{cid}, *, *)$ was never sent by another honest party to $\mathcal{P}$;
  - $(\mathbf{G}_{13})\mathcal{F}_{\mathsf{cbSC}}$ already produced output $(\textsc{key}, *, *, *, \mathsf{pk})$;
  - $(\mathbf{G}_{12})\mathsf{Vfy}_{\mathsf{pk}}(\sigma, (\mathsf{HSC}_{\mathsf{role}}, \mathsf{ssid}, \mathsf{ctx})) = 1$, where $\mathsf{HSC}_{\mathsf{role}}$ is taken from the record;
- $(\mathbf{G}_{16})$Abort if all of the following conditions hold:
  - $\mathcal{P}', \mathcal{P}''$ are both honest;
  - $(\mathsf{ssid}, \mathsf{cid}, \sigma, \mathsf{mac})$ was never sent by pid, where $\mathsf{pid} \in \{\mathcal{P}', \mathcal{P}''\}$ and $\mathsf{pid} \neq \mathcal{P}$;
  - $\mathsf{MAC.Vfy}_{\mathsf{MK}_{\mathsf{role}'}}(\mathsf{mac}, m) = 1$ for $m = (\mathsf{HSC}_{\mathsf{role}'}, \mathsf{ssid}, \mathsf{ctx}, \sigma)$, where $\mathsf{role} \neq \mathsf{role}'$.
- $(\mathbf{G}_{18})$If σ or mac are adversarially-generated, send $(\textsc{ActiveAttack}, \mathcal{P}, \mathsf{cid}, \mathsf{ssid}, \mathsf{ctx}^*, \mathsf{pk}^*)$ to $\mathcal{F}_{\mathsf{PHA}}$ for randomly chosen $\mathsf{ctx}^*, \mathsf{pk}^*$.

On $(\mathsf{ssid}, \mathsf{ak}, \mathsf{ske})$ from $\mathcal{F}_{\mathsf{PHA}}$:

- $(\mathbf{G}_7)$Parse $(\mathsf{nonce}, \mathsf{ae}) \leftarrow \mathsf{ske}$. If $((\mathsf{ak}, \mathsf{nonce}), k) \in T_{\mathsf{H}}$ for some k, set $\mathsf{sk} \leftarrow \mathsf{ADec}_k(\mathsf{ae})$. If $(\mathsf{sk}, \mathsf{pk})$ valid, then store $(k, \mathsf{sk}, (\mathsf{nonce}, \mathsf{ae}), \mathsf{pk})$, choose fresh kid and send $(\textsc{KeyGen}, \mathsf{kid}, \mathsf{ak}, \mathtt{tk})$ to $\mathcal{F}_{\mathsf{PHA}}$. Upon $(\textsc{KeyGen}, \mathsf{kid}, \mathcal{A}, \mathtt{tk})$ from $\mathcal{F}_{\mathsf{PHA}}$, reply with $(\mathsf{kid}, \mathsf{ske}, \mathsf{pk})$.

On $(\textsc{AuthVerify}, \mathcal{P}, \mathsf{cid}, \mathsf{ssid}, \mathsf{ctx}, \mathsf{pk})$ from $\mathcal{F}$:

- $(\mathbf{G}_{17})$if $\exists$ record $(\textsc{session}, \mathcal{P}, \mathsf{cid}, \mathsf{role}')$ marked $\mathtt{conn}(\mathcal{P}', \mathsf{cid}')$, and record $(\textsc{session}, \mathcal{P}', \mathsf{cid}', \mathsf{role})$ marked $\mathtt{conn}(\mathcal{P}, \mathsf{cid})$, and $\mathcal{P}, \mathcal{P}'$ are both honest, and $\mathcal{A}$ delivered message $\mathsf{ssid}, \mathsf{cid}', \sigma, \mathsf{mac}$ from honest $\mathcal{P}$ to $\mathcal{P}'$ unchanged, then $\mathcal{S}$ replies with flag 1. If $\mathcal{A}$ changed any of σ, mac, then $\mathcal{S}$ replies with flag 0.
- $(\mathbf{G}_{18})$if $\exists$ record $(\textsc{session}, \mathcal{P}, \mathsf{cid})$ marked $\mathtt{att}$, or if $\exists$ record $(\textsc{session}, \mathcal{P}, \mathsf{cid}, \mathsf{role}')$ marked $\mathtt{conn}(\mathcal{P}', \mathsf{cid}')$, and record $(\textsc{session}, \mathcal{P}', \mathsf{cid}', \mathsf{role})$ marked $\mathtt{conn}(\mathcal{P}, \mathsf{cid})$ with $\mathcal{P}'$ corrupt, then do: let $\mathsf{ssid}, \mathsf{cid}, \sigma^*, \mathsf{mac}^*$ denote the message delivered to honest $\mathcal{P}$, and let $m = (\mathsf{HSC}, \mathsf{ssid}, \mathsf{ctx})$, where HSC is produced from binder EMS of channel cid. If $\mathsf{SIG.Vfy}_{\mathsf{pk}}(\sigma, m) = 1$ and $\mathsf{MAC.Vfy}_{\mathsf{MK}_C}(\mathsf{mac}, (m, \sigma)) = 1$ then send $(\textsc{ActiveAttack}, \mathcal{P}, \mathsf{cid}, \mathsf{ssid}, \mathsf{ctx}, \mathsf{pk})$ to $\mathcal{F}$.

Figure 20: Simulator for Theorem 5.1. We omit session identifier sid of Π_{EA} in all inputs, outputs and messages.

B.3 Proof of Theorem 6.1

The proof is simplified by the following facts: (1) there are no messages contents to simulate (all the messages sent during registration and authentication are adversarially-determined, i.e., their contents are given either as input by the environment or to $\mathcal{F}_{\mathsf{OPRF}}$ or $\mathcal{F}_{\mathsf{PHA}}$ by the adversary), and (2) apart from sampling and table lookups, the protocol only uses calls to ideal functionalities. Hence, the challenge lies in simulating the instances of $\mathcal{F}_{\mathsf{PHA}}$ and $\mathcal{F}_{\mathsf{OPRF}}$ *without the knowledge of secret inputs* pw, such that these instances appear to the distinguishing environment $\mathcal{Z}$ as in the real protocol execution, running on passwords. Our simulator compensates for not knowing passwords by using the adversarial interfaces of $\mathcal{F}_{\mathsf{pwPHA}}$.

Before describing simulation, we mention two subtleties of modularity that at first sight make $\Pi_{\mathsf{TLS-OPAQUE}}$ appear insecure. First, $\mathcal{Z}$ determines all public keys via $\mathcal{F}_{\mathsf{PHA}}$ – so $\mathcal{Z}$ may know all secret keys! It hence looks like we cannot guarantee anything since all keys are compromised. However, with the modular usage of $\mathcal{F}_{\mathsf{PHA}}$, we abstract from objects such as secret keys and signatures; these objects become meaningless on the level of $\Pi_{\mathsf{TLS-OPAQUE}}$. Indeed, a party running the code of $\Pi_{\mathsf{TLS-OPAQUE}}$ does not "understand" anything from receiving a signature under some pk. Instead, it can only be convinced of an authentication by receiving $(\textsc{AuthVerify}, ..., 1)$ from $\mathcal{F}_{\mathsf{PHA}}$.

