The OPTLS Protocol and TLS 1.3

(extended abstract)

Hugo Krawczyk[*] Hoeteck Wee[†]

October 9, 2015

Abstract

We present the OPTLS key-exchange protocol, its design, rationale and cryptographic analysis. OPTLS design has been motivated by the ongoing work in the TLS working group of the IETF for specifying TLS 1.3, the next-generation TLS protocol. The latter effort is intended to revamp the security of TLS that has been shown inadequate in many instances as well as to add new security and functional features. The main additions that influence the cryptographic design of TLS 1.3 (hence also of OPTLS) are a new “0-RTT requirement" (0-RTT stands for “zero round trip time") to allow clients that have a previously retrieved or cached public key of the server to send protected data already in the first flow of the protocol; making forward secrecy (PFS) a mandatory requirement; and moving to elliptic curves as the main cryptographic basis for the protocol (for performance and security reasons). Accommodating these requirements calls for moving away from the traditional RSA-centric design of TLS in favor of a protocol based on Diffie-Hellman techniques. OPTLS offers a simple design framework that supports all the above requirements with a uniform and modular logic that helps in the specification, analysis, performance optimization, and future maintenance of the protocol. The current (draft) specification of TLS 1.3 builds upon the OPTLS framework as a basis for the cryptographic core of the handshake protocol, adapting the different modes of OPTLS and its HKDF-based key derivation to the TLS 1.3 context.

[*] IBM Research

[†] ENS, CNRS, INRIA and Columbia University. Supported in part by NSF Awards CNS-1237429 and CNS-1445424 and a Google Faculty Research Award.

Contents

1 Introduction

TLS (Transport Layer Security) [23, 24], often referred to as SSL, is the most important cryptographic protocol in current practice, responsible for providing security to web communications and many other applications. In spite of this fundamental role in the Internet security infrastructure, the protocol has been repeatedly shown to suffer of security weaknesses (see below list of references) that have been addressed with occasional updates and ad-hoc "patches". These recurrent modifications to the protocol add non-trivial complexity and are hard to implement and deploy due to the huge use basis of TLS and the myriad of different settings where it is applied. This state of affairs calls for a major revamping of the protocol design and the need to found it on on a clean and sound cryptographic design. This is a main goal of the work currently underway in the TLS Working Group of the IETF which is charged with defining the "next generation TLS", named TLS version 1.3.

In addition to the need to clean up the protocol, a major force behind the work on TLS 1.3 is the need to address new performance and security requirements from the TLS key-exchange protocol, namely, the *handshake protocol*[1]. These include: (i) a new "0-RTT option" that would reduce latency in cases where the client has a previously retrieved or cached public key of the server by allowing the client to transmit protected information already in the first flow of the protocol; (ii) adding forward secrecy (PFS) as a mandatory feature of the protocol; and (iii) moving to elliptic curves as the main cryptographic tool for the implementation of the protocol. These requirements call for moving away from the traditional RSA-centric design of the TLS Handshake in favor of a protocol that is based on Diffie-Hellman techniques. In particular, the 0-RTT requirements requires the inclusion of a one-message key-exchange protocol which, in turn, requires a KEM-like mechanism for authentication (digital signatures cannot provide authentication in a one-message protocol). Naturally, this mechanism needs to be based on Diffie-Hellman if implemented via elliptic curves.

At the same time, these new elements in the protocol need to mesh with further mechanisms to support other, more traditional protocol settings essential to TLS, including a one round-trip (1-RTT) protocol supporting the common case where a client does not have a pre-cached public key of the server, a protocol mode where client and server have a pre-shared key (PSK), an inexpensive means of resuming a session based on a previously exchanged session key, and more. Therefore, there is a need to design a cryptographic protocol simple enough to be practical and easy to specify (and manage), but at the same time flexible enough to accommodate all the above requirements and modes of operation of TLS.

In this paper we present OPTLS, a key-exchange (KE) protocol developed to inform the design of the handshake protocol in TLS 1.3 and which serves as a basis for the current (draft) specification of the TLS 1.3 handshake [51]. OPTLS is designed to respond to the above TLS 1.3 needs, namely, a sound cryptographic framework on which the different functionalities and requirements of the TLS handshake protocol can be based upon. Due to the complexity of the TLS environments, one cannot over-emphasize the importance of laying the cryptographic foundation of TLS on a simple protocol that provides sufficient flexibility to adapt to the many different use scenarios of TLS while offering a modular and uniform logic across its different modes. Such modularity and uniform logic is essential for aiding the specification of TLS 1.3 and its analysis, and as a basis for the many other protocols and applications that are based on TLS. We note that while the immediate motivation of our work on OPTLS is to support the specification of TLS 1.3, its flexibility and modularity are likely to help its application to other key exchange settings. Most of the presentation here is independent of TLS details.

[1] TLS 1.3 will also address shortcomings from the *record protocol*, the part of TLS that protects data with the keys produced by the handshake protocol, but our work focuses on the handshake protocol.

Our presentation starts with Section 2 where we provide a high-level and informal description of OPTLS. It is intended to highlight the design approach, the common logic of its different variations and modes, and the flexibility of the design that allows for different optimizations depending on the use case. In Section 3 we present the security model on which the OPTLS analysis is based and in Section 4 we formally describe OPTLS and present an analysis of its modes and underlying key derivation scheme, where the latter serves as a unifying element across the different protocol modes. Finally, Section 5 describes the adopted uses of OPTLS in the specification of TLS 1.3.

We consider the simplicity of the OPTLS protocol and its analysis as strengths of this work. It is important that the basic cryptographic core of an "uber standard" like TLS be simple and amenable to analysis by both cryptographers and existing tools for automated verification. Using standard, well understood techniques is yet another advantage of OPTLS.

Related work. In view of its importance, TLS has long been the subject of intense research analysis and attacks, including, in chronological order, [53, 16, 50, 37, 34, 52, 22, 35, 5, 47, 31, 6, 48, 26, 49, 11, 2, 32, 46, 27, 18, 46, 3, 12, 4, 13, 10]. Particular attention has been given in recent works to the analysis of TLS 1.2, starting with variants of the protocol in [34, 48], to the cryptographic core in [32, 41, 29, 43] and then a more complete specification [14]. The recent QUIC protocol from Google by Langley and Chang [42], designed to add support to the 0-RTT setting, has also been the subject of formal analysis by Fischlin and Günther [28] and by Lychev et al. [44]. In recent concurrent works, Kohlweiss et al. [36] and Dowling et al. [25] analyzed earlier candidates of TLS 1.3 drafts which predate the full-fledged adoption of OPTLS; in particular, both works covered a variant of our 1-RTT non-static protocol, but do not cover the 0-RTT, PSK and 1-RTT semi-static modes.

OPTLS borrows from QUIC a similar mechanism for supporting the 0-RTT case and for providing forward secrecy to the protocol, but extends this mechanism to cover other use cases in TLS, for example, several 1-RTT settings with and without server's signatures, a non-public-key mode based on a shared symmetric key between client and server, and more. The 1-RTT non-static protocol is also similar to (a one-way authenticated variant of) the SIGMA-I protocol from [20, 38]. The OPTLS approach is to build the protocols for these different authentication and keying settings in a modular way and under a common logic, using key derivation as a unifying element for all modes.

For the analysis of OPTLS, we use the Canetti-Krawczyk security model [19] that we adapt to the server-only authentication setting. For the analysis of the 0-RTT mode, we use the specialization of this model to one-message key-exchange protocols from [30].

Future work. This paper focuses on the design of OPTLS and its applicability to the cryptographic core of TLS 1.3. The analysis presented here, carried in a basic security model such as [19], is essential for driving protocol design and to support the soundness of the core protocol. Yet, this analysis is not intended to satisfy the more comprehensive task of analyzing a complex specification such as TLS 1.3 and its use in the real world. Future analysis work will be needed to address these more complex settings and more involved security requirements. In particular, while we analyze each protocol mode in isolation, it is necessary to further analyze the interaction between the different modes and between multiple versions of TLS (e.g. a client initiating a TLS 1.3 connection with a server that only supports TLS 1.2). Recent analysis tools developed for TLS 1.2 and tools from automated verification (cf. the miTLS project [12, 14]) are well suited for these more comprehensive analytical tasks. In addition, our work does not include some of the functionalities included in the full TLS handshake specification such as client authentication and resumption. See Section 5.2 for further discussion of these issues.

2 OPTLS Overview

$$\xrightarrow{\overbrace{n_C, \mathsf{negot}_C}^{\mathsf{chello}},\ \overbrace{g^x}^{\mathsf{ceks}}}$$

$$\xleftarrow{\overbrace{n_S, \mathsf{negot}_S}^{\mathsf{shello}},\ \overbrace{g^y}^{\mathsf{seks}},\overbrace{\mathsf{cert}_S}^{\mathsf{ssks}^+},\overbrace{\mathsf{MAC}_{\mathsf{sfk}}(n_C, \mathsf{negot}_C, g^x, n_S, \mathsf{negot}_S, g^y, \mathsf{id}_S)}^{\mathsf{sfin}}}$$

Figure 1: OPTLS Basis (c.f. 1-RTT semi-static). sfk is derived from g^{xs} and atk from g^{xy} and g^{xs}. See Fig 2 for the full OPTLS protocol.

We provide a high-level overview of OPTLS intended to convey the general logic and rationale of the protocol and its main features. Details are given in Section 4 where a formal treatment of the protocol and its analysis are presented; in particular, Figure 2 describes the different modes of OPTLS. Section 5 explains the uses of OPTLS in TLS 1.3.

OPTLS basics. OPTLS is based on the simple protocol depicted in Figure 1 run between a client C, on the left side, and a server S; it corresponds to the 1-RTT semi-static mode described in Section 4. This version provides server-only authentication (the most common use case in TLS) and assumes the possession by the server S of a CA-signed DH certificate cert_S binding the server's identity to a DH key g^s (where s is the server static secret key). We note that while authentication via DH certificates will not be the typical use case in TLS, this mode is useful to illustrate the logic of OPTLS (in the more common setting, this logic is preserved but the server itself will sign a static or ephemeral key using a signature certificate). The values n_C, n_S represent unique session nonces and $\mathsf{negot}_C, \mathsf{negot}_S$ represent protocol parameters negotiated by the parties to agree on a protocol version, cryptographic algorithms, etc. Authentication is provided by a MAC[2] computed with key sfk derived from the value g^{xs}; values x and y are ephemeral secrets chosen by the client and server, respectively. The information shown under the MAC represents the information that must be authenticated but in the actual protocol other elements may be included (in TLS, the input to the MAC is a hash of the handshake transcript).

