Handshake Privacy for TLS 1.3 - Technical report

Karthikeyan Bhargavan[1], Vincent Cheval[1], and Christopher Wood[2]

[1]Inria Paris
[2]Cloudflare

September 1, 2022

Abstract

TLS 1.3, the newest version of the Transport Layer Security (TLS) protocol, provides stronger authentication and confidentiality guarantees than prior TLS version. Despite additional encryption of handshake messages, some parts of the TLS 1.3 handshake, including the `ClientHello`, are still in the clear. For example, the protocol reveals the identity of the target server to network attackers, allowing the passive surveillance and active censorship of TLS connections. A recent privacy extension called Encrypted Client Hello (ECH, previously called ESNI) addresses this problem and offers more comprehensive handshake encryption and privacy for TLS 1.3. Surprisingly however, although the security of the TLS 1.3 handshake has been comprehensively analyzed in a variety of formal models, the privacy guarantees of handshake encryption have never been formally studied. This gap has resulted in several mis-steps: several of the initial designs for ECH were found to be vulnerable to passive and active network attacks.

In this paper, we present the first mechanized formal analysis of privacy properties for the TLS 1.3 handshake. We study all standard modes of TLS 1.3, with and without ECH, using the symbolic protocol analyzer ProVerif. We discuss attacks on ECH, some found during the course of this study, and show how they are accounted for in the latest version. Our analysis has helped guide the standardization process for ECH and we provide concrete privacy recommendations for TLS implementors. We also contribute the most comprehensive model of TLS 1.3 to date, which can be used by designers experimenting with new extensions to the protocol. Ours is one of the largest privacy proofs attempted in ProVerif and our modeling strategies may be of general interest to protocol analysts.

1 Introduction

The Transport Layer Security (TLS) protocol is a widely-deployed Internet standard for establishing secure channels across untrusted networks. Notably, it is used for HTTPS connections between web browsers and servers. A typical deployment scenario is depicted in Figure 1: a web browser A connects to a website S_1 across the Internet. In practice, S_1 is often run by a hosting service or content-delivery network that provides a shared client-facing server (F) for multiple websites. The goal of TLS is to provide a secure channel between A and S_1, which is often instantiated as a secure channel between A and F operating on behalf of S_1, even if the attacker fully controls the network, other clients like B, and other servers like S_2. I rewrote this bit – please review

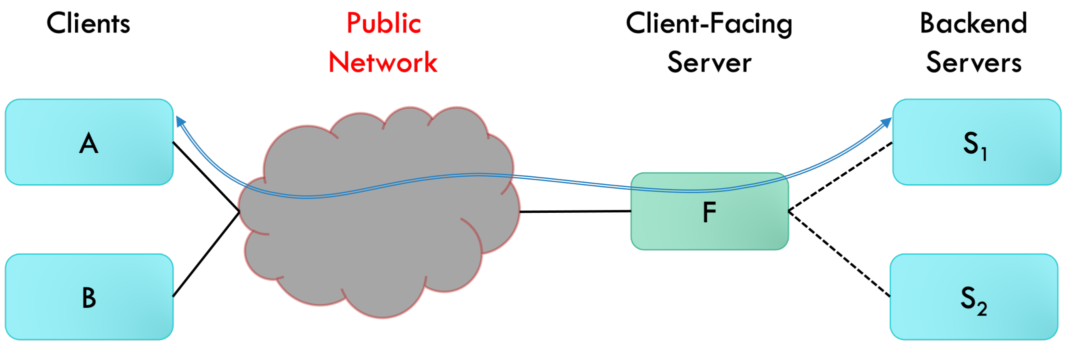


Figure 1: Example TLS 1.3 Deployment Scenario: two TLS clients A and B (e.g. web browsers) connect via a public network to TLS servers S_1 and S_2 (e.g. websites) that are both hosted by the same client-facing server F (e.g. a content delivery network).

In 2018, the Internet Engineering Task Force (IETF) standardized TLS 1.3 [Res18], which improves upon the previous version (TLS 1.2) in several significant ways. In terms of efficiency, TLS 1.3 reduces the latency of connection setup (called the *handshake*) from two round-trips to one round-trip before the client can start sending application data. In terms of security, the protocol provides forward secrecy as default, deprecates obsolete cryptographic constructions [BL16b, BL16a, VP15, MDK14, AP13], and improves transcript authentication to prevent downgrade and key synchronization attacks [ASS+16, BDF+14, ABD+15, BBDL+15].

Perhaps most remarkably, TLS 1.3 was designed in collaboration with the academic research community with the explicit goal of having formal security proofs for the protocol *before* standardization. Consequently, a series of publications have analyzed the security of TLS 1.3 using a variety of proof techniques and formal definitions. A real-world protocol standard like TLS supports dozens of protocol flows depending on how the client and server are configured and analyzing all these flows by hand can be tedious. As a result, pen-and-paper cryptographic proofs tend to focus on a small subset of features supported by the protocol [DFGS15, KW16, LXZ+16, BBF+16, BFG19, FG17, KMO+15]. More comprehensive analyses of TLS 1.3 rely on semi-automated verification tools like Tamarin [CHSvdM16, CHH+17], ProVerif [BBK17], CryptoVerif [BBK17], and F* [DFK+17]. All of these works focus on the three primary secure channel goals of TLS; no prior work, with the notable exception of [ABF+19], studies the *privacy* guarantees of TLS 1.3.

Privacy Goals for TLS. Many secure channel protocols seek to protect the identity of one or both participants from passive or active network adversaries. The SIGMA family of key exchange protocols [Kra03], which influenced both IKEv2 [KHN+14] and TLS, includes two identity hiding variants. The more modern Noise protocol framework offers dozens of protocols with multiple levels of identity hiding [Per18], one of which is used in WireGuard [Don17].

In the TLS scenario of Figure 1, we can informally state two privacy goals:

- *Client identity privacy*: the attacker cannot distinguish between a connection from A to S_1 and one from B to S_1;
- *Server identity privacy*: the attacker cannot distinguish between a connection from A to S_1 and one from A to S_2.

In addition to identities, we may also be interested in protecting other handshake metadata and sensitive protocol extensions. We can then ask whether these privacy guarantees hold for arbitrary sequences of connections between honest participants, in the presence of passive or active network adversaries.

Before TLS 1.2, the identities of the client and server were sent in the clear and hence all these privacy goals were trivially false. However, in TLS 1.3, most of the handshake messages, including those containing client and server certificates, are encrypted, in order to provide better privacy guarantees against pervasive surveillance [BSJ$^+$15].

Despite this improvement, TLS 1.3 still leaks the server identity in plaintext. In the TLS scenario of Figure 1, clients indicate which service S_i they wish to open a channel with in their first message to the client-facing server, and this message – the `ClientHello` – is sent in the clear. In particular, the `ClientHello` contains the name of the server (S_i) in the clear, and this name is used to allow the client-facing server (F) to route the connection to the right server. Even though clients are usually unauthenticated, client identity privacy can be subtle, since a network adversary may be able to correlate different connections made by the same client to the same server, if these connections use the same pre-shared key.

To improve the privacy of TLS 1.3, the IETF TLS working group is working on standardizing a new protocol extension called Encrypted Client Hello (ECH) [ROSW21], which was previously called Encrypted Server Name Indication (ESNI). The key idea of ECH is to encrypt parts of the first handshake message (the `ClientHello`) in order to protect the server name, pre-shared key, and other handshake metadata.

Formal Privacy for TLS+ECH. One of the main goals of ECH is to provide server identity privacy against active network attackers. However many early designs for ECH proved to be vulnerable to subtle attacks, some of which we shall detail later in this report. The TLS working group invited researchers to formally analyze the ECH design before standardization and this work is the first to provide such an analysis.

ECH is the first major extension to TLS 1.3 since it was standardized. It introduces new cryptographic constructions to the handshake and subtly changes the meaning of the handshake transcript. Given that TLS 1.3 has seen such extensive security analysis, it is a valid concern that the changes introduced by ECH may break the security guarantees of TLS 1.3.

To safely extend TLS 1.3 with ECH, we propose a three-step analysis methodology, which we believe should be followed by all future security extensions for TLS 1.3:

1. **Security Preservation**: Prove that the extended protocol preserves the authentication, confidentiality, and integrity properties that have been previously proved for TLS 1.3;

2. **Stronger Guarantees:** Define new security goals for TLS, and prove that vanilla TLS 1.3 does not satisfy these goals while the extended protocol does;

3. **Downgrade Resistance:** Prove that an active network attacker cannot downgrade the extended protocol to vanilla TLS 1.3, hence removing the extended guarantees.

Furthermore, we advocate that each of these analyses should be applied to a model of TLS 1.3 that covers as many of its optional modes as possible, to ensure that we do not miss attacks lurking in less-understood corners of the protocol.

Contributions. We begin with the ProVerif model of TLS 1.3 in [BBK17] and make it more precise and more comprehensive: we carefully model all the configuration options and resulting protocol branches, we add support for `HelloRetryRequest` and post-handshake authentication, and we define stronger confidentiality and integrity goals. We believe that the result of these extensions is the most comprehensive formal model of TLS 1.3 to date, which may be of independent interest to protocol designers and analysts. We prove, using ProVerif, that this model satisfies the classic secure channel goals of TLS 1.3. We note that ProVerif is a symbolic prover based on the Dolev-Yao model [DY06], so proofs in ProVerif are not directly comparable with cryptographic pen-and-paper proofs.

Next, we formally define a series of privacy goals for TLS 1.3, including client identity privacy (for the certificate-based client authentication mode), client unlinkability (for the pre-shared key resumption mode), server identity privacy, and metadata privacy for additional extensions sent by the client and server. We prove that vanilla TLS 1.3 satisfies some of these properties. Our formulation of privacy for TLS 1.3 is novel, and our proofs are machine-checked with ProVerif. The main prior work in this space is [ABF$^+$19], which presents a pen-and-paper proof of client identity privacy for two protocol flows of TLS 1.3. Our results are for more privacy properties and more protocol flows, albeit using an abstract model of cryptography.

Finally, we extend our model with ECH and prove that TLS 1.3 with ECH preserves the security properties of TLS 1.3, that it achieves the privacy goals, and that it protects against downgrades to TLS 1.3. We also show that previous draft versions of ECH did not satisfy the privacy goals and demonstrate attacks on these versions.