The second "scary subtlety" is $\mathcal{Z}$ knowing all password files, even without compromising or corrupting anybody: all file values are given by $\mathcal{A}$. While this makes the simulation of such files trivial, it does not mean that $\mathcal{Z}$ can freely run dictionary attacks against these files. To complete such an attack, $\mathcal{Z}$ must turn each password guess into a file decryption key by interacting with $\mathcal{F}_{\mathsf{OPRF}}$. Unless the file is compromised, this requires interaction with the honest party storing the file. Hence, guesses are still limited to one guess per session (= the unavoidable online dictionary attack in password-based protocols) with an honest server $\mathcal{P}$, which is the exact same number of guesses that $\mathcal{F}_{\mathsf{pwPHA}}$ allows $\mathcal{S}$ to make through $\textsc{TestPwd}$ with "honest session" counter $\mathsf{ctr}[\mathcal{P}, \mathsf{uid}]$. We note that this proof strategy of controlling adversarial guesses strongly resembles usage of idealized assumptions for analyzing password-based protocols, such as a random oracle. Since instantiating $\mathcal{F}_{\mathsf{OPRF}}$ is believed to require an idealized assumption itself, it seems natural that we can exploit $\mathcal{F}_{\mathsf{OPRF}}$ as such.

We now describe our simulation in more detail. To simulate the registration phase, $\mathcal{S}$ omits password inputs completely and starts by simulating $\mathcal{F}_{\mathsf{OPRF}}$ output $(\textsc{Tr}, \mathsf{sid}, \mathsf{ssid}, a)$ towards parties, with a given by the adversary $\mathcal{A}$. The simulation of messages is trivial, as none are generated by honest parties: uid is received from $\mathcal{F}_{\mathsf{pwPHA}}$, a, b are both received from $\mathcal{A}$ through $\mathcal{F}_{\mathsf{OPRF}}$, and $\mathsf{pk}_S, \mathsf{pk}_C, \mathsf{ske}$ are received from $\mathcal{A}$ via $\mathcal{F}_{\mathsf{PHA}}$.

$\mathcal{Z}$ can make a password guess against a file $(\mathsf{uid}, \mathsf{ske}, \mathsf{pk}_S, \mathsf{pk}_C)$ stored by $\mathcal{P}$ by querying $\mathcal{F}_{\mathsf{OPRF}}$ with $\mathsf{sid} = \mathsf{uid}||\mathcal{P}$ and pw (through a corrupt evaluator, or interface $\textsc{OfflineEval}$) and receiving back rw; then, $\mathcal{Z}$ can test whether envelope ske can be decrypted with rw to $(\mathsf{pk}_S, \mathsf{pk}_C)$ by querying $\mathcal{F}_{\mathsf{PHA}}$ with $(\textsc{GetAuxData}, \mathsf{rw}, \mathsf{ske})$. Since $\mathcal{S}$ did not know neither password nor PRF value used during registration, it now extracts the password guess pw from $\mathcal{Z}$'s query to $\mathcal{F}_{\mathsf{OPRF}}$ and submits $(\textsc{TestPwd}, \mathcal{P}, \mathsf{uid}, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{pwPHA}}$. Upon answer "correct guess", $\mathcal{S}$ answers the $\textsc{GetAuxData}$ query of $\mathcal{Z}$ with $(\mathsf{pk}_S, \mathsf{pk}_C)$, otherwise $\mathcal{S}$ replies with $(\perp, \perp)$. We note that password guesses are not per individual authentication sessions but per file. Consequently, $\mathcal{F}_{\mathsf{pwPHA}}$ allows $\mathcal{S}$ to successfully finish all pending authentication sessions for uid with honest $\mathcal{P}$ as soon as a successful password guess for the corresponding ske was issued. Depending on whether the real-world adversary successfully finishes those sessions or not (via sending $f = 1$ for success, and $f = 0$ for failure in $\textsc{AuthVerify}$), $\mathcal{S}$ finishes the corresponding session at $\mathcal{F}_{\mathsf{pwPHA}}$ likewise using $\textsc{pwDeliver}$ and $b = 1$ for success, $b = 0$ for failure.

Impersonation attacks are dictionary attacks mounted *with* a compromised password file. In $\Pi_{\mathsf{TLS-OPAQUE}}$, their simulation is more straightforward than the above guesses *against* a file. This is because an impersonator must commit to which file to use already when sending the first authentication message including ske (note: $\mathcal{F}_{\mathsf{PHA}}$ enforces uniqueness of envelopes, otherwise there would be no commitment here). Hence, $\mathcal{S}$ simply derives which $\mathcal{P}, \mathsf{uid}$ such ske belongs to and uses $(\textsc{Impersonate}, \mathsf{ssid}, \mathcal{P}, \mathsf{uid}, \varepsilon)$ of $\mathcal{F}_{\mathsf{pwPHA}}$, leaving the authentication decision up to $\mathcal{F}_{\mathsf{pwPHA}}$. If $\mathcal{Z}$ happens to use an envelope ske that was not created during registration, but one that $\mathcal{Z}$ created locally, $\mathcal{S}$ extracts password guess pw from the corresponding ske generation at $\mathcal{F}_{\mathsf{PHA}}$ and $\mathcal{F}_{\mathsf{OPRF}}$ and issues $(\textsc{Impersonate}, \mathsf{ssid}, \varepsilon, \mathsf{uid}, \mathsf{pw})$ for session ssid under attack.

The formal proof proceeds in a series of games, starting with the real execution and ending up with the ideal execution.

Game $\mathbf{G}_0$: The real execution. $\mathcal{Z}$ interacts with parties running protocol $\Pi_{\mathsf{TLS-OPAQUE}}$ depicted in Figure 14, making calls to ideal functionalities $\mathcal{F}_{\mathsf{PHA}}, \mathcal{F}_{\mathsf{OPRF}}$ and interacting with a dummy adversary $\mathcal{A}$.

Game $\mathbf{G}_1$: Introducing simulator and ideal functionality. We group all machines except for $\mathcal{Z}$ into one new machine and call it the cimulator $\mathcal{S}$. For each party, a dummy party is added between $\mathcal{Z}$ and $\mathcal{S}$. We also add a machine $\mathcal{F}$ between dummy parties and $\mathcal{S}$, and add to it NewSession, Attack, Connect, Send, Deliver, ExpireSession as in $\mathcal{F}_{\mathsf{PHA}}$. $\mathcal{F}$ serves as a relay for all messages that are not send to these interfaces. To account for these changes, we let $\mathcal{S}$ send exact copies of all adversarial NewSession, Attack, Connect, Send, Deliver, ExpireSession queries directed at $\mathcal{F}_{\mathsf{PHA}}$ to $\mathcal{F}$, and forward all messages from $\mathcal{F}$ intended for $\mathcal{A}$ to the internal simulation. More formally:

- On $(\text{Attack}, \mathcal{P}, \mathsf{cid}, \mathsf{cid}^*)$ from $\mathcal{A}$, send $(\text{Attack}, \mathcal{P}, \mathsf{cid}, \mathsf{cid}^*)$ to $\mathcal{F}_{\mathsf{PHA}}$ and $(\text{Attack}, \mathcal{P}, \mathsf{cid}, \mathsf{cid}^*)$ to $\mathcal{F}$;
- On $(\text{Connect}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}', \mathsf{cid}^*)$ from $\mathcal{A}$, if there is a $(\text{session}, \mathcal{P}, \mathsf{cid}, \mathsf{role})$ record labeled `wait`, then do:
  - if $\nexists$ record $(\text{session}, \mathcal{P}', \mathsf{cid}', \mathsf{role}')$ that is labeled $\mathtt{conn}(\mathcal{P}, \mathsf{cid})$ then ignore the query;
  - otherwise send $(\text{Connect}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}', \mathsf{cid}^*)$ to $\mathcal{F}_{\mathsf{PHA}}$ and $(\text{Connect}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{cid}', \mathsf{cid}^*)$ to $\mathcal{F}$.
- On $(\text{Deliver}, \mathcal{P}', \mathsf{cid}', m^*)$ from $\mathcal{A}$ forward the query to $\mathcal{F}$;
- On $(\text{NewSession}, \mathcal{P}, \mathsf{cid}, \mathcal{P}', \mathsf{role})$ from $\mathcal{F}$, send $(\text{NewSession}, \mathsf{cid}, \mathcal{P}', \mathsf{role})$ to $\mathcal{F}_{\mathsf{PHA}}$;
- On $(\text{Send}, \cdot, \cdot)$ from $\mathcal{F}$ forward the query to $\mathcal{Z}$;
- On $(\text{ExpireSession}, \mathcal{P}, \mathsf{cid})$ from $\mathcal{F}$ send the query $(\text{ExpireSession}, \mathsf{cid})$ to $\mathcal{F}_{\mathsf{PHA}}$.

These changes do not influence the output distribution of this game, as the set of channels is simply replicated from $\mathcal{F}_{\mathsf{PHA}}$ to $\mathcal{F}$ and hence we have

$$\Pr[\mathsf{win}(\mathbf{G}_1)] = \Pr[\mathsf{win}(\mathbf{G}_0)].$$

Game $\mathbf{G}_2$: Let $\mathcal{F}$ maintain a password file.
Changes to $\mathcal{F}$: We add the interfaces StorePwdFile, StorePwdComplete, StealPwdFile, OfflTestPwd to $\mathcal{F}$ as in $\mathcal{F}_{\mathsf{pwPHA}}$ (Fig. 13), but let $\mathcal{F}$ still inform the simulator about the passwords within StorePwdFile queries of honest parties.
Changes to simulation: $\mathcal{S}$ relays StorePwdFile queries with passwords as input to simulated honest parties, and send $(\text{StorePwdComplete}, \mathcal{P}, \mathsf{ssid})$ to $\mathcal{F}$ whenever a simulated party $\mathcal{P}$ stores a file in session ssid. Upon $\mathcal{Z}$ sending StealPwdFile for $\mathcal{P}, \mathsf{uid}$, $\mathcal{S}$ sends $(\text{StealPwdFile}, \mathcal{P}, \mathsf{uid})$ to $\mathcal{F}$, $(\text{Compromise}, \mathcal{P})$ to $\mathcal{F}_{\mathsf{OPRF}}$ and to $\mathcal{F}_{\mathsf{PHA}}$, and the password file stored by the simulated $\mathcal{P}$ to $\mathcal{A}$. Further, $\mathcal{S}$ extracts registration passwords pw of corrupt client $\mathcal{P}$ with honest server $\mathcal{P}'$ from $\mathcal{F}_{\mathsf{OPRF}}$ where $\mathsf{sid} = \mathcal{P}'||\mathsf{uid}$, where uid is sent by $\mathcal{P}$. In case the corrupt client uses an unknown rw, $\mathcal{S}$ sets $\mathsf{pw} = \perp$. $\mathcal{S}$ then submits $(\text{StorePwdFile}, \mathsf{ssid}, \mathcal{P}', \mathsf{uid}, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{pwPHA}}$.

This concludes the changes of this game. For indistinguishability, we need to argue that introduction of the new interfaces in $\mathcal{F}$ and the changes in the simulation both go unnoticed by the environment. First, note that $\mathcal{F}_{\mathsf{pwPHA}}$ does not make use yet of any files. The new interfaces affect (1) password files in the simulation, (2) registration transcripts, and the changes in the simulation additionally affect (3) outputs of $\mathcal{F}_{\mathsf{OPRF}}, \mathcal{F}_{\mathsf{PHA}}$. For (1), indistinguishability is trivial since all elements of the files are known to $\mathcal{Z}$ already since they are either input to $\mathcal{F}_{\mathsf{pwPHA}}$ or adversarially-determined values in $\mathcal{F}_{\mathsf{PHA}}$. Likewise, registration transcripts (2) are trivially indistinguishable since they are purely $\mathcal{Z}$-generated. For (3), party outputs remain the same as in the last game since party simulation still runs with the password. For adversarial outputs of $\mathcal{F}_{\mathsf{OPRF}}$ and $\mathcal{F}_{\mathsf{PHA}}$, all adversarial outputs generated from the protocol run, e.g., the Init and SndrComplete outputs of $\mathcal{F}_{\mathsf{OPRF}}$, are produced automatically by the simulation running the protocol code as of game $\mathbf{G}_1$, and the same holds for $\mathcal{F}_{\mathsf{PHA}}$'s communication with the adversary. We hence have

$$\Pr[\mathsf{win}(\mathbf{G}_2)] = \Pr[\mathsf{win}(\mathbf{G}_1)].$$

.

We note that the changes made in this game demonstrate already how $\mathcal{S}$ exploits the OFFLTESTPWD interface to answer to a $\mathcal{Z}$ who wants to check whether an honestly generated password file by $\mathcal{P}'$ (for which $\mathcal{Z}$ gave inputs pw and ske) is "good". Checking whether a file "works" can be done by first turning pw into rw via $\mathcal{F}_{\mathsf{OPRF}}$ with $\mathsf{sid} = \mathsf{uid}||\mathcal{P}'$, and then testing whether $\mathsf{rw}, \mathsf{ske}$ is a working key pair via interface GETAUXDATA at $\mathcal{F}_{\mathsf{PHA}}$. Looking ahead, this is already the first step towards simulating without passwords.

Game $\mathbf{G}_3$: Let $\mathcal{F}$ handle initialization.