The output of the protocol is a session key derived jointly from g^{xs} and g^{xy} with the former ensuring securing for as long as the server's private key s is secure and the latter providing security in case of leakage of s, namely, *perfect forward secrecy (PFS)*. Details of key derivation are discussed below. We note that the session key in the TLS setting is the *application traffic key (*atk*)* used to protect data transmitted over the record protocol, via the use of authenticated encryption (AEAD) [49, 45, 9, 37].

In spite of its simplicity and schematic representation, the underlying logic of the protocol in Figure 1 is demonstrated and preserved throughout all variations and modes of OPTLS. That is, a two-layer authentication logic where the server's DH key g^s is authenticated by a signature (in the example of Figure 1 this is done via a CA-signed certificate) and this key g^s is then used to authenticate the transcript of the protocol through a MAC (which makes this MAC, i.e., the mandatory server's Finished message in TLS 1.3, an essential piece in handshake authentication). As we see below, due to functional requirements of TLS 1.3 (e.g., 0-RTT) and the limited deployment of DH certificates, OPTLS supports other ways

[2] This MAC is called the *server's Finished message (*sfin*)* and the key to the MAC is denoted sfk for "server's finished key".

of implementing the top-layer authentication, namely, different ways of managing the authentication of the server's key g^s. The flexibility and uniform logic of the protocol is not only demonstrated through the many protocol variants but also in the ability to accommodate different optimization trade-offs (more below).

From DH certificates to signature certificates. Since in today's practice servers typically possess certified signature keys, we adapt the above protocol to more available settings by using the server's signature key to sign the static DH key g^s. Thus, instead of a DH certificate, we now consider the transmitted cert_S to be a certificate binding the server's identity to a signing key, and add a signature of the server on the static DH key g^s. This signature can take one of two forms: (i) the signature covers g^s and a validity period through which a client can accept server authentication based on g^s; or (ii) the signature covers g^s and the client's nonce n_C. The latter can be thought of a validity period that lasts for the current session only, i.e., ensures the freshness of the signature. A clear advantage of option (i) is that it saves the cost of a per-session signature, only requiring the server to periodically (offline) sign g^s values (after which a client can cache g^s); it also provides much flexibility to the management of static keys by a server including the use of short-lived g^s values. On the other hand, offline signing empowers the server's signature with certification capabilities (for certifying g^s) something the current TLS 1.3 specification does not support[3] although an extension for such a mode is in the plans. Option (ii) resolves the concerns with offline signing at the cost of requiring an *online* signature, one per session, that can be costly, especially while using RSA signing keys (the prevalent case today). As we see next, this cost can be amortized via caching of keys g^s.

Cached keys and 0-RTT. An operational mode required by TLS 1.3 and supported by OPTLS is the use of server's static keys cached by a client or retrieved out of band. This is needed to support a "0-RTT mode" in which the client can send data already in the first message of the protocol, following the value g^x (see early data in Figure 2), and protect it using authenticated encryption under a key derived from g^{xs}. Such a 0-RTT mode is mandatory for TLS 1.3 and is intended to decrease the latency introduced by the rounds of communication imposed by prior versions of TLS. A typical application example [42] is that of a user sending a TLS-protected query to a search engine; with current TLS, the browser and server need to exchange handshake-specific messages before they can establish the secure channel on which to transmit the query. Using the 0-RTT mode, the client can send its protected query already in its first message. In addition, cached server's keys (received by the client in a previous session with the server) also allow for handshake sessions without signatures at all, something that may be particularly beneficial when using RSA signatures (a 2048-bit RSA signature is an order of magnitude more costly than a variable-base 256-bit ECDH exponentiation).

Replay protection. An important consideration for the use of a 0-RTT mode is the fact that data (early data) transmitted in the first message under the protection of g^{xs} is open to replay by an attacker. OPTLS does not include a specific anti-replay mechanism as this can be implemented in different forms by different implementations, independently of the KE mechanism. A typical anti-replay approach is for the server to keep state about the 0-RTT messages (e.g., client nonces) it has seen and reject duplicates. QUIC [42], for example, includes such an elaborate mechanism. We caution that managing an anti-replay mechanism at the "Internet scale" is very non-trivial as recently observed in the TLS mailing list[4] and much care needs to be exercised when assessing a particular mechanism.

[3] A potential weakness arises from server implementations that protect the signing key, e.g., via hardware protection, but may be vulnerable to occasional unauthorized use of the signing capability that would allow an attacker to obtain a signature on a value g^s of her choice and use it to impersonate the server for a long period of time.

[4] `http://www.ietf.org/mail-archive/web/tls/current/msg15594.html`

Optimizations. One of the salient properties of OPTLS is its flexibility and the way it can support different performance needs. For example, consider the use of server signatures in the protocol as discussed above. OPTLS provides variants where server signatures are not used at all (the case of DH certificates), or only used offline, or used online but amortized over repeated uses of cached g^s keys. These are all very attractive options for saving in the number of signing operations which is particularly important when considering RSA signatures. On the other hand, for ECDSA signatures where signing is very efficient, one may not want to focus on saving signing operations and prefer saving on DH exponentiations. Here is how this can be achieved with OPTLS. Consider for simplicity a server that does not support cached keys or DH certificates. The client sends g^x and the server responds with a signed g^s, the ephemeral g^y, and a MAC computed using a key derived from g^{xs}. This costs the server a signature and two DH variable-base exponentiations (the latter for computing g^{xs} and g^{xy}). However, since the client does not cache the server's key, the server can use g^s as a one-time value (used only for this particular session) and eliminate g^y, where the one-time g^s provides PFS. Authentication is still obtained via the MAC computation as in the basic logic of OPTLS. The cost is one signature and one DH exponentiation, hence accomplishing the saving of one DH exponentiation as intended. This is how the 1-RTT non-static mode is defined (except that in the description of this mode in Figures 2 and 3 we follow the TLS 1.3 notation and use g^y in the role of g^s).

Key derivation. The two central ingredients in the key derivation of OPTLS are the DH values g^{xs}, from which the MAC key sfk for server authentication is derived, and g^{xy} that serves as input to the derivation of the session key atk for providing PFS. More precisely, OPTLS strengthens the session key by deriving it from both g^{xy} and g^{xs}, hence ensuring the security of the session key even upon compromise of one of the secrets y or s (assuming the compromise of s happens after the handshake is complete). In the 0-RTT case where the client sends protected data in the first protocol message, the authenticated-encryption key is derived from g^{xs} only. OPTLS implements these key derivations using the HMAC-based construction HKDF [40, 39] that serves as a unifying tool for the multiple derivations in the protocol. This results in a uniform *derivation tree*, based on the HKDF extract and expand components, common to all of the protocol modes (see Figure 2 and note the notation ssk and esk for g^{xs} and g^{xy}, respectively).

Recall that HKDF takes as inputs: an initial keying material value (IKM) from which keys are derived, a constant or random salt value, a context value info, and the total length of keys to be output. In OPTLS, the info value is the concatenation of a unique label (indicating the purpose of the derived key) with information from the protocol's transcript that includes nonces, identities, security parameters, cryptographic algorithms, etc. For deriving the server's MAC key sfk and the client's authenticated-encryption key for protection of 0-RTT data, IKM is set to $\mathsf{ssk} = g^{xs}$ and the salt value to 0. For the derivation of the session key atk, IKM is set to mES which is derived from $\mathsf{esk} = g^{xy}$ and the salt is set to mSS that is derived from $\mathsf{ssk} = g^{xs}$; this achieves the mixing of g^{xy} and g^{xs} in the derivation of the session key atk for ensuring, as explained above, the security of this key even if one of y or s leak.

Interestingly, for analyzing these key derivations we resort to the full modeling power of HKDF: The derivation from g^{xs} uses a salt value of 0 which is equivalent to modeling HMAC as a random oracle (this is necessary to ensure CCA security of the key g^{xs} [1])[5].

For the derivation of the session key atk we consider two cases. First, if s is uncompromised then mSS is pseudorandom (by virtue of the secrecy of g^{xs} and the modeling of $\mathsf{HKDF}(0,\cdot)$ as a random oracle) and

[5] We believe that one should confine the use of the random oracle model to a minimum and even if one protocol component requires RO abstraction for its proof, the analysis of other components should strive to avoid it. Note that [39, 40] recommend the use of a random salt value, e.g., the parties' contributed nonces; however, in our case only the client's nonce would be available for this but this value is not authenticated.

we only need to assume HMAC to be a PRF; note that in this case atk is secure even if y is disclosed, all we need is g^y to be unique. Second, if s leaks (after the session key is computed, hence ensuring that g^{xs} was not adversarially chosen) then mSS leaks too but it is still the output from a random oracle on a non-adversarial input, hence it serves as a public random salt in the derivation of atk; in this case it suffices to assume that HMAC is a (strong) randomness extractor seeded by the salt mSS.

Pre-shared key. While the main operational mode of TLS (and OPTLS) assumes that the server possesses a long-term public key (typically a signature verification key), there is an important use case in which the client and server enter the protocol with a previously shared secret key. This mode, called Pre-Share Key needs to be supported in TLS 1.3 too (where it is also used as a session resumption mechanism). OPTLS naturally accommodates this mode by simply replacing the key g^{xs} (for the MAC computation and key derivation) with the pre-shared secret. The other input to key derivation, g^{xy}, is preserved hence providing PFS (the resultant mode is called PSK-DHE). One can also obtain pre-shared key mode (PSK) without PFS by eliminating g^x and g^y which results in a very efficient mode with only symmetric key operations.

Handshake encryption. TLS 1.3 requires its handshake messages to be protected to the extent possible with authenticated encryption (AEAD). This is intended to hide the contents of some messages such as the server identity/certificate from an observer. For this, OPTLS derives an additional handshake traffic key (htk) from g^{xy} and starts encrypting all handshake traffic following the g^y value sent in the server's message (the first client's message is not protected, except for client's $\mathsf{early\ data}$ in the 0-RTT case). In OPTLS and TLS 1.3, the key htk for encrypting handshake traffic is computational independent of that for application traffic, even though both keys are derived from g^{xy}. Using separate keys for handshake and application traffic allows the use of standard KE models [7, 19] that require that session keys remain indistinguishable from random at the end of the KE run. In addition, it enables modular and composable design where one can analyze the KE protocol independently of any specific application [19, 17]. In contrast, TLS 1.2 uses the application key to encrypt handshake messages, adding complexity to the analysis and violating the generic security and modular composability of the key exchange protocol. We note that OPTLS guarantees the security of the KE protocol even without encryption of the handshake traffic. In fact, we first analyze the KE protocol without handshake encryption, and then extend our analysis to the setting with handshake encryption.