Our analysis provides concrete feedback for the TLS ECH proposal for the TLS working group and our results have already influenced the standardization of ECH. We also provide guidelines for implementors on how to implement TLS 1.3 and ECH safely.

Our proofs of privacy are (to our knowledge) the largest automated privacy proofs for any protocol in the literature, and at the cutting edge of what is achievable by modern verification tools. Our modeling strategies and verification tricks are of independent interest for future privacy analyses for cryptographic protocols.

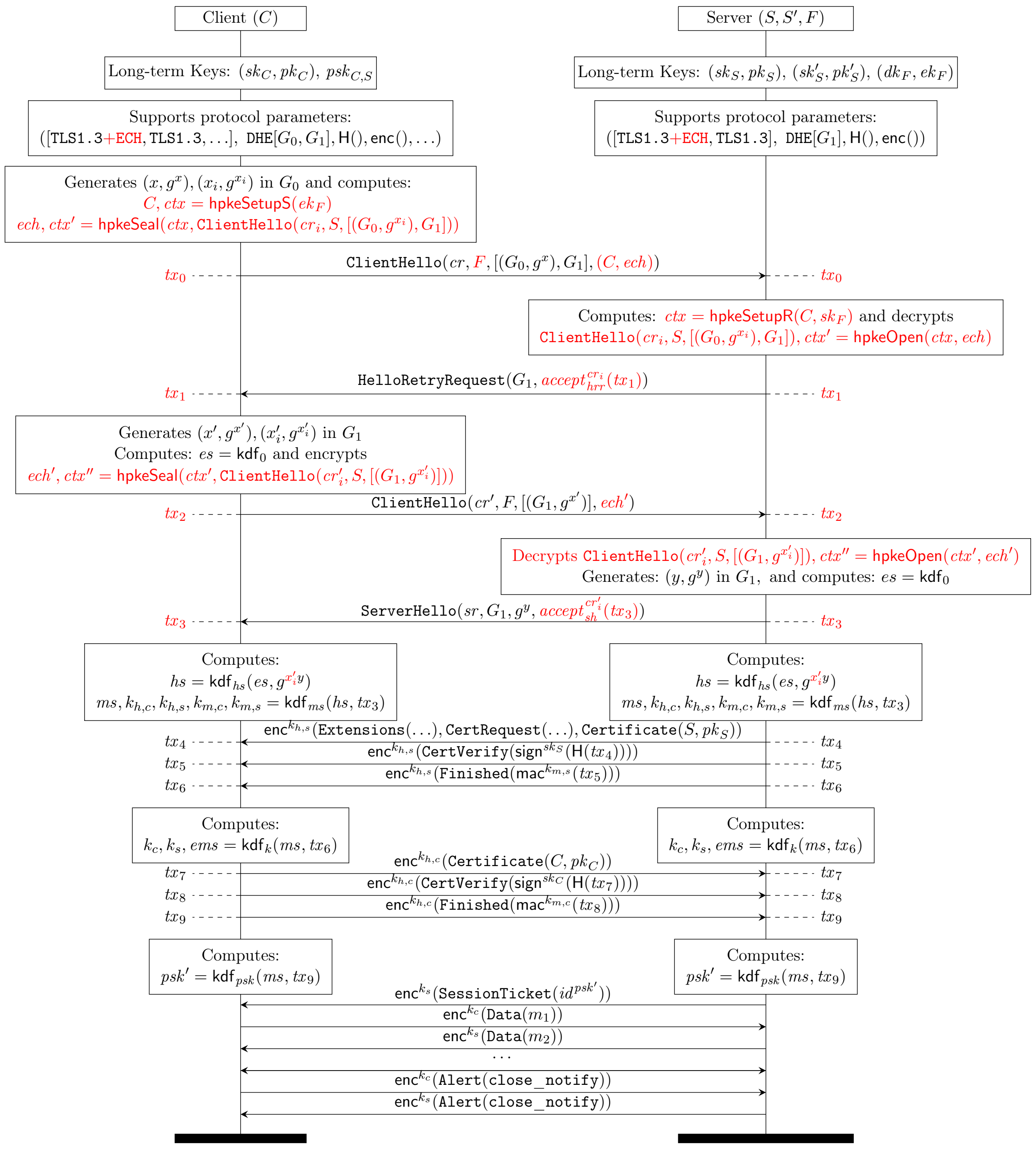
2 Transport Layer Security Background

TLS 1.3 is specified in IETF RFC 8446 [Res18] as a two-party secure channel protocol between a client and a server. It composes three sub-protocols: the *handshake* sub-protocol performs an authenticated key exchange (AKE) to establish a fresh symmetric key, the *record* protocol uses this key for authenticated encryption of application data (and handshake messages), and the *alert* sub-protocol is used to indicate connection closure and fatal errors.

Figure 2 shows an example flow of messages in a TLS 1.3 connection. For now, we ignore the parts in red, which will be used to explain the ECH extension in later sections. Many of the details of TLS 1.3 have been extensively covered and analyzed in prior work. Here, we focus on the features that are most relevant for the privacy modeling and analysis in this report.

We begin by looking at the minimal messages needed to establish a connection, and then describe optional features and configuration options available at the client and server.

Server Authenticated (EC)DHE Handshake with 1-RTT Data. When a TLS 1.3 client (C) connects to a server (S) for the first time, it typically runs a Diffie-Hellman key exchange (using an elliptic curve) where the server is authenticated with an X.509 public key certificate and the client is unauthenticated.



Client (C)
Server (S, S', F)
Long-term Keys: (sk_C, pk_C), $psk_{C,S}$
Long-term Keys: (sk_S, pk_S), (sk'_S, pk'_S), (dk_F, ek_F)
Supports protocol parameters: $([\texttt{TLS1.3+ECH}, \texttt{TLS1.3}, \ldots], \texttt{DHE}[G_0, G_1], \mathsf{H}(), \mathsf{enc}(), \ldots)$
Supports protocol parameters: $([\texttt{TLS1.3+ECH}, \texttt{TLS1.3}], \texttt{DHE}[G_1], \mathsf{H}(), \mathsf{enc}())$
Generates $(x, g^x), (x_i, g^{x_i})$ in G_0 and computes:
$C, ctx = \mathsf{hpkeSetupS}(ek_F)$
$ech, ctx' = \mathsf{hpkeSeal}(ctx, \texttt{ClientHello}(cr_i, S, [(G_0, g^{x_i}), G_1]))$
$\texttt{ClientHello}(cr, F, [(G_0, g^x), G_1], (C, ech))$
tx_0
Computes: $ctx = \mathsf{hpkeSetupR}(C, sk_F)$ and decrypts
$\texttt{ClientHello}(cr_i, S, [(G_0, g^{x_i}), G_1]), ctx' = \mathsf{hpkeOpen}(ctx, ech)$
$\texttt{HelloRetryRequest}(G_1, accept^{cr_i}_{hrr}(tx_1))$
tx_1
Generates $(x', g^{x'}), (x'_i, g^{x'_i})$ in G_1
Computes: $es = \mathsf{kdf}_0$ and encrypts
$ech', ctx'' = \mathsf{hpkeSeal}(ctx', \texttt{ClientHello}(cr'_i, S, [(G_1, g^{x'_i})]))$
$\texttt{ClientHello}(cr', F, [(G_1, g^{x'})], ech')$
tx_2
Decrypts $\texttt{ClientHello}(cr'_i, S, [(G_1, g^{x'_i})]), ctx'' = \mathsf{hpkeOpen}(ctx', ech')$
Generates: (y, g^y) in G_1, and computes: $es = \mathsf{kdf}_0$
$\texttt{ServerHello}(sr, G_1, g^y, accept^{cr'_i}_{sh}(tx_3))$
tx_3
Computes:
$hs = \mathsf{kdf}_{hs}(es, g^{x'_i y})$
$ms, k_{h,c}, k_{h,s}, k_{m,c}, k_{m,s} = \mathsf{kdf}_{ms}(hs, tx_3)$
$\mathsf{enc}^{k_{h,s}}(\texttt{Extensions}(\ldots), \texttt{CertRequest}(\ldots), \texttt{Certificate}(S, pk_S))$
tx_4
$\mathsf{enc}^{k_{h,s}}(\texttt{CertVerify}(\mathsf{sign}^{sk_S}(\mathsf{H}(tx_4))))$
tx_5
$\mathsf{enc}^{k_{h,s}}(\texttt{Finished}(\mathsf{mac}^{k_{m,s}}(tx_5)))$
tx_6
Computes:
$k_c, k_s, ems = \mathsf{kdf}_k(ms, tx_6)$
$\mathsf{enc}^{k_{h,c}}(\texttt{Certificate}(C, pk_C))$
tx_7
$\mathsf{enc}^{k_{h,c}}(\texttt{CertVerify}(\mathsf{sign}^{sk_C}(\mathsf{H}(tx_7))))$
tx_8
$\mathsf{enc}^{k_{h,c}}(\texttt{Finished}(\mathsf{mac}^{k_{m,c}}(tx_8)))$
tx_9
Computes:
$psk' = \mathsf{kdf}_{psk}(ms, tx_9)$
$\mathsf{enc}^{k_s}(\texttt{SessionTicket}(id^{psk'}))$
$\mathsf{enc}^{k_c}(\texttt{Data}(m_1))$
$\mathsf{enc}^{k_s}(\texttt{Data}(m_2))$
$\ldots$
$\mathsf{enc}^{k_c}(\texttt{Alert}(\texttt{close\_notify}))$
$\mathsf{enc}^{k_s}(\texttt{Alert}(\texttt{close\_notify}))$

Figure 2: TLS 1.3 Protocol Flow for (EC)DHE Handshake with Encrypted Client Hello (ECH) and 1-RTT Data.
This protocol flow uses a Diffie-Hellman key exchange with certificate-based server authentication. It also shows `HelloRetryRequest`-based group negotiation and certificate-based client authentication. Other optional features like version and ciphersuite negotiation, post-handshake client authentication and `PSK`-based mutual authentication are not depicted.
Message components in red are introduced by the ECH extension: `ClientHello` messages now have an inner `ClientHello` encrypted with the HPKE public-key pk_F of the front-end server (F). If the server accepts ECH, then the inner `ClientHello` is used in the transcripts $(tx_0, tx_2, \ldots)$, and the inner key-share $(g^{x'_i})$ is used for Diffie-Hellman. Otherwise, the connection falls back to standard TLS 1.3 and the outer `ClientHello` and its key-share $(g^{x'})$ are used.

