Key-Schedule Security for the TLS 1.3 Standard

Chris Brzuska[1](✉), Antoine Delignat-Lavaud[2], Christoph Egger[3], Cédric Fournet[2], Konrad Kohbrok[1], and Markulf Kohlweiss[4]

[1] Aalto University, Espoo, Finland
{chris.brzuska,konrad.kohbrok}@aalto.fi
[2] Microsoft Research Cambridge, Cambridge, UK
{antdl,fournet}@microsoft.com
[3] IRIF, Université Paris Cité, Paris, France
christoph.egger@alumni.fau.de
[4] University of Edinburgh, Edinburgh, UK
mkohlwei@ed.ac.uk

Abstract. Transport Layer Security (TLS) is the cryptographic backbone of secure communication on the Internet. In its latest version 1.3, the standardization process has taken formal analysis into account both due to the importance of the protocol and the experience with conceptual attacks against previous versions. To manage the complexity of TLS (the specification exceeds 100 pages), prior reduction-based analyses have focused on some protocol features and omitted others, e.g., included session resumption and omitted agile algorithms or vice versa.

This article is a major step towards analysing the TLS 1.3 key establishment protocol as specified at the end of its rigorous standardization process. Namely, we provide a full proof of the TLS *key schedule*, a core protocol component which produces output keys and internal keys of the key exchange protocol. In particular, our model supports all key derivations featured in the standard, including its negotiated modes and algorithms that combine an optional Diffie-Hellman exchange for forward secrecy with optional pre-shared keys supplied by the application or recursively established in prior sessions.

Technically, we rely on *state-separating proofs* (Asiacrypt '18) and introduce techniques to model large and complex derivation graphs. Our key schedule analysis techniques have been used subsequently to analyse the key schedule of Draft 11 of the MLS protocol (S&P '22) and to propose improvements.

Keywords: TLS 1.3 · Key schedule · Protocol analysis · State-separating proofs

1 Introduction

Transport Layer Security (TLS) is the most widely used authenticated secure channel protocol on the Internet, protecting the communications of billions of

S. Agrawal and D. Lin (Eds.): ASIACRYPT 2022, LNCS 13791, pp. 621–650, 2022.
https://doi.org/10.1007/978-3-031-22963-3_21

users. Previous versions of TLS have suffered from impactful attacks against weaknesses in their design, including legacy algorithms (e.g. FREAK for export RSA [9], LogJam [2] for export Diffie-Hellman, WeakDH for ill-chosen groups, and exploits against Mantin biases of RC4 [21]); the RSA key encapsulation (e.g. the ROBOT [19] variant of Bleichenbacher's PKCS1 padding oracle); the fragile MAC-encode-encrypt construction leading to many variants of Vaudenay's padding oracles against CBC cipher suites (e.g. BEAST [38], Lucky13 [3]); the weak signature over nonces allowing protocol version downgrades (e.g. DROWN [5] and POODLE); attacks on other negotiated parameters [11], the key exchange logic (e.g. the cross-protocol attack of [49] and 3SHAKE [12]); exploitations of collisions on the hash transcript (e.g. SLOTH [15]). TLS 1.3 intends both to fix the weaknesses of previous versions and to improve the protocol performance, notably by lowering the latency of connection establishment from two roundtrips down to one, or even zero when resuming a connection.

Historically, the IETF process to adopt a standard involves an open consortium of contributors mostly coming from industry, with a bias towards early implementers. The TLS working group at the IETF acknowledged that this process puts too much emphasis on deployment and implementation concerns, and tends to address security issues reactively [51]. For TLS 1.3, it decided to address security upfront by welcoming feedback from various cryptographic efforts, including symbolic [29,30] and computational protocol models [34,35,48], both on paper and implemented in tools such as Tamarin or CryptoVerif. Early drafts of TLS 1.3 also drew much inspiration from Krawczyk's OPTLS protocol [47], which comes with a detailed security proof, although later versions diverged from it (in particular in the design of resumption). This proactive approach has certainly improved the overall design of TLS 1.3, and uncovered flaws along its 28 intermediate drafts. However, many of these efforts are incomplete (focusing, e.g., on fixed protocol configurations) or do not account for the final version published in RFC 8446, see Sect. 6 for a more detailed discussion of related work. Since final adoption, further questions have been raised about pre-shared keys, potential reflection attacks [37], and difficulties in separating resumption PSKs (produced internally by the key exchange) from external ones installed by the application. In short: we still miss provable security for the final Internet standard.

TLS can be decomposed into sub-protocols: the *record layer* manages the multiplexing, fragmentation, padding and encryption of data into packets (also called *records*) from three separate streams of handshake, alert, and application data. Incoming handshake messages are passed to the *handshake* sub-protocol, which in turn produces fresh record keys and outgoing handshake messages. Taking advantage of this well-understood modularity, other protocols re-use the TLS 1.3 handshake with different record layers: for instance, DTLS 1.3 is a variant based on UDP datagrams instead of TCP streams, while the IETF version of QUIC replaces the record layer with a much extended transport [42], adding features such as dynamic application streams and fine-grained flow control. Detailed security proofs for the TLS 1.3 record layer have been proposed by Patton et al. [52] (extending the work of Fischlin et al. [40] on stream-based channels),

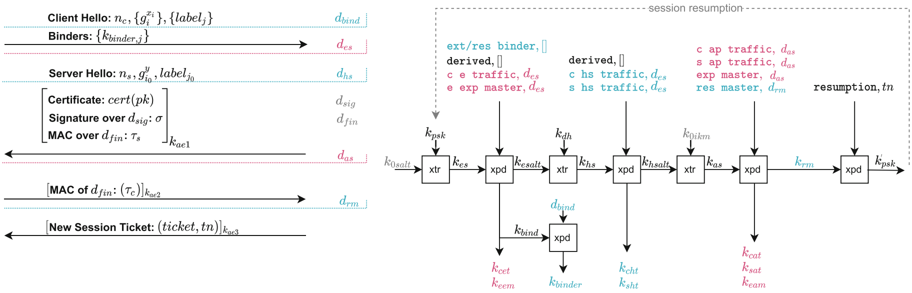


Fig. 1. Overview over the TLS 1.3 Handshake (left) and its key schedule (right). $[m]_k$ denotes encryption of message m under key k. k_{ae1} and τ_c are derived from k_{cht}, k_{ae2} and τ_s are derived from k_{sht}, and k_{ae3} is derived from k_{sat}. We color digests and keys in alternating pink and blue to clarify digest-key dependency. E.g., label `c e traffic` and digest d_{as} is used to derive k_{cet}. (Color figure online)

Badertscher et al. [6], and Bhargavan et al. [32], who also provide a verified reference implementation. Therefore, we defer to these works for the record layer, and focus on the handshake protocol.

1.1 TLS 1.3 Handshake and Key Schedule

The top of Fig. 1 gives an abstract view of the TLS 1.3 protocol message flow. In the client hello message, the client sends a nonce n_c, its Diffie-Hellman (DH) share g^x, a PSK *label* and a *binder* value for domain separation and session resumption. As a means of negotiation, the client may offer shares for different groups and different PSK options (thus the indices i, j in $g_i^{x_i}, label_j, binder_j$). The server communicates its choice of the DH group and the PSK when sending the server hello message which contains the server nonce n_s, its share $g_{i_0}^y$ (including the group description) and the label $label_{j_0}$ of the chosen PSK. The remaining messages consist of server certificate, signature $(\mathrm{C}(pk), \mathrm{CV}(\sigma))$, key confirmation messages in the forms of messages authentication codes (MACs) τ_s and τ_c computed over the transcript, and a *ticket* which is used on the client side to store a resumption key (later referred to as *resumption PSK*) derived from the key material of the current key exchange session.

The *key schedule* is the core part of the handshake that performs all key computations. It takes as main input PSK and DH key materials and, at each phase of the handshake, it derives keys, e.g., to encrypt client early traffic (k_{cet}), to compute the binder value (k_{binder}), to encrypt server handshake traffic (k_{sht}) and to encrypt client handshake traffic (k_{cht}).

The key schedule relies on the hashed key derivation function (HKDF) standard [45], which uses HMAC [7] to implement *extract* (xtr) and *expand* (xpd) operations. In addition, the key schedule makes calls to xpd to expand keys into further subkeys. The key schedule thus consists of a collection of xtr an xpd operations, organized in a graph. Each of the operations takes as input a *chaining* key and/or new key material, (k_{psk} in the xtr in the early phase and k_{dh} in the xtr in the in the handshake phase), together with the latest digest and auxiliary inputs such as a resumption status r and a ticket nonce tn.

In this article, we consider eight output keys of the TLS key schedule: k_{cet}, k_{eem}, k_{binder}, k_{cht}, k_{sht}, k_{cat}, k_{sat}, k_{eam}. They constitute a natural boundary, inasmuch as all other TLS keys and IVs are further derived from them in a transcript-independent manner.

1.2 Key Schedule Model and Key Exchange Model

We model the security of the key schedule as an indistinguishability game between a real and an ideal game. The real game allows the adversary to use their own dishonest application PSKs and Diffie-Hellman shares. In addition, it allows the adversary to instruct the game to sample honest PSKs and Diffie-Hellman shares. From these base keys, the adversary can then instruct the model to derive further keys. The adversary cannot see internal keys, but it can obtain the 8 output keys from the model. In turn, in the ideal game, the output keys are replaced by unique, random keys which are sampled independently from the input key material.

The interface of this model captures how the key exchange protocol uses the key schedule. The key exchange protocol should, indeed, not use the internal keys, but instead only use the output keys. Moreover, the final session keys are to be used only by the Record Layer to implement a secure channel. In a companion paper [25], we show that key exchange security of the TLS 1.3 handshake protocol reduces to the key schedule security established in this paper. Note that authentication is proved based on *keys* and does not capture binding between keys and identities, as needed, e.g., for reflection attacks [29].

Outline. We introduce our overall technical approach in Sect. 2. We define our assumptions for collision-resistance, pseudorandomness and pre-image resistance in Sect. 3. Section 4 defines syntax and security of the TLS key schedule. Section 5 states the main key schedule theorem and provides its proof. This article gives proof sketches of all lemmata, highlighting their conceptual insights. The complete proofs are provided in the full version [23]. Finally, Sect. 7 includes proposals for (late) changes to the TLS 1.3 standard.

2 Technical Approach

2.1 Handles

Complex derivation steps make it crucial to maintain administrative *handles* in the model state, both for internal bookkeeping and security modeling as well as for communication with the adversary. Namely, to instruct the model to perform further computations on keys, the adversary can point to the keys to be used via handles. Such handles are particularly important for honest keys, i.e., honest psks, honest Diffie-Hellman shares and honest internal keys derived via xtr and xpd from honest base keys, because the model cannot provide the adversary with the actual values of these secrets.

Our model constructs handles as nested data records where each nesting step keeps track of the inputs which were used to compute the associated key. We have base handles for PSKs and DH secrets, including handles for dummy zero values to be used in noDH and noPSK mode as well as base handles for a fixed *0salt* and fixed *0ikm*.

$\mathsf{dh}\langle\mathsf{sort}(X,Y)\rangle$	Diffie-Hellman secret
$h = \mathsf{psk}\langle ctr, alg\rangle$	application PSK
$\mathsf{noDH}\langle alg\rangle$	fixed $0^{\mathsf{len}(alg)}$ Diffie-Hellman secret
$\mathsf{noPSK}\langle alg\rangle$	fixed $0^{\mathsf{len}(alg)}$ PSK
$\mathsf{0salt}$	fixed 0 salt
$\mathsf{0ikm}\langle alg\rangle$	fixed $0^{\mathsf{len}(alg)}$ initial key material (IKM)

The model then inductively applies the following constructors to build all other handles from the base handles:

$$\mathsf{xtr}\langle name, \textit{left parent handle}, \textit{right parent handle}\rangle.$$
$$\mathsf{xpd}\langle name, label, \textit{parent handle}, \textit{other arguments}\rangle.$$

For example, given a handle to the early master secret h_{es}, the handle h_{cet} to the client early transport secret is defined as

$$h_{cet} = \mathsf{xpd}\langle cet, \texttt{c e traffic}, h_{es}, t_{es}\rangle$$

where t_{es} is the transcript of the protocol messages exchanged so far, and '$\texttt{c e traffic}$' is the constant byte string label prescribed in the RFC [53] for this derivation step.

Agility. Our model is *agile*, i.e., it supports multiple algorithms. Thus, we tag the handles $h = \mathsf{psk}\langle ctr, alg\rangle$, $\mathsf{noPSK}\langle alg\rangle$ and $\mathsf{0ikm}\langle alg\rangle$ with the algorithm alg for which the keys are intended. Jumping ahead, we note that we also tag *keys* with their intended algorithm so that in the key derivation

$$k_{cet} = \mathsf{xpd}(k_{es}, \texttt{c e traffic}, d_{es}),$$

the agile xpd function can retrieve the correct hash algorithm alg to use within hmac from the key's tag. We write $\mathsf{alg}(h_{cet})$ for the algorithm descriptor of h_{cet} and $\mathsf{tag}_h(k)$ for key k tagged with this algorithm.

Length. The handle determines the algorithm, and the algorithm determines the length of keys and outputs of a hash-algorithm alg. For convenience, we write $\mathsf{len}(h_{cet})$ as an alias for $\mathsf{len}(\mathsf{alg}(h_{cet}))$.