Protocol analysis. In Section 4 we present a formal specification and analysis of the OPTLS protocol. The analysis is carried in the Canetti-Krawczyk model of KE protocols [19, 21] that allows for the consideration of leakage not only from session keys but also from the session's temporary values such as ephemeral exponents and intermediate keys. More specifically, we use the CK model with server-only authentication, as well as its specialization to one-message KE protocols [30] as needed for analyzing the 0-RTT mode of the protocol. Much of the analysis is standard and, thanks to the uniform logic of OPTLS, it applies with relatively small adaptations to the different variants of the protocol. As discussed in the introduction, the analysis of OPTLS in such a model is essential for gaining confidence in its security and for guiding the very design of the protocol as well as its use in TLS 1.3. Additionally, we purposely minimize the elements in the protocols to understand what is essential for security and what is not: not only for the sake of analysis but also to know what's "the cryptographic core" that must be preserved when adapting it to new uses.

At the same time, we stress that a complete analysis of a standard like TLS 1.3 will require major additional efforts in several directions. Considerations should include: additional pieces of TLS such as client authentication and resumption; security requirements not contemplated by the basic security model used here, such as resistance to “key synchronization attacks”; the interaction between different modes and components of TLS, including the co-existence with previous versions of TLS; specification and implementation issues as well as varying use cases and application-specific semantics; and more. Fortunately, much work has been done in these directions with TLS 1.2 (see introduction) and we expect TLS 1.3 (partly thanks to its OPTLS basis) to be significantly more amenable to analysis. We anticipate that the analysis presented here will serve as a basis for the above more comprehensive studies, in particular as a way to frame in a formal way the cryptographic logic of the protocol, as well as to guide future extensions to the protocol (e.g., the use of DH certificates, offline signatures, and further optimizations not currently supported in the TLS 1.3 spec).

OPTLS in TLS 1.3. The main motivation for the design of OPTLS is to serve as a basis for the handshake protocol of TLS 1.3 and, in particular, to respond to the new requirements of the TLS working group as discussed earlier, namely, the provision of a 0-RTT mode, making forward secrecy an integral part of the protocol, and accommodating elliptic curves as the preferred cryptographic machinery. OPTLS delivers these requirements while offering a combination of conceptual simplicity, design flexibility, and a uniform logic that unifies the different protocol variants. These are all attractive properties for the analysis, specification and future adaptations of a complex protocol like TLS.

Due to these properties and flexibility, OPTLS is being adopted and adapted as a basis for the design of the TLS 1.3 Handshake protocol [51]. At the same time, a specification of a complex protocol like TLS needs to manage the tension between the need to support multiple use scenarios and the need to reduce the number of options and execution paths allowed by the protocol. The latter is needed for reducing the overall complexity of the protocol, simplifying its specification, and avoiding implementation errors.

In the case of TLS 1.3, its designers can choose the subset of OPTLS variants that best fit the functioning and security requirements of the protocol. These choices mainly refer to the properties of g^s, such as specifying the method for authenticating this key and whether it is static, semi-static or non-static key (roughly correspond to a long-term g^s as in the case of DH certificate, a short term cacheable key, or a one-time key as discussed in the optimization section above and in more detail in Section 4.4). Importantly, the execution paths of the protocol are similar for all these cases as exemplified by the uniform key derivation strategy for the different protocol modes (see Figure 7).

Moving from the theoretical core of the protocol to the TLS environment requires adaptations and extensions of the protocol that we discuss in Section 5. These changes are in the form of additions to OPTLS, e.g., more inputs to the key derivation function or additional server-side signatures, that preserve the core OPTLS modes and therefore enjoy the security guarantees provided by our formal analysis from Section 4. Another element added in TLS 1.3 is a third message, the client’s Finished, where the client sends a MAC computed on the handshake transcript. As our analysis shows, this message is not necessary for achieving the basic key exchange security captured by the basic security model. Such message does have significance as it serves to confirm before the end of the handshake that server and client share the same view of the protocol execution, including shared key material and negotiated parameters. Finally, we point out to two modes supported by TLS 1.3 and not treated here. One is session resumption that is essentially reduced to the pre-shared mode PSK and an optional client authentication mode whose design (and requirements) are still under discussion. See more on these issues in Section 5.2.

3 Cryptographic preliminaries

Cryptographic groups. We consider cyclic groups of order p generated by a generator g. We assume that we can efficiently verify membership in the group, and that both parties will always verify membership in the group throughout the protocol. The Gap Diffie-Hellman (gap-DH) assumption [1] asserts that computing g^{uv} given (g, g^v, g^u) is hard on average, even given access to a verification oracle $(g, g^v, \cdot, \cdot)$ for DDH tuples. In other words, CDH is hard given a DDH oracle.

HKDF assumptions. HKDF [40, 39] uses a two-step mechanism by which one first extracts randomness from a possibly non-uniform secret input to create a uniformly random string that is then used with a regular PRF to generate keying material. Here, we denote the two steps as HKDF.Extract and HKDF.Expand. The rationale of the extraction part is to concentrate the "secret entropy" of the input on a single strong cryptographic key (from which more keys can be derived).

- When the first input is 0, we treat $\mathsf{HKDF.Extract}(0, \cdot)$ as a random oracle, and as a deterministic extractor when applied to flat high-entropy sources. In particular, if the input is uniformly random, then the output is uniformly random. When the first input is a random string K, we treat $\mathsf{HKDF.Extract}(K, \cdot)$ as a pseudo-random function with key K if K is secret, and as a (strong) extractor with seed K if K is public.
- We treat $\mathsf{HKDF.Expand}(K, \cdot)$ as a (variable-length output) pseudo-random function with key K. HKDF.Expand receives as an additional input a string denoting context information related to the derived keys (called info in Section 2). We use unique labels for domain separation when using HKDF.Expand to derive multiple keys.

3.1 Security model for key exchange

This section presents an abridged description of the Canetti-Krawczyk security model for key-exchange protocols [19, 21, 8] on which all the analysis work in this paper is based. The reader familiar with this model can skip this section. On the other hand, please consult [19] for complete details.

A key-exchange (KE) protocol is run in a network of interconnected parties where each party can be activated to run an instance of the protocol called a session. Within a session a party can be activated to initiate the session or to respond to an incoming message. As a result of these activations, and according to the specification of the protocol, the party creates and maintains a session state, generates outgoing messages, and eventually completes the session by outputting a session key atk and erasing the session state. A session may also be aborted without generating a session key. A KE session is associated with its owner (the party at which the session exists), a peer (the party with which the session key is intended to be established), and a session identifier.

Server authentication. We assume that each server owns a long-term pair of private and public keys, and that other parties can verify the authentic binding between an identity and a public key. For concreteness, we will assume that this binding assurance is provided by a certification authority (CA) which is only trusted to correctly verify the identity of the registrant of a public key before issuing the certificate that binds this identity and the public key. No other actions by the CA are required or assumed; in particular, we make no assumption on whether the CA requires a proof-of-possession of the private key from a registrant of a public key, and we do not assume any specific checks on the value of a public key. In particular, a corrupted party can choose (at any point during the protocol) to register any public key of its choice, including public keys equal or related to keys of other parties in the system. In this paper, a public key will always be a static Diffie-Hellman value and the private key its secret exponent. We use $\mathsf{ssks} = g^s$ to denote the server's DH key share, and we use ssks^+ to denote the server's authenticated DH key share. That is, ssks^+ includes both the server's identity sid, along with a signature chain starting with the CA authenticating both ssks and sid.

Attacker model. The attacker is modeled to capture realistic attack capabilities in open networks, including the control of communication links and the access to some of the secret information used or generated in the protocol. The basic principle is that since security breaches happen in practice (e.g., via break-ins, mishandling of information, insider attacks, cryptanalysis), a well-designed protocol needs to *confine* the damage of such breaches to a minimum. In particular, the leakage of session-specific ephemeral secret information should have no adverse effect on the security of any other *non-matching* sessions.

The attacker, denoted $\mathscr{A}$, is an active "man-in-the-middle" adversary with full control of the communication links between parties. $\mathscr{A}$ can intercept and modify messages sent over these links, it can delay or prevent their delivery, inject its own messages, interleave messages from different sessions, etc. (Formally, it is $\mathscr{A}$ to whom parties hand their outgoing messages for delivery.) $\mathscr{A}$ also schedules all session activations and session-message delivery. In addition, in order to model potential disclosure of secret information, the attacker is allowed access to secret information via the following attacks:

- A $\mathsf{StateReveal}()$ query is directed at a single session while still incomplete (i.e., before outputting the session key) and its result is that the attacker learns the session state for that particular session (which may include, for example, the secret exponent of an ephemeral DH value but not the long-term private key used across all sessions at the party).

- A $\mathsf{Reveal}()$ query can be performed against an individual session after completion and the result is that the attacker learns the corresponding session key atk (this models leakage on the session key either via usage of the key by applications, cryptanalysis, break-ins, known-key attacks, etc.).

- A $\mathsf{Corrupt}()$ means that the attacker learns the long-term secret of that party; in addition, from the moment a party is corrupted all its actions may be controlled by the attacker.

Basic security. To define security, we need a notion of $\mathsf{matching}$. For the protocols in this paper, we use a prefix of the transcript as the session identifier. That is, we say that two sessions are $\mathsf{matching}$ if the transcripts at both sessions begin with the same $\mathsf{sessionid} = (\mathsf{chello}, \mathsf{ceks}, \mathsf{shello}, \mathsf{seks}, \mathsf{ssks}^+)$. Recall that ssks^+ includes the server's identity sid.

Sessions against which any one of the above attacks
$\mathsf{StateReveal}(), \mathsf{Reveal}(), \mathsf{Corrupt}()$ is performed (including sessions compromised via party corruption) are called $\mathsf{exposed}$. In addition, a session is also called exposed if the matching session has been exposed (since matching sessions output the same session key, the compromise of one inevitably implies the compromise of the other).