In this flow, the client first sends a `ClientHello` message containing a nonce (cr) and Diffie-Hellman key share (g^x) in some group (G_0). The server responds with a `ServerHello` containing its own nonce (sr) and key share (g^y). The client and server both compute the Diffie-Hellman shared secret (g^{xy}) and use it to derive a pair of handshake encryption keys ($k_{h,c}, k_{h,s}$) and a pair of MAC keys ($k_{m,c}, k_{m,s}$). This is kind of misleading as the endpoints must wait for more transcript before deriving the MAC keys – do we need that level of detail?

The server then sends a sequence of four handshake messages (encrypted using $k_{h,s}$): `Extensions` contains additional server parameters, `Certificate` contains the server's public-key certificate (for pk_S), `CertVerify` contains a signature over the handshake transcript so far using the server's private key (sk_S), and `Finished` contains a MAC (using $k_{m,s}$) over the handshake transcript up to `CertVerify`. Here, The `Certificate` and `CertVerify` messages serve to authenticate the server, while `Finished` provides key and transcript confirmation, following the classic Sign-and-MAC protocol design pattern from SIGMA [Kra03].

The client responds by sending its own `Finished` message (encrypted using $k_{h,c}$) containing a MAC (using $k_{m,c}$) over the handshake transcript so far (up to server `Finished`).

At this point, both client and server derive a master secret (ms) and data encryption keys (k_c, k_s), and then switch over to these data encryption keys to exchange a stream of encrypted application data messages ($m_1, m_2, \ldots$) in both directions. Application data sent after `Finished` is typically called *1-RTT Data* since it is sent after one set of handshake messages has been sent in each direction. When the data exchange is complete, the client and server send `close_notify` alerts to close the connection.

Cryptographic computations for the various key derivations and MACs in the TLS 1.3 handshake are detailed in Figure 3.

Negotiating Connection Parameters. The basic handshake described above assumes that the client and server already agree on the TLS protocol version, Diffie-Hellman group, and the cryptographic algorithms they will use for signing, MAC, encryption, etc. In practice, the client and server *negotiate* the values of these parameters in the beginning of the handshake.

If the client supports multiple protocol versions (e.g. TLS 1.3 and TLS 1.2), it lists these versions in the `ClientHello` and the server chooses the version it prefers (typically the highest version it supports). Similarly, the `ClientHello` lists the signature algorithms, Diffie-Hellman groups, encryption algorithms, and hash algorithms that the client supports, and the server chooses the combination it prefers and indicates it in the `ServerHello`.

If the server chooses a Diffie-Hellman group (G_1) that the client supports but for which the client has not provided a key share in the `ClientHello`, the server sends back an `HelloRetryRequest` message indicating the chosen group (G_1); the client responds with a new `ClientHello` containing a key-share ($g^{x'}$) for G_1. The subsequent handshake transcript includes the full transcript with both `ClientHello` messages. Otherwise, the rest of the handshake after the initial `HelloRetryRequest` exchange proceeds as normal.

Certificate-based Client Authentication. Although server authentication is far more common, TLS 1.3 also allows clients to be authenticated using public key certificates. During the handshake, between its `Certificate` and `CertVerify` messages, the server may send a `CertRequest` message indicating that it wishes for the client to authenticate itself. If the client agrees to this request, then in its response, it sends three messages instead of just a `Finished`: `Certificate` contains the client's public-key certificate (for pk_C), `CertVerify` contains the client's signature over the transcript (using sk_C), `Finished` contains a MAC over the transcript up to the client's `CertVerify` (using $k_{m,c}$). All these messages are encrypted

Key Derivation Functions:

$\mathsf{hkdf\text{-}extract}(k, s) = \mathsf{HMAC\text{-}H}^{k}(s)$

$\mathsf{hkdf\text{-}expand\text{-}label}_1(s, l, h) =$
$\quad \mathsf{HMAC\text{-}H}^{s}(len_{\mathsf{H}()} \| \texttt{"TLS 1.3, "} \| l \| h \| \mathsf{0x01})$

$\mathsf{derive\text{-}secret}(s, l, m) = \mathsf{hkdf\text{-}expand\text{-}label}_1(s, l, \mathsf{H}(m))$

Non-PSK Key Schedule:

$\mathsf{kdf}_0 = \mathsf{hkdf\text{-}extract}(0^{len_{\mathsf{H}()}}, 0^{len_{\mathsf{H}()}})$

$\mathsf{kdf}_{hs}(es, e) = \mathsf{hkdf\text{-}extract}(es, e)$

$\mathsf{kdf}_{ms}(hs, tx_3) = ms, k_c^h, k_s^h, k_c^m, k_s^m$ where
$\quad ms = \mathsf{hkdf\text{-}extract}(hs, 0^{len_{\mathsf{H}()}})$
$\quad hts_c = \mathsf{derive\text{-}secret}(hs, \mathsf{hts}_c, tx_3)$
$\quad hts_s = \mathsf{derive\text{-}secret}(hs, \mathsf{hts}_s, tx_3)$
$\quad k_{h,c} = \mathsf{hkdf\text{-}expand\text{-}label}(hts_c, \mathsf{key}, \texttt{""})$
$\quad k_{m,c} = \mathsf{hkdf\text{-}expand\text{-}label}(hts_c, \mathsf{finished}, \texttt{""})$
$\quad k_{h,s} = \mathsf{hkdf\text{-}expand\text{-}label}(hts_s, \mathsf{key}, \texttt{""})$
$\quad k_{m,s} = \mathsf{hkdf\text{-}expand\text{-}label}(hts_s, \mathsf{finished}, \texttt{""})$

$\mathsf{kdf}_{k}(ms, tx_6) = k_c, k_s, ems$ where
$\quad ats_c = \mathsf{derive\text{-}secret}(ms, \mathsf{ats}_c, tx_6)$
$\quad ats_s = \mathsf{derive\text{-}secret}(ms, \mathsf{ats}_s, tx_6)$
$\quad ems = \mathsf{derive\text{-}secret}(ms, \mathsf{ems}, tx_6)$
$\quad k_c = \mathsf{hkdf\text{-}expand\text{-}label}(ats_c, \mathsf{key}, \texttt{""})$
$\quad k_s = \mathsf{hkdf\text{-}expand\text{-}label}(ats_s, \mathsf{key}, \texttt{""})$

$\mathsf{kdf}_{psk}(ms, tx_9) = psk_{C,S}{}'$ where
$\quad psk_{C,S}{}' = \mathsf{derive\text{-}secret}(ms, \mathsf{rms}, tx_9)$

PSK Key Schedule:

$\mathsf{kdf}_{es}(psk_{C,S}) = es, k_b$ where
$\quad es = \mathsf{hkdf\text{-}extract}(0^{len_{\mathsf{H}()}}, psk_{C,S})$
$\quad k_b = \mathsf{derive\text{-}secret}(es, \mathsf{pbk}, \texttt{""})$

$\mathsf{kdf}_{0RTT}(es, tx_1) = k_{c,0}$ where
$\quad ets_c = \mathsf{derive\text{-}secret}(es, \mathsf{ets}_c, tx_1)$
$\quad k_{c,0} = \mathsf{hkdf\text{-}expand\text{-}label}(ets_c, \mathsf{key}, \texttt{""})$

MAC Computations:

$binder^{psk_{C,S}} = \mathsf{HMAC\text{-}H}^{k_b}(tx_{ch})$ where
$\quad tx_{ch}$ is the transcript up to current `ClientHello`
$\qquad$ with all binders removed

$\mathsf{mac}^{k_m}(tx) = \mathsf{HMAC\text{-}H}^{k_m}(\mathsf{H}(tx))$

$accept_{sh}^{cr_i} = \mathsf{hkdf\text{-}expand\text{-}label}(k_{cr}, \mathsf{ech}_{sh}, tx_{sh}, 8)$ where
$\quad k_{cr} = \mathsf{hkdf\text{-}extract}(0^{len_{\mathsf{H}()}}, cr_i)$
$\quad tx_{sh}$ is the transcript up to `ServerHello`
$\qquad$ with the last 8 bytes of sr set to 0

$accept_{hrr}^{cr_i} = \mathsf{hkdf\text{-}expand\text{-}label}(k_{cr}, \mathsf{ech}_{hrr}, tx_{sh}, 8)$ where
$\quad k_{cr} = \mathsf{hkdf\text{-}extract}(0^{len_{\mathsf{H}()}}, cr_i)$
$\quad tx_{sh}$ is the transcript up to `HelloRetryRequest`
$\qquad$ with the payload of the ECH extension set to 0

HPKE Computations:

$\mathsf{LabeledExtract}(s, l, i) = \mathsf{hkdf\text{-}extract}(s, \texttt{"HPKE-v1"} \| \mathsf{suite} \| l \| i)$

$\mathsf{LabeledExpand}(prk, l, i, L) = \mathsf{hkdf\text{-}expand\text{-}label}(prk, l', i)$ with
$\quad l' = \mathsf{I2OSP}(L, 2) \| \texttt{"HPKE-v1"} \| \mathsf{suite} \| l$

$\mathsf{KeySchedule{<}R{>}}(ss, i) = \mathsf{Context{<}R{>}}(k, bn, 0, es)$ where
$\quad p_h = \mathsf{LabeledExtract}(\texttt{""}, \texttt{"psk\_id\_hash"}, \texttt{""})$
$\quad i_h = \mathsf{LabeledExtract}(\texttt{""}, \texttt{"info\_hash"}, i)$
$\quad ks_{ctx} = 0x00 || p_h || i_h$
$\quad s = \mathsf{LabeledExtract}(ss, \texttt{"secret"}, \texttt{""})$
$\quad k = \mathsf{LabeledExpand}(s, \text{"key"}, ks_{ctx}, N_k)$
$\quad bn = \mathsf{LabeledExpand}(s, \text{"base\_nonce"}, ks_{ctx}, N_n)$
$\quad es = \mathsf{LabeledExpand}(s, \text{"exp"}, ks_{ctx}, len_{\mathsf{H}()})$
$\quad \mathsf{context} = \mathsf{ContextS}(k, bn, 0, es)$