Changes to $\mathcal{F}$: We add the PWINIT and ACTIVEATTACK interfaces of $\mathcal{F}_{\mathsf{pwPHA}}$ to $\mathcal{F}$, but still let $\mathcal{F}$ relay passwords of PWINIT to the simulator.
Changes to the simulation: Upon receiving (PWINIT, $\mathcal{P}, \mathcal{P}', \mathsf{cid}, \mathsf{ssid}, \mathsf{uid}$) and password pw from the functionality $\mathcal{F}_{\mathsf{pwPHA}}$, $\mathcal{S}$ starts the simulated party $\mathcal{P}$ with input (PWINIT, $\mathcal{P}', \mathsf{cid}, \mathsf{ssid}, \mathsf{uid}, \mathsf{pw}$) to $\mathcal{F}_{\mathsf{pwPHA}}$. Upon $\mathcal{A}$ or a corrupt $\mathcal{P}$ sending (AUTHSEND, $\mathcal{P}', \mathsf{cid}, \mathsf{ssid}, (\mathsf{uid}, a)$) or a corresponding query (ACTIVEATTACK, $\mathcal{P}, \mathsf{cid}, \mathsf{ssid}, (\mathsf{uid}, a), *$) to $\mathcal{F}_{\mathsf{PHA}}$, the simulator queries $\mathcal{F}_{\mathsf{pwPHA}}$ with (ACTIVEATTACK, $\mathsf{ssid}, \mathcal{P}', \mathsf{cid}, \mathsf{uid}$).

For indistinguishability, we first note that (1) interface PWDELIVER is not used yet, so PWAUTH records in $\mathcal{F}$ do not have any effect yet, (2) interface PWINIT produces the same output towards parties as the $\Pi_{\mathsf{TLS-OPAQUE}}$ protocol/the previous game, and (3) $\mathcal{S}$ receives the same inputs as in the previous game. (1) and (3) are straightforward to see. For (2), note that $\Pi_{\mathsf{TLS-OPAQUE}}$ lets parties output whatever is received as message from the other party.

Overall, this game does not change the output distribution towards $\mathcal{Z}$, and hence we have

$$\Pr[\mathsf{win}(\mathbf{G}_3)] = \Pr[\mathsf{win}(\mathbf{G}_2)].$$

Game $\mathbf{G}_4$: Rule out rw collisions.

Changes to the simulation: We let $\mathcal{S}$ abort whenever $\mathcal{F}_{\mathsf{OPRF}}$ samples a random value rw that was already sampled before.

This and the previous game are identical except if a collision in drawing a random value from $\{0,1\}^\lambda$ occurs, which is negligible in λ by the Birthday Bound and hence

$$|\Pr[\mathsf{win}(\mathbf{G}_4)] - \Pr[\mathsf{win}(\mathbf{G}_3)]| \leq \frac{q(q-1)}{2 \cdot 2^{-\lambda}},$$

with $q = q_I + q_R$, q_I being the number of PWINIT queries issued by $\mathcal{Z}$ towards honest parties and q_R the number of STOREPWDFILE queries.

Game $\mathbf{G}_5$: Abort upon rw guesses.

Changes to the simulation: We let $\mathcal{S}$ abort whenever the adversary sends an rw that $\mathcal{F}_{\mathsf{OPRF}}$ has already sampled and which was not previously given to the adversary.

This and the previous game are identical except if any of the adversarially-submitted rw coincides with any of the randomly sampled rw of the OPRF instances. The adversary has at most $q = q_I + q_T$ guesses, q_I being the number of PWINIT queries issued by $\mathcal{Z}$ towards honest parties and q_T being the number of adversarial GETAUXDATA queries. GETAUXDATA queries count to adversarial guesses because they contain rw values, and PWINIT queries count because the adversary can send rw within AUTHSEND in attacked sessions. Hence, the probability of an abort is upper bounded by the Birthday bound over $q := 2q_I + q_T$ as

$$|\Pr[\mathsf{win}(\mathbf{G}_5)] - \Pr[\mathsf{win}(\mathbf{G}_4)]| \leq \frac{q(q-1)}{2 \cdot 2^{-\lambda}}.$$

We now demonstrate how the simulator extracts password guesses from a corrupt server. For this, note that the authentication phase of $\Pi_{\mathsf{TLS-OPAQUE}}$ does not send a single message over the network. Indeed, it only makes the parties use the hybrid functionalities. Hence, the adversary cannot mount any meaningful attack over an attacked channel, since there is simply no message that $\mathcal{A}$ can send to an honest party that this party would even consider processing. Therefore, we can restrict ourselves to extraction from corrupt parties (who, opposed to the adversary, can actually call the user interfaces of $\mathcal{F}_{\mathsf{OPRF}}$ and $\mathcal{F}_{\mathsf{PHA}}$), and ignore all messages that the adversary sends over the network during authentication.

Game $\mathbf{G}_6$: Extracting from corrupt server (Impersonate).
Changes to $\mathcal{F}$: We add the Impersonate interface to $\mathcal{F}$.
Changes to $\mathcal{S}$: Upon (AuthSend, $\mathcal{P}, \mathsf{cid}, \mathsf{ssid}, \varepsilon, \varepsilon, \varepsilon, \mathsf{pk}_S, \mathtt{std}$) as input to $\mathcal{F}_{\mathsf{PHA}}$ from corrupt $\mathcal{P}'$ and message $\mathsf{ssid}, \mathsf{cid}, (\mathsf{ske}, b)$ for a channel cid connecting $\mathcal{P}'$ and simulated $\mathcal{P}$, who formerly received a corresponding input (pwInit, $\mathcal{P}', \mathsf{cid}', \mathsf{ssid}, \mathsf{uid}, *$) for such session, $\mathcal{S}$ distinguishes the following cases:

- $\mathcal{S}$ finds a registration transcript $\mathsf{uid}, \mathsf{ske}, \mathsf{pk}_S$ between $\mathcal{P}$ and some compromised party $\mathcal{P}''$. If such a transcript is found, and corrupt $\mathcal{P}'$ subsequently follows $\Pi_{\mathsf{TLS-OPAQUE}}$ (i.e., sends (SndrComplete, $\mathcal{P}''||\mathsf{uid}, \mathsf{ssid}', a$) for a, b received from $\mathcal{F}_{\mathsf{OPRF}}$ with $\mathsf{sid} = \mathcal{P}''||\mathsf{uid}$ and $\mathcal{A}$ acknowledges the authentication by sending bit 1 in AuthVerify, then simulator $\mathcal{S}$ queries $\mathcal{F}_{\mathsf{pwPHA}}$ with (Impersonate, ssid, $\mathcal{P}'', \mathsf{uid}, \perp$).
- $\mathcal{S}$ finds a former (key, $\mathsf{kid}, \mathsf{rw}, \mathsf{ske}, \mathsf{pk}_S, *$) output of $\mathcal{F}_{\mathsf{PHA}}$ towards a corrupt party or the adversary. If $\mathcal{S}$ finds pw corresponding to rw in $\mathcal{F}_{\mathsf{OPRF}}$ for $\mathsf{sid} = \mathcal{P}', \mathsf{uid}$), and additionally $\mathcal{A}$ acknowledges the authentication by sending bit 1 in AuthVerify, then $\mathcal{S}$ queries $\mathcal{F}_{\mathsf{pwPHA}}$ with (Impersonate, $\mathsf{ssid}, \varepsilon, \mathsf{uid}, \mathsf{pw}$).
- If none of the above happens, $\mathcal{S}$ sends (Impersonate, $\mathsf{ssid}, \varepsilon, \mathsf{uid}, \perp$) to $\mathcal{F}_{\mathsf{pwPHA}}$.