The security of session keys generated in unexposed sessions is captured via the inability of the attacker $\mathscr{A}$ to distinguish the session key of a $\mathsf{test\ session}$, chosen by $\mathscr{A}$ among all complete sessions in the protocol, from a random value. This is captured via a $\mathsf{Test}()$ query. This query may only be asked once during the security game. Sets $K_0 := \mathsf{atk}$ (or random if $\mathsf{atk} = \perp$) as the real session key, and sets $K_1 \leftarrow_{\mathrm{R}} \{0,1\}^\lambda$. Then, it picks $b \leftarrow_{\mathrm{R}} \{0,1\}$ and returns K_b. In the case of server-only authentication, the $\mathsf{Test}()$ query is always issued against a client session whose peer corresponds to the identity of an honest server.

The attacker can continue with the regular actions against the protocol also after the Test() query; at the end of its run $\mathscr{A}$ outputs a bit b', which is meant as a guess for the value of b.

A key-exchange protocol π is called secure [19] if for polynomial-time attackers $\mathscr{A}$ running against π it holds:

1. If two uncorrupted parties complete matching sessions in a run of protocol π under attacker $\mathscr{A}$ then they output the same key (except for a negligible probability).
2. The probability that the test session is not exposed and $\mathscr{A}$ output a correct guess $b = b'$ is at most $1/2$ plus a negligible fraction.

Forward secrecy. An stronger notion of security is that of perfect forward secrecy (PFS), namely, established session keys remain secure even if the parties are corrupted and their long-term secrets are exposed. More formally, a key-exchange protocol is said to achieve PFS security if it achieves basic security even if we relax the definition of unexposed sessions to allow a Corrupt() query upon completion of the session.

Resilience to disclosure of ephemeral exponents. Another stronger notion of security is that of resilience to disclosure of ephemeral exponents, namely, establish session keys remain secure even if some ephemeral exponents are leaked. More formally, a key-exchange protocol is said to achieve resilience to disclosure of ephemeral exponents if it achieves basic security even if we relax the definition of unexposed sessions to allow a StateReveal() only on the server side. (For the protocols in this paper, it is impossible to protect the session key upon a StateReveal() on the client side.)

Simplifications and conventions. Following [41], we assume selective security in the choice of the test session, which implies fully adaptive security at a security loss which is linear in the total number of sessions. In our proofs we will assume the existence of *a single honest client and a single honest server*. This simplification is without loss of generality since we may simply simulate all other clients and servers.

4 Cryptographic Core of OPTLS

The OPTLS protocols have two flows as shown in Fig 2. Before going into detail, we reiterate that the OPTLS protocols are designed to achieve basic KE security (c.f. Section 3.1) and do not incorporate many elements in TLS 1.3 that are necessary for operating in the real world; we elaborate on these latter issues in Section 5.

OPTLS modes. There are four primary modes for OPTLS:

- 1-RTT semi-static, where the server has a semi-static g^s which can also be used to support early application data in a 0-RTT mode;
- 1-RTT non-static, where there is no static g^s and the ephemeral g^y plays the role of g^s;
- PSK for use with a pre-shared secret key (an existing secret shared between the server and the client);
- PSK-DHE which augments the PSK mode with a DH exchange to achieve forward secrecy.

OPTLS protocol flows

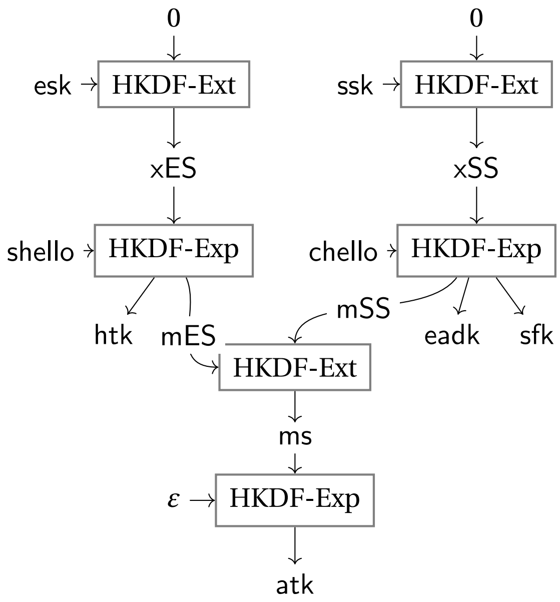
$$\xrightarrow{\text{chello},\ \overbrace{\text{ceks}}^{g^x},\ \overbrace{\{\text{early data}\}_{\text{eadk}}}^{\text{optional}^{1,3,4}}}$$

$$\xleftarrow{\text{shello},\ \overbrace{\text{seks}}^{g^y},\ \{\ \overbrace{\text{ssks}}^{g^s \text{ or } g^y}{}^{+},\ \overbrace{\text{MAC}_{\text{sfk}}(\text{chello}, \text{ceks}, \text{shello}, \text{seks}, \text{ssks}^{+})}^{\text{sfin}}\ \}_{\text{htk}}}$$

OPTLS modes

[1] 1-RTT semi-static (ssks = g^s)
[2] 1-RTT non-static (ssks = g^y)
[3] PSK (ssks$^+$, ceks, seks omitted)
[4] PSK-DHE (ssks$^+$ omitted)

OPTLS key derivation



Protocol notation

chello, shello	nonce + cryptographic negotiation parameters
ceks, seks	ephemeral key shares g^x, g^y
ssks	server's key share g^s
ssks$^+$	signed server's key share, includes server's identity sid
sfin	server's Finished message MAC$_{\text{sfk}}$(chello, ceks, shello, seks, ssks$^+$)
xSS, xES, mES, mSS	intermediate results in key derivation

Key derivation inputs (per-mode values ssk and esk)

1-RTT semi-static	ssk = g^{xs}, esk = g^{xy}
1-RTT non-static	ssk = esk = g^{xy}
PSK	ssk = esk = psk
PSK-DHE	ssk = psk, esk = g^{xy}

Output keys

eadk	early application data key
htk	handshake traffic keys
atk	application traffic keys
sfk	server Finished key

Figure 2: OPTLS Summary. Modes, Terminology and Key Derivation (unique labels used as inputs to HKDF-Exp are not shown; ε denotes the empty string).

mode	ceks	seks	ssks	sfk	eadk	atk	htk	cost
1-RTT semi-static	g^x	g^y	g^s	g^{xs}	g^{xs}	g^{xs}, g^{xy}	g^{xy}	2DH
1-RTT non-static	g^x	g^y	g^y	g^{xy}	–	g^{xy}	g^{xy}	1DH + 1sig
PSK	–	–	–	psk	psk	psk	psk	–
PSK-DHE	g^x	g^y	–	psk	psk	psk, g^{xy}	g^{xy}	1DH

Figure 3: The entries in the table specify the keying material from which each key is derived - e.g., in the first-row mode sfk is derived from g^{xs} while in the second it is derived from g^{xy}.

These four modes have also been adapted and adopted in the TLS draft as described in [51, Section 7.1] (see Section 5 for more details).

Key derivation. We derive the keys for all the different modes in a unified manner as shown in Fig 2. We designed the key derivation schedule following the following principles from HKDF:

- We always use HKDF.Extract before HKDF.Expand and we use multiple invocations of HKDF.Expand with different labels to derive multiple keys;
- Each intermediate key is only used at most once as an input to either HKDF.Extract or HKDF.Expand.

In the protocols, we minimized the input to HKDF.Expand to understand what is essential for security and what is not. We refer to Section 5 for discussion on using session hash (i.e. a hash of all handshake messages up to the point of derivation) for key derivation to thwart key synchronization attacks.

Server configuration. The 1-RTT protocols refer to the server configuration ssks^+, namely an authenticated $\mathsf{ssks} = g^s$ attached to a server identity id_S. There are several different ways to implement such an authenticated configuration. First, it could be transmitted out-of-band, e.g. via DNSSEC, DANE, a website or a previous TLS execution. Second, it could be transmitted during the protocol execution, e.g. via DH certificates, or offline signatures, or online signatures that covers g^s together with the client's nonce. The specification of the server configuration in the current TLS draft [51, Section 6.3.7] also includes an expiration date, which we left out here as it is not captured by our security model.

Notation. We denote algorithms $\mathsf{MAC}, \mathsf{HKDF}$ using all-caps and strings $\mathsf{chello}, \ldots$ using all-lower-case. We use $\{ M \}_K$ to denote an authenticated-encryption (AEAD) of a message M under key K.

4.1 Detailed overview

We present a detailed overview of the modes that rely on DH key exchange: 1-RTT semi-static, 1-RTT non-static and PSK-DHE. Here, we deviate slightly from the overview in Section 2, treating each of the three modes in parallel. For simplicity, we first carry out the analysis ignoring handshake encryption and then address handshake encryption in Section 4.6.

The starting point is the textbook protocol for key exchange with signatures: the client sends (chello, g^x), and the server responds with (shello, g^y) and a signature over the entire transcript, and the shared session key is g^{xy}. As noted in the introduction, if the server has an authenticated g^s instead of a signature key, then it could instead authenticate the entire transcript with a MAC using a key derived from g^{xs}. Note that both of these protocols achieve PFS, since g^{xy} remains pseudorandom even if the server's long-term signing key or DH exponent s is compromised upon completion of the key exchange protocol.

The message flow for TLS handshakes [51, Figure 1] mandates that the server provides a Finished message across all modes of operation, whose role is to "providing authentication of the server side of the handshake and computed keys" [51, Section 6.3.9]. In TLS, the Finished message corresponds to a MAC computation, since not all modes admit a server's signing key.

In OPTLS, we also adopt the server's Finished message in TLS, and we require that the server authenticate the entire transcript via the server's Finished message, which corresponds to computing a MAC –instead of a signature– over the entire transcript. We continue to derive the session key from g^{xy} (possibly along with additional keying material to enable security even if y is compromised), but the question remains: how do we derive the MAC key sfk?

- In 1-RTT semi-static, the server has an authenticated semi-static key g^s and sfk is derived from g^{xs}.

- In 1-RTT non-static, there is no semi-static key g^s. Instead, the server would sign g^y (this can be pre-processed offline), and then derive sfk from g^{xy}. An alternative interpretation of this mode is as follows: once the server signs g^y, it plays a role similar to the semi-static key g^s and this mode may be thought of replacing (g^s, g^{xs}) in 1-RTT semi-static with (g^y, g^{xy}).

- In PSK-DHE mode, sfk is derived from the pre-shared key psk; the session key is derived from a combination of psk and g^{xy}, which allows us to achieve PFS in the case where psk gets compromised.

Proof overview. The proofs all follow the following high-level structure in the PFS setting.