Note that we introduced handles $\mathsf{0ikm}\langle alg\rangle$ for the dummy key value $0^{\mathsf{len}(alg)}$ as well as $\mathsf{0salt}$ for the 1-bit-long 0-key. This is because hmac pads keys with zeroes up to their block length and thus, storing multiple zero values would introduce redundancy in the model without a correspondence in real-life.

Name and Level. In addition to the algorithm and its key length, the handle determines the key name (cet) and a *level*. The level is the number of resumptions the handle records, counting from 0 and adding one for each node with a `resumption` label. We write $\mathsf{level}(h_{cet})$ for this level. We will often need to refer to the *parent* names of a particular key (name) n, and write the pair of parent names as $\mathsf{prntn}(n)$. In the case of xpd, the key is only derived from one key and thus, in this case, $\mathsf{prntn}(n) = (n_1, \bot)$. Conversely, we refer by $\mathsf{chldrnn}(n_1)$ to the set of all key names which are derived from n_1. In particular, if $\mathsf{prntn}(n) = (n_1, \bot)$, then $n \in \mathsf{chldrnn}(n_1)$. We refer to all names which share a parent with n as $\mathsf{sblngn}(n)$.

Handshake Mode. Jumping ahead, we note that we use handle data also to communicate the handshake mode to the key schedule model. A $\mathsf{noDH}\langle alg\rangle$ Diffie-Hellman handle signals a `psk_ke` mode, while a $\mathsf{noPSK}\langle alg\rangle$ PSK handle signals a `dh_ke` mode.

2.2 Application Key Registration and Honesty

Honesty of a handle is a crucial concept to model that the key associated with the handle, when returned to the adversary, looks pseudorandom. Honesty is inductively computed, starting from the base keys: All zero keys have dishonest handles. Handles of application PSKs are honest if their key was sampled by the security model and dishonest if their key was sampled by the security model. Diffie-Hellman handles are honest if both shares are honest. Derived handles are honest if and only if at least one of their input handles are honest. Considering the derivation graph (cf. right side of Fig, 1), we obtain that the h_{esalt} handles and the handles which appear *before* have the same honesty as the last PSK handle, while the handles after h_{esalt} are honest if the last PSK handle was honest *or* the last Diffie-Hellman handle was honest.

2.3 State-Separating Proofs (SSPs)

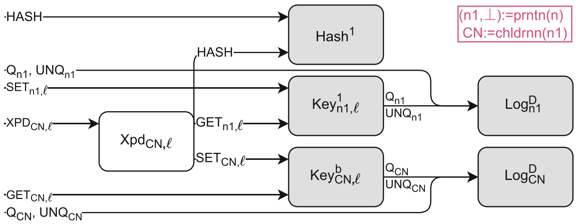


Fig. 2. Game $\mathtt{Gxpd}^b_{n,\ell}$ for $b \in \{0, 1\}$

In the following we use the the pseudorandomness game $\mathtt{Gxpd}^0_{n,\ell}$ for the xpd function (depicted in Fig. 2) as a running example to introduce core concepts. As is common in cryptography, security is modeled as an interaction between an adversary $\mathcal{A}$ (which can be thought of as sitting left of the picture) and a program which we call the *game*. This interaction happens via so-called *oracles*—which we describe in pseudo-code—corresponding to the arrows from the left side of the picture. The task of the adversary consists in *distinguishing* two variants of the game $\mathtt{G}^0$ and $\mathtt{G}^1$ with identical interfaces and we measure the success probability of any such adversary $\mathcal{A}$ and call it *advantage.*

Definition 1 (Advantage). *For adversary $\mathcal{A}$, we define the advantage*

$$\mathsf{Adv}(\mathcal{A}; \mathtt{G}^0, \mathtt{G}^1) := \left|\Pr\left[1 = \mathcal{A} \rightarrow \mathtt{G}^0\right] - \Pr\left[1 = \mathcal{A} \rightarrow \mathtt{G}^1\right]\right|.$$

In particular, for the pseudorandomness game $\mathtt{Gxpd}^b_{n,\ell}$ for xpd, the analogous definition is as follows.

Definition 2 (XPD). *For adversary $\mathcal{A}$, we define the* xpd *pseudorandomness advantage* $\mathsf{Adv}(\mathcal{A}, \mathtt{Gxpd}^0_{n,\ell}, \mathtt{Gxpd}^1_{n,\ell})$ *as*

$$\left|\Pr\left[1 = \mathcal{A} \rightarrow \mathtt{Gxpd}^0_{n,\ell}\right] - \Pr\left[1 = \mathcal{A} \rightarrow \mathtt{Gxpd}^1_{n,\ell}\right]\right|,$$

where Fig. 2 defines $\mathtt{Gxpd}^0_{n,\ell}$.

The graphs specifying such a security game suggest a natural flow downwards. While we discuss the details of the game later in this section, one can extract a conceptual picture already from the graph alone. Concretely the intended usage (by the adversary) of $\mathtt{Gxpd}^b_{n,\ell}$ consists on first registering input values using the $\mathsf{SET}_{n_1,\ell}$ oracle, executing key derivation using the $\mathsf{XPD}_{CN,\ell}$ oracle and finally retrieving and testing the output using the $\mathsf{GET}_{n,\ell}$ oracle. In addition, the adversary gets access to auxiliary oracles, namely the HASH oracle modeling a cryptographic hash function as well as the Q and UNQ oracles.[1] Finally, $\mathtt{Gxpd}^b_{n,\ell}$ is structured in individual components which we call *packages.*

Definition 3 (Package). *A* package $\mathtt{M}$ *consists of a set of oracles* $[\rightarrow \mathtt{M}] = \{\mathsf{O1}, .., \mathsf{Ot}\}$*, specified by pseudo-code and operating on a set of* state variables Σ*, specified on the top of each package description. All other variables used by oracles are temporary and their values are forgotten after each call. The oracles of* $\mathtt{M}$ *may* depend *on oracles* $[\mathtt{M} \rightarrow] = \{\mathsf{O'1}, .., \mathsf{O't'}\}$*, i.e., make calls to oracles in* $[\mathtt{M} \rightarrow]$*. We say that a package* $\mathtt{M}$ *is stateless if* $\Sigma = \emptyset$*. We say that a package* $\mathtt{M}$ *is a game if* $[\mathtt{M} \rightarrow] = \emptyset$*.*

While some oracles of a package are exposed to the adversary, others are used only internally within the game. A monolithic version of a game such as $\mathtt{Gxpd}^b_{n,\ell}$ can be obtained by *inlining* all internal oracle calls. With the concept of packages we can now discuss the individual parts of $\mathtt{Gxpd}^b_{n,\ell}$. $\mathtt{Xpd}_{CN,\ell}$ is a parallel composition of $\mathtt{Xpd}_{n,\ell}$ for all children of n_1 exposing the oracles $\mathsf{XPD}_{n,\ell}$ for $n \in CN$, we write $\mathsf{XPD}_{CN,\ell}$ as shorthand for these oracles. The $\mathtt{Xpd}_{CN,\ell}$ packages are the only stateless packages in the game, indicated by the white color as opposed to the gray of stateful packages (Fig. 2).

[1] These two oracles in particular are necessary for composition: Note that the main oracles the adversary interacts with are subscripted by a name n and a level ℓ while the Q and UNQ oracles only take the name n as subscript. We will share the same Q and UNQ oracles between many instances of $\mathtt{Gxpd}^b_{n,\ell}$ and therefore need to allow reductions access to these oracles.

The $\mathsf{XPD}_{n,\ell}$ oracle of package $\mathtt{Xpd}_{n,\ell}$ (Fig. 3) computes a new handle $h \leftarrow \mathsf{xpd}\langle n, \mathit{label}, h_1, \mathit{args}\rangle$ alongside a new key $k \leftarrow \mathsf{xpd}(k_1, (\mathit{label}, d))$ based on the parent handle h_1, the arguments (e.g. transcript) and the bit r indicating whether this is a resumption session. The evaluation also includes a *label* which depends on the name of the package as well as the resumption bit. Note that the oracle only receives the *handle* of the input key from the adversary and only returns the newly constructed *handle* of the newly derived key. Concrete secrets are passed to $\mathtt{Key}^b_{n,\ell}$ packages using the GET and SET oracles. Here we can distinguish the upper $\mathtt{Key}^1_{n_1,\ell}$ package and the lower $\mathtt{Key}^b_{CN,\ell}$ packages (for all n in CN). We defer discussion about the Q and UNQ oracle calls to the description of the $\mathtt{Log}$ package.

The upper $\mathtt{Key}^1_{n_1,\ell}$ package offers oracle $\mathsf{SET}_{n_1,\ell}(h, \mathit{hon}, k)$ to the adversary which allows it to register a key. The oracle first verifies that the handle h matches the name n and level ℓ of this key package and—modeling algorithmic agility—verifies that the algorithm tag matches the value of the key, and else, assert throws an *abort*. As this is an ideal key package (indicated by superscript $^{b=1}$) for honest keys, instead of using the value provided by the adversary a fresh value is sampled—as indicated by using $\leftarrow_{\$}$ in contrast to $\leftarrow$ used for assignments. Finally the key is stored in this package's state and the handle returned to the caller. The GET oracle simply restores algorithm tagging on the key value and returns it to the caller (in this case the $\mathtt{Xpd}$ package). The lower $\mathtt{Key}^b_{CN,\ell}$ packages work the other way round in that they expose the GET oracle to the adversary while the SET oracle is used by $\mathtt{Xpd}$. We encode the distinguishing task for the adversary in the $\mathtt{Key}^b_{CN,\ell}$ package: In $\mathtt{Gxpd}^0_{n,\ell}$ ($b = 0$), the keys returned from the GET oracle of the $\mathtt{Key}^0_{CN,\ell}$ is honestly computed based on the input keys while in the ideal game $\mathtt{Gxpd}^1_{n,\ell}$ the values of honest keys are sampled in the $\mathtt{Key}$ package ignoring the value computed by $\mathtt{Xpd}$.

```
$\mathtt{Xpd}_{n,\ell}$

Parameters
$n$ : name
$\ell$ : level

$\mathsf{prntn} : N \to (N_\perp \times N_\perp)$
$\mathsf{label} : N \times \{0,1\} \to \{0,1\}^{96}$

State
no state

$\mathsf{XPD}_{n,\ell}(h_1, r, \mathit{args})$
$n_1, \_ \leftarrow \mathsf{prntn}(n)$
$\mathit{label} \leftarrow \mathsf{label}(n, r)$
$h \leftarrow \mathsf{xpd}\langle n, \mathit{label}, h_1, \mathit{args}\rangle$
$(k_1, \mathit{hon}) \leftarrow \mathsf{GET}_{n_1,\ell}(h_1)$
if $n = \mathit{psk}$ :
    $\ell \leftarrow \ell + 1$
    $k \leftarrow \mathsf{xpd}(k_1, (\mathit{label}, \mathit{args}))$
else
    $\mathit{alg} \leftarrow \mathsf{alg}(h_1)$
    $d \leftarrow \mathsf{HASH}(\mathsf{tag}_{\mathit{alg}}(\mathit{args}))$
    $k \leftarrow \mathsf{xpd}(k_1, (\mathit{label}, d))$
$h \leftarrow \mathsf{SET}_{n,\ell}(h, \mathit{hon}, k)$
return $h$
```

Fig. 3. Xpd package

Finally, queries Q_n and UNQ_n to the $\mathtt{Log}_n$ package (Fig. 4) model collisions. The Q query simply returns if a *handle* is re-used while UNQ concerns itself with collisions between keys via an abort pattern and a mapping method. In slightly nonstandard notation, we use existential quantors here to express searching for *indices* into tables. The pattern models conditions on states where the game aborts (i.e. terminates and outputs a special symbol), cf. Sect. 5.3 for their use.

```
$\mathsf{Key}^b_{n,\ell}$

Parameters              State
$n$ : name              $K_{n,\ell}$ : Keytable
$\ell$ : level

$\mathsf{SET}_{n,\ell}(h, hon, k^\star)$
assert $\mathsf{name}(h) = n$
assert $\mathsf{level}(h) = \ell$
assert $\mathsf{alg}(k^\star) = \mathsf{alg}(h)$
$k \leftarrow \mathsf{untag}(k^\star)$
assert $\mathsf{len}(h) = |k|$
if $\mathsf{Q}_n(h) \neq \bot$ : return $\mathsf{Q}_n(h)$
if $b \wedge hon$ :
    $k \leftarrow_{\$} \{0,1\}^{\mathsf{len}(h)}$
$h' \leftarrow \mathsf{UNQ}_n(h, hon, k)$
if $h' \neq h$ : return $h'$
$K_{n,\ell}[h] \leftarrow (k, hon)$
return $h$

$\mathsf{GET}_{n,\ell}(h)$
assert $K_{n,\ell}[h] \neq \bot$
$(k^*, hon) \leftarrow K_{n,\ell}[h]$
$k \leftarrow \mathsf{tag}_h(k^*)$
return $(k, hon)$
```

```
$\mathsf{Log}^{P,map}_n$

Parameters
$n$ : name

State
$L_n$ : Log

$\mathsf{Q}_n(h)$
if $L_n[h] = \bot$ : return $\bot$
else
    $(h', \_, \_) \leftarrow L_n[h]$
    return $h'$

$\mathsf{UNQ}_n(h, hon, k)$
if $(\exists\ h' : L_n[h'] = (h', hon', k)$
        $\wedge\ \mathsf{level}(h) = r \wedge \mathsf{level}(h^\star) = r')$ :
    if $map(r, hon, r', hon' J_n[k])$ :
        $L_n[h] \leftarrow (h', hon, k)$
        $\mathrm{J}_n[k] \leftarrow 1$
        return $h'$
if $(\exists\ h^\star : L_n[h^\star] = (h', hon', k)$
        $\wedge\ \mathsf{level}(h) = r \wedge \mathsf{level}(h^\star) = r')$ :
    $P(r, hon, r', hon')$
$L_n[h] \leftarrow (h, hon, k)$
return $h$
```

P	the command $P(r, hon, r', hon')$ is
Z	$\emptyset$
A	**if** $hon = hon' = 0 \wedge r = r' = 0$: **throw** *abort*
D	**if** $hon = hon' = 0$: **throw** *abort*
R	**if** $hon = hon' = 0$: **throw** *abort* **else throw** *win*
F	**throw** *abort*

map	the command $map(r, hon, r', hon', \mathrm{J}_n[k])$ is
0	0
1	$hon = hon' = 0 \wedge r \neq r' \wedge 0 \in \{r, r'\} \wedge \mathrm{J}_n[k] \neq 1$
∞	$hon = hon' = 0$

Fig. 4. Code for the `Key` and `Log`. In addition we use `Nkey` for a single key package that answers queries for all levels from the same table and `Okey` for a `NKey` package which consistently answers with the constant all-zeros key.