The changes only affect pwAuth records in $\mathcal{F}_{\mathsf{pwPHA}}$, and hence the output distribution is unchanged compared to the previous game. We have

$$\Pr[\mathsf{win}(\mathbf{G}_6)] = \Pr[\mathsf{win}(\mathbf{G}_5)].$$

Game $\mathbf{G}_7$: Abort upon ssid collision.
Changes to $\mathcal{S}$: We let $\mathcal{S}$ abort if a simulated party samples a value ssid^* in registration that was already used as ssid before anywhere else.

Due to the Birthday bound we have

$$|\Pr[\mathsf{win}(\mathbf{G}_7)] - \Pr[\mathsf{win}(\mathbf{G}_6)]| \leq \frac{q(q-1)}{2 \cdot 2^{-\lambda}},$$

with $q = q_I + q_R$, q_I being the number of pwInit queries and q_R being the number of StorePwdFile queries issued by $\mathcal{Z}$ towards honest parties.

Game $\mathbf{G}_8$: $\mathcal{F}$ decides connected honest channels.
Changes to $\mathcal{F}$: We add the pwProceed and pwDeliver interfaces to $\mathcal{F}$.
Changes to $\mathcal{S}$: On $\mathcal{A}$ sending f to $\mathcal{F}_{\mathsf{PHA}}$ during AuthVerify, if $\mathcal{P}, \mathsf{cid}$ corresponds to an unattacked channel with some $\mathcal{P}'$ where both $\mathcal{P}, \mathcal{P}'$ are honest, and simulated $\mathcal{P}$ had already sent (AuthVerify, cid,

$\mathsf{ssid}, \mathsf{ctx}, \mathsf{pk}$) to $\mathcal{F}_{\mathsf{PHA}}$, then $\mathcal{S}$ does the following. Let uid denote the user identifier used in authentication session ssid. Let ssid' denote the session identifier of the registration session between $\mathcal{P}$ and $\mathcal{P}'$ where uid got delivered, and set $b^* = f$.

- If $\mathcal{A}$ modified at least one of $\mathsf{pk}_S, \mathsf{ske}$ in session ssid', or if $\mathcal{A}$ delivered a message $\mathsf{ssid}, \mathsf{cid}, \mathsf{ske}', *$ for some $\mathsf{ske}' \neq \mathsf{ske}$ to $\mathcal{P}$, where ske was sent by $\mathcal{P}$ in session ssid', then set $b^* = 0$;
- If $\mathcal{A}$ uses i in session ssid' of $\mathcal{F}_{\mathsf{OPRF}}$, and i' in session ssid of $\mathcal{F}_{\mathsf{OPRF}}$ with $i \neq i'$, then $\mathcal{S}$ sets $b^* = 0$.
- If $\mathcal{A}$ uses $i = \mathcal{P} = \mathcal{P}''$ and additionally $\mathcal{A}$ modified any of a, b in ssid or of a', b' in authentication session ssid via AuthSend, then $\mathcal{S}$ sets $b^* = 0$.
- Send (pwDeliver, $\mathsf{ssid}, \mathcal{P}, b^*$) to $\mathcal{F}_{\mathsf{pwPHA}}$.

For indistinguishability, note that the pwDeliver interface lets $\mathcal{F}$ only create outputs for parties if it is called by $\mathcal{S}$, and hence the changes in this game only concern connected and fully honest channels. We consider several cases conditioned on which role $\mathcal{P}$ plays (initiator or responder), and which output it produced in the previous game.

First, we consider the trivial case of $\mathcal{P}$ not producing any output at all in the previous game. This happens only if $\mathcal{P}$ receives output AuthVerify of $\mathcal{F}_{\mathsf{PHA}}$ but is not expecting it, because it did not yet provide a public key for verification via query AuthVerify to $\mathcal{F}_{\mathsf{PHA}}$. The simulation in this game ensures that $\mathcal{P}$ in this game will also not produce any output, since $\mathcal{S}$ conditions the sending of pwDeliver on input AuthVerify being already sent.

1. Initiator $\mathcal{P}$ outputs `fail` in the previous game. This can happen because $\mathcal{F}_{\mathsf{PHA}}$ either detects mismatching passwords, does not have a corresponding Auth record, or the adversary sent $f = 0$ to $\mathcal{F}_{\mathsf{PHA}}$'s AuthVerify interface.
   (i) **Due to $f = 0$ by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{PHA}}$:** The initiator outputs `fail` in this game as well due to $\mathcal{S}$ sending $b^* = 0$ via pwDeliver.
   (ii) **Due to $f = 1$ and an Auth record with a different public key:** In $\mathbf{G}_7$, this happens iff $\mathcal{P}$ submits (AuthVerify, $\mathsf{ssid}, \mathsf{cid}, \mathsf{pk}_S$) but $\mathcal{F}_{\mathsf{PHA}}$ has stored a different pk'_S in the corresponding Auth record. $\mathcal{F}_{\mathsf{PHA}}$ stores pk'_S input by $\mathcal{P}'$ via AuthSend, where pk'_S is an uncompromised public key that was generated by $\mathcal{P}'$ via KeyGen. $\mathcal{P}$ retrieves via GetAuxData either (a) an adversarial $\mathsf{pk}_S \neq \mathsf{pk}'_S$ that was sent during registration, or (b) $\mathsf{pk}_S = \bot$. In case (b), since rw is unique and not known to the adversary, we can further partition in cases (b-i) $\mathcal{A}$ modifying ske (during registration by message tampering, or during authentication via an adversarial AuthSend or ActiveAttack), or $\mathcal{P}$ using a different rw than in registration due to (b-ii) $\mathcal{A}$ sending differing i, i' to $\mathcal{F}_{\mathsf{OPRF}}$, (b-iii) $\mathcal{A}$ sending $i = i' = \mathcal{P}''$ to $\mathcal{F}_{\mathsf{OPRF}}$ and additionally tampering with any of the OPRF transcript values a, b, a', b'. Finally, we have (b-iv) $\mathcal{P}$ using a different password in authentication than in registration. In cases (a),(b-i),(b-ii),(b-iii), our $\mathcal{S}$ of this game sets $b^* = 0$ and hence $\mathcal{P}$ outputs `fail` in this game as well. In the latter case (b-iv), $\mathcal{F}_{\mathsf{pwPHA}}$ has set `state` = `fail` in the pwAuth record. Altogether, $\mathcal{P}$ will output `fail` in this game as well.
   (iii) **Due to $f = 1$ but no Auth record:** In the previous game, at the time where honest party has sent query AuthVerify, non-existence of an Auth record cannot be due to missing inputs or adversarial acknowledgements, but only because the authenticating party used a public key not owned by it, or an invalid transportable key pair (see [S.2] in $\mathcal{F}_{\mathsf{PHA}}$ for the conditions under which the Auth record is created). In this game, in case $\mathcal{P}$ is the initiator, honest $\mathcal{P}'$ will always use the correct key, and hence this case cannot occur.
2. Initiator $\mathcal{P}$ outputs success in the previous game. This implies that $\mathcal{F}_{\mathsf{PHA}}$ generated output (AuthVerify, ..., 1) towards $\mathcal{P}$ after sending (AuthVerify, $\ldots, \mathsf{ctx}_S, \mathsf{pk}_S$), where pk_S was previously received via GetAuxData by $\mathcal{P} \Rightarrow$ Auth record with pk_S exists in $\mathcal{F}_{\mathsf{PHA}}$ and $\mathcal{A}$ sends $f = 1$ via AuthVerify $\Rightarrow \mathcal{P}'$ sent AuthSend. Additionally, $\mathcal{A}$ sends the same i in SndrComplete for the two corresponding $\mathcal{F}_{\mathsf{OPRF}}$ evaluations, and it does not set it to be $i = \mathcal{P}'$ in case $\mathcal{A}$ tampered with the registration transcript (except maybe pk_C). In this case, in this game, we