- We rely on the security of server authentication and sfk to argue that g^y at the client test session must have come from an honest server. The exact security reduction depends on how sfk is derived: for instance, in 1-RTT non-static, this follows quite readily from the security of the server's signature, and in PSK-DHE mode, this follows quite readily from PRF security with seed psk. The formal security reduction requires a bit more care as we need to properly account for replay attacks.

- We then apply the DH assumption to key shares g^x, g^y at the client test session to replace g^{xy} with random, upon which we are essentially done.

Key derivation. To achieve uniform key derivation, we associate each mode with a pair of keys $\mathsf{esk}, \mathsf{ssk}$ (denoting ephemeral secret and static secret keys). For 1-RTT semi-static, these correspond naturally to (g^{xy}, g^{xs}). The remaining keys are then derived from $(\mathsf{esk}, \mathsf{ssk})$ in a unified manner as shown in Fig 2.

- We note that sfk is derived from ssk and not from esk. In 1-RTT semi-static, only $\mathsf{ssk} = g^{xs}$ can authenticate the handshake as it is the only element to involve the server's semi-static key g^s. One of the main elements to be authenticated by the server is g^y, thus deriving sfk from $\mathsf{esk} = g^{xy}$ (or from mixing esk and ssk) would create a circularity issue in the logic of the derivation.

- We encrypt handshake traffic under a key derived from $\mathsf{esk} = g^{xy}$ and not from both esk and ssk. This allows the server to start encrypting handshake traffic as soon as g^y is transmitted; in particular, it allows the encryption of the server's semi-static key g^s.

- We derive the application key from a mix of ssk and esk via HKDF; deriving from esk provides forward secrecy whereas deriving from ssk provides resilience to disclosure of ephemeral exponents (the latter in 1-RTT semi-static and PSK-DHE modes). One can contemplate straightforward combinations of ssk and esk such as a simple XORing of these values. We choose to use HKDF instead as a more robust option against future changes and misuse. In particular, an XOR could be problematic in the 1-RTT non-static and PSK modes where we set $\mathsf{ssk} = \mathsf{esk}$. See the "Key derivation" text in Section 2 for more detailed rationale of the particular way in which ssk and esk are mixed through HKDF.

4.2 1-RTT semi-static

The 1-RTT semi-static mode is summarized in Fig 4.

As described above, the server transmits its configuration ssks^+ in the second message. The protocol and the security proof also works if the client uses a cached ssks^+, as long as the client and the server

$$\xrightarrow{\quad \mathsf{chello},\ \overbrace{g^x}^{\mathsf{ceks}},\ \overbrace{\{\,\mathsf{early\ data}\,\}_{\mathsf{eadk}}}^{\text{optional}} \quad}$$

$$\xleftarrow{\quad \mathsf{shello},\ \overbrace{g^y}^{\mathsf{seks}},\overbrace{\text{signed } g^s}^{\mathsf{ssks}^+},\ \overbrace{\mathsf{MAC}_{\mathsf{sfk}}(\cdots)}^{\mathsf{sfin}} \quad}$$

Figure 4: 1-RTT semi-static mode. Here, $(\mathsf{sfk}, \mathsf{eadk})$ are derived from g^{xs} and atk from (g^{xs}, g^{xy}). See Fig 2 for details on computation of these keys and sfin.

agree on the same ssks^+ and it is included in the computation of sfin. We first analyze the protocol assuming there is no early data, and we analyze the security of early data in Section 4.3.

Theorem 1 (1-RTT semi-static) *The OPTLS protocol in the 1-RTT semi-static mode achieves security with perfect forward security (PFS) and resilience to disclosure of ephemeral exponents. This holds in the random oracle model assuming security of HKDF, hardness of Gap-DH and hardness of DDH.*

Here, PFS security refers to the server's secret s and long-term signing key in ssks^+, and resilience to disclosure of ephemeral exponents refers to the server's ephemeral exponent y. The Gap-DH assumption and the random oracle are used to ensure non-malleability across multiple sessions that use the same fixed g^s, as was the case in the DHIES CCA-secure encryption scheme in [1].

Correctness. Suppose two honest peers have the same matching conversation $(\mathsf{chello}, \mathsf{ceks}, \mathsf{shello}, \mathsf{seks}, \mathsf{ssks}^+)$. Observe that this uniquely determines $\mathsf{esk}, \mathsf{ssk}$ and thus the session key atk.

Security. Let $\mathsf{sessionid}^* = (\mathsf{chello}^*, \mathsf{ceks}^*, \mathsf{shello}^*, \mathsf{seks}^*, \mathsf{ssks}^*)$ denote the view of the client at the test session. Throughout, we write $(\mathsf{ceks}^*, \mathsf{seks}^*, \mathsf{ssks}^*) = (g^x, g^y, g^s)$. Throughout the proofs, we treat $\mathsf{HKDF.Extract}(0, \cdot)$ as a random oracle, which we simulate via lazy sampling. We consider two cases. The first addresses the setting where g^{xs} is secure (e.g. s is secure but y may be leaked via a $\mathsf{StateReveal}$ query, and the second addresses the setting where g^{xy} is secure (e.g. y is secure but s may be leaked after the handshake via a $\mathsf{Corrupt}$ query, i.e. the PFS setting).

Case 1: s is never compromised. Here, we rely on security of $\mathsf{ssk} = g^{xs}$ for the privacy of atk. We proceed via a sequence of games as follows:

- Game 0: Real game.
- Game 1: Abort if ssks^* does not match g^s for the honest server whose identity is embedded in ssks^+. This follows from the security of the signature scheme used to sign ssks^* (or the security of whatever mechanism used to authenticate ssks^+).
- Game 2: Abort if the adversary queries $\mathsf{HKDF.Extract}(0, \cdot)$ at g^{xs}. In addition, pick a uniformly random string xSS^* and implicitly program $\mathsf{HKDF.Extract}(0, g^{xs})$ to be xSS^*. By the Gap-DH assumption[6] which tells us that g^{xs} is hard to compute given g, g^x, g^s and a verification oracle $(g, g^s, \cdot, \cdot)$ for DDH tuples, this affects the advantage by a negligible quantity. In the security reduction,

[6] We can replace the use of random oracles and the Gap-DH assumption with the following interactive assumption, which stipulates that given random g, g^x, g^s, the quantity $\mathsf{HKDF.Extract}(0, g^{xs})$ is pseudorandom, even given access to an oracle that on input $u \neq g^x$, returns $\mathsf{HKDF.Extract}(0, u^s)$.

  - We pick $y \leftarrow_{\mathrm{R}} \mathbb{Z}_p$ ourselves, setting $(\mathsf{ceks}^*, \mathsf{seks}^*, \mathsf{ssks}^*) = (g^x, g^y, g^s)$ (here, we use the fact that $(\mathsf{ceks}^*, \mathsf{seks}^*) = (g^x, g^s)$ are sampled uniformly by an honest client and honest server respectively);
  - We simulate the honest server by simulating $\mathsf{HKDF.Extract}(0, \cdot)$ ourselves via lazy sampling to compute xSS, using the DDH verification oracle to ensure that the queries are simulated consistently.

  Note that any leakage of xSS^* to the adversary must come from interacting with the honest server in a session with $\mathsf{ceks} = g^x$ (and there could be many such sessions at the same server).

- Game 3: Replace $(\mathsf{sfk}^*, \mathsf{mSS}^*, \mathsf{eadk}^*)$ by uniformly random strings. This follows readily from PRF security of $\mathsf{HKDF.Expand}(\mathsf{xSS}^*, \cdot)$. (Again, multiple sessions at the same server could share the same $g^x, \mathsf{xSS}^*, \mathsf{sfk}^*$.)

- Game 4: Abort if there is no matching conversation, i.e., no server session begins with the view $\mathsf{sessionid}^*$. In Game 2, the only access the adversary has to sfk^* comes from interacting with the honest server, which computes $\mathsf{MAC}_{\mathsf{sfk}^*}(\cdot)$. By the security of the MAC, the client will only reach accept state if the adversary forwards $\mathsf{sfin} = \mathsf{MAC}_{\mathsf{sfk}^*}(\mathsf{sessionid}^*)$ as computed by an honest server, in which case we have a matching conversation. In the security reduction,

  - We pick $x, s \leftarrow_{\mathrm{R}} \mathbb{Z}_p$ ourselves, setting $(\mathsf{ceks}^*, \mathsf{ssks}^*) = (g^x, g^s)$;
  - We program sfk^* to be the MAC key;
  - We simulate all messages from the honest server ourselves using s, except that we use the $\mathsf{MAC}_{\mathsf{sfk}^*}(\cdot)$ oracle to compute sfin whenever $\mathsf{ceks} = g^x$ and thus $\mathsf{xSS} = \mathsf{xSS}^*$;
  - When the honest client receives sfin^* from the adversary, the reduction outputs $(\mathsf{sessionid}^*, \mathsf{sfin}^*)$.

  Whenever client accepts without a matching conversation, the reduction outputs a valid forgery for $\mathsf{MAC}_{\mathsf{sfk}^*}(\cdot)$. Finally, note that the matching conversation is unique since all other server sessions use $\mathsf{shello} \neq \mathsf{shello}^*$.

- Game 5: Replace ms^* by a uniformly random string. This follows from PRF-security of $\mathsf{HKDF.Extract}(\mathsf{mSS}^*, \cdot)$ and the uniqueness of mES^* across different sessions via a birthday analysis.

- Game 6: Replace atk^* by a uniformly random string. This follows readily from PRF-security of $\mathsf{HKDF.Expand}$ with seed ms^*.

Observe that in the final game, the view of the adversary is completely independent of the challenge bit. Furthermore, observe that throughout the proof, we always pick $y \leftarrow_{\mathrm{R}} \mathbb{Z}_p$ ourselves at the honest server, which means that security holds even when y is compromised at the honest server via a $\mathsf{StateReveal}$ query.

Case 2: s is compromised after handshake. Next, we consider the case where s is compromised after the client receives g^y and reaches accept state. Here, we rely on security of $\mathsf{esk} = g^{xy}$ for the privacy of atk. Also, we only rely on the security of s in Game 2 before the completion of the handshake. We start out as before:

- Game 0: Real game.