We use the **throw** notation here to allow special symbols in addition to *abort* which is also used by assert. In the game $\mathtt{Gxpd}^{b}_{n,\ell}$, the D pattern aborts on collisions between dishonest keys. The F and R pattern abort if there is a collision between key values, regardless of their honesty, and they return different abort messages. Z does not abort at all, and A aborts upon a collision of two dishonest level 0 keys (which we use to constrain the adversary's psk registrations in the key schedule model).

Mapping methods filter certain collisions (preventing an *abort* event. ∞ allows collisions between Diffie-Hellman secrets (the adversary can construct colliding values via $X^zY = XY^z$) and the 1 method allows the adversary to register a dishonest application PSK colliding with an dishonest resumption PSK. The mapping methods are only used in the proof and not in the security model.

3 Assumptions

3.1 Collision-Resistance

Figure 5 defines the collision-resistance game $\mathtt{Gcr}^{\mathsf{f}\text{-}alg,b}$ for a given function $\mathsf{f}\text{-}alg$, where $\mathsf{f} \in \{\mathsf{hash}, \mathsf{xtr}, \mathsf{xpd}\}$ and $alg \in \mathcal{H}$ which TLS 1.3 currently defines as

$$\mathcal{H} = \{\mathtt{sha256}, \mathtt{sha384}, \mathtt{sha512}\}$$

(see FIPS 180-2). The HASH oracle takes as input a text t from the domain of $\mathsf{f}\text{-}alg$ and returns its digest d. If that text t has not been queried before, the digest is stored in table H at index t. In the ideal game ($b = 1$), the oracle first checks whether d already occurs in H, and if so, throws an abort. Hence, the adversary can distinguish between the real and the ideal game if and only if it can submit two different texts with the same digest. Our definition generalizes to n-ary functions by letting the text t be the tuple of their arguments.

```
$\mathtt{Gcr}^{\mathsf{f}\text{-}alg,b}$
$\mathsf{HASH}(t)$
assert $t \in dom(\mathsf{f}\text{-}alg)$
$d \leftarrow \mathsf{f}\text{-}alg(t)$
if $H[t] = \bot$ :
    if $b \wedge d \in range(H)$ :
        throw $abort$
    $H[t] \leftarrow d$
return $d$
```

Fig. 5. $\mathtt{Gcr}^{\mathsf{f}\text{-}alg,b}$ code.

Definition 4 (Collision-Resistance). *For an adversary $\mathcal{A}$, a function $\mathsf{f} \in \{\mathsf{hash}, \mathsf{xtr}, \mathsf{xpd}\}$ and algorithm $alg \in \mathcal{H}$, define collision-resistance advantage* $\mathsf{Adv}(\mathcal{A}, \mathtt{Gcr}^{\mathsf{f}\text{-}alg,0}, \mathtt{Gcr}^{\mathsf{f}\text{-}alg,1})$ *is*

$$\left|\Pr\left[1 = \mathcal{A} \rightarrow \mathtt{Gcr}^{\mathsf{f}\text{-}alg,0}\right] - \Pr\left[1 = \mathcal{A} \rightarrow \mathtt{Gcr}^{\mathsf{f}\text{-}alg,1}\right]\right|.$$

Agile Collision-Resistance. It is convenient to define the *agile* collision-resistance game $\mathtt{Gacr}^{\mathsf{f},b}$ as well, where $\mathsf{f} \in \{\mathsf{hash}, \mathsf{xtr}, \mathsf{xpd}\}$ takes *tagged* inputs, i.e., hash takes a single input, tagged with the algorithm to use, xpd takes three inputs $(k, label, args)$, where k is tagged, and xtr takes inputs (k_1, k_2) where one is tagged, and if both are tagged, they are tagged consistently. The adversary can then make queries to HASH with values in the domain of the *agile* functions. We write $\mathtt{Hash}^b := \mathtt{Gacr}^{\mathsf{hash},b}$. See Sect. 2.1 for further discussion of tagging.

3.2 Pseudorandomness of xpd

For most key names n, Definition 2 already captures pseudorandomness of xpd. We now cover two special cases.

XPD to Derive PSK. For $n = psk$ (cf. Fig. 6a), the *layer* index increases from ℓ to $\ell+1$. Thus, the $\mathsf{XPD}_{psk,\ell}$ oracle reads keys via $\mathsf{GET}_{rm,\ell}$ queries, but writes keys using the level $\ell+1$ query $\mathsf{SET}_{psk,\ell+1}$. Another difference in $\mathtt{Gxpd}^{b}_{psk,\ell}$ compared to the general $\mathtt{Gxpd}^{b}_{n,\ell}$ is that the lower $\mathtt{Log}^{D1}_{psk}$ package uses a $D1$ pattern for logging which ignores level 0 $\mathsf{UNQ}_{psk}(h, hon, k)$ queries with $hon = 0$ whenever there already exists a dishonest handle h' for key value k at level 0. Since $\mathsf{XPD}_{psk,\ell}$ writes only on level $\ell + 1 > 0$, this difference in logging does not affect the strength of the assumption, but it makes the assumption code align with the key schedule game, cf. Sect. 4.1. Finally, for deriving the psk, no hash-operation is performed.

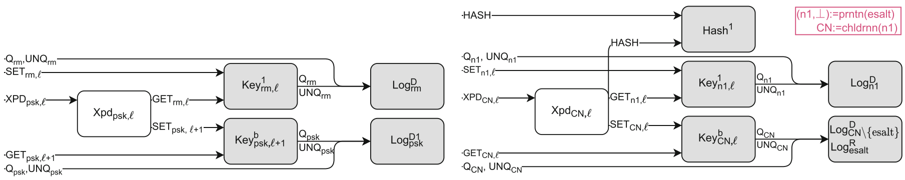
Definition 5 (XPD for psk). *For an adversary $\mathcal{A}$, we define the* xpd *pseudorandomness advantage for psk derivation* $\mathsf{Adv}(\mathcal{A}, \mathtt{Gxpd}^{0}_{psk,\ell}, \mathtt{Gxpd}^{1}_{psk,\ell})$ *as*

$$\left|\Pr\left[1 = \mathcal{A} \rightarrow \mathtt{Gxpd}^{0}_{psk,\ell}\right] - \Pr\left[1 = \mathcal{A} \rightarrow \mathtt{Gxpd}^{1}_{psk,\ell}\right]\right|$$

XPD to Derive Esalt. For $n = esalt$, the lower $\mathtt{Log}^{R}_{esalt}$ package uses an R pattern instead of a D pattern, sending abort messages whenever the same key value k is registered as an *esalt* under two distinct handles h and h' (across all levels and regardless of honesty). Note that the adversary could simulate the R pattern itself (by retrieving all keys and checking for equality) and thus, the R pattern only *weakens* the adversary since it can no longer query the game after triggering an R abort and since the adversary does not learn the value of the collision which caused the abort.

Definition 6 (XPD for esalt). *For an adversary $\mathcal{A}$, we define the* xpd *pseudorandomness advantage for esalt derivation* $\mathsf{Adv}(\mathcal{A}, \mathtt{Gxpd}^{0}_{esalt,\ell}, \mathtt{Gxpd}^{1}_{esalt,\ell})$ *as*

$$\left|\Pr\left[1 = \mathcal{A} \rightarrow \mathtt{Gxpd}^{0}_{esalt,\ell}\right] - \Pr\left[1 = \mathcal{A} \rightarrow \mathtt{Gxpd}^{1}_{esalt,\ell}\right]\right|.$$


$\mathsf{Q}_{rm}, \mathsf{UNQ}_{rm}$
$\mathsf{SET}_{rm,\ell}$
$\mathsf{XPD}_{psk,\ell}$
$\mathsf{Xpd}_{psk,\ell}$
$\mathsf{GET}_{rm,\ell}$
$\mathsf{SET}_{psk,\ell+1}$
$\mathsf{Key}^{1}_{rm,\ell}$
$\mathsf{Key}^{b}_{psk,\ell+1}$
Q_{rm}
UNQ_{rm}
Q_{psk}
UNQ_{psk}
Log^{D}_{rm}
Log^{D1}_{psk}
$\mathsf{GET}_{psk,\ell+1}$
$\mathsf{Q}_{psk}, \mathsf{UNQ}_{psk}$
HASH
Hash^{1}
$(n1,\perp)$:=prntn(esalt)
CN:=chldrnn(n1)
HASH
$\mathsf{Q}_{n1}, \mathsf{UNQ}_{n1}$
$\mathsf{SET}_{n1,\ell}$
$\mathsf{XPD}_{CN,\ell}$
$\mathsf{Xpd}_{CN,\ell}$
$\mathsf{GET}_{n1,\ell}$
$\mathsf{SET}_{CN,\ell}$
$\mathsf{Key}^{1}_{n1,\ell}$
$\mathsf{Key}^{b}_{CN,\ell}$
Q_{n1}
UNQ_{n1}
Q_{CN}
UNQ_{CN}
Log^{D}_{n1}
$\mathsf{Log}^{D}_{CN\setminus\{esalt\}}$
Log^{R}_{esalt}
$\mathsf{GET}_{CN,\ell}$
$\mathsf{Q}_{CN}, \mathsf{UNQ}_{CN}$

(a) Game $\mathtt{Gxpd}^{b}_{psk,\ell}$ for $b \in \{0,1\}$

(b) Game $\mathtt{Gxpd}^{b}_{esalt,\ell}$ for $b \in \{0,1\}$

Fig. 6. xpd assumptions

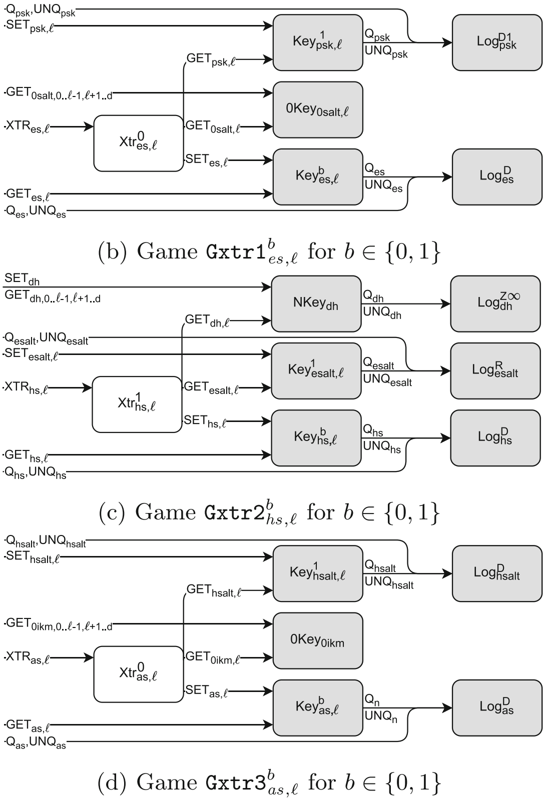
```
$\mathtt{Xtr}^b_{n,\ell}$

Parameters
$n$ : name
$\ell$ : level
$b$ : bit
$\mathsf{prntn} : N \to (N_\perp \times N_\perp)$
$\mathsf{label} : N \times \{0,1\} \to \{0,1\}^{96}$

State
no state

$\mathsf{XTR}_{n,\ell}(h_1, h_2)$
$n_1, n_2 \leftarrow \mathsf{prntn}(n)$
if $\mathsf{alg}(h_1) \neq \perp \wedge \mathsf{alg}(h_2) \neq \perp$ :
    assert $\mathsf{alg}(h_1) = \mathsf{alg}(h_2)$
$h \leftarrow \mathsf{xtr}\langle n, h_1, h_2 \rangle$
$(k_1, hon_1) \leftarrow \mathsf{GET}_{n_1,\ell}(h_1)$
$(k_2, hon_2) \leftarrow \mathsf{GET}_{n_2,\ell}(h_2)$
$k \leftarrow \mathsf{xtr}(k_1, k_2)$
$hon \leftarrow hon_1 \vee hon_2$
if $b \wedge hon_2$ :
    $k^\star \leftarrow_\$ \{0,1\}^{\mathsf{len}(k)}$
    $k \leftarrow \mathsf{tag}_{\mathsf{alg}(k)}(k^\star)$
$h \leftarrow \mathsf{SET}_{n,\ell}(h, hon, k)$
return $h$
```

(a) Code of Xtr



(b) Game $\mathtt{Gxtr1}^b_{es,\ell}$ for $b \in \{0,1\}$

(c) Game $\mathtt{Gxtr2}^b_{hs,\ell}$ for $b \in \{0,1\}$

(d) Game $\mathtt{Gxtr3}^b_{as,\ell}$ for $b \in \{0,1\}$

Fig. 7. xtr Pseudorandomness Assumption

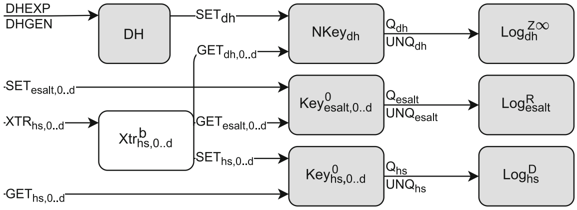
3.3 Pseudorandomness of xtr

The TLS 1.3 key schedule performs three xtr operations (cf. Fig. 1), and the modeling is analogous to the XPD assumptions, except that for the early secret *es*, xtr security relies on the *psk* which is the *right* input to xtr, and for the application secret *as*, xtr security relies on *esalt* which is the *left* input to xtr. The derivation of the handshake secret *hs* is a special case, because its security is an *OR* of the honesty of its left and right input. We here state the xtr security assumption required for *hs* security based on its *left* input *esalt* and turn to the security based in its right input (the Diffie-Hellman (DH) secret) shortly. Note that the security of *esalt* will be applied *after* the security of the DH secret and thus, the bit b in the $\mathtt{Xtr}^b_{hs,\ell}$ is already set to 1 and samples output keys uniformly at random whenever the Diffe-Hellman secret is honest. The security of *esalt* thus only increases security for those keys where the Diffie-Hellman secret is dishonest.