$\mathsf{SetupBaseS}(e, i) = C, ctxt$ where
$\quad s, C = \mathsf{Encap}(e)$
$\quad ctxt = \mathsf{KeyScheduleS}(s, i)$

$\mathsf{hpkeSetupS}(e) = \mathsf{SetupBaseS}(e, \texttt{"tls ech"} || 0x00 || e)$

$\mathsf{SetupBaseR}(C, x, i) = \mathsf{KeyScheduleR}(\mathsf{Decap}(C, x), i)$

$\mathsf{hpkeSetupR}(C, x) = \mathsf{SetupBaseR}(C, x, \texttt{"tls ech"} || 0x00 || g^x)$

$\mathsf{hpkeSeal}(\mathsf{ctx}, \mathsf{pt}) = \mathsf{enc}^{k}(pt), ctx'$ with
$\quad ctx = \mathsf{ContextS}(k, bn, seq, es),$
$\quad$ uses $\mathsf{I2OSP}(seq, N_n) \oplus bn$ as AEAD nonce
$\quad ctx' = \mathsf{ContextS}(k, bn, seq + 1, es)$

$\mathsf{hpkeOpen}(\mathsf{ctx}, \mathsf{ct}) = \mathsf{dec}^{k}(ct), ctx'$ with
$\quad ctx = \mathsf{ContextR}(k, bn, seq, es),$
$\quad$ uses $\mathsf{I2OSP}(seq, N_n) \oplus bn$ as AEAD nonce
$\quad ctx' = \mathsf{ContextR}(k, bn, seq + 1, es)$

Figure 3: TLS 1.3 Cryptographic Computations.
The standard TLS 1.3 key schedule is on the left. MAC computations are on the right. The hash function $\mathsf{H}()$ used in the key schedule is typically `SHA-256`, which has length $len_{\mathsf{H}()} = 32$ bytes.

using the client's handshake encryption key ($k_{h,c}$).

In some scenarios, the server may wish to request client authentication after the handshake is finished. This feature is called *post-handshake authentication*: the server sends a `CertRequest` message in the middle of the 1-RTT application data exchange, and the client responds with the `Certificate-CertVerify-Finished` message combination. These messages are encrypted under the client's data encryption key (k_c).

Pre-Shared Keys (PSK). If a client (C) and server (S) have been configured with a pre-shared symmetric key ($psk_{C,S}$), then they can avoid public-key signature operations and instead use this PSK to authenticate each other. This pre-shared key may be an *external PSK* provided by the application or it may be a *resumption PSK* output by a prior handshake between the client and server.

Figure 4 depicts a typical PSK-DHE handshake: the client lists the identifiers (e.g. $id^{psk_{C,S}}$) of the PSKs (e.g. $psk_{C,S}$) that it shares with the server in the `ClientHello`. It proves its knowledge of each PSK by using it to compute a *binder* ($binder^{psk_{C,S}}$), i.e. a MAC (using $psk_{C,S}$) over the current transcript (including the current `ClientHello` but with binders removed).

The server responds by choosing one of the PSKs from the `ClientHello` and indicates its index in the `ServerHello`. The chosen PSK is mixed into the key schedule that derives encryption and MAC keys. The server then sends an `Extensions` message followed directly by `Finished`. The MAC in the `Finished` serves to authenticate the server since it could only have been produced by someone who knows the PSK. Note that if the PSK is known to someone other than C or S, or if C is willing to use the PSK both as a client and server, then the authentication guarantees provided by this MAC are very weak [DG19].

The client completes the handshake with its `Finished` message, hence authenticated itself by proving its knowledge of the PSK.

After the end of each handshake, whether authenticated using certificates or PSKs, the server may send the client a `SessionTicket` message containing a *session ticket* that serves as a PSK identifier. This message is encrypted under the application data key (k_s). The client and server then derive a fresh PSK called a *resumption master secret* using the master secret (ms) and final handshake transcript and store it along with the ticket (as identifier) and other connection meta-data in their pre-shared key database.[1]

In a PSK handshake, the client and server share a key from the beginning of the handshake and so the client does not have to wait for a round-trip before sending messages. The client can optimistically assume that the server will pick a certain PSK and start sending application data immediately after the `ClientHello` message. This data is called *0-RTT Data* and is encrypted using a key derived from the PSK. If the server does not accept 0-RTT data or does not choose the PSK, then this data is discarded.

TLS Extensions. We have described the main TLS 1.3 protocol flows and commonly-used optional features, but the protocol itself is extensible. The `ClientHello` message may indicate protocol extensions that the client supports and the server may choose some of these extensions in the `ServerHello`. In fact, even standard TLS 1.3 fields, such as supported versions, key shares, and PSK identifiers are sent as `ClientHello` extensions for backwards compatibility with TLS 1.2.

[1]In practice, a server can avoid maintaining such a database by encrypting the PSK and its metadata using a ticket encryption key (known only to the server) and embedding the resulting ciphertext in the session ticket.