know that $\mathcal{S}$ acknowledged PWINIT and hence the PWAUTH record got initialized in $\mathcal{F}_{\mathsf{pwPHA}}$. The PWPROCEED interface, which must have been queried before $\mathcal{P}'$ sent AUTHSEND, rewrites the state in that record: since ske is unique, the only entry $\mathsf{tkey}[\mathsf{ak}, \mathsf{ske}]$ for ske has $\mathsf{ak} = \mathsf{rw}$ with rw output by $\mathcal{F}_{\mathsf{OPRF}}$ on input pw, where pw is the password that $\mathcal{P}$ used when file $*, \mathsf{ske}, \mathsf{pk}_S, *$ was stored. Hence, for GETAUXDATA to not return $(\perp, \perp)$, $\mathcal{P}$ must have input pk to PWINIT $\Rightarrow$ $\mathcal{F}_{\mathsf{pwPHA}}$ rewrites `state` to `match` in the PWAUTH record. Together with Sim sending $b^* = 1$, we conclude that $\mathcal{P}$ outputs match in this game as well.

3. Responder $\mathcal{P}$ outputs `fail` in the previous game. This works almost as the initiator case above, with the difference that initiator $\mathcal{P}'$ can indeed use an invalid transportable key pair $(\perp, \perp)$, which can happen if $\mathcal{A}$ tampered with ske or the OPRF evaluation as described above. In all cases these cases, $\mathcal{S}$ of this game sets $b^* = 0$ and hence $\mathcal{P}$ outputs `fail` in this game as well.
4. Responder $\mathcal{P}$ outputs success in the previous game. The argument is almost analoguous to the initiator case.

Hence we have

$$\Pr[\mathsf{win}(\mathbf{G}_8)] = \Pr[\mathsf{win}(\mathbf{G}_7)].$$

Game $\mathbf{G}_9$: $\mathcal{F}$ decides sessions with corrupt server.
Changes to $\mathcal{S}$: Upon $\mathcal{A}$ sending f to $\mathcal{F}_{\mathsf{PHA}}$ during AUTHVERIFY, if simulated $\mathcal{P}$ had already sent AUTHVERIFY to $\mathcal{F}_{\mathsf{PHA}}$ and $\mathcal{P}, \mathsf{cid}, \mathsf{ssid}$ correspond to a channel with honest $\mathcal{P}$ who initiated authentication session ssid, where the other endpoint, say, $\mathcal{P}'$, is corrupt, then $\mathcal{S}$ does the following. Let uid denote the uid used by $\mathcal{P}$ in authentication session ssid. Let ssid' denote the session identifier of the registration session between $\mathcal{P}$ and $\mathcal{P}'$. Set $b^* = f$.

- If $\mathcal{A}$ submits pk_S in AUTHSEND that it had not previously generated via KEYGEN, or one that differs from the public key sent to $\mathcal{P}$ during registration session ssid, then set $b^* = 0$.
- If corrupt $\mathcal{P}'$ sent message $\mathsf{ssid}, \mathsf{cid}, , \mathsf{ske}', *$ for a $\mathsf{ske}' \neq \mathsf{ske}$, where ske was sent by $\mathcal{P}$ in session ssid', then set $b^* = 0$;
- If $\mathcal{A}$ uses i in session ssid' of $\mathcal{F}_{\mathsf{OPRF}}$, and i' in session ssid of $\mathcal{F}_{\mathsf{OPRF}}$ with $i \neq i'$, then $\mathcal{S}$ sets $b^* = 0$.
- If $\mathcal{A}$ uses $i = \mathcal{P} = \mathcal{P}''$ and additionally $\mathcal{A}$ modified any $\mathsf{prfx}, \mathsf{pstfx}$ value of the two corresponding $\mathcal{F}_{\mathsf{OPRF}}$ sessions (either by sending different values to $\mathcal{P}$ than to $\mathcal{F}_{\mathsf{OPRF}}$ via its adversarial interface there, or by not taking the values received from $\mathcal{P}$ as input to FINALIZE), then $\mathcal{S}$ sets $b^* = 0$.
- Send (PWDELIVER, $\mathsf{ssid}, \mathcal{P}, b^*$) to $\mathcal{F}_{\mathsf{pwPHA}}$.

For indistinguishability, the argument in case of $\mathcal{P}$ not producing any output is the same as in the previous game. We now do the same case distinction as before, but with partly differing arguments of indistinguishability.

- $\mathcal{P}$ outputs `fail` in the previous game.
  (i) **Due to f by $\mathcal{A}$ to $\mathcal{F}_{\mathsf{PHA}}$:** As before.
  (ii) **Due to f and an AUTH record with different public key:** We now have to distinguish a different set of cases since, opposed to the previous game, rw values are not secret anymore. Case (a) remains unchanged, as well as its argument for indistinguishability. To argue for case (b), namely $\mathcal{P}$ obtaining $(\perp, \perp)$ via GETAUXDATA, we have to consider Game $\mathbf{G}_6$, where $\mathcal{S}$ issues (IMPERSONATE, $\mathsf{ssid}, *, \mathsf{uid}, *$). We show that this query cannot result in $\mathcal{F}_{\mathsf{pwPHA}}$ rewriting the corresponding record's state to `match`. In case $\mathcal{P}$ obtains $(\perp, \perp)$ upon (GETAUXDATA, $\mathsf{rw}, \mathsf{ske}$), $\mathcal{F}_{\mathsf{PHA}}$ does not have $(\mathsf{rw}, \mathsf{ske})$ registered as transportable key pair. Consequently, there was no output of such key pair towards $\mathcal{A}$ yet, and we know that $\mathcal{S}$ of Game $\mathbf{G}_6$ will not use an extracted password. We consider the first bulletpoint in Game $\mathbf{G}_6$. Assuming $\mathcal{S}$ finds a transcript containing ske submitted by $\mathcal{P}$. GETAUXDATA returning $(\perp, \perp)$ implies that this ske was not computed from rw, and hence in this case, $\mathcal{P}$ was using a different password in registration than in authentication. Therefore, the PWAUTH record

in $\mathcal{F}_{\mathsf{pwPHA}}$ will get rewritten to `fail` due to mismatching passwords. In all other cases, the Impersonate query with empty server and password fields will result in the pwAuth record to be rewritten to `fail`. Altogether, $\mathcal{P}'$ outputs `fail` also in this game.