Definition 7 (XTR advantages). *For adversary $\mathcal{A}$, level $\ell \in \mathbb{N}_0$, we define the* xtr *pseudorandomness advantage for es as* $\mathsf{Adv}(\mathcal{A}, \mathtt{Gxtr1}^0_{es,\ell}, \mathtt{Gxtr1}^1_{es,\ell})$*, the pseudorandomness advantage for hs as* $\mathsf{Adv}(\mathcal{A}, \mathtt{Gxtr2}^0_{hs,\ell}, \mathtt{Gxtr2}^1_{hs,\ell})$ *and the pseudorandomness advantage for as as* $\mathsf{Adv}(\mathcal{A}, \mathtt{Gxtr3}^0_{as,\ell}, \mathtt{Gxtr3}^1_{as,\ell})$*, where Fig. 7b–7d define the games* $\mathtt{Gxtr1}^b_{es}$*,* $\mathtt{Gxtr2}^b_{hs}$ *and* $\mathtt{Gxtr}^b_{as}$ *and Definition 1 defines advantage.*

3.4 Salted ODH

Our salted oracle Diffie-Hellman assumption (SODH) is a stronger variant of the oracle Diffie-Hellman assumption introduced by Abdalla et al. [1] and the PRF oracle Diffie-Hellman assumption studied by Brendel et al. [20]. Most importantly, SODH is an *agile*, i.e., it requires pseudorandomness of the derived keys even when the adversary can see hash-values of the same Diffie-Hellman secret under *different* hash-functions and different, possibly adversarially chosen salts. In practice, different salts can emerge from disagreement between server and client about the PSK to use since the early salt *esalt* (and possibly also the *alg*) changes when the PSK changes (see Fig. 1). The $\mathtt{Gsodh}^b$ game (cf. Fig. 8) allows the adversary to generate honest Diffie-Hellman shares via DHGEN, to combine them (or an honest and a dishonest share) into a Diffie-Hellman secret via DHEXP and to derive keys from them via $\mathsf{XTR}_{n,\ell}$ for an arbitrary level $\ell \in \{0, .., d\}$. Oracle $\mathsf{GET}_{n,\ell}$ then allows to retrieve the derived keys. Note that pseudorandomness is modeled, this time, by a bit in the $\mathtt{Xtr}^b_{n,\ell}$ package (Fig. 7a).



DH

```
Parameters
$G$ : set of groups
$\mathsf{ord} : G \to \mathbb{N}$

State
$E$ : table
```

```
DHGEN($grp$)
assert $grp \in G$
$g \leftarrow \mathsf{gen}(grp)$
$x \leftarrow_{\$} Z_{\mathsf{ord}(grp)}$
$X \leftarrow g^x$
$E[X] \leftarrow x$
return $X$
```

```
DHEXP($X, Y$)
assert $\mathsf{grp}(X) = \mathsf{grp}(Y)$
$h \leftarrow \mathsf{dh}\langle \mathsf{sort}(X, Y) \rangle$
$hon_X \leftarrow E[X] \neq \bot$
$hon_Y \leftarrow E[Y] \neq \bot$
assert $hon_X = 1$
$x \leftarrow E[X]; k \leftarrow Y^x$
$hon \leftarrow hon_X \wedge hon_Y$
$h \leftarrow \mathsf{SET}_{dh}(h, hon, k)$
return $h$
```

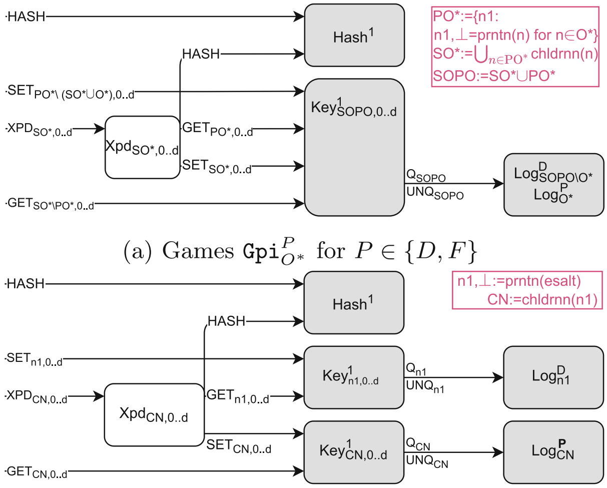
Fig. 8. Game $\mathtt{Gsodh}^b$ (top), package $\mathtt{Dh}$ (bottom)

Definition 8 (SODH). *For an adversary $\mathcal{A}$, we define the Salted Oracle Diffie Hellman (SODH) advantage* $\mathsf{Adv}(\mathcal{A}, \mathtt{Gsodh}^0, \mathtt{Gsodh}^1) :=$

$$\left|\Pr\left[1 = \mathcal{A} \to \mathtt{Gsodh}^0\right] - \Pr\left[1 = \mathcal{A} \to \mathtt{Gsodh}^1\right]\right|,$$

3.5 Pre-image Resistance for xpd

Pseudorandomness and collision resistance of xpd also imply that it is hard to find pre-images for *honest* output keys. We prove this implication in the full version of this article [23, Lemma E.7] and in this conference version rely on pre-image resistance as a separate assumption for convenience.



(a) Games $\texttt{Gpi}^P_{O*}$ for $P \in \{D, F\}$

(b) Games $\texttt{Gpi}^P_{esalt}$ for $P \in \{R, D\}$

Fig. 9. Pre-image resistance assumptions

Definition 9 (Pre-image resistance advantages). *For an adversary* $\mathcal{A}$ *and level* $\ell \in \mathbb{N}_0$ *we define the pre-image resistance advantage for deriving keys in* O^* *(a set to be specified later)* $\mathsf{Adv}(\mathcal{A}, \texttt{Gpi}^D_{O*}, \texttt{Gpi}^F_{O*}) :=$

$$\left|\Pr\left[1 = \mathcal{A} \rightarrow \texttt{Gpi}^D_{O*}\right] - \Pr\left[1 = \mathcal{A} \rightarrow \texttt{Gpi}^F_{O*}\right]\right|,$$

the pre-image resistance advantage for deriving keys with the same parent as esalt by $\mathsf{Adv}(\mathcal{A}, \texttt{Gpi}^D_{esalt}, \texttt{Gpi}^F_{esalt}) :=$

$$\left|\Pr\left[1 = \mathcal{A} \rightarrow \texttt{Gpi}^D_{esalt}\right] - \Pr\left[1 = \mathcal{A} \rightarrow \texttt{Gpi}^F_{esalt}\right]\right|.$$

Figure 9b and Fig. 9b define $\texttt{Gpi}^P_{O*}$ *and* $\texttt{Gpi}^P_{esalt}$.

Our modular assumptions for xpd and xtr are agile, multi-instance security assumptions with registration of dishonest keys. They reduce to their non-agile, single-instance, monolithically written counterparts with a security loss equal to the number of honest keys. Since TLS 1.3 currently only supports hash-algorithms of different length, indeed, our agile assumptions for xtr and xpd reduce to *non-agile* assumptions. In turn, we can only reduce our modular agile SODH assumption to an *agile* monolithic SODH assumption, because TLS 1.3 indeed requires such a strong, agile SODH assumption (cf. Sect. 3.4 and Sect. 7) for further discussion. See full version [23, Appendix E] for the reduction proofs.

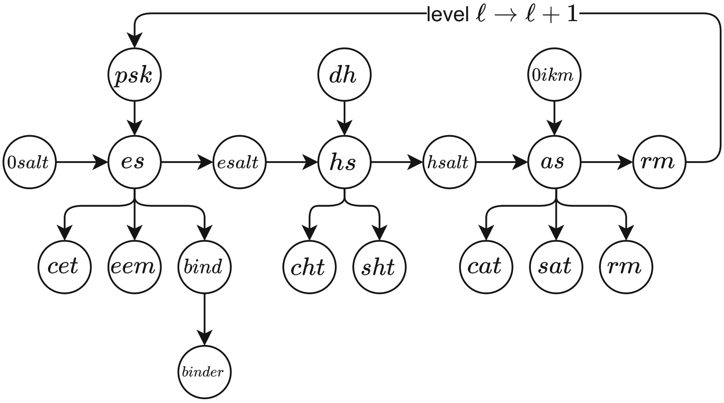


Fig. 10. Parent names prntn in TLS 1.3

4 Key Schedule

We reason about the TLS 1.3 key schedule in terms of its three elementary operations extract (xtr), expand (xpd) and computation of Diffie-Hellman secrets. This section first introduces an abstract key schedule syntax and refines it to capture TLS 1.3 as part of a bigger class of *TLS-like* key schedules. We then define key schedule security and state our theorem for all TLS-like key schedules.

4.1 Key Schedule Syntax

Our formalization interprets the key schedule as a directed graph where nodes describe *key names* (cf. Fig. 10 for the case of TLS 1.3). In addition to the set of names N and the graph description (encoded as prntn function, cf. Sect. 2.1), a key schedule has a function label which maps the name and a resumption bit to a derivation label. We conveniently model hmac operations by using xpd with *empty label* as an alias for hmac. By sound cryptographic practice, a key should be either used for xpd or for hmac but not both, so if a node has an empty label, it is not allowed to have siblings. Similarly, xtr operations only yield a single child, and the multiple children of xpd operations are derived using distinct labels.

Definition 10 (Key Schedule Syntax). *A key schedule* $ks = (N, \mathsf{label}, \mathsf{prntn})$ *consists of a set of names* N *and two functions*

$$\begin{aligned} \mathsf{label}: &\quad N \times \{0,1\} \to \{0,1\}^{96} \cup \{\bot\} \\ \mathsf{prntn}: &\quad N \to (N \cup \bot) \times (N \cup \bot) \end{aligned}$$

with the previously described restrictions.

Figure 10 describes the prntn function of the TLS 1.3 key schedule as a graph. Stating and proving our theorem in terms of the concrete TLS key schedule would require listing and treating each xpd operation individually. Instead, we prove our theorem for all *TLS-like* key schedules (of which the TLS key schedule is an instance). We consider a key schedule as *TLS-like* if it aligns with TLS in terms of base keys and xtr operations and treats the *psk* name as the main root from which all keys except for the base keys can be reached. Moreover, a TLS-like key schedule only has a single loop. This loop contains the edge from *rm* to *psk* and models resumptions. This edge has the special property of increasing the associated level as the *psk* is computed in an earlier session to be used in a later key schedule session. As such the cycle does not contradict an ordering on key computations.