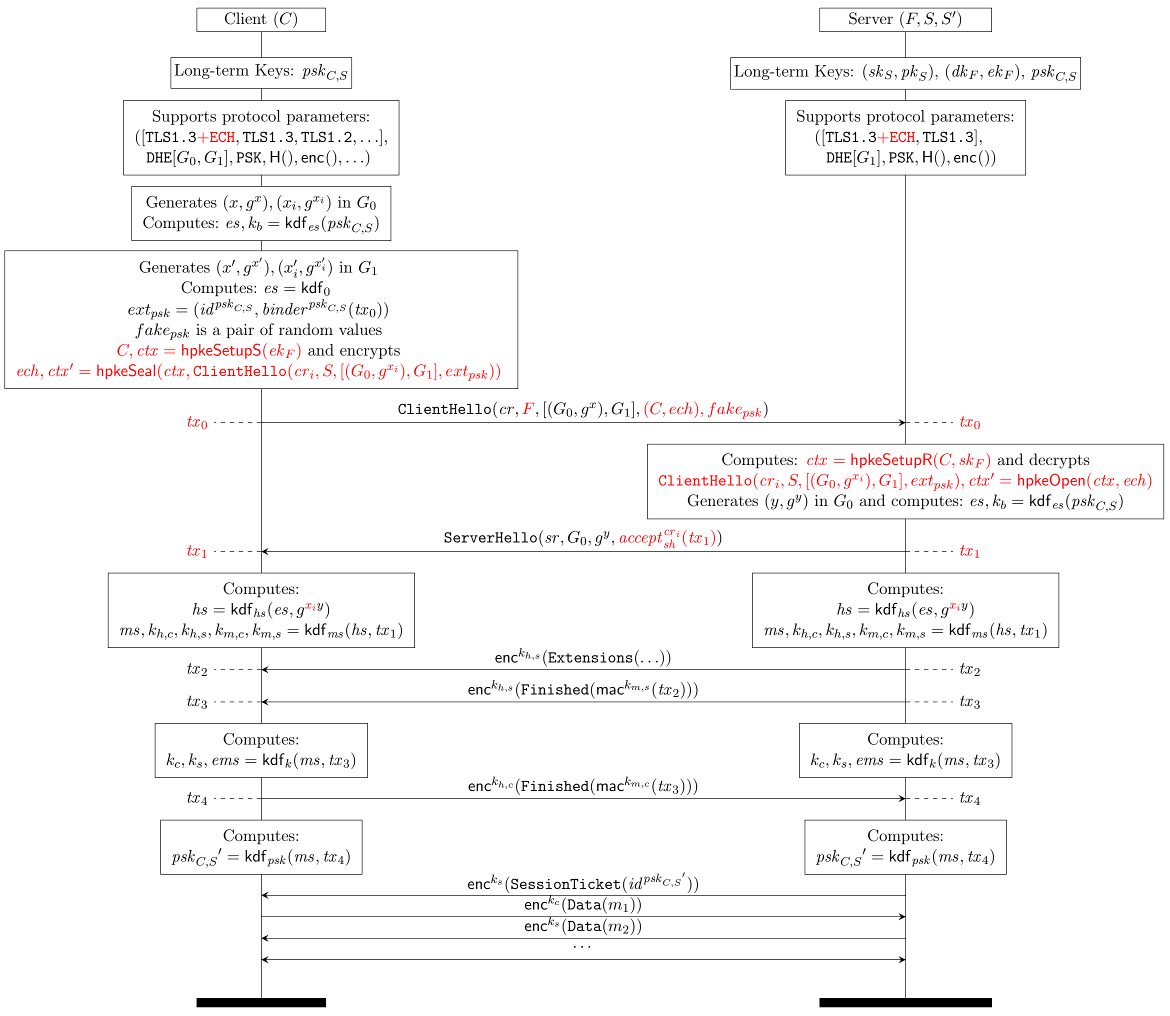

Client (C)
Server (F, S, S')
Long-term Keys: $psk_{C,S}$
Long-term Keys: (sk_S, pk_S), (dk_F, ek_F), $psk_{C,S}$
Supports protocol parameters: $([\mathtt{TLS1.3+ECH}, \mathtt{TLS1.3}, \mathtt{TLS1.2}, \ldots], \mathtt{DHE}[G_0, G_1], \mathtt{PSK}, \mathsf{H}(), \mathsf{enc}(), \ldots)$
Supports protocol parameters: $([\mathtt{TLS1.3+ECH}, \mathtt{TLS1.3}], \mathtt{DHE}[G_1], \mathtt{PSK}, \mathsf{H}(), \mathsf{enc}())$
Generates $(x, g^x), (x_i, g^{x_i})$ in G_0
Computes: $es, k_b = \mathsf{kdf}_{es}(psk_{C,S})$
Generates $(x', g^{x'}), (x'_i, g^{x'_i})$ in G_1
Computes: $es = \mathsf{kdf}_0$
$ext_{psk} = (id^{psk_{C,S}}, binder^{psk_{C,S}}(tx_0))$
$fake_{psk}$ is a pair of random values
$C, ctx = \mathsf{hpkeSetupS}(ek_F)$ and encrypts
$ech, ctx' = \mathsf{hpkeSeal}(ctx, \mathtt{ClientHello}(cr_i, S, [(G_0, g^{x_i}), G_1], ext_{psk}))$
tx_0
$\mathtt{ClientHello}(cr, F, [(G_0, g^x), G_1], (C, ech), fake_{psk})$
tx_0
Computes: $ctx = \mathsf{hpkeSetupR}(C, sk_F)$ and decrypts
$\mathtt{ClientHello}(cr_i, S, [(G_0, g^{x_i}), G_1], ext_{psk}), ctx' = \mathsf{hpkeOpen}(ctx, ech)$
Generates (y, g^y) in G_0 and computes: $es, k_b = \mathsf{kdf}_{es}(psk_{C,S})$
tx_1
$\mathtt{ServerHello}(sr, G_0, g^y, accept^{cr_i}_{sh}(tx_1))$
tx_1
Computes:
$hs = \mathsf{kdf}_{hs}(es, g^{x_i y})$
$ms, k_{h,c}, k_{h,s}, k_{m,c}, k_{m,s} = \mathsf{kdf}_{ms}(hs, tx_1)$
Computes:
$hs = \mathsf{kdf}_{hs}(es, g^{x_i y})$
$ms, k_{h,c}, k_{h,s}, k_{m,c}, k_{m,s} = \mathsf{kdf}_{ms}(hs, tx_1)$
tx_2
$\mathsf{enc}^{k_{h,s}}(\mathtt{Extensions}(\ldots))$
tx_2
tx_3
$\mathsf{enc}^{k_{h,s}}(\mathtt{Finished}(\mathsf{mac}^{k_{m,s}}(tx_2)))$
tx_3
Computes:
$k_c, k_s, ems = \mathsf{kdf}_k(ms, tx_3)$
Computes:
$k_c, k_s, ems = \mathsf{kdf}_k(ms, tx_3)$
tx_4
$\mathsf{enc}^{k_{h,c}}(\mathtt{Finished}(\mathsf{mac}^{k_{m,c}}(tx_3)))$
tx_4
Computes:
$psk_{C,S'} = \mathsf{kdf}_{psk}(ms, tx_4)$
Computes:
$psk_{C,S'} = \mathsf{kdf}_{psk}(ms, tx_4)$
$\mathsf{enc}^{k_s}(\mathtt{SessionTicket}(id^{psk_{C,S'}}))$
$\mathsf{enc}^{k_c}(\mathtt{Data}(m_1))$
$\mathsf{enc}^{k_s}(\mathtt{Data}(m_2))$
$\ldots$

Figure 4: Protocol Flow for TLS 1.3 PSK-(EC)DHE with Encrypted Client Hello (ECH). This protocol instance uses an (Elliptic-Curve) Diffie-Hellman key exchange with `PSK`-based mutual authentication. Message components in red are introduced by the ECH extension.

A commonly used TLS `ClientHello` extension on the web is *Server Name Indication* (SNI), which includes the name of the server (S) to which the client wishes to connect, e.g. `website.example`. This extension is particularly important in scenarios where a web hosting service or content-delivery network hosts a number of backend servers and needs to choose which server to direct the connection to.

By default, all extensions sent in the `ClientHello` (e.g SNI), `HelloRetryRequest`, and `ServerHello` are unencrypted, but the server can encrypt some extension data in `Extensions`. As we shall see, the ECH extension allows the client to encrypt elements of the `ClientHello`. Without this encryption, TLS 1.3 leaks potentially sensitive information to passive attackers during a handshake, including, though not limited to:

- Server name: The `ServerNameIndiation` (SNI) extension carries the name of the target server in cleartext. Given recent and related efforts to encrypt these names during DNS resolution, e.g., via DNS-over-HTTPS (DoH) [HM18], this leak undermines the positive

	(EC)DHE + Server Auth		Negotiation			Client Auth		Pre-Shared Keys				Extensions	
Method	Handshake	1RTT	VER	CS	HRR	CC	PHA	PSK-DHE	TKT	PSKO	0RTT	SNI	ECH
Pen-and-Paper [DFGS21]	✓	✗	✗	✗	✗	✓	✗	✓	✗	✓	✗	✗	✗
Pen-and-Paper [ABF$^+$19]	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗
CryptoVerif [BBK17, Bla18]	✓	✓	✗	✗	✗	✓	✗	✓	✓	✓	✓	✗	✗
F* [BDLF$^+$17, DFP$^+$21]	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
Tamarin [CHSvdM16, CHH$^+$17]	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✗
ProVerif [BBK17]	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✗	✗
This work (ProVerif)	✓	✓	✗	✗	✓	✓	✓	✓	✓	✗	✓	✓	✓

Table 1: TLS 1.3 Features Captured by Formal Analyses.
Handshake: (EC)DHE handshake; 1RTT: 1RTT Data; VER: multiple TLS versions; CS: multiple ciphersuites; HRR: group negotiation; CC: client certificates; PHA: post-handshake authentication; PSK-DHE: PSK-(EC)DHE handshake; TKT: session tickets; PSKO: psk-only mode; 0RTT: 0RTT Data; SNI: server name indication; ECH: encrypted client hello

		Authentication						Confidentiality				Privacy			
Method	Model	SAUTH	CAUTH	AGR	INT	UNIQ	DOWN	SEC	FS	IND	SEC0	CIP	UNL	SIP	EXT
Pen-and-Paper [DFGS21]	Comp.	✓	✓	✓	✗	✓	✗	✓	✓	✓	✗	✗	✗	✗	✗
Pen-and-Paper [ABF$^+$19]	Comp.	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗
CryptoVerif [BBK17, Bla18]	Comp.	✓	✓	✓	✓	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗
F* [BDLF$^+$17, DFP$^+$21]	Comp.	✗	✗	✗	✓	✗	✗	✓	✗	✗	✓	✗	✗	✗	✗
Tamarin [CHSvdM16, CHH$^+$17]	Symb.	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✗	✗	✗	✗
ProVerif [BBK17]	Symb.	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗
This work (ProVerif)	Symb.	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 2: TLS 1.3 Security Properties Covered by Formal Analyses

privacy goals of name encryption on the Internet.

- PSK identities: Resumption PSKs often carry server-provided tickets obtained from prior connection attempts. Moreover, clients use these tickets at most once, which means that the same PSK identity (ticket) is never sent in cleartext more than once. However, external PSKs do not have this restriction. Applications may provision external PSKs usable for multiple connections. Moreover, the identities of these PSKs may contain application- or endpoint-specific information, e.g., "Alice's PSK."

- Supported applications: A client's supported applications are conveyed in cleartext. In some cases, these applications may reveal private information about a client's intent. For example, if a client were to send "dot" as a supported application protocol, this would reveal to the network that the client intends to use the corresponding connection for DNS queries.

- Cached certificate digest: RFC 7924 [ST16] defines an extension wherein clients may send a hash of the certificate it expects from the server, thereby saving the server from re-sending this message during the handshake. This hash digest leaks information about the client's previous connections, as well as the current connection.

The high-level goal of ECH is to encrypt this information that is otherwise available in cleartext to passive attackers. Section 3 contains a detailed discussions of this goal and the relevant threat model in which it is evaluated.

3 TLS 1.3 Goals and Formal Analyses

In this section, we outline the main security and privacy goals of TLS 1.3 and describe various prior analyses of the protocol.

Threat Model. The threat model considered by TLS 1.3 includes passive and active network attackers and malicious or compromised clients and servers. We say that a client or server is *compromised* if any of its private keys or pre-shared keys is known to the adversary; otherwise we say that it is *honest*. The attacker can use compromised keys to impersonate a client or server to an honest party. Furthermore, the attacker is free to use any cryptographic construction to create and decode messages.

The security assumptions on the cryptographic constructions can be formally stated in many different ways. In the *symbolic* or Dolev-Yao model [DY06], cryptographic functions are perfect black-boxes that obey well-defined algebraic rules; protocol participants and the attacker are modeled as non-deterministic processes that can participate in an unbounded number of sessions but they cannot violate the algebraic rules of cryptography. In the *computational* or complexity-theoretic model, cryptographic constructions are probabilistic functions over bitstrings; protocol participants and the attacker are probabilistic polynomial-time turing machines. To better understand the relationship between these two models and their applications, see [AR00, CKW11, BBB^{+}21]. At a high-level, computational models are more cryptographically precise but symbolic models are easier to analyze and can scale to analyze large protocols with many optional features. The strengths of both models are complementary.

Authentication and Integrity Goals. The TLS 1.3 standard [Res18] list several authentication and integrity goals, both for application data and for the handshake protocol:

- **Server Authentication** (SAUTH): If a client C receives application data message m over a TLS 1.3 connection (supposedly) with a server S, then either there must be an honest server S that sent the message m, or else (a long-term key of) S must be compromised.

- **Client Authentication** (CAUTH): If a server S receives application data m over a client-authenticated TLS 1.3 connection (supposedly) with a client C, then either there must be an honest client C that sent the message m, or else C must be compromised.

- **Key and Transcript Agreement** (AGR): If a client (or server) completes a handshake with an honest peer, then both parties must agree on the connection keys (e.g. k_c, k_s) and the handshake transcript.

- **Data Stream Integrity** (INT): If a client (or server) receives a sequence of messages $m_0, m_i, \ldots$ over a TLS 1.3 connection, then either an honest peer must have sent the same sequence of messages, or else the peer must be compromised.

- **Key Uniqueness** (UNIQ): Two different TLS 1.3 connections cannot result in the same encryption keys.

- **Downgrade Resilience** (DOWN): If a client (or server) completes a handshake with an honest peer and both parties prefer to use a certain protocol parameter over another (e.g a version or ciphersuite or Diffie-Hellman group), then the handshake cannot end with the less-preferred parameter.

Note that the authentication properties (CAUTH, SAUTH) apply to both certificate-based handshakes and PSK handshakes and prevent impersonation without compromise. In the case of certificate-based authentication, these properties forbid key compromise impersonation (KCI): even if an attacker knows (say) sk_C, it cannot impersonate S to C. The transcript agreement property (AGR) provides different guarantees at the client and server: after the client `Finished`, the server gets full transcript agreement, whereas the client only gets transcript agreement up to the server `Finished`. Data stream integrity (INT) implies that the attacker cannot reorder or selectively drop application data messages. Key uniqueness (UNIQ) prevents unknown key share and key synchronization attacks [BDF$^+$14]. Downgrade resilience (DOWN) prevents a network attacker from fooling modern clients and servers into using an obsolete protocol version or cryptographic construction that is supported only for backwards compatibility.