- Game 1: Same as Case 1.
- Game 2: As in Case 1, except the abort condition only applies to queries made before the client reaches accept state (and thus before s is compromised).
- Game 3: Same as Case 1.
- Game 4: Same as Case 1.
- Game 5: Replace $\mathsf{esk}^* = g^{xy}$ with g^z where $z \leftarrow_{\mathrm{R}} \mathbb{Z}_p$. This follows from the DDH assumption, which tells us that g^{xy} is pseudorandom given g, g^x, g^y. In the security reduction,
  - We pick $s \leftarrow_{\mathrm{R}} \mathbb{Z}_p$ ourselves, setting $(\mathsf{ceks}^*, \mathsf{seks}^*, \mathsf{ssks}^*) = (g^x, g^y, g^s)$ (here, we use the fact that $(\mathsf{ceks}^*, \mathsf{seks}^*) = (g^x, g^y)$ are sampled uniformly by the honest client and honest server respectively);
  - We simulate the honest client ourselves;
  - We simulate all messages from the honest server ourselves using s, except that we use the challenge g^{xy} or g^z for computing esk^*.
- Game 6: Replace $(\mathsf{htk}^*, \mathsf{mES}^*)$ with uniformly random strings. This follows from PRF security of $\mathsf{HKDF.Expand}$ with seed xES^* which is derived from g^z.
- Game 7: Replace
$$\mathsf{ms}^* = \mathsf{HKDF.Extract}(\mathsf{mSS}^*, \mathsf{mES}^*)$$
  with a uniformly random string. This follows from modeling $\mathsf{HKDF.Extract}$ as a strong extractor, where mES^* is the entropy source and mSS^* is the public seed.
- Game 8: Same as Game 6 in Case 1.

Observe that in the final game, the view of the adversary is completely independent of the challenge bit.

4.3 Early application data (0-RTT)

We refer to the protocol in Figure 4 and analyze the protocol where early data is transmitted. If the client has a cached ssks^+, then it can start transmitting early application data in the first flow, encrypted under eadk. This is also known as 0-RTT exchange, which is a requirement for TLS 1.3, c.f. [51, Section 6.2.2]. We reiterate that the early data is not part of the input to $\mathsf{MAC}_{\mathsf{sfk}}$ in computing sfin. Here, eadk is open to replay attacks (but atk is not). We prove privacy of eadk in the one-pass model from [30]. Note that it is impossible to achieve PFS security in the one-pass setting.

One-pass security model. We modify the security model in Section 3.1 as follows:

- the adversary's queries $\mathsf{Reveal}(), \mathsf{Test}()$ refer to eadk instead of atk;
- We define matching conversations based on $(\mathsf{chello}, \mathsf{ssks}^+)$ (this applies only to eadk and not to atk).

Note that we cannot achieve PFS in this setting, so we require that the adversary never runs a corrupt query on the test session. The protocol also achieves a slightly stronger notion, namely eadk remains private even if we learn atk in the same session. Similarly, in the previous proof, atk remains private even if we corrupt eadk in the same session.

Theorem 2 (early data in 1-RTT semi-static) *The OPTLS protocol in the 1-RTT semi-static mode achieves one-pass security for early application data. This holds in the random oracle model assuming security of HKDF and hardness of Gap-DH.*

The theorem refers to the first message of the protocol in Fig 4 and to the session key eadk.

Correctness. Correctness assumes an anti-replay mechanism where all sessions at an honest sender have distinct incoming chello, so that chello uniquely identifies $\mathsf{ceks} = g^x$. This means that when there is a matching conversation, both parties share $\mathsf{chello}, \mathsf{ceks}, \mathsf{ssks}$ and thus compute the same eadk.

Security. We want to prove privacy of eadk^* in the case where the server's static key ssks is never compromised. We start out similar to the proof of Case 1 in Theorem 3.

- Game 0: Real game.
- Game 1: Same as Case 1 in Theorem 3.
- Game 2: Same as Case 1 in Theorem 3.
- Game 3: Replace eadk^* with a uniformly random string. This follows from PRF security of $\mathsf{HKDF.Expand}(\mathsf{xSS}^*, \cdot)$, since
  - Reveal queries for eadk in other sessions do not affect eadk^* since $\mathsf{chello} \neq \mathsf{chello}^*$;
  - Leakage from other types of keys $\mathsf{atk}, \mathsf{sfk}$ do not affect eadk^* due to domain separation enforced by the different labels. In particular, this implies security of eadk^* even if atk^* gets compromised.

4.4 1-RTT non-static

The 1-RTT semi-static mode is summarized in Fig 5.

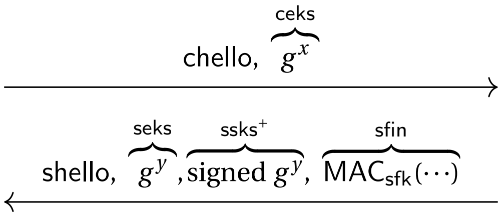

ceks
chello, g^x
seks
ssks$^+$
sfin
shello, g^y, signed g^y, $\mathsf{MAC}_{\mathsf{sfk}}(\cdots)$

Figure 5: 1-RTT non-static mode. Here, $\mathsf{sfk}, \mathsf{atk}$ are derived from g^{xy}. See Fig 2 for details on computation of these keys and sfin. In the actual protocol, there is no need to transmit g^y twice.

Theorem 3 (1-RTT non-static) *The OPTLS protocol in the 1-RTT non-static mode achieves security with perfect forward security (PFS). This holds in the standard model assuming security of HKDF and hardness of DDH.*

Here, PFS security refers to the server's long-term signing key, and we do not achieve resilience to disclosure of ephemeral exponents. Recall from Section 2 that we may think of the ephemeral g^y as a "single use" semi-static g^s in the semi-static mode. Indeed, the proof is similar to that of Case 1 of Theorem 1 where s is never compromised. Note however that we avoid the use of the random oracle model and the Gap-DH assumption in Theorem 1 for 1-RTT semi-static. The intuition is as follows: in 1-RTT semi-static mode, we needed the Gap-DH assumption to simulate the DDH oracle for a fixed g^s which is used across multiple sessions. Here, a fresh g^y is picked for each session and there is no fixed g^s. In 1-RTT semi-static, we also needed the random oracle to extract a random key xSS from g^{xs}; here, we may simply use the DDH assumption to extract a random xSS from g^{xy}, again because g^y is fresh per session.

The security proof in this section relies on completely standard techniques and follows closely Case 1 of Theorem 1 except for Games 2 and 3, so we leave out details of the security reduction.

Security. Let $(\mathsf{chello}^*, \mathsf{ceks}^*, \mathsf{shello}^*, \mathsf{seks}^*, \mathsf{ssks}^*)$ denote the view of the client at the test session. Throughout, we write $(\mathsf{ceks}^*, \mathsf{seks}^*) = (g^x, g^y)$.

- Game 0: Real game.

- Game 1: Abort if $\mathsf{seks}^* = g^y$ does not match a seks coming from an honest server session. This follows from security of the server's signature.

- Game 2: Replace $\mathsf{esk}^* = \mathsf{ssk}^* = g^{xy}$ with g^z where $z \leftarrow_{\mathrm{R}} \mathbb{Z}_p$. This follows from the DDH assumption.

- Game 3: Replace $(\mathsf{htk}^*, \mathsf{mES}^*, \mathsf{mSS}^*, \mathsf{sfk}^*)$ with uniformly random strings. This follows from PRF security of $\mathsf{HKDF.Expand}$ with seed $\mathsf{xES}^* = \mathsf{xSS}^*$ which is derived from g^z.

- Game 4: Abort if there is no matching conversation, i.e., no server session begins with the view $\mathsf{sessionid}^*$. This follows from the security of $\mathsf{MAC}_{\mathsf{sfk}^*}(\cdot)$ as in Game 4 in Case 1 of Theorem 1.

- Game 5: Replace ms^* by a uniformly random string. This follows from PRF-security of $\mathsf{HKDF.Extract}(\mathsf{mSS}^*, \cdot)$ and the uniqueness of mES^* across different sessions via a birthday analysis. This is the same as Game 5 in Case 1 of Theorem 1.

- Game 6: Replace atk^* with a uniformly random string. This follows readily from PRF-security of $\mathsf{HKDF.Expand}$ with seed ms^*. This is the same as Game 6 in Case 1 of Theorem 1.

4.5 PSK and PSK-DHE modes

TLS provides a pre-shared key (PSK) mode which allows a client and server who share an existing secret (e.g., a key established out of band) to establish a connection authenticated by that key [51, Section 6.2.3]. PSKs can also be established in a previous session and then reused ("session resumption"). The basic PSK mode does not achieve forward secrecy, but can be augmented with a DH exchange to provide forward secrecy in combination with shared keys.

Following the TLS specification, we assume that chello contains a unique identifier for pre-shared key psk, in addition to the client nonce. We say that two sessions are matching if the transcripts at both sessions begin with $(\mathsf{chello}, \mathsf{ceks}, \mathsf{shello}, \mathsf{seks})$ and we require that distinct identifiers amongst honest parties correspond to independent pre-shared keys.

The PSK and PSK-DHE modes are summarized in Fig 6.

In the PSK and PSK-DHE modes, we use $\mathsf{HKDF.Extract}$ on psk, which allows us to handle psk distributions which have high entropy but may not necessarily be the uniform distributions.

$$\xrightarrow{\qquad \mathsf{chello},\ \overbrace{[\,g^x\,]}^{\mathsf{ceks}},\ \overbrace{\{\,\text{early data}\,\}_{\mathsf{eadk}}}^{\text{optional}} \qquad}$$

$$\xleftarrow{\qquad \mathsf{shello},\ \overbrace{[\,g^y\,]}^{\mathsf{seks}},\ \overbrace{\mathsf{MAC}_{\mathsf{sfk}}(\cdots)}^{\mathsf{sfin}} \qquad}$$

Figure 6: PSK and PSK-DHE modes. g^x, g^y are only used in PSK-DHE modes. Here, $(\mathsf{sfk}, \mathsf{eadk})$ are derived from psk. atk is derived from psk in PSK mode, and from (psk, g^{xy}) in PSK-DHE mode.

Theorem 4 (PSK and PSK-DHE modes) *The OPTLS protocol in the PSK mode achieves basic security in the standard model assuming security of HKDF. The OPTLS protocol in the PSK-DHE mode achieves security with perfect forward security (PFS) and resilience to disclosure of ephemeral exponents in the standard model assuming security of HKDF and hardness of DDH.*

Here, PFS security refers to the server's psk, and resilience to disclosure of ephemeral exponents refers to the server's ephemeral exponent y.