Definition 11 (TLS-like Key Schedule Syntax). *A key schedule* $ks = (N, \mathsf{label}, \mathsf{prntn})$ *is* TLS-like *if its* prntn *graph satisfies the above restrictions, its set of names* N *contains at least the names* $0salt, psk, es, esalt, dh, hs, hsalt, 0ikm, as, rm$ *and the* prntn *function maps* $0salt$*,* dh *and* $0ikm$ *to* $(\bot, \bot)$*, maps es, hs and as according to Fig. 10, maps psk to* $(rm, \bot)$ *and each of the remaining names* n *to some pair* $(n_1, \bot)$ *with* $n_1 \neq \bot$*.*

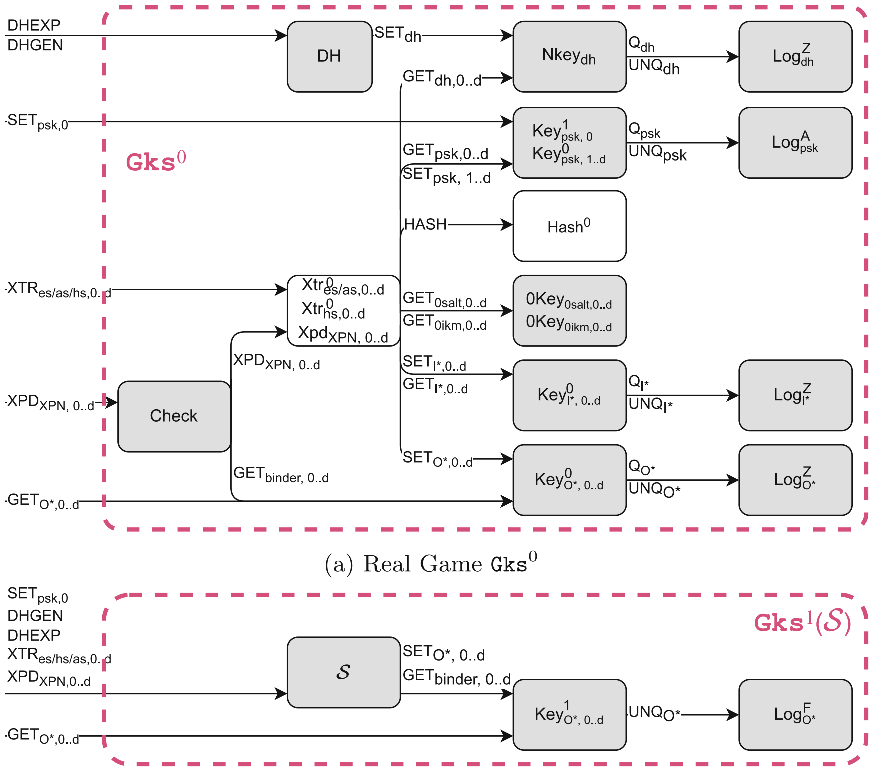
We use several subsets of N which we summarize in Table 1.

4.2 Key Schedule Security Model

Our key schedule security model captures that the key schedule produces keys which are pseudorandom and unique. We formulate security as indistinguishability between a real and an ideal game where the real game implements the actual key schedule derivations, while in the ideal game, output keys are unique, and honest keys are sampled uniformly at random. Concretely, we follow a simulation approach (somewhat similar to the Canetti and Krawczyk [26] approach to key exchange), where the ideal game is defined as a composition of a simulator $\mathcal{S}$ and an ideal functionality. The simulator instructs the ideal functionality to produce output keys of certain length, however the *value* of the output keys is sampled independently from the simulator. As we require that no adversary can distinguish these two settings this captures security: The protocol determines when an output key becomes available and which type of key but no information about the concrete value is disclosed in the protocol (as the simulator does not have such information).

Concretely, in our ideal game $\mathtt{Gks}^1(\mathcal{S})$ (Fig. 11b), the simulator $\mathcal{S}$ is a parameter and the $\mathtt{Key}^1_{O^*,0..d}$ and $\mathtt{Log}_{O^*}$ packages (cf. Sect. 2.3) constitute the ideal functionality. Namely, the $\mathtt{Key}^1_{O^*,0..d}$ package samples a uniformly random key for handles which correspond to honest keys with a name $n \in O^*$ and some level $0 \leq \ell \leq d$. The $\mathtt{Log}_{O^*}$ package, in turn, ensures that each handle corresponds to a *different* key, modeling key uniqueness for both honest and dishonest keys.

Similarly, we describe the real execution of the key schedule as a game $\mathtt{Gks}^0$, written in pseudocode. Following the SSP methodology outlined in Sect. 2.3, we split the pseudocode of the game $\mathtt{Gks}^0$ into several packages most of which ($\mathtt{Xpd}$, $\mathtt{Xtr}$, $\mathtt{DH}$, $\mathtt{Key}$, and $\mathtt{Log}$) have been introduced before and $\mathtt{Check}$ is described in



(a) Real Game $\mathtt{Gks}^0$

(b) Ideal Game $\mathtt{Gks}^1(\mathcal{S})$

Fig. 11. Key schedule security games with internal keys I^*, output keys O^* and XPN, the set of key names produced by $\mathtt{xpd}$. We write $\mathtt{0K}_n$ as an abbreviation for $\mathtt{Nk}_n \rightarrow \mathtt{L}^Z_n$. We initialize $\mathtt{K}$ and $\mathtt{Nk}_n$ with suitable 0 values (cf. Sect. 2.1).

Table 1. Notation

N	The set of all (key) names
N^*	$N \setminus \{psk, dh\}$
I^*	The set of internal keys $\{n \in N^* \mid \mathsf{chldrnn}(n) = \emptyset\}$
O^*:	The set of output keys $\{n \in N^* \mid \mathsf{chldrnn}(n) = \emptyset\}$
O:	$O^* \cup \{psk\}$
S:	The set of separation points (Definition 13)
XPN:	The set of expand names $\{n \in N : \mathsf{prntn}(n) = (\_, \bot)\}$
XPR:	The set of representatives (Sect. 4.3)

Sect. 4.3. Figure 11a depicts the composed game $\texttt{Gks}^0$—recall that this graph is not merely an illustration, it is part of the formal definition of $\texttt{Gks}^0$.

The game $\texttt{Gks}^0$ exposes $\mathsf{SET}_{psk,0}$ and DHGEN oracles which sample honest Diffie-Hellman shares, honest application PSKs and enable the adversary to register dishonest application PSKs with a chosen value. The XTR and XPD oracles trigger key derivations. Finally, the adversary can access output keys via the GET oracle on the (real) key package $\texttt{Key}^0_{O^*,0..d}$.

Definition 12 (Key Schedule Advantage). *For a key schedule $ks = (N, \mathsf{label}, \mathsf{prntn})$, a natural number d, a simulator $\mathcal{S}$ and an adversary $\mathcal{A}$ which makes queries for at most d levels we define the advantage* $\mathsf{Adv}(\mathcal{A}, \texttt{Gks}^0, \texttt{Gks}^1(\mathcal{S})) :=$

$$\left|\Pr\left[1 = \mathcal{A} \to \texttt{Gks}^0\right] - \Pr\left[1 = \mathcal{A} \to \texttt{Gks}^1(\mathcal{S})\right]\right|,$$

where Fig. 11b defines $\texttt{Gks}^1(\mathcal{S})$ and Fig. 11a defines $\texttt{Gks}^0$.

4.3 Front-End Checks

The Check package acts as a restriction on the adversary since the **assert** conditions in the Check code force the adversary to use the correct Diffie-Hellman shares and binder value in its transcript when the transcript is included in a derivation step. In terms of composability, the **assert** conditions in Check force the key exchange to call the key schedule with consistent values, i.e., derive the Diffie-Hellman secret from a pair of shares that is included in the transcript and not from an unrelated pair of shares. The TLS 1.3 specification ensures these innocent conditions, and requiring them formally means that the proof breaks down when session memory in TLS 1.3 is unsafely implemented.

In addition to enforcing the use of consistent shares in the transcript, the XPD oracle of the Check package (Fig. 12) ensures that the resumption flag is consistent with the level of the PSK; and that the binder tag included in the transcript of later stages (at the end of the last ClientHello message) is the same that was computed and checked in the early stage. The transcript is not included into all xpd derivations, but only once on the path from *psk* to output key, and Check only filters queries on these particular derivation steps.

```
Check
$\mathsf{XPD}_{n,\ell}(h_1, r, args)$
if $n = bind$ :
    if $r = 0$, assert $\mathsf{level}(h_1) = 0$
    if $r = 1$, assert $\mathsf{level}(h_1) > 0$
elseif $n \in S \cap early$ :
    $binder \leftarrow \mathsf{BinderArgs}(args)$
    $h_{bndr} \leftarrow \mathsf{BinderHand}(h_1, args)$
    $(k, \_) \leftarrow \mathsf{GET}_{binder,\ell}(h_{bndr})$
    assert $binder = k$
elseif $n \in S$ :
    $X, Y \leftarrow \mathsf{DhArgs}(args)$
    $h_{dh} \leftarrow \mathsf{DhHand}(h_1)$
    assert $h_{dh} = \mathsf{dh}\langle\mathsf{sort}(X, Y)\rangle$
    $binder \leftarrow \mathsf{BinderArgs}(args)$
    $h_{bndr} \leftarrow \mathsf{BinderHand}(h_1, args)$
    $(k, \_) \leftarrow \mathsf{GET}_{binder,\ell}(h_{bndr})$
    assert $binder = k$
$h \leftarrow \mathsf{XPD}_{n,\ell}(h_1, r, args)$
return $h$
```

Fig. 12. Code of Check

Since including the transcript ensures domain separation between different protocol runs and derivation pathes, we refer to the derivation steps which include the transcript as a *separation point*.

Definition 13 (Separation Points). *For a key schedule* $ks = (N, \mathsf{label}, \mathsf{prntn})$*, we call* $S \subseteq N$ *a* set of separation points*, if it satisfies the following two requirements:*

- $\forall\, n \in O$*: the path from* psk *to* n *contains an* $n' \in S$*.*
- *If there exists a path from* dh *to an* $n \in O$*, then it contains an* $n' \in S$*.*

In addition, for each xpd operation, we choose one representative child. I.e., $XPR \subseteq N$ is a *representative set* for ks if $psk, esalt \in XPR$ and for each name $n \in N$ with only a single parent (these are the xpd nodes), either n or exactly one sibling of n is contained in XPR.

5 Key Schedule Theorem

Theorem 1. *Let* ks *be a TLS-like key schedule with representative set* XPR *and separation points* S*. Let* $d \in \mathbb{N}$*. There is an efficient simulator* $\mathcal{S}$ *such that for all adversaries* $\mathcal{A}$ *which make queries for at most* d *resumption levels,*

$$\begin{aligned}
\mathsf{Adv}(\mathcal{A}, \mathtt{Gks}^0, \mathtt{Gks}^1(\mathcal{S})) \leq{}& \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^{main}_{cr}, \mathtt{Gacr}^{\mathsf{hash},b}) \\
&+ \sum_{j \in \{Z,D\}, \mathsf{f} \in \{\mathsf{xtr},\mathsf{xpd}\}} \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^{main}_{j,\mathsf{f}}, \mathtt{Gacr}^{\mathsf{f},b}) \\
&+ \max_{i \in \{0,1\}} \Big[\mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}^{main}_{sodh}, \mathtt{Gsodh}^b) \\
&\quad \sum_{\ell=0}^{d-1} \big(\mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}^{main}_{es,\ell}, \mathtt{Gxtr}^b_{es,\ell}) \\
&\quad\quad +\mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}^{main}_{hs,\ell}, \mathtt{Gxtr}^b_{hs,\ell}) \\
&\quad\quad +\mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}^{main}_{as,\ell}, \mathtt{Gxtr}^b_{as,\ell}) \\
&\quad + \sum_{n \in XPR} \big(\mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}^{main}_{n,\ell}, \mathtt{Gxpd}^b_{n,\ell})\big)\big) \\
&\quad\quad +\mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}^{main}_{esalt,pi}, \mathtt{Gpi}^b_{esalt}) \\
&\quad\quad +\mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}^{main}_{O^*,pi}, \mathtt{Gpi}^b_{O^*})\Big],
\end{aligned}$$

where $\mathcal{A}_i$ *behaves as* $\mathcal{A}$ *except that it returns bit* i *on a so-called* win *abort (cf. [23, Lemma D.4]);* $\mathcal{R}^{main}_* := \mathcal{R}^{ch\text{-}map} \to \mathcal{R}_*$ *when replacing* $*$ *by* cr*,* (Z, f)*,* (D, f)*,* $sodh$*,* es*,* hs*,* as*,* n*,* O^*, pi *or* $esalt, pi$*, the simulator* $\mathcal{S}$ *is marked in grey in [23, Fig. 26b], [23, Fig. 32a] defines* $\mathcal{R}_{sodh}$*, [23, Fig. 34a] defines* $\mathcal{R}_{es,\ell}$*,* $\mathcal{R}_{hs,\ell}$ *and* $\mathcal{R}_{as,\ell}$ *are defined analogously, and [23, Fig. 34b] defines* $\mathcal{R}_{n,\ell}$ *for* $n \in XPR$*,* $0 \leq \ell \leq d$*, [23, Fig. 32c] defines* $\mathcal{R}_{esalt,pi}$ *and [23, Fig. 32d] defines* $\mathcal{R}_{O^*,pi}$*.*

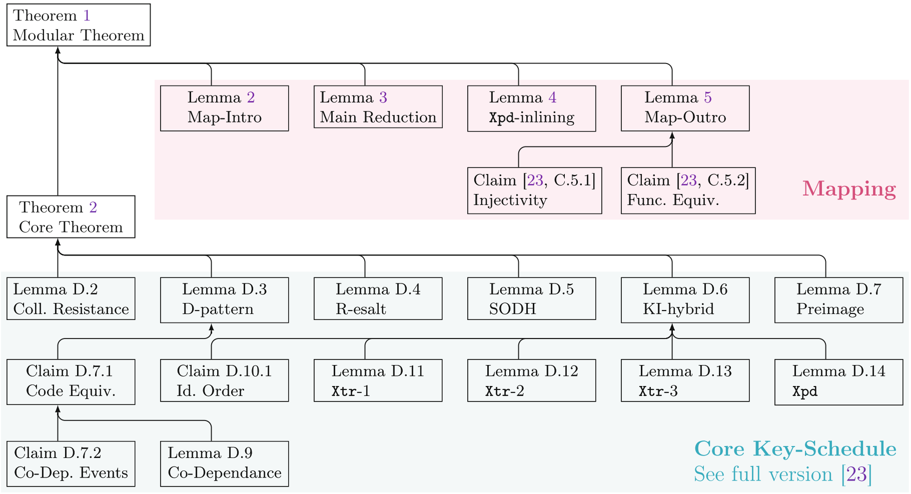


Fig. 13. Proof structure

5.1 Proof Technique

A recurrent proof technique which we use are *reductions*, written in SSP style. As usually, we want to show that if there is an adversary $\mathcal{A}$ which successfully distinguishes between two games $\mathtt{G}^0_{\mathrm{big}}$ and $\mathtt{G}^1_{\mathrm{big}}$, then based on $\mathcal{A}$, we can construct an adversary $\mathcal{B}$ of similar complexity as $\mathcal{A}$ which successfully distinguishes between two games $\mathtt{G}^0_{\mathrm{sml}}$ and $\mathtt{G}^1_{\mathrm{sml}}$. Our reductions will have the following form.