Confidentiality. The confidentiality guarantees of TLS 1.3 can either be stated in terms of the keys established by the handshake or the application data exchanged by the client and server:

- **Key Secrecy** (SEC): If a client (or server) completes a handshake with data encryption keys (k_c, k_s), then either the peer is honest and keys are unknown to the adversary, or the peer is compromised.
- **Key Indistinguishability** (IND): If a client (or server) completes a handshake with an honest peer to obtain data encryption keys (k_c, k_s), then these keys are indistinguishable from fresh random values generated at the end of the handshake.
- **1-RTT Data Forward Secrecy** (FS): If a client (or server) sends a secret application data message m over a TLS 1.3 connection with an honest peer, then the message m remains secret even if the peer is compromised after the connection has closed.
- **0-RTT Data Secrecy** (SEC0): If a client sends secret 0RTT application data message m over a TLS 1.3 connection, then this message remains secret unless the PSK used in the connection is compromised.

Note that key indistinguishability (IND) is strictly stronger than key secrecy (SEC), and that forward secrecy (FS) is also stronger than secrecy. TLS 1.3 only guarantees FS for 1-RTT data. These are four example properties taken from the literature on TLS 1.3, but other confidentiality properties, such as data indistinguishability and post-compromise secrecy, can also be formulated for various TLS 1.3 scenarios.

Privacy. Appendix E.1 of the TLS 1.3 RFC [Res18] mentions the "protection of endpoint identities" as one of the goals of the handshake. We elaborate on this goal and extend it to capture four specific privacy goals:

- **Client Identity Privacy** (CIP): If a client holds long-term keys for users A and B and connects using client-authenticated TLS 1.3 to an honest server S, then the attacker cannot tell whether the client authenticated as A or as B.
- **Client Unlinkability** (UNL): If two anonymous clients make TLS 1.3 connections to a server S, and then one of the two clients makes a new TLS 1.3 connection to the same server S, then the attacker cannot tell which of the two clients made the new connection.

- **Server Identity Privacy** (SIP): If a server holds long-term keys for two servers S_1 and S_2 and the server receives to a TLS 1.3 connection from an honest client, then the attacker cannot tell whether the server authenticated as S_1 or S_2.
- **Client and Server Extension Privacy** (EXT): If a client or server sends an extension with a sensitive payload over a TLS 1.3 handshake with an honest peer, then the attacker cannot distinguish between different values of the payload.

Note that client identity privacy (CIP) is stated for both certificate-based and PSK-based client authentication. Client unlinkability (UNL) is stronger in that it works even for anonymous clients and even if the client makes a series of connections that are cryptographically linked via resumption PSKs. TLS 1.3 only asks for Server identity privacy (SIP) against passive attackers, while our formulation is stronger. Extension privacy (EXT) corresponds to secrecy or indistinguishability for extension payloads, but is treated here as a privacy property since extensions typically carry metadata, not application secrets.

Prior Security and Privacy Analyses. Many prior works analyze various parts of TLS 1.3 for authentication and confidentiality properties. We survey some of the most advanced works in this space and compare the TLS 1.3 features they cover and the properties they analyze in Tables 1 and 2.

Pen-and-paper cryptographic proofs of TLS 1.3 typically focus on a few handshake modes to highlight and analyze a particular feature of the protocol. For example, the most recent cryptographic proof of the TLS 1.3 handshake appears in [DFGS21] and subsumes many prior analyses. It covers the certificate-based DHE handshake, the PSK-DHE handshake, and the PSK-only handshake. However, it does not cover data or handshake encryption, negotiation, session tickets, or extensions. It proves confidentiality and integrity properties about handshake keys, but does not prove properties about application data (e.g. stream integrity) and does not consider privacy.

Even with these limitations, this is an impressive proof described carefully in 65 pages of text. Suppose there were an attack that exploited a combination of `HelloRetryRequest` and session tickets to only manifest in a sequence of three handshakes, where the resumption PSK generated in the first handshake is used in both subsequent handshakes. Such attacks are not always unrealistic (see e.g. [BDF$^+$14]), but are well outside the scope of [DFGS21].

The problem is that pen-and-paper proofs cannot scale to the full complexity of protocols like TLS. Even if we could write hundreds of pages of proofs, checking them would be infeasible. One option is to build machine-checked proofs, and such a cryptographic proof has been built for TLS 1.3 using CryptoVerif [BBK17, Bla18]. This proof covers more TLS features than pen-and-paper analyses but the proofs have to guided manually and the tool does not scale to the analysis of features like negotiation.

The most comprehensive analyses of TLS rely on automated tools based on the symbolic Dolev-Yao model. For example, Tamarin has been used to develop a comprehensive model of most features of TLS 1.3 [CHSvdM16, CHH$^+$17]. This model, however, does not cover version and ciphersuite negotiation, probably to reduce the analysis complexity. A different symbolic model was developed in ProVerif [BBK17], which supports multiple versions and weak ciphersuites and hence can find downgrade attacks on TLS 1.3. However, neither of these models analyzes stronger properties like key indistinguishability (IND) or privacy. The probable reason is that although these properties are within the scope of symbolic tools, they require significant modeling effort, precision, and computation time. Even without these properties, analyzing the symbolic models takes several hours.

A different angle of research is to analyze the security of TLS 1.3 *implementations*, to ensure that the proof applies to the deployed protocol without modeling abstractions. This approach has resulted in proofs for the TLS 1.3 record layer in F* [BDLF$^+$17], but has not yet been applied to the handshake.

As we have seen, most prior works on TLS 1.3 do not consider privacy. A notable exception is [ABF$^+$19] which describes several privacy attacks on TLS 1.3, and gives a pen-and-paper proof of server identity privacy and client unlinkability under certain implementation conditions. The model does not cover client authentication and does not enable SNI (which would falsify server identity privacy). On the whole, this work serves to emphasize the weak privacy guarantees of TLS 1.3. In order to get stronger privacy, a new extension to TLS 1.3 is needed, and to analyze such an extension, we need a more comprehensive machine-checkable model of TLS 1.3.

Our analysis results on TLS 1.3. In this paper, we present a new configurable model of TLS 1.3 in ProVerif. Our model extends the previous proverif model of [BBK17] in several ways: we add support for post-handshake authentication, extensions like SNI and ECH, and we make the model significantly more configurable so that the analyst can easily turn various features on and off and rerun ProVerif. Indeed, all the analyses in this paper are run using automated scripts that run each analysis under various combinations of features, until they find the maximal set of features under which ProVerif is able to find a proof in a few hours.

The main technical advance we make over the TLS 1.3 model of [BBK17] is to add support for equivalence-based reasoning. We prove key indistinguishability for TLS 1.3, stated as the indistinguishability of the resumption master secret from a random value. This kind of *strong secrecy* property was not proved before for comprehensive models of TLS 1.3 in ProVerif or in Tamarin. We also analyze TLS 1.3 for privacy properties, finding that TLS 1.3 with SNI immediately fails server identity privacy (SIP), does not provide client extension privacy (C-EXT), but does provide client identity privacy (CIP), unlinkability (UNL), and server extension privacy (S-EXT).

We summarize our results for TLS 1.3 in Table 2. The precise details of the model and properties will be explained in Section 5. Due to heavy memory consumption, the ✓and ✗ marks represent the features activated and deactivated respectively during the analysis in addition to Handshake and SNI. Hence, we prove secrecy and authenticaiton for the full model of TLS 1.3 but to prove handshake-oriented equivalence properties for key indistinguishability and privacy, we disable application data and post-handshake authentication. To keep the analysis feasible, we also disable TLS 1.2 and weak ciphersuites, and hence do not model downgrade attacks to prior versions and legacy cryptographic algorithms.

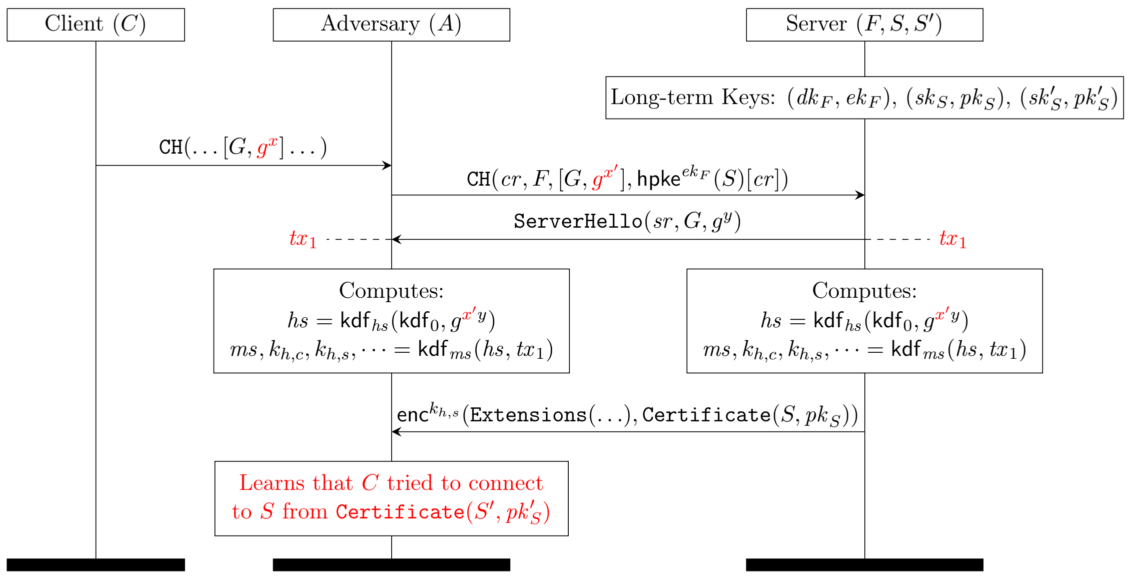
Our model of TLS 1.3 is one of the most comprehensive symbolic models of the protocol and is of independent interest. Its configurability makes it a strong foundation for automatically analyzing extensions and advanced properties of TLS 1.3. Our verification results for the secrecy and privacy of TLS 1.3 are also stronger than prior symbolic analyses. The next step is to obtain the missing privacy properties of TLS 1.3 (SIP,C-EXT) via the ECH extension.