Note that the PSK and PSK-DHE modes are similar to the 1-RTT modes in that psk essentially replaces g^{xs}: in PSK mode, we have $\mathsf{esk} = \mathsf{ssk} = \mathsf{psk}$ instead of $\mathsf{esk} = \mathsf{ssk} = g^{xy}$ in 1-RTT non-static, whereas in PSK-DHE mode, we have $(\mathsf{esk}, \mathsf{ssk}) = (g^{xy}, \mathsf{psk})$ instead of $(\mathsf{esk}, \mathsf{ssk}) = (g^{xy}, g^{xs})$. We highlight some of the main differences between the analysis of the PSK and PSK-DHE modes from the 1-RTT ones. Here, we need to simulate multiple sessions at the honest client with the same secret psk and we need to explicitly handle key reveal queries at clients. In the 1-RTT modes, different sessions at the client do not share any secret state and we can simulate each client session in the reduction. Nonetheless, we do effectively get independence across the different honest clients due to PRF-security with seed derived from psk applied to distinct chello's across the sessions.

Correctness. Suppose two honest peers have the same matching conversation $(\mathsf{chello}, \mathsf{ceks}, \mathsf{shello}, \mathsf{seks})$. This determines psk and thus the session key atk.

Security for PSK mode. The proof is similar to that for Theorem 3 for 1-RTT non-static. There, we have $\mathsf{esk} = \mathsf{ssk} = g^{xy}$, and here, we have $\mathsf{esk} = \mathsf{ssk} = \mathsf{psk}$. That is, we simply bypass Games 1 and 2 and go directly from Game 0 to Game 3. In Game 3, we can immediately rely on PRF-security of $\mathsf{HKDF.Expand}$ with seed $\mathsf{xES}^* = \mathsf{xSS}^*$ which is derived from psk.

Security for PSK-DHE mode. The proof is similar to that for Theorem 1 for 1-RTT semi-static. Again, we consider two cases. In Case 1, psk is never compromised. This is the same as Case 1 from before except we can bypass Game 1 and immediately rely PRF-security of $\mathsf{HKDF.Expand}$ with seed xSS^* which is derived from psk. In Case 2, psk is compromised after the handshake and security of atk comes from $\mathsf{esk} = g^{xy}$. This is the same as Case 2 from before.

Security for early application data. In both modes, security for early data application key follows readily from PRF-security of $\mathsf{HKDF.Expand}$ with seed xSS^* which is derived from psk.

4.6 Encrypting handshake traffic

We need to show that encrypting the handshake traffic as shown in Fig 2 does not compromise security of the key exchange protocols. That is, all handshake messages after seks are encrypted under a key htk derived from esk. Note that we still define matching conversations as before $(\mathsf{chello}, \mathsf{ceks}, \mathsf{shello}, \mathsf{seks}, \mathsf{ssks}^+)$, that is, with the unencrypted ssks^+.

Clearly, the only changes we need to make to the security proofs from before are in the games that refer to $\mathsf{sfin}, \mathsf{ssks}^+$ and htk. For $\mathsf{sfin}, \mathsf{ssks}^+$ (which is where the main challenge lies), we need to be able to decrypt incoming messages to an honest client in order to verify the signature on ssks^+ and the MAC in sfin. For the 1-RTT semi-static, 1-RTT non-static and PSK-DHE modes, we we use the fact that the honest client knows x and can compute htk and recover the unencrypted $\mathsf{ssks}^+, \mathsf{sfin}$. That is,

- In Game 1 in Case 1 and 2 of Theorem 1 and in Game 1 of Theorem 3, we use x to compute htk and to recover the unencrypted ssks^+, upon which we can rely on the security of the server's signature (that is, the honest server which is the peer of the honest client at the test session).

- In Game 4 in Case 1 and 2 of Theorem 1 and in Game 4 in Case 1 of Theorem 3, we use x to compute htk and to recover the unencrypted sfin, upon which we can rely on security of the MAC.

For the PSK mode, we use the fact that the honest client can compute htk using the PRF oracle whose seed xES is derived from psk, thanks to domain separation enforced by chello. Interestingly, the above argument does not rely on AEAD security of the encryption scheme used for handshake encryption.

It is also straight-forward to verify that in the prior proofs, for the honest server, we can always compute htk and then encrypt $\mathsf{ssks}^+, \mathsf{sfin}$. For instance, in Theorem 1 Case 2 Game 5, we will derive htk^* from esk^* (either g^{xy} or g^z) and then use htk^* to encrypt $\mathsf{ssks}^+, \mathsf{sfin}$. The rest of the proofs then go through unchanged.

In fact, the prior proofs also established security of $(\mathsf{atk}, \mathsf{htk})$ (in that we can treat $(\mathsf{atk}, \mathsf{htk})$ as the session key), as long as y is not compromised. Concretely, we replaced htk^* with a uniformly random string in the proofs of Theorem 1 Case 2 Game 6 and Theorem 3 Game 3 (which carry over to the proofs of Theorem 4). AEAD security then ensures privacy of ssks^+ once it is encrypted under htk. From a cryptographic stand-point, there is no security advantage to encrypting sfin; the reason for doing so is to simplify implementation. We omit further details about protecting privacy of the handshake messages as this is not captured by our security model.

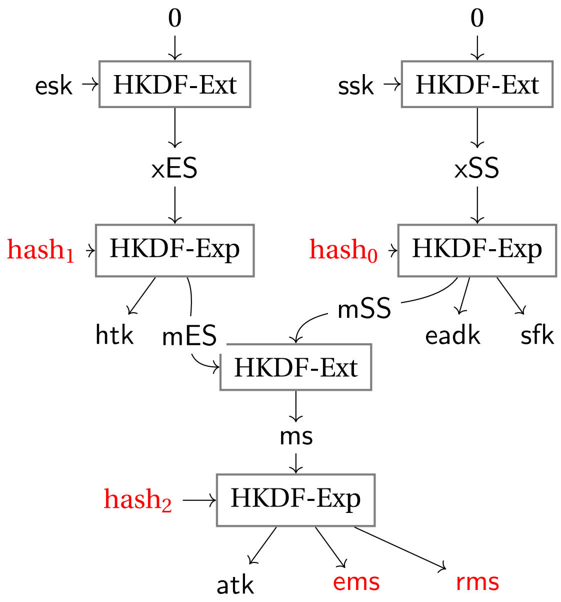
5 OPTLS in TLS 1.3

We show how the basic OPTLS design is adapted into the current TLS 1.3 specification, along with a uniform treatment of all of the major modes, including the PSK mode. Here, we follow the maximalist approach of TLS, namely hashing everything to the point of derivation (a.k.a. "session hash" [15]) as part of the context for $\mathsf{HKDF.Expand}$.

The different modes as summarized in Fig 7 build upon those in OPTLS, and the only modifications pertain to the input to $\mathsf{HKDF.Expand}$ for key derivation (the hash inputs which also include unique labels not shown in the figure). The analysis is exactly the same as before, since the only changes are that we add additional context information as input to $\mathsf{HKDF.Expand}$ (and since the role of these additions are to enforce domain separation, adding more things to the context does not affect the security proofs).

The security definition in Section 3.1 does not preclude key synchronization attacks, in which a person-in-the-middle $\mathscr{A}$ with whom both client and server are willing to interact can make both honest

OPTLS in TLS 1.3 draft-09



eadk	early application data key
htk	handshake traffic keys
atk	application traffic keys
ems	exporter master secret
rms	resumption master secret
sfk	server Finished key

1-RTT s.s.	ssk = g^{xs}, esk = g^{xy}
1-RTT n.s.	ssk = esk = g^{xy}
PSK	ssk = esk = psk
resumption	ssk = esk = rms
PSK-DHE	ssk = psk, esk = g^{xy}

hash$_0$	chello + ceks
hash$_1$	... + shello + seks
hash$_2$	... + ssks$^+$

Figure 7: Diagram for key derivation (unique labels used as inputs to HKDF.Expand are not shown), as adopted in [51, Sec 7.1]. The session hash refers to all handshake messages sent or received to the point of derivation, with the exception of the Finished messages. Here, online signatures of ssks cover hash$_1$. We derive the keys for the different modes by setting (esk, ssk) appropriately as a function of g^{xy}, g^{xs}, psk, in the order ssk, xSS, mSS, eadk, sfk; esk, xES, mES, htk; ms, atk, ems, rms. The changes from Fig 2 are highlighted in red.

instances agree on the same key with $\mathscr{A}$ (in the CK model, there is no guarantee about the distribution of a key exchanged with a corrupted party). This form of attack was the source of the triple handshake attack [13]. The session hashes added in the TLS 1.3 key derivation should preclude this attack.

5.1 Relation to TLS 1.3 draft-09 [51]

As mentioned earlier, the four major modes of OPTLS and our uniform key derivation have been adopted in TLS 1.3 draft-09, with a number of additions as described below. The default 1-RTT mode in the draft is the 1-RTT non-static. 1-RTT with known configuration in the draft instantiates 1-RTT semi-static with a cached ssks$^+$.

Server-side signing. In the current spec, the server always signs, even in 1-RTT semi-static. This has a performance cost due to additional sign/verify operations and the need to transmit/verify certificates, yet the simplification of “always sign" has been prioritized over the performance consideration (which is more significant with RSA signatures than with ECDSA). This change does not affect our security proofs for OPTLS or their applicability to TLS 1.3 as the addition of a signature to 1-RTT semi-static does not eliminate any of the elements that ensure the key exchange security of OPTLS. That said, online signatures that cover the client nonce can add security features on top of the proven security properties of OPTLS. For instance, online signatures guarantee that the server has continuous access to the signing key, which would help address vulnerabilities considered in [33] where a server incorrectly uses the same

RSA certificate for RSA encryption in TLS 1.2 and offline signing in TLS 1.3.

We stress that eliminating online signatures may enhance the privacy features of the protocol by providing "plausible deniability" of the peer's identity. Note that our analysis of OPTLS shows that future uses of TLS could indeed safely forgo the use of online signatures when clients cache a server's semi-static key g^s or when authentication is performed using DH certificates or offline signatures with dedicated signature keys. This is in contrast to recent analysis of TLS 1.3 candidates in [36, 36] which crucially rely on the use of online signatures for security.

Key derivation. Key derivation in draft-09 uses additional hash inputs as mentioned above. Note that since each HKDF.Expand operation is paired with an HKDF.Extract operation, all of the keys can also be computed using black-box invocations of HKDF at the cost of additional invocations of HKDF.Extract (instead of separate invocations of HKDF.Extract and HKDF.Expand as depicted in our KDF representation). This provides compatibility with a black-box HKDF API, e.g. in PKCS #11 for cryptographic tokens, such as hardware security modules (HSM) and smart cards. Basic KE security of the protocol is not changed by the additional hash inputs.