Lemma 1 (Reduction Technique). *If we can define a reduction $\mathcal{R}$ such that*

$$\mathtt{G}^0_{big} \overset{code}{\equiv} \mathcal{R} \to \mathtt{G}^0_{sml} \quad (1) \qquad \textit{and} \qquad \mathtt{G}^1_{big} \overset{code}{\equiv} \mathcal{R} \to \mathtt{G}^1_{sml} \quad (2)$$

then

$$\mathsf{Adv}(\mathcal{A}; \mathtt{G}^0_{big}, \mathtt{G}^1_{big}) = \mathsf{Adv}(\mathcal{B}; \mathtt{G}^0_{sml}, \mathtt{G}^1_{sml}), \tag{3}$$

where

$$\mathcal{B} := \mathcal{A} \to \mathcal{R}. \tag{4}$$

Proof. Assuming Eq. 1, 2 and 4, we deduce Eq. 3 as follows:

$$\begin{aligned}
&\mathsf{Adv}(\mathcal{A}, \mathtt{G}^0_{big}, \mathtt{G}^1_{big}) \\
&\overset{\text{def.}}{=} \left|\Pr\left[1 = \mathcal{A} \to \mathtt{G}^0_{big}\right] - \Pr\left[1 = \mathcal{A} \to \mathcal{R} \to \mathtt{G}^1_{big}\right]\right| \\
&\overset{\text{Eq.}1\&2}{=} \left|\Pr\left[1 = \mathcal{A} \to (\mathcal{R} \to \mathtt{G}^0_{sml})\right] - \Pr\left[1 = \mathcal{A} \to (\mathcal{R} \to \mathtt{G}^1_{sml})\right]\right| \\
&= \left|\Pr\left[1 = (\mathcal{A} \to \mathcal{R}) \to \mathtt{G}^0_{sml})\right] - \Pr\left[1 = (\mathcal{A} \to \mathcal{R}) \to \mathtt{G}^1_{sml}\right]\right| \\
&\overset{\text{def.}}{=} \mathsf{Adv}(\mathcal{A} \to \mathcal{R}, \mathtt{G}^0_{sml}, \mathtt{G}^1_{sml}) \overset{\text{Eq. }4}{=} \mathsf{Adv}(\mathcal{B}, \mathtt{G}^0_{sml}, \mathtt{G}^1_{sml})
\end{aligned}$$

```
Map

$\mathsf{SET}_{\mathrm{psk},0}(h, hon, k)$
    $h' \leftarrow \mathsf{SET}_{\mathrm{psk},0}(h, hon, k)$
    $M_{\mathrm{psk}}[h] \leftarrow h'$
    return $h$

$\mathsf{GET}_{n \in O^*,\ell}(h)$
    assert $M_{n,\ell}[h] \neq \bot$
    $h' \leftarrow M_{n,\ell}[h]$
    return $\mathsf{GET}_{n,\mathsf{level}(h')}(h')$
```

```
$\mathsf{XPD}_{n \in XPN,\ell}(h_1, r, args)$
    $i_1, \_ \leftarrow \mathsf{prntidx}(n, \ell)$
    assert $M_{i_1}[h_1] \neq \bot$
    $label \leftarrow \mathsf{label}(n, r)$
    $\ell_1 \leftarrow \mathsf{level}(M_{i_1}[h_1])$
    $h \leftarrow \mathsf{xpd}\langle n, label, h_1, args\rangle$
    $h' \leftarrow \mathsf{XPD}_{n,\ell_1}\begin{pmatrix} M_{i_1}[h_1], \\ r, args \end{pmatrix}$
    if $n = psk$ : $\ell \leftarrow \ell + 1$
    $M_{n,\ell}[h] \leftarrow h'$
    return $h$
```

```
$\mathsf{DHGEN}()$
    return $\mathsf{DHGEN}()$

$\mathsf{DHEXP}(X, Y)$
    $h \leftarrow \mathsf{dh}\langle \mathsf{sort}(X, Y)\rangle$
    $h' \leftarrow \mathsf{DHEXP}(X, Y)$
    if $M_{dh}[h] = \bot$ :
        $M_{dh}[h] \leftarrow h'$
    return $h$
```

```
$\mathsf{XTR}_{n \in \{es,hs,as\},\ell}(h_1, h_2)$
    $i_1, i_2 \leftarrow \mathsf{prntidx}(n, \ell)$
    assert $M_{i_1}[h_1] \neq \bot$
    assert $M_{i_2}[h_2] \neq \bot$
    $\ell' \overset{\text{choose}}{\leftarrow} \mathsf{level}(M_{i_1}[h_1]), \mathsf{level}(M_{i_2}[h_2])$
    $h \leftarrow \mathsf{xtr}\langle n, h_1, h_2\rangle$
    $h' \leftarrow \mathsf{XTR}_{n,\ell'}\begin{pmatrix} M_{i_1}[h_1], \\ M_{i_2}[h_2] \end{pmatrix}$
    $M_{n,\ell}[h] \leftarrow h'$
    return $h$
```

Fig. 14. Oracles of `Map`. Here, $\ell \in \{0 \ldots d\}$. $\ell' \overset{\text{choose}}{\leftarrow} \mathsf{level}(M_{n_1}[h_1]), \mathsf{level}(M_{n_2}[h_2])$ assigns to ℓ' the value $\mathsf{level}(M_{n_1}[h_1])$ if it is not $\bot$ and $\mathsf{level}(M_{n_2}[h_2])$, else.

Importantly, throughout this article, we define reductions graphically as composition of previously defined packages so that the reduction *re-uses* code, as opposed to the usual technique which introduces new code for a reduction. As a result, we can argue Eqs. 1 and 2 graphically. E.g., in [23, Fig. 31a] we highlight the reduction in gray and observe that the only change from Fig. 15a is the collision resistance assumption—the $\mathtt{G}^b_{\mathrm{sml}}$ in this case. Observing the graph of $\mathtt{Gks}^0$ (cf. Fig. 11a) closely and comparing it with the graphs of the assumptions introduced in Sect. 3, one can identify that the assumptions are almost sub-graphs of $\mathtt{Gks}^0$, and by an appropriately chosen sequence of reduction arguments, the graphs of the assumptions will appear as actual subgraphs.

5.2 Proof of Theorem 1

We need to show the indistinguishability of the real game $\mathtt{Gks}^0$ and the ideal game $\mathtt{Gks}^1(\mathcal{S})$. [23, Fig. 25a] depicts the real game $\mathtt{Gks}^0$ (cf. Fig. 11a), with slightly different graph layouting. [23, Fig. 26b] depicts the ideal game $\mathtt{Gks}^1(\mathcal{S})$ (cf. Fig. 11b) where the simulator $\mathcal{S}$ is described in concrete code. To show the indistinguishability between $\mathtt{Gks}^0$ ([23, Fig. 25a]) and $\mathtt{Gks}^1(\mathcal{S})$ ([23, Fig. 26b]), we make 4 *game hops*, depicted as the sequence of the five games depicted in [23, Fig. 25a], [23, Fig. 25b], [23, Fig. 25c], [23, Fig. 26a] and [23, Fig. 26b]. We now describe each of the game hops and state the corresponding lemma, see Fig. 13 for a proof overview.

First, recall that the key schedule security model stores keys in a redundant fashion (a) due to possible equal values of a dishonest resumption psk ($\mathsf{level}(h) > 0$) and an adversarially registered application psk ($\mathsf{level}(h) = 0$) and (b) due to the equal values of the (dishonest) DH keys corresponding to (X^a, Y) and (X, Y^a).

Lemma 2 introduces a `Map` package (see [23, Fig. 25b] for the game and the left column of Fig. 14 for the code of `Map`) to remove the redundantly stored keys—note that the $\mathtt{Log}^{A1}_{psk}$ and the $\mathtt{Log}^{Z\infty}_{dh}$ package now use the $map = 1$ and the $map = \infty$ code of `Log` (see Fig. 4 for its code). As a result, any adversary playing against $\mathtt{Gcore}^0$ (defined in [23, Fig. 25b]) cannot create (this particular) redundancy anymore since the $\texttt{Key}_{psk,\ell}$ and $\texttt{DHKey}_{dh}$ packages do not store the key again when the mapping code is triggered. We defer the proof of code equality proof of Lemma 2 to the full version [23]. It relies on proving the invariant that whenever $\texttt{Gks}^0$ stores key k with honesty hon under handle h, then game $\texttt{Gks}^{0\text{Map}}$ stores key k with honesty hon under the mapped handle $h' = M[h]$. The proof proceeds by induction over the oracle calls.

Lemma 2 (Map-Intro). *For all adversaries $\mathcal{A}$ which make queries for at most d resumption levels,*

$$\Pr\left[1 = \mathcal{A} \to \texttt{Gks}^0\right] = \Pr\left[1 = \mathcal{A} \to \texttt{Gks}^{0Map}\right].$$

In particular $\texttt{Gks}^0 \overset{func}{\equiv} \texttt{Gks}^{0Map}$*.*

Lemma 3 then reduces the indistinguishability of $\texttt{Gks}^{\texttt{0Map}}$ ([23, Fig. 25b]) and $\texttt{Gks}^{\texttt{1Map}}$ ([23, Fig. 25c]) to the indistinguishability of $\texttt{Gcore}^0$ and $\texttt{Gcore}^1(\mathcal{S}^{\text{core}})$ using reduction $\texttt{R}^{\text{core}}$. The indistinguishability of $\texttt{Gcore}^0$ and $\texttt{Gcore}^1(\mathcal{S}^{\text{core}})$ will be established in Theorem 2 in Appendix 5.3 and contains the main technical argument of this article.

Lemma 3 (Main). *For all PPT adversaries $\mathcal{A}$ which make queries for at most d resumption levels,*

$$\begin{aligned}&\mathsf{Adv}(\mathcal{A}, \texttt{Gks}^{0Map}, \texttt{Gks}^{1Map})\\ =&\mathsf{Adv}(\mathcal{A} \to \mathcal{R}^{ch\text{-}map}, \texttt{Gcore}^0, \texttt{Gcore}^1(\mathcal{S}^{core})),\end{aligned}$$

where [23, Fig. 25b] defines $\texttt{Gks}^{0Map}$, [23, Fig. 25c] defines $\texttt{Gks}^{1Map}$, $\mathcal{R}^{ch\text{-}map}$ and $\mathcal{S}^{core}$ are marked in grey in [23, Fig. 25c], and Fig. 15a and Fig. 15b define $\texttt{Gcore}^0$ *and* $\texttt{Gcore}^1(\mathcal{S}^{core})$*, respectively.*

Proof. The proof of Lemma 3 is an instance of Lemma 1 with $\texttt{G}^0_{\text{big}} = \texttt{Gks}^{0\text{Map}}$, $\texttt{G}^1_{\text{big}} = \texttt{Gks}^{1\text{Map}}$, $\texttt{G}^0_{\text{sml}} = \texttt{Gcore}^0$, $\texttt{G}^1_{\text{sml}} = \texttt{Gcore}^1(\mathcal{S}^{\text{core}})$ and $\mathcal{R} = \mathcal{R}^{\text{ch-map}}$.

By Lemma 1, it suffices to show that

$$\texttt{Gks}^{0\text{Map}} \overset{\text{code}}{\equiv} \mathcal{R}^{\text{ch-map}} \to \texttt{Gcore}^0 \tag{5}$$

$$\texttt{Gks}^{1\text{Map}} \overset{\text{code}}{\equiv} \mathcal{R}^{\text{ch-map}} \to \texttt{Gcore}^1(\mathcal{S}^{\text{core}}) \tag{6}$$

Equation 5 follows by definition, since [23, Fig. 25b] defines $\texttt{Gks}^{0\text{Map}}$ as the composition of $\mathcal{R}^{\text{ch-map}}$ and $\texttt{Gcore}^0$. Similarly, for Eq. 6, [23, Fig. 25c]

In Lemma 4, we inline the $\texttt{Xpd}_{n,0..d}$ code into $\texttt{Map}$ for $n \in O^*$ and call the result $\texttt{Map-Xpd}$ (see [23, Fig. 25c] and [23, Fig. 26a] for the two games). The proof is a simple inlining argument and included into the full version [23] for completeness.