4 ECH: Encrypted Client Hello

The goal of the ECH extension is simple: given a public key of the client-facing server, encrypt the contents of `ClientHello` such that only the client-facing server can use the true value. This turns out to be a non-trivial challenge in realistic threat models. The rest of this section describes some prior designs of ECH (and ESNI) that had flaws due to subtle yet important attacker capabilities.

4.1 ESNI draft 0

Consider perhaps the simplest variant of ESNI, the predecessor to ECH that focused solely on encrypting the SNI extension. First, generate a normal `ClientHello`, except that instead of using the SNI extension that carries the plaintext SNI, insert a different extension that carries an encryption of the SNI under the client-facing server public key. This is vulnerable to an obvious cut-and-paste attack, wherein an attacker can extract the ESNI extension from this target `ClientHello`, insert it into its own ClientHello, and then observe the server's response, which includes the server Certificate message. This attack is demonstrated in Figure 5.



Client (C)
Adversary (A)
Server (F, S, S')
Long-term Keys: (dk_F, ek_F), (sk_S, pk_S), (sk'_S, pk'_S)
$\mathtt{CH}(\ldots[G, g^x]\ldots)$
$\mathtt{CH}(cr, F, [G, g^{x'}], \mathsf{hpke}^{ek_F}(S)[cr])$
$\mathtt{ServerHello}(sr, G, g^y)$
tx_1
tx_1
Computes:
$hs = \mathsf{kdf}_{hs}(\mathsf{kdf}_0, g^{x'y})$
$ms, k_{h,c}, k_{h,s}, \cdots = \mathsf{kdf}_{ms}(hs, tx_1)$
Computes:
$hs = \mathsf{kdf}_{hs}(\mathsf{kdf}_0, g^{x'y})$
$ms, k_{h,c}, k_{h,s}, \cdots = \mathsf{kdf}_{ms}(hs, tx_1)$
$\mathsf{enc}^{k_{h,s}}(\mathtt{Extensions}(\ldots), \mathtt{Certificate}(S, pk_S))$
Learns that C tried to connect to S from $\mathtt{Certificate}(S', pk'_S)$

Figure 5: Cut-and-paste attack (draft-ietf-tls-esni-00)

The first ESNI design went beyond this simple variant and bound the encryption of the SNI to `ClientHello.random`. However, this was vulnerable to the same basic problem, since an attacker can simply replay both the ESNI extension *and* `ClientHello.random` with its own key share to observe the server response.

Fundamentally, this simple issue points at the crux of the design challenge: how does one encrypt the SNI such that it is bound *forward* to the rest of the handshake, as well as *backward* to the client's `ClientHello` value. In the following two sections, we will describe two variants of this design that demonstrate the need for both types of bindings, using real-world attacks to highlight their importance.

4.2 Backward Binding

Binding backwards means that changes to anything *before* the server's response should cause the TLS handshake to fail. The attack above wherein the ESNI was not bound to the client's key share was one instance of failing to bind backwards completely. There are other more subtle forms of backward binding, however.

Consider TLS session resumption. The typical flow involves a client connecting to a server and being presented a resumption ticket in response. In subsequent connection attempts, clients opportunistically offer this ticket in the form of a PSK to the server. If the server can process and accept the ticket, e.g., by decrypting it and recovering session state, the server will complete the handshake without re-sending the server certificate. Some of this state may include, for example, the SNI value used in the original handshake. Servers which do not admit resumption across hostnames will necessarily check that the SNI of the previous state coincides with the SNI of the client's `ClientHello` and, if they disagree, fall back to a full handshake.

This behavior admits an oracle for learning the SNI contained in a ticket. Specifically, an attacker that obtains a ticket for a domain of its choosing, e.g., `test.example`, may inject it in a ticket-free `ClientHello` carrying an encrypted SNI payload, and then forward the modified message to the server. If the server completes the handshake in an abbreviated form, which may be visible because the server's total handshake flight is smaller given that there is no certificate chain, then the attacker learns that its guess of `test.example` was correct. This is shown in Figure 6. The fix to this type of problem is to ensure that the client's `ClientHello` cannot be modified by anyone other than the client itself, meaning that the encryption is bound to the entire `ClientHello`.

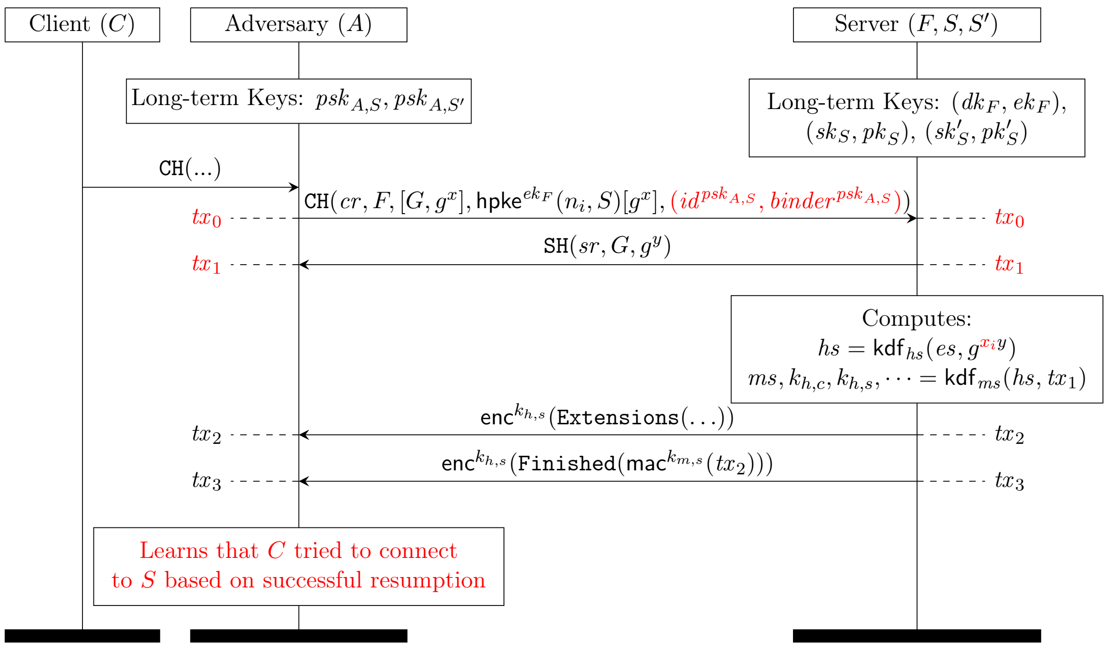
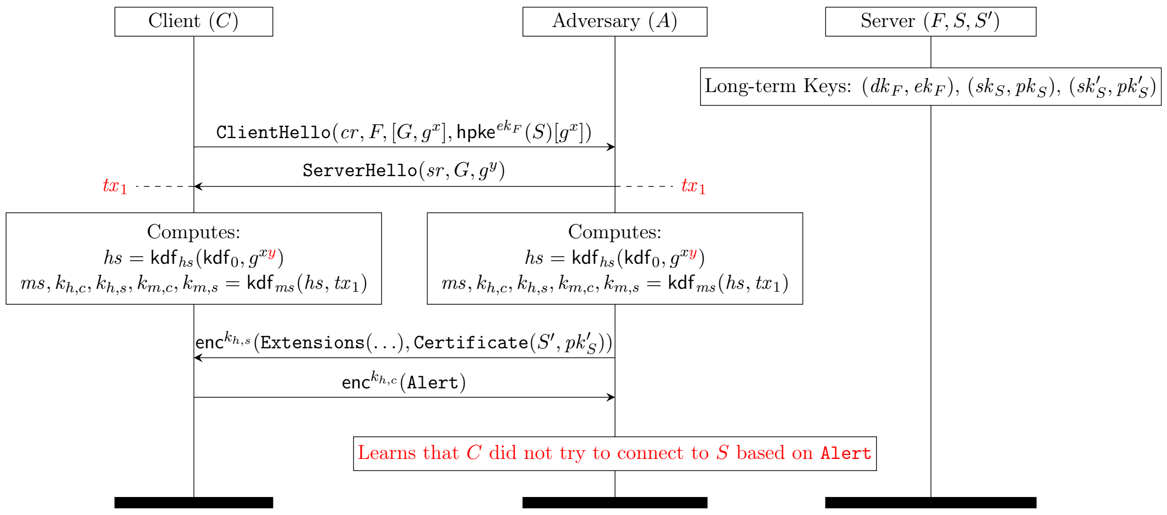

Client (C)
Adversary (A)
Server (F, S, S')
Long-term Keys: $psk_{A,S}, psk_{A,S'}$
Long-term Keys: (dk_F, ek_F), (sk_S, pk_S), (sk'_S, pk'_S)
$\mathtt{CH}(\ldots)$
$\mathtt{CH}(cr, F, [G, g^x], \mathsf{hpke}^{ek_F}(n_i, S)[g^x], (id^{psk_{A,S}}, binder^{psk_{A,S}}))$
tx_0
tx_0
$\mathtt{SH}(sr, G, g^y)$
tx_1
tx_1
Computes:
$hs = \mathsf{kdf}_{hs}(es, g^{x_i y})$
$ms, k_{h,c}, k_{h,s}, \cdots = \mathsf{kdf}_{ms}(hs, tx_1)$
$\mathsf{enc}^{k_{h,s}}(\mathtt{Extensions}(\ldots))$
tx_2
tx_2
$\mathsf{enc}^{k_{h,s}}(\mathtt{Finished}(\mathsf{mac}^{k_{m,s}}(tx_2)))$
tx_3
tx_3
Learns that C tried to connect to S based on successful resumption

Figure 6: PSK oracle attack (draft-ietf-tls-esni-05)

4.3 Forward Binding

Forward binding means that the only entities which can proceed with the handshake are those which can decrypt and recover the private information encrypted by ECH. To illustrate the need for this requirement, consider the following scenario. A client connects to a server using ECH and receives upon success an encrypted `Certificate` message and then `CertVerify` message. The `Certificate` message contains the server certificate chain that clients use to authentiate against the server name. The `CertVerify` contains a signature over the handshake transcript using the private key corresponding to the leaf public key in `Certificate`. If clients process and verify `Certificate` before verifying `CertVerify`, it is possible for clients to reveal that `Certificate` is invalid before `CertVerify` is invalid.