(iii) **Due to $f = 1$ but no Auth record:** This happens if $\mathcal{A}$ uses an uncompromised pk_S in AuthSend. $\mathcal{S}$ of $\mathbf{G}_6$ will neither find a registration transcript with a compromised server, nor a former key output towards $\mathcal{A}$ containing pk_S. Hence, $\mathcal{S}$ will issue Impersonate with empty server and password fields, resulting in `fail` in the corresponding pwAuth record in $\mathcal{F}_{\mathsf{pwPHA}}$ and hence $\mathcal{P}$ outputs `fail` also in this game.

- $\mathcal{P}$ outputs `match` in the previous game. This happens iff $\mathcal{F}_{\mathsf{PHA}}$ outputs (AuthVerify, ..., 1) to $\mathcal{P}$ who previously sent (AuthVerify, ..., pk_S), where $\mathsf{pk}_S \neq \perp$ was previously received via (GetAuxData, $\mathsf{rw}, \mathsf{ske}$) by $\mathcal{P} \Rightarrow$ Auth record with pk_S exists in $\mathcal{F}_{\mathsf{PHA}}$ and $\mathcal{A}$ sends $f = 1$ via AuthVerify $\Rightarrow \mathcal{P}'$ sent (AuthSend, $\mathcal{P}, \mathsf{cid}, \mathsf{ssid}, \varepsilon, \varepsilon, \varepsilon, \mathsf{pk}_S, \mathtt{std}$). Additionally, $\mathcal{A}$ sends the same i in SndrComplete for the two corresponding $\mathcal{F}_{\mathsf{OPRF}}$ evaluations, and it does not set it to be $i = \mathcal{P}'$ in case $\mathcal{A}$ tampered with the registration transcript (except maybe pk_C). For $\mathsf{rw}, \mathsf{ske}, \mathsf{pk}_S$ to be a valid password file with compromised pk_S, either $\mathsf{rw}, \mathsf{ske}$ was generated by $\mathcal{A}$, or it was generated by an honest party that got compromised (these are the only two ways in which $\mathcal{F}_{\mathsf{PHA}}$ puts public keys in the list pkComp, which decides whether the adversary may use it or not). Our $\mathcal{S}$ of Game $\mathbf{G}_6$ correspondingly runs Impersonate on either the compromised party, or the password extracted from the adversarial OPRF evaluation. $\mathcal{F}_{\mathsf{pwPHA}}$ will rewrite the corresponding record to `match`, which $\mathcal{P}$ hence outputs in this game as well.

Altogether, we have

$$\Pr[\mathsf{win}(\mathbf{G}_9)] = \Pr[\mathsf{win}(\mathbf{G}_8)].$$

Note that the simulation at this point still works with real passwords, e.g., $\mathcal{F}_{\mathsf{PHA}}$'s GetAuxData interface, which can be accessed by $\mathcal{Z}$ via $\mathcal{A}$ or via corrupt parties, runs on rw values produced from the real passwords, ensuring indistinguishability from the real execution. We change this particular aspect in the next game and simulate GetAuxData replies and registration without knowledge of passwords. For this, note that GetAuxData allows to test whether envelopes ske can be "openend" with a value rw. The interface returns $(\perp, \perp)$ for all wrong rw/pw, and public keys for the correct rw. The simulation will exploit the link between rw and pw, established by $\mathcal{F}_{\mathsf{OPRF}}$: each password guess has to be submitted to $\mathcal{F}_{\mathsf{OPRF}}$ (and hence is seen by the simulator), to compute the corresponding rw value. Consequently, our simulator will submit adversarial OPRF evaluations as password guess via TestPwd, and uses the answer to simulate the GetAuxData reply correctly. The challenge is to ensure that there cannot be more such adversarial OPRF evaluations as available TestPwd queries.

Game $\mathbf{G}_{10}$: Extracting from a corrupt client (TestPwd).
Changes to $\mathcal{F}$: We add the TestPwd interface to $\mathcal{F}$.
Changes to $\mathcal{S}$: We let $\mathcal{S}$ simulate honest parties who input StorePwdFile without a password and leave the corresponding input field in the PRF table of $\mathcal{F}_{\mathsf{OPRF}}$ empty. Upon (GetAuxData, $\mathsf{rw}, \mathsf{ske}$) from $\mathcal{A}$, if ske was previously output by an honest party $\mathcal{P}$, let $\mathsf{uid}, \mathsf{pk}_S, \mathsf{pk}_C$ denote the rest of the registration transcript as seen by $\mathcal{P}$, and let $\mathcal{P}'$ denote the server. $\mathcal{S}$ retrieves input pw corresponding to rw from $\mathcal{F}_{\mathsf{OPRF}}$ with $\mathsf{sid} = \mathcal{P}'||\mathsf{uid}$. If such input is found, if $\mathcal{P}''$ is declared compromised in that OPRF instance, $\mathcal{S}$ sends (OfflTestPwd, $\mathcal{P}', \mathsf{uid}, \mathsf{pw}$) to $\mathcal{F}_{\mathsf{pwPHA}}$. If $\mathcal{P}''$ is honest, $\mathcal{S}$ sends (TestPwd, $\mathcal{P}', \mathsf{uid}, \mathsf{pw}$) to $\mathcal{F}_{\mathsf{pwPHA}}$. If no pw is found or the reply to the query is "wrong guess", then $\mathcal{S}$ replies with $(\perp, \perp)$. Otherwise, $\mathcal{S}$ replies with $(\mathsf{pk}_S, \mathsf{pk}_C)$. If $\mathcal{S}$ submits a password guess but does not get a reply, it aborts.

For indistinguishability, first note that TestPwd affects flags in pwAuth records in $\mathcal{F}_{\mathsf{pwPHA}}$ where the client is corrupt, which are however not used yet by $\mathcal{F}_{\mathsf{pwPHA}}$ to create output (only pwAuth records of fully honest sessions or ones with a corrupt server have an effect at this point). An abort happens only if $\mathcal{F}_{\mathsf{pwPHA}}$ does not reply to $\mathcal{S}$, which happens if there is either no record (file, $\mathcal{P}', \mathsf{uid}, *$), or $\mathsf{ctr}[\mathcal{P}', \mathsf{uid}] = 0$ and $\mathcal{P}'$ is not compromised. The former does not happen, because a completed registration transcript with both uid and ske implies that simulator $\mathcal{S}$ sent StorePwdComplete to $\mathcal{F}_{\mathsf{pwPHA}}$ (as of Game $\mathbf{G}_2$), and hence there exists a file $(\text{FILE}, \mathcal{P}', \mathsf{uid}, *)$. Regarding the counter, we need to ensure that there cannot be more adversarial evaluations of the OPRF with $\mathsf{sid} = \mathcal{P}'||\mathsf{uid}$ than password guesses available via $\mathcal{F}_{\mathsf{pwPHA}}$ if $\mathcal{P}'$ is not corrupt or compromised. Since $\mathcal{F}_{\mathsf{OPRF}}$ grants one PRF evaluation per input SndrComplete of $\mathcal{P}'$, which in turn is only issued by the honest $\mathcal{P}'$ after receiving pwProceed, and pwProceed increases the counter in this case, an abort never happens. It remains to argue that the output of GetAuxData is not distinguishable from the one in the last game. In case the reply was $(\mathsf{pk}_S, \mathsf{pk}_C)$ in the previous game, $\mathcal{A}$ used the same value rw in GetAuxData that $\mathcal{P}$ used in the registration, which was computed from pw that $\mathcal{P}$ received as input. Due to uniqueness and secrecy of rw values (cf. Games $\mathbf{G}_4$ and $\mathbf{G}_5$), there must be an entry pw, rw in $\mathcal{F}_{\mathsf{OPRF}}$ which was queried by the adversary. Hence, $\mathcal{S}$ of this game finds pw, TestPwd will result in "correct guess", and $\mathcal{S}$ outputs $(\mathsf{pk}_S, \mathsf{pk}_C)$ in this game as well. In case GetAuxData response was $(\perp, \perp)$ in the previous game, $\mathcal{A}$ did not use the same rw as used by $\mathcal{P}$ in registration, and there is either no pair $(\mathsf{pw}', \mathsf{rw}')$ in $\mathcal{F}_{\mathsf{OPRF}}$ or one where $\mathsf{pw}' \neq \mathsf{pw}$. Consequently, $\mathcal{S}$ also replies with $(\perp, \perp)$ in this game.