Lemma 4 (Xpd-Inlining). *For all PPT adversaries $\mathcal{A}$ which make queries for at most d resumption levels,*

$$\Pr\left[1 = \mathcal{A} \to \texttt{Gks}^{1Map}\right] = \Pr\left[1 = \mathcal{A} \to \texttt{Gks}^{Mapxpd}\right].$$

In particular $\texttt{Gks}^{1Map} \overset{code}{\equiv} \texttt{Gks}^{Mapxpd}$*.*

Finally, Lemma 5 establishes the (perfect) indistinguishability of $\mathtt{Gks}^{\mathtt{Map\text{-}Xpd}}$ and $\mathtt{Gks}^1(\mathcal{S})$. The proof of Lemma 5, essentially, removes or rather *inverts* the mapping on the output keys in order to recover the ideal functionality. Inverting the handle mapping, however, requires that it is *injective*. Conceptually, it is also clear that injectivity of the handle mapping needs to play a role in the proof: We prove uniqueness of output keys which means that equal keys imply equal handles. The injectivity proof ensures that the mapping did not introduce additional collisions and that the proof of Theorem 2 indeed suffices to establish the uniqueness of output keys in $\mathtt{Gks}^1(\mathcal{S})$.

Lemma 5 (Map-Outro). *For all PPT adversaries $\mathcal{A}$ which make queries for at most d resumption levels,*

$$\Pr\left[1 = \mathcal{A} \to \mathtt{Gks}^{Mapxpd}\right] = \Pr\left[1 = \mathcal{A} \to \mathtt{Gks}^1(\mathcal{S})\right].$$

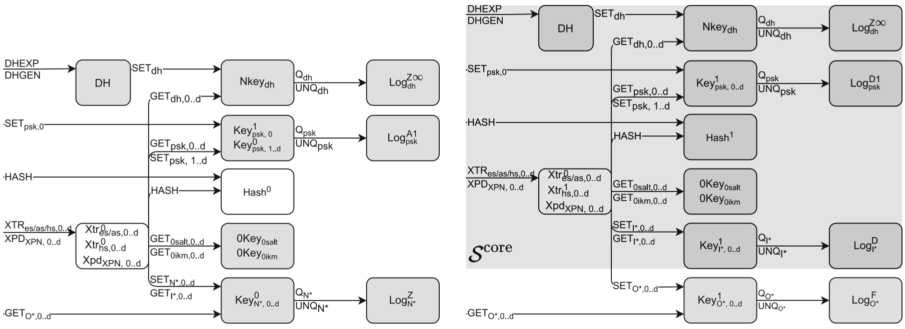
In particular, $\mathtt{Gks}^{Mapxpd} \overset{func}{\equiv} \mathtt{Gks}^1(\mathcal{S})$.

In summary, Lemma 3 is the core argument, Lemma 2 is proven via a mechanical invariant proof, Lemma 5 is proven via a conceptually interesting invariant proof and Lemma 4 is a straightforward inlining argument.

Theorem 1 directly follows from Lemma 2–Lemma 5 and Theorem 2 (stated in Sect. 5.3).

$$\begin{aligned}
\mathsf{Adv}(\mathcal{A}, \mathtt{Gks}^0, \mathtt{Gks}^1(\mathcal{S})) &\overset{\text{Lm. } 2}{=} \mathsf{Adv}(\mathcal{A}, \mathtt{Gks}^{0\text{Map}}, \mathtt{Gks}^1(\mathcal{S}))\\
&\overset{\text{Lm. } 5}{=} \mathsf{Adv}(\mathcal{A}, \mathtt{Gks}^{0\text{Map}}, \mathtt{Gks}^{\text{Mapxpd}})\\
&\overset{\text{Lm. } 4}{=} \mathsf{Adv}(\mathcal{A}, \mathtt{Gks}^{0\text{Map}}, \mathtt{Gks}^{1\text{Map}})
\end{aligned}$$

$$\begin{aligned}
&\overset{\text{Lm. } 3}{=} \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^{\text{ch-map}}, \mathtt{Gks}^{0\text{core}}, \mathtt{Gks}^{1\text{core}}(\mathcal{S}^{\text{core}}))\\
&\overset{\text{Th. } 2}{\leq} \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^{\text{main}}_{\text{cr}}, \mathtt{Gacr}^{\mathsf{hash},b})\\
&\qquad + \sum_{j\in\{Z,D\},\mathsf{f}\in\{\mathsf{xtr},\mathsf{xpd}\}} \mathsf{Adv}(\mathcal{A} \to \mathcal{R}^{\text{main}}_{j,\mathsf{f}}, \mathtt{Gacr}^{\mathsf{hash},b})\\
&\qquad + \max_{i\in\{0,1\}} \mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}^{\text{main}}_{\text{sodh}}, \mathtt{Gsodh}^b)\\
&\qquad + \mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}^{\text{main}}_{esalt,pi}, \mathtt{Gpi}^b_{esalt})\\
&\qquad + \mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}^{\text{main}}_{O^*,pi}, \mathtt{Gpi}^b_{O^*})\\
&\qquad + \sum_{\ell=0}^{d-1} \Big(\mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}^{\text{main}}_{\text{es},\ell}, \mathtt{Gxtr}^b_{es,\ell})\\
&\qquad\qquad + \mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}^{\text{main}}_{\text{hs},\ell}, \mathtt{Gxtr}^b_{hs,\ell})\\
&\qquad\qquad + \mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}^{\text{main}}_{\text{as}}, \mathtt{Gxtr}^b_{as,\ell})\\
&\qquad\qquad + \sum_{n\in XPR} \big(\mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}^{\text{main}}_{\text{n},\ell}, \mathtt{Gxpd}^b_{n,\ell})\big)\Big),
\end{aligned}$$



(a) Game $\texttt{Gcore}^0$.

(b) Game $\texttt{Gcore}^1(\mathcal{S}^{\text{core}})$, $\mathcal{S}^{\text{core}}$ in grey.

Fig. 15. Games for Theorem 2

where *XPR* is the representative set required by the theorem, $\mathcal{R}^{\text{main}}_* := \mathcal{R}^{\text{ch-map}} \to \mathcal{R}_*$ when replacing $*$ by cr, (Z, f), (D, f) sodh, es, hs, as, n, O^*, pi or $esalt, pi$.

5.3 Core Key Schedule Theorem

It remains to show that the *core* key schedule game $\texttt{Gcore}^0$ without the $\texttt{Map}$ and $\texttt{Check}$ package in front (Fig. 15a is indistinguishable from an ideal game $\texttt{Gcore}^1(\mathcal{S}^{\text{core}})$ which consists of an ideal functionality with a simulator $\mathcal{S}^{\text{core}}$ (Fig. 15b). The proof of Theorem 2 can be found in the full version [23, Appendix D]

Theorem 2 (Core). *Let ks be a TLS-like key schedule with XPR. Let d be an integer. Let $\mathcal{S}^{core}$ be the efficient simulator defined in [23, Fig. 26b]. Then, for all adversaries $\mathcal{A}$ which make queries for at most d resumption levels, we have that*

$$\begin{aligned}
&\mathsf{Adv}(\mathcal{A}, \texttt{Gcore}^0, \texttt{Gcore}^1(\mathcal{S}^{core}))\\
\leq &\sum_{\mathcal{R}\in\{\mathcal{R}_{cr},\mathcal{R}_Z,\mathcal{R}_D\}} \mathsf{Adv}(\mathcal{A}\to\mathcal{R}, \texttt{Gacr}^b)\\
&+ \max_{i\in\{0,1\}} \mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}_{sodh}, \texttt{Gsodh}^b)\\
&+ \mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}_{esalt,pi}, \texttt{Gpi}^b_{esalt})\\
&+ \mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}_{O^*,pi}, \texttt{Gpi}^b_{O^*})\\
&+ \sum_{\ell=0}^{d-1} \big(\mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}_{es,\ell}, \texttt{Gxtr}^b_{es,\ell})\\
&\qquad + \mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}_{hs,\ell}, \texttt{Gxtr}^b_{hs,\ell})\\
&\qquad + \mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}_{as}, \texttt{Gxtr}^b_{as,\ell})\\
&\qquad + \sum_{n\in XPR} \big(\mathsf{Adv}(\mathcal{A}_i \to \mathcal{R}_{n,\ell}, \texttt{Gxpd}^b_{n,\ell})\big)\big),
\end{aligned}$$

where XPR is the required representation set (cf. Table 1), Fig. 15a defines $\texttt{Gcore}^0$ *and Fig. 15b defines* $\texttt{Gcore}^1(\mathcal{S}^{core})$*, [23, Fig. 31a] defines* $\mathcal{R}_{cr}$*, [23, Fig. 32a] defines* $\mathcal{R}_{sodh}$*, [23, Fig. 34a] defines* $\mathcal{R}_{es,\ell}$*,* $\mathcal{R}_{hs,\ell}$ *and* $\mathcal{R}_{as,\ell}$ *are defined analogously, and* $\mathcal{R}_{n,\ell}$ *for* $n \in XPR$ *and* $0 \leq \ell \leq d$ *is defined in [23, Fig. 34b],* $\mathcal{R}_{esalt,pi}$ *is defined in [23, Fig. 32c] and* $\mathcal{R}_{O^*,pi}$ *is defined in [23, Fig. 32d].*

6 Related Work

The following discussion focuses on attacker capabilities and security guarantees, and glosses over the exact encoding into security games and the use of multiple keys and stages.

Dowling et al. [34–36] present a multi-stage security model of `draft-05`, `draft-10`, and the final version of the standard. Their multi-stage model considers `psk_ke`, `dh_ke`, and `psk_dhe_ke` modes in isolation. Li et al. [48] adapt the multi-stage security model to also capture the recursive nature of the TLS 1.3 key schedule, by accounting for the re-use of resumption secrets between different modes (`psk_ke`, `psk_dhe_ke`, and the now removed semi-static share 0-RTT).

Cremers et al. [29,30] investigate the security of `draft-10` and `draft-21`, using the automated Tamarin prover (in the symbolic model). Their work investigates the proposed post-handshake client authentication and finds an attack that exploited a missing binding between PSKs and transcripts that led to the addition of binders to the standard. To our knowledge ours is the first reduction proof that models the additional security afforded by binder values.

Bhargavan et al. [10] also model TLS 1.3, decomposed into 3 separate pieces: `dh_ke` 1-RTT handshake, the 0-RTT handshake, and the record protocol. They verify these models using both ProVerif [18] and CryptoVerif [16]. A limitation of their model is the informal way in which the separate guarantees for the three components are combined to justify the overall security of the protocol.

Blanchet [17] introduces a new proof modularization framework in CryptoVerif, which bears significant similarities with the state-separating proof framework [24] that our work is based on. The work also updates some of the model from `draft-18` to `draft-28`; however, the model still assumes that all pre-shared keys are derived from resumption secrets and does not capture adaptively-created dishonest application PSKs, or the security of PSK binders.

Many other works focus on analysing certain properties of the TLS 1.3 handshake protocol. For instance, Arfaou et al. [4] specifically analyse the privacy of the TLS 1.3 `psk_ke`, `dh_ke`, and `psk_dhe_ke` handshakes. Fischlin et. al. [41] analyse the `draft-06` TLS 1.3 handshake, and show that its modes achieve key confirmation in isolation. Fischlin et. al. [39] considers replay attacks against various drafts of TLS 1.3 0-RTT handshakes such as `draft-14`'s `psk_ke` mode, similarly considering versions and modes in isolation. Other relevant papers on TLS handshake analysis are [27,37,46].

The idea of analyzing a key schedule (rather than a key exchange protocol) is conceptually similar to the SIGMA-I pattern of Krawczyk [44] and Krawczyk and Wee [47]. These works prove a reduction from key exchange security to key schedule security analogously to our companion paper [25].

Recent work also looked at the tightness of TLS 1.3 security proofs [31,33]. Besides natural birthday bounds for collision resistance, our reductions avoid the common quadratic loss in the number of sessions. We remark however, that tightness was not the principal focus of our analysis.

Subsequent work to the present article [22] uses our methodology, e.g., our recursive handle structure and the style of encoding security guarantees in `Log` packages to analyse the key schedule security of the Messaging Layer Security (MLS) protocol whose conclusions were integrated into the IETF standard, e.g., [28]. In the present paper, in addition to key techniques which were picked up by [22], we introduce a plethora of techniques to tackle *indirect* domain separation by late hashing of Diffie-Hellman shares and binders such as the notion of separation points and the `Check` component introduced in Sect. 4.3. In a similar way, the additional mapping step (Lemma 2, 4 and 5) handle redundancy not present in MLS. See Sect. 7 for simplifications of the TLS protocol which would allow for a much simpler analysis than the one presented in this article.

7 Lessons Learned and Afterthoughts on the Key Schedule

We now discuss changes to the key schedule that would improve its security and simplify its analysis and may be of independent interest for other protocols.

Simplify SODH. The salted Diffie-Hellman computation extracts entropy from the DH secret and mixes it with the PSK-derived salt (which is under adversarial influence). A separate DH extraction, preferably hashing the (sorted) public shares together with the secret, followed by a dual PRF, would enable a proof based on the simpler and better understood Oracle Diffie-Hellman assumption. The hashing of shares would also remove the need to map DH secrets (currently computable from multiple pairs of shares), and would enable the use of a more abstract functionality such as a CCA-secure KEM (as in TLS 1.2 [14]). These changes would thus also ease the integration of post-quantum secure primitives.