Binding keys to identities. The final keys $(\mathsf{atk}, \mathsf{rms}, \mathsf{ems})$ are bound to server's identities, even if the server's identity is not transmitted on the wire as is the case in the 1-RTT semi-static mode. This is a prudent addition addressed in [51, Section 7.2.1] which defines the handshake hash to always include the server's configuration.

5.2 Further work

Our treatment here does not include some of the elements/modes that are part of TLS 1.3 and which we list here. We anticipate that our analysis work can be extended to cover these elements.

Resumption. Resumption is an important mode in TLS that allows to cache session information (including keying material) and enable a fast resumption of a session without having to go through the full cost of a new session establishment. TLS 1.3 simplifies (and adds security to) resumption by reducing it [51, Section 6.2.3] to the PSK mode.

Client authentication. Our treatment focuses on server authentication only, the most common case in TLS practice. Yet, client authentication is necessary and supported in other cases. OPTLS can be augmented with such capability but we don't do it here, particularly since the full requirements and operational details of this mode are not yet worked out by the TLS working group. This includes issues such as providing support of client authentication only after server authentication or doing so already in the first message from client to server, something that raises replay considerations. We leave the augmentation of OPTLS with client authentication as a future work item.

Client's Finished Message. TLS 1.3 includes a mandatory third message in the handshake in which the client sends a MAC value (client's Finished) computed on the prior transcript similar to the one sent by the server in the second message. As our analysis shows this is not needed for the basic key exchange security in the case of server-only authentication. Yet, this message serves several purposes that are valuable in the TLS 1.3 setting. It serves as the confirmation from the client that both server and client have the same view of the transcript, including keying material, server identity and negotiated security parameters (part of the hello messages). Interestingly, the sending of this message has been shown in [21] to "upgrade" security from the indistinguishability-based model we use here to universally-composable security.

Acknowledgments.

We would like to thank Eric Rescorla for essential contributions to the design of the protocol, the TLS Working Group for insightful discussions and feedback on the use of OPTLS in TLS 1.3, and Björn Tackmann for helpful comments and suggestions. This work was done in part while the authors were visiting the Simons Institute for the Theory of Computing, supported by the Simons Foundation and NSF grant #CNS-1523467.

References

[1] M. Abdalla, M. Bellare, and P. Rogaway. The oracle Diffie-Hellman assumptions and an analysis of DHIES. In *CT-RSA*, pages 143–158, 2001.

[2] N. AlFardan and K. G. Paterson. Plaintext-recovery attacks against Datagram TLS. In *Network and Distributed System Security Symposium (NDSS 2012)*, 2012.

[3] N. AlFardan and K. G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy*, 2013. URL www.isg.rhul.ac.uk/tls/Lucky13.html.

[4] N. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. Schuldt. On the security of RC4 in TLS and WPA. In *USENIX Security Symposium*, 2013. URL www.isg.rhul.ac.uk/tls.

[5] G. V. Bard. The vulnerability of SSL to chosen plaintext attack. *IACR Cryptology ePrint Archive*, 2004:111, 2004.

[6] G. V. Bard. A challenging but feasible blockwise-adaptive chosen-plaintext attack on SSL. In *SECRYPT*, pages 99–109, 2006.

[7] M. Bellare and P. Rogaway. Entity authentication and key distribution. In D. R. Stinson, editor, *CRYPTO*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249. Springer, 1993. ISBN 3-540-57766-1.

[8] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *CRYPTO*, pages 232–249, 1993.

[9] M. Bellare, T. Kohno, and C. Namprempre. Authenticated encryption in SSH: provably fixing the SSH binary packet protocol. In *ACM Conference on Computer and Communications Security*, pages 1–11, 2002.

[10] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *IEEE Symposium on Security and Privacy*, 2015.

[11] K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu. Verified cryptographic implementations for TLS. *ACM Trans. Inf. Syst. Secur.*, 15(1):3, 2012.

[12] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub. Implementing TLS with verified cryptographic security. In *IEEE Symposium on Security and Privacy*, 2013. URL http://mitls.rocq.inria.fr/.

[13] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Pironti, and P. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *2014 IEEE Symposium on Security and Privacy, SP*, pages 98–113, 2014.

[14] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and S. Z. Béguelin. Proving the TLS handshake secure (as it is). In *CRYPTO II*, pages 235–255, 2014.

[15] K. Bhargavan, A. Delignat-Lavaud, A. Pironti, A. Langley, and M. Ray. Transport layer security (TLS) session hash and extended master secret extension, Sept. 2015. URL http://www.rfc-editor.org/rfc/rfc7627.txt.

[16] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *CRYPTO*, pages 1–12, 1998.

[17] C. Brzuska, M. Fischlin, B. Warinschi, and S. C. Williams. Composability of Bellare-Rogaway key exchange protocols. In *ACM Conference on Computer and Communications Security*, pages 51–62, 2011.

[18] C. Brzuska, M. Fischlin, N. Smart, B. Warinschi, and S. Williams. Less is more: Relaxed yet composable security notions for key exchange. Cryptology ePrint Archive, Report 2012/242, 2012.

[19] R. Canetti and H. Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *EUROCRYPT*, pages 453–474, 2001. See also Cryptology ePrint Archive, Report 2001/040.

[20] R. Canetti and H. Krawczyk. Security analysis of IKE's signature-based key-exchange protocol. In *CRYPTO*, pages 143–161, 2002. Also Cryptology ePrint Archive, Report 2002/120.

[21] R. Canetti and H. Krawczyk. Universally composable notions of key exchange and secure channels. In *EUROCRYPT*, pages 337–351, 2002. See also Cryptology ePrint Archive, Report 2002/059.

[22] B. Canvel, A. P. Hiltgen, S. Vaudenay, and M. Vuagnoux. Password Interception in a SSL/TLS Channel. In *CRYPTO*, pages 583–599, 2003.

[23] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1, Apr. 2006. URL http://www.rfc-editor.org/rfc/rfc4346.txt.

[24] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2, Aug. 2008. URL http://www.rfc-editor.org/rfc/rfc5246.txt.

[25] B. Dowling, M. Fischlin, F. Günther, and D. Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In *ACM CCS*, 2015. Also, Cryptology ePrint Archive, Report 2015/914.

[26] T. Duong and J. Rizzo. Here come the ⊕ Ninjas. Unpublished manuscript, 2011.

[27] T. Duong and J. Rizzo. The CRIME attack. Presentation at ekoparty Security Conference, 2012. URL http://www.ekoparty.org/eng/2012/juliano-rizzo.php.

[28] M. Fischlin and F. Günther. Multi-stage key exchange and the case of Google's QUIC protocol. In *ACM CCS*, pages 1193–1204, 2014.

[29] F. Giesen, F. Kohlar, and D. Stebila. On the security of TLS renegotiation. In *ACM CCS*, pages 387–398, 2013.

[30] S. Halevi and H. Krawczyk. One-pass HMQV and asymmetric key-wrapping. In *PKC 2011*, pages 317–334, 2011.

[31] C. He, M. Sundararajan, A. Datta, A. Derek, and J. C. Mitchell. A modular correctness proof of IEEE 802.11i and TLS. In *ACM CCS*, pages 2–15, 2005.

[32] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk. On the security of TLS-DHE in the standard model. In *CRYPTO*, pages 273–293, 2012. Also Cryptology ePrint Archive, Report 2011/219.

[33] T. Jager, J. Schwenk, and J. Somorovsky. On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption. In *ACM CCS*, 2015.

[34] J. Jonsson and B. S. Kaliski Jr. On the security of RSA encryption in TLS. In *CRYPTO*, pages 127–142, 2002.

[35] V. Klíma, O. Pokorný, and T. Rosa. Attacking RSA-based sessions in SSL/TLS. In *CHES*, pages 426–440, 2003.

[36] M. Kohlweiss, U. Maurer, C. Onete, B. Tackmann, and D. Venturi. (De-)constructing TLS. Cryptology ePrint Archive, Report 2014/020, 2014. revised Apr 2015.

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

[38] H. Krawczyk. SIGMA: The "SIGn-and-MAc" approach to authenticated Diffie-Hellman and its use in the IKE protocols. In *CRYPTO*, pages 400–425, 2003.

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

[40] H. Krawczyk and P. Eronen. HMAC-based extract-and-expand key derivation function (HKDF), May 2010. URL http://www.rfc-editor.org/rfc/rfc5869.txt.

[41] H. Krawczyk, K. G. Paterson, and H. Wee. On the security of the TLS protocol: A systematic analysis. In *CRYPTO (1)*, pages 429–448, 2013. Also, Cryptology ePrint Archive, Report 2013/339.

[42] A. Langley and W.-T. Chang. QUIC crypto, 2013. URL http://tinyurl.com/lrrjyjs.

[43] Y. Li, S. Schäge, Z. Yang, F. Kohlar, and J. Schwenk. On the security of the pre-shared key ciphersuites of TLS. In *PKC*, pages 669–684, 2014.

[44] R. Lychev, S. Jero, A. Boldyreva, and C. Nita-Rotaru. How secure and quick is QUIC? provable security and performance analyses. In *IEEE Symposium on Security and Privacy*, pages 214–231, 2015.

[45] U. Maurer and B. Tackmann. On the soundness of authenticate-then-encrypt: formalizing the malleability of symmetric encryption. In *ACM CCS*, pages 505–515, 2010.

[46] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel. A cross-protocol attack on the TLS protocol. In *ACM CCS*, pages 62–72, 2012.

[47] B. Moeller. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures. Unpublished manuscript, May 2004. http://www.openssl.org/~bodo/tls-cbc.txt.

[48] P. Morrissey, N. P. Smart, and B. Warinschi. A modular security analysis of the TLS handshake protocol. In *ASIACRYPT*, pages 55–73, 2008.

[49] K. G. Paterson, T. Ristenpart, and T. Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In *ASIACRYPT*, pages 372–389, 2011.

[50] L. C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Trans. Inf. Syst. Secur.*, 2(3):332–351, 1999.

[51] E. Rescorla. The transport layer security (TLS) protocol version 1.3 (draft 09), Oct. 2015. URL https://tools.ietf.org/html/draft-ietf-tls-tls13-09.

[52] S. Vaudenay. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ... In *EUROCRYPT*, pages 534–546, 2002.

[53] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *USENIX Workshop on Electronic Commerce*, pages 29–40, 1996.