Hence we have

$$\Pr[\mathsf{win}(\mathbf{G}_{10})] = \Pr[\mathsf{win}(\mathbf{G}_9)].$$

Game $\mathbf{G}_{11}$: $\mathcal{F}$ decides sessions with corrupt client.

Changes to $\mathcal{S}$: Upon $\mathcal{A}$ sending f to $\mathcal{F}_{\mathsf{PHA}}$ during AuthVerify, if simulated $\mathcal{P}$ had already sent AuthVerify to $\mathcal{F}_{\mathsf{PHA}}$ and $\mathcal{P}, \mathsf{cid}, \mathsf{ssid}$ correspond to a channel with honest $\mathcal{P}$ who did *not* initiate authentication session ssid, where the other endpoint, say, $\mathcal{P}'$, is corrupt, then $\mathcal{S}$ does the following. Let again uid denote the uid used in authentication session ssid. Let ssid' denote the session identifier of the registration session of $\mathcal{P}$ where uid got delivered.

- In case $\mathcal{P}'$ uses a different pk_C (either in `std` or `tk` mode) in AuthSend in session ssid than what was registration session ssid', then set $b^* = 0$.
- In case $\mathcal{P}'$ uses an uncompromised pk_C in `std` mode in AuthSend in session ssid, or a `tk` key pair that was not previously output to $\mathcal{A}$, then set $b^* = 0$.
- In case $\mathcal{P}'$ uses a compromised pk_C in AuthSend (in any mode), $\mathcal{S}$ submits $(\text{TestPwd}, \mathcal{P}', \mathsf{uid}, \mathsf{pw})$ if it finds pw, rw in $\mathcal{F}_{\mathsf{OPRF}}$ corresponding to rw used to generate pk_C in $\mathcal{F}_{\mathsf{PHA}}$.
- Send $(\text{pwDeliver}, \mathsf{ssid}, \mathcal{P}, b^*)$ to $\mathcal{F}_{\mathsf{pwPHA}}$.

We proceed similar to the previous games regarding indistinguishability, and consider the following cases:

- $\mathcal{P}$ outputs `fail` in the previous game.
  (i) **Due to $f = 0$ from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{PHA}}$:** As before.
  (ii) **Due to $f = 1$ and Auth record with different public key:** This corresponds to the corrupt party using inconsistent keys pk_C in registration and authentication, in which case the simulation of this game will set $b^* = 0$.
  (iii) **Due to $f = 1$ but no Auth record:** This corresponds to corrupt $\mathcal{P}'$ using an uncompromised public key, or an invalid transportable key pair, in AuthSend. The simulation of this game ensures that $\mathcal{S}$ will overwrite b^* with 0.
- $\mathcal{P}$ outputs `match` in the previous game. This happens iff $\mathcal{F}_{\mathsf{PHA}}$ outputs $(\text{AuthVerify}, ..., 1)$ towards $\mathcal{P}$, where $\mathcal{P}$ previously received input $(\text{AuthVerify}, ..., \mathsf{pk}_C)$, where pk_C was received by $\mathcal{P}$ in the registration session for uid of ssid. Hence, $\mathcal{F}_{\mathsf{PHA}}$ holds an Auth record with pk_C, meaning that pk_C is a compromised key. Additionally, $\mathcal{A}$ acknowledged the authentication by sending bit 1 in AuthVerify. If pk_C was generated by the adversary, due to uniqueness of public keys enforced by $\mathcal{F}_{\mathsf{PHA}}$, the registration must have been through a corrupt client. The simulation of Game $\mathbf{G}_2$ ensures that the password used by the client is registered in $\mathcal{F}_{\mathsf{pwPHA}}$ as a file, and the simulation of this game ensures that a TestPwd query for that password is submitted

to $\mathcal{F}_{\mathsf{pwPHA}}$, rewriting the corresponding PWAUTH record to `match`. If pk_C was not generated by the adversary but an honest client, the simulation of Game $\mathbf{G}_{10}$ ensures that $\mathcal{S}$ submits TESTPWD with an extracted password pw used by $\mathcal{A}$ to generate rw and to unlock usage of pk_C in $\mathcal{F}_{\mathsf{PHA}}$ by using GETAUXDATA. The TESTPWD query will rewrite the corresponding PWAUTH record in $\mathcal{F}_{\mathsf{pwPHA}}$ to `match` and hence $\mathcal{P}$ outputs `match` in this game as well.

Altogether, we have

$$\Pr[\mathsf{win}(\mathbf{G}_{11})] = \Pr[\mathsf{win}(\mathbf{G}_{10})].$$

Game $\mathbf{G}_{12}$: Remove passwords from the simulation

Changes to $\mathcal{F}$: We stop the forwarding of passwords of honest party to $\mathcal{S}$.

Changes to $\mathcal{S}$: Whenever a simulated party initializes registration, $\mathcal{S}$ submits a dummy value to $\mathcal{F}_{\mathsf{OPRF}}$ instead, and it leaves the rw field empty in the key generation of $\mathcal{F}_{\mathsf{PHA}}$. For simulating honest clients in authentication, $\mathcal{S}$ submits a dummy value to $\mathcal{F}_{\mathsf{OPRF}}$. AUTHVERIFY messages of $\mathcal{F}_{\mathsf{PHA}}$ are simulated according to "correct guess" or "wrong guess" answers of TESTPWD and IMPERSONATE.

For indistinguishability, note that outputs of simulated parties are not used anymore as of Games $\mathbf{G}_8$, $\mathbf{G}_9$ and $\mathbf{G}_{11}$. Since AUTHVERIFY queries are simulated using $\mathcal{F}_{\mathsf{pwPHA}}$'s answers, $\mathcal{Z}$ does not see any rw-dependent values from the internal simulation, and hence we conclude that the changes do not affect the output distribution of the previous game.

It can be seen from the sequence of games that $\mathcal{F}$ of this game is equal to $\mathcal{F}_{\mathsf{PHA}}$, and the overall distinguishing advantage between the first and this game is negligible in λ. This concludes our proof.