Eliminate PSK Mapping. Similarly, *directly* applying domain-separation for computations based on application and resumption PSKs via distinct labels would remove the need to map PSKs and argue via inclusion of binders at separation points indirectly. Both proposals follow the same design pattern: first sanitize input key materials to prevent malleability (DH secrets) and collisions (dishonest resumption PSKs and adversarially-chosen application PSKs).

Avoid Agile Assumptions. Our development supports multiple hash algorithms without requiring any hash-agile assumptions, by observing that the hash functions currently used by TLS 1.3 have pairwise-distinct digest lengths. This is brittle, e.g. adding support for SHA3 with the same lengths as SHA2 would require to formally account for cross-algorithm collisions. This may be prevented by tagging the outputs of all extractors and KDFs with hash algorithms. Similarly, we may avoid the current need for agile (S)ODH assumptions by tagging group elements with both a group descriptor and a single extraction algorithm.

Prevent PSK Reflections. Drucker and Gueron note that TLS 1.3 is subject to reflection attacks due to its symmetric use of PSKs [37]. Hence, in our model, the same PSK handle may either be used by two parties, as intended, or by the same party acting both as a client and as a server. This is a security risk, inasmuch as applications may embed identity information in PSK identifiers to benefit from their early authentication. It may also enable key synchronization attacks and other variants of key compromise impersonation [13] when identities are also symmetrical. When using PSKs, the standard unfortunately forbids certificate-based authentication, which would otherwise provide more detailed, role-specific identity information. At the key schedule level, it may be possible to enforce better separation by tagging PSK identifiers with roles.

Enforce Stronger Modularity. Applied cryptographers often complain that, in TLS 1.2, the subtle interleaving of the handshake with the record layer hinders its analysis based on the well-established Bellare-Rogaway [8] security model [43]. While TLS 1.3 tries to enforce cleaner separation between handshake and record keys, it still fails in some important places. Notably, the handshake traffic secrets, meant to be released to the record layer (be it TLS, DTLS, or QUIC) are also used by the handshake to derive finished keys. Similarly, some handshake messages are encrypted under keys derived from application traffic secrets (e.g. New Session Ticket, carrying resumption PSKs, late client authentication, and key updates). This complicates the modeling of data stream security, as application data may be interleaved with handshake messages (e.g. the same QUIC packet may contain both data and session tickets). To prevent such issues, and many others, we suggest the RFC documents more explicitly its application interface and, in particular, recommends not to derive keys from keys released to the record layer.

References

1. Abdalla, M., Bellare, M., Rogaway, P.: The oracle diffie-hellman assumptions and an analysis of DHIES. In: Naccache, D. (ed.) CT-RSA 2001. LNCS, vol. 2020, pp. 143–158. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45353-9_12
2. Adrian, D., et al.: Imperfect forward secrecy: how Diffie-Hellman fails in practice. In: ACM CCS 2015, pp. 5–17. ACM Press (2015). https://doi.org/10.1145/2810103.2813707
3. AlFardan, N.J., Paterson, K.G.: Lucky thirteen: breaking the TLS and DTLS record protocols. In: 2013 S&P, pp. 526–540. IEEE (2013). https://doi.org/10.1109/SP.2013.42
4. Arfaoui, G., Bultel, X., Fouque, P.A., Nedelcu, A., Onete, C.: The privacy of the TLS 1.3 protocol. PoPETs **2019**(4), 190–210 (2019). https://doi.org/10.2478/popets-2019-0065

5. Aviram, N., et al.: DROWN: breaking TLS using SSLv2. In: USENIX Security 2016, pp. 689–706. USENIX (2016)
6. Badertscher, C., Matt, C., Maurer, U., Rogaway, P., Tackmann, B.: Augmented secure channels and the goal of the TLS 1.3 record layer. In: Au, M.-H., Miyaji, A. (eds.) ProvSec 2015. LNCS, vol. 9451, pp. 85–104. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26059-4_5
7. Bellare, M.: New proofs for NMAC and HMAC: security without collision resistance. J. Cryptol. **28**(4), 844–878 (2014). https://doi.org/10.1007/s00145-014-9185-x
8. Bellare, M., Rogaway, P.: Entity authentication and key distribution. In: Stinson, D.R. (ed.) CRYPTO 1993. LNCS, vol. 773, pp. 232–249. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-48329-2_21
9. Beurdouche, B., et al.: A messy state of the union: taming the composite state machines of TLS. In: 2015 S&P, pp. 535–552. IEEE (2015). https://doi.org/10.1109/SP.2015.39
10. Bhargavan, K., Blanchet, B., Kobeissi, N.: Verified models and reference implementations for the TLS 1.3 standard candidate. In: 2017 S&P, pp. 483–502. IEEE (2017). https://doi.org/10.1109/SP.2017.26
11. Bhargavan, K., Brzuska, C., Fournet, C., Green, M., Kohlweiss, M., Zanella-Béguelin, S.: Downgrade resilience in key-exchange protocols. In: 2016 S&P, pp. 506–525. IEEE (2016). https://doi.org/10.1109/SP.2016.37
12. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Pironti, A., Strub, P.Y.: Triple handshakes and cookie cutters: Breaking and fixing authentication over tls. In: IEEE Symposium on Security & Privacy (Oakland) (2014). `pubs/triple-handshakes-and-cookie-cutters-sp14.pdf`
13. Bhargavan, K., Delignat-Lavaud, A., Pironti, A.: Verified contributive channel bindings for compound authentication. In: NDSS 2015. ISOC (2015)
14. Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.-Y., Zanella-Béguelin, S.: Proving the TLS handshake secure (as it is). In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 235–255. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44381-1_14
15. Bhargavan, K., Leurent, G.: Transcript collision attacks: breaking authentication in TLS, IKE and SSH. In: NDSS 2016. ISOC (2016)
16. Blanchet, B.: CryptoVerif: computationally sound mechanized prover for cryptographic protocols. In: Formal Protocol Verification, vol. 117, p. 156 (2007)
17. Blanchet, B.: Composition theorems for CryptoVerif and application to TLS 1.3. In: CSF, pp. 16–30 (2018). https://doi.org/10.1109/CSF.2018.00009
18. Blanchet, B., Smyth, B., Cheval, V., Sylvestre, M.: ProVerif 2.00: automatic cryptographic protocol verifier. User Manual (2018)
19. Böck, H., Somorovsky, J., Young, C.: Return of bleichenbacher's oracle threat (ROBOT). In: USENIX Security 2018, pp. 817–849. USENIX (2018)
20. Brendel, J., Fischlin, M., Günther, F., Janson, C.: PRF-ODH: relations, instantiations, and impossibility results. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10403, pp. 651–681. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63697-9_22
21. Bricout, R., Murphy, S., Paterson, K.G., van der Merwe, T.: Analysing and exploiting the mantin biases in RC4. Cryptology ePrint Archive, Report 2016/063 (2016). http://eprint.iacr.org/2016/063
22. Brzuska, C., Cornelissen, E., Kohbrok, K.: Security analysis of the mls key derivation. In: 2022 IEEE Symposium on Security and Privacy, pp. 595–613. IEEE Computer Society, Los Alamitos (2022). https://doi.org/10.1109/SP46214.2022.00035

23. Brzuska, C., Delignat-Lavaud, A., Egger, C., Fournet, C., Kohbrok, K., Kohlweiss, M.: Key-schedule security for the TLS 1.3 standard. Cryptology ePrint Archive, Report 2021/467 (2021). https://eprint.iacr.org/2021/467
24. Brzuska, C., Delignat-Lavaud, A., Fournet, C., Kohbrok, K., Kohlweiss, M.: State separation for code-based game-playing proofs. In: ASIACRYPT 2018, Part III. LNCS, vol. 11274, pp. 222–249. Springer, Heidelberg (Dec 2018). https://doi.org/10.1007/978-3-030-03332-3_9
25. Brzuska, C., Egger, C.: Key exchange to key schedule reduction for TLS 1.3 (2022). preprint
26. Canetti, R., Krawczyk, H.: Universally composable notions of key exchange and secure channels. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 337–351. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46035-7_22
27. Chen, S., Jero, S., Jagielski, M., Boldyreva, A., Nita-Rotaru, C.: Secure communication channel establishment: TLS 1.3 (over TCP Fast Open) vs. QUIC. In: Sako, K., Schneider, S., Ryan, P.Y.A. (eds.) ESORICS 2019. LNCS, vol. 11735, pp. 404–426. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29959-0_20
28. Cornelissen, E.: Pull request 453: Use the GroupContext to derive the joiner_secret. https://github.com/mlswg/mls-protocol/pull/453
29. Cremers, C., Horvat, M., Hoyland, J., Scott, S., van der Merwe, T.: A comprehensive symbolic analysis of TLS 1.3. In: ACM CCS 2017, pp. 1773–1788. ACM Press (2017)
30. Cremers, C., Horvat, M., Scott, S., van der Merwe, T.: Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In: 2016 S&P, pp. 470–485. IEEE (2016). https://doi.org/10.1109/SP.2016.35
31. Davis, H., Günther, F.: Tighter proofs for the SIGMA and TLS 1.3 key exchange protocols. In: Sako, K., Tippenhauer, N.O. (eds.) ACNS 2021. LNCS, vol. 12727, pp. 448–479. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78375-4_18
32. Delignat-Lavaud, A., et al.: Implementing and proving the TLS 1.3 record layer. In: IEEE Security & Privacy. IEEE (2017)
33. Diemert, D., Jager, T.: On the tight security of TLS 1.3: theoretically sound cryptographic parameters for real-world deployments. J. Cryptol. **34**(3), 1–57 (2021). https://doi.org/10.1007/s00145-021-09388-x
34. Dowling, B., Fischlin, M., Günther, F., Stebila, D.: A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In: ACM CCS 2015, pp. 1197–1210. ACM Press (2015). https://doi.org/10.1145/2810103.2813653
35. Dowling, B., Fischlin, M., Günther, F., Stebila, D.: A cryptographic analysis of the TLS 1.3 draft-10 full and pre-shared key handshake protocol. Cryptology ePrint Archive, Report 2016/081 (2016). http://eprint.iacr.org/2016/081
36. Dowling, B., Fischlin, M., Günther, F., Stebila, D.: A cryptographic analysis of the TLS 1.3 handshake protocol. J. Cryptol. **34**(4), 1–69 (2021). https://doi.org/10.1007/s00145-021-09384-1
37. Drucker, N., Gueron, S.: Selfie: reflections on TLS 1.3 with PSK. J. Cryptol. **34**(3), 1–18 (2021). https://doi.org/10.1007/s00145-021-09387-y
38. Duong, T., Rizzo, J.: Here come the ⊕ ninjas (2011). http://nerdoholic.org/uploads/dergln/beast_part2/ssl_jun21.pdf
39. Fischlin, M., Günther, F.: Replay attacks on zero round-trip time: the case of the TLS 1.3 handshake candidates. In: 2017 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 60–75. IEEE (2017)

40. Fischlin, M., Günther, F., Marson, G.A., Paterson, K.G.: Data is a stream: security of stream-based channels. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 545–564. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48000-7_27
41. Fischlin, M., Günther, F., Schmidt, B., Warinschi, B.: Key confirmation in key exchange: a formal treatment and implications for TLS 1.3. In: 2016 S&P, pp. 452–469. IEEE (2016). https://doi.org/10.1109/SP.2016.34
42. Iyengar, J., Thomson, M.: QUIC. IETF draft (2019)
43. Jager, T., Kohlar, F., Schäge, S., Schwenk, J.: Authenticated confidential channel establishment and the security of TLS-DHE. J. Cryptol. **30**(4), 1276–1324 (2017). https://doi.org/10.1007/s00145-016-9248-2
44. Krawczyk, H.: SIGMA: the "SIGn-and-MAc" approach to authenticated diffie-hellman and its use in the IKE protocols. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 400–425. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45146-4_24
45. Krawczyk, H.: Cryptographic extraction and key derivation: the HKDF scheme. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 631–648. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14623-7_34
46. Krawczyk, H.: A unilateral-to-mutual authentication compiler for key exchange (with applications to client authentication in TLS 1.3). In: ACM CCS 2016, pp. 1438–1450. ACM Press (2016). https://doi.org/10.1145/2976749.2978325
47. Krawczyk, H., Wee, H.: The OPTLS protocol and TLS 1.3. Cryptology ePrint Archive, Report 2015/978 (2015). http://eprint.iacr.org/2015/978
48. Li, X., Xu, J., Zhang, Z., Feng, D., Hu, H.: Multiple handshakes security of TLS 1.3 candidates. In: 2016 S&P, pp. 486–505. IEEE (2016). https://doi.org/10.1109/SP.2016.36
49. Mavrogiannopoulos, N., Vercauteren, F., Velichkov, V., Preneel, B.: A cross-protocol attack on the TLS protocol. In: ACM CCS 2012, pp. 62–72. ACM Press (2012). https://doi.org/10.1145/2382196.2382206
50. Möller, B., Duong, T., Kotowicz, K.: This poodle bites: exploiting the SSL 3.0 fallback (2014). https://www.openssl.org/~bodo/ssl-poodle.pdf
51. Paterson, K.G., van der Merwe, T.: Reactive and proactive standardisation of TLS. In: Security Standardisation Research, pp. 160–186 (2016)
52. Patton, C., Shrimpton, T.: Partially specified channels: the TLS 1.3 record layer without elision. Cryptology ePrint Archive, Report 2018/634 (2018)
53. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3 (2018). https://tools.ietf.org/html/rfc8446