This type of reaction could be abused to attack ECH in the following way. Assume the client sends a `ClientHello` to the server using ECH. Assume further that there exists an on-path attacker between client and server. This attacker could hijack the connection and continue it using a valid `Certificate` message, i.e., one matching `Certificate` from the server that it wants to check against. (The attacker could obtain `Certificate` by connecting to the target server as a client and observing `Certificate`.) Importantly, the attacker *does not* provide the `CertVerify` message, as it cannot produce a signature without the private key. If the client does not abort the connection after receiving the valid `Certificate` message, the attacker learns that it presented a valid chain matching the client's expectation, i.e., one that matches validates against the client's chosen SNI. This type of reaction attack is shown in Figure 7.



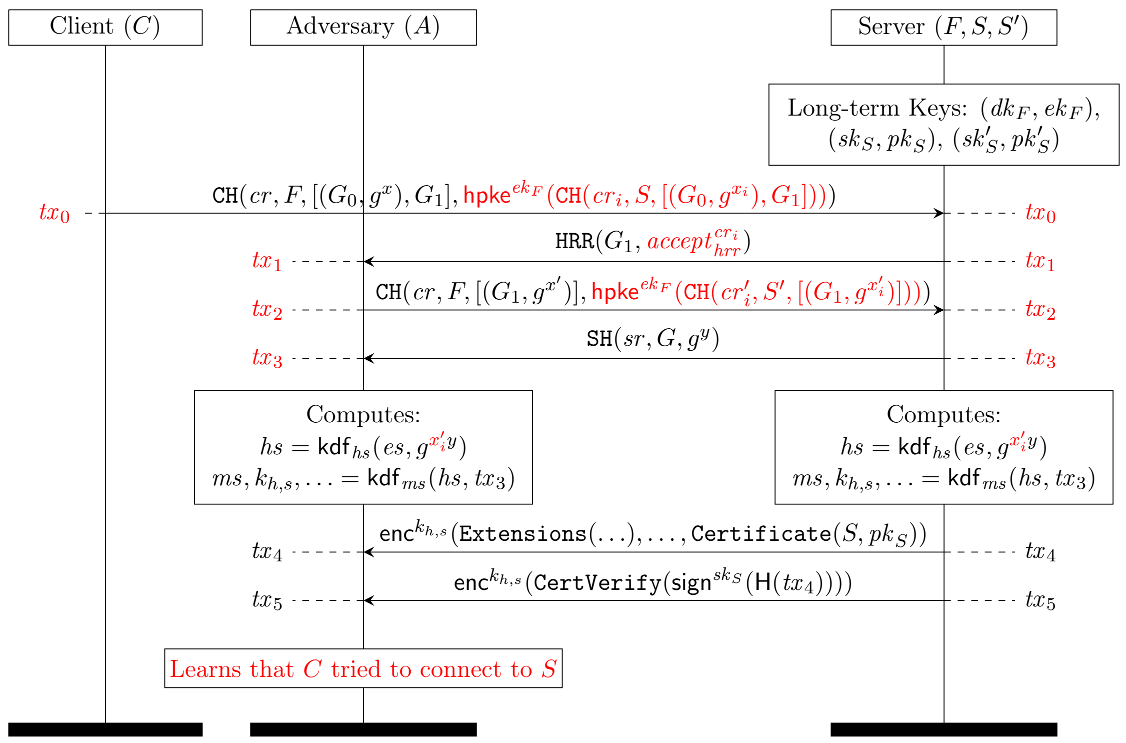
Client (C)
Adversary (A)
Server (F, S, S')
Long-term Keys: (dk_F, ek_F), (sk_S, pk_S), (sk'_S, pk'_S)
$\texttt{ClientHello}(cr, F, [G, g^x], \mathsf{hpke}^{ek_F}(S)[g^x])$
$\texttt{ServerHello}(sr, G, g^y)$
tx_1
tx_1
Computes:
$hs = \mathsf{kdf}_{hs}(\mathsf{kdf}_0, g^{xy})$
$ms, k_{h,c}, k_{h,s}, k_{m,c}, k_{m,s} = \mathsf{kdf}_{ms}(hs, tx_1)$
Computes:
$hs = \mathsf{kdf}_{hs}(\mathsf{kdf}_0, g^{xy})$
$ms, k_{h,c}, k_{h,s}, k_{m,c}, k_{m,s} = \mathsf{kdf}_{ms}(hs, tx_1)$
$\mathsf{enc}^{k_{h,s}}(\texttt{Extensions}(\ldots), \texttt{Certificate}(S', pk'_S))$
$\mathsf{enc}^{k_{h,c}}(\texttt{Alert})$
Learns that C did not try to connect to S based on Alert

Figure 7: Client reaction attack (draft-ietf-tls-esni-05)

The fix to this type of problem is to ensure that the client's encrypted values are always bound forward in the handshake. Specifically, the transcript used to derive handshake encryption keys must only be accessible to the client and client-facing server. This prevents the attacker from producing an encrypted `Certificate` for the client to validate.

There are also more subtle examples of failing to bind forward. Consider the `HelloRetryRequest` flow in normal TLS 1.3 from Figure 2. Servers reply to a `ClientHello` with a special message indicating they could not complete the handshake with the client's provided parameters. In response to the `HelloRetryRequest` message, the client generates a new `ClientHello` message to the server with the desired parameters and values. If the server can accept the new parameters, it provides its own key share and continues with the remainder of the protocol. As input to the handshake encryption key derivation, the transcript used is based only on *public* data exchanged between client and server. This means it is possible for an on-path attacker to hijack a HRR message and continue the handshake with its own updated second `ClientHello`, unbeknownst to the client.

Now consider this same flow in the context of ECH. Moreover, imagine further that the client's encrypted value was processed and recorded by the server upon receipt of the first `ClientHello`. If the server processes the second `ClientHello` using parameters from the first `ClientHello`, such as the plaintext SNI value, it would be possible for the on-path hijacking attacker to force the server to reveal the plaintext SNI. This is shown in Figure 8.



Client (C)
Adversary (A)
Server (F, S, S')
Long-term Keys: (dk_F, ek_F), (sk_S, pk_S), (sk'_S, pk'_S)
$\mathtt{CH}(cr, F, [(G_0, g^x), G_1], \mathsf{hpke}^{ek_F}(\mathtt{CH}(cr_i, S, [(G_0, g^{x_i}), G_1])))$
tx_0
$\mathtt{HRR}(G_1, accept^{cr_i}_{hrr})$
tx_1
$\mathtt{CH}(cr, F, [(G_1, g^{x'})], \mathsf{hpke}^{ek_F}(\mathtt{CH}(cr'_i, S', [(G_1, g^{x'_i})])))$
tx_2
$\mathtt{SH}(sr, G, g^y)$
tx_3
Computes: $hs = \mathsf{kdf}_{hs}(es, g^{x'_i y})$ $ms, k_{h,s}, \ldots = \mathsf{kdf}_{ms}(hs, tx_3)$
Computes: $hs = \mathsf{kdf}_{hs}(es, g^{x'_i y})$ $ms, k_{h,s}, \ldots = \mathsf{kdf}_{ms}(hs, tx_3)$
$\mathsf{enc}^{k_{h,s}}(\mathtt{Extensions}(\ldots), \ldots, \mathtt{Certificate}(S, pk_S))$
tx_4
$\mathsf{enc}^{k_{h,s}}(\mathtt{CertVerify}(\mathsf{sign}^{sk_S}(\mathsf{H}(tx_4))))$
tx_5
Learns that C tried to connect to S

Figure 8: `HelloRetryRequest` hijack attack on ECH

The fix to this type of problem is to ensure that the client's encrypted values are always bound forward in the handshake.

4.4 ECH

The current design of ECH [ROSW21] is based on a simple idea: given a public key, encrypt the entire `ClientHello` such that only the owner of the private key, i.e., the client-facing server, can decrypt the response and complete the handshake. However, for practical availability reasons, the protocol must accommodate cases where the client-facing server rotates or forgets its private key. This led to a design wherein there are two `ClientHello` messages, an inner and outer one called `ClientHelloInner` and `ClientHelloOuter`, respectively. `ClientHelloInner` contains the private parameters for the handshake with the *backend server* that a client wishes to keep secret from the attacker. `ClientHelloOuter` contains parameters for completing the handshake with the *client-facing server*, and is necessary in the event that the client-facing server cannot decrypt and process `ClientHelloInner`.

Importantly, `ClientHelloOuter` carries an encryption of `ClientHelloInner` as an extension, and is constructed such that the outer message is not malleable by anyone other than the client or client-facing server. This is done by creating a synthetic `ClientHelloOuter`$'$ with an empty placeholder of length equal to `ClientHelloInner`, and using this as additional authenticated data when encrypting `ClientHelloInner`. The encryption of `ClientHelloInner` is then written to the placeholder in `ClientHelloOuter`$'$, yielding `ClientHelloOuter`. If a client does not have the client-facing server public key, `ClientHelloOuter` carries a *random string* whose contents are indistinguishable from an encrypted `ClientHelloInner`. This is referred to as a fake `ClientHello`.

Upon receipt of a `ClientHelloOuter`, a server takes the following actions. If it does not support ECH or cannot decrypt `ClientHelloInner`, it completes the handshake with `ClientHelloOuter`. This branch is referred to as *rejecting* ECH. Otherwise, if the server successfully decrypts the extension, it forwards `ClientHelloInner` to the backend server, which completes the handshake. This branch is referred to as *accepting* ECH. The transcript used in the TLS key schedule varies depending on this branch. For ECH rejection the transcript includes `ClientHelloOuter`, which covers `ClientHelloInner`, whereas for ECH acceptance the transcript only includes `ClientHelloInner`.

Upon receiving the server's response, the client determines whether or not ECH was accepted and proceeds with the handshake accordingly. Servers indicate ECH acceptance in such a way that an attacker cannot distinguish between a successful handshake where ECH is accepted and a successful handshake where the client offered a fake `ClientHello`. Indicating acceptance is done with $accept_{sh}^{cr_i}$, which is computed as described in Figure 3. The output of this PRF is placed in the `ServerHello`.*random* field. Clients re-compute $accept_{sh}^{cr_i}$ and check it against `ServerHello`.*random*. When the values match, the client concludes acceptance, otherwise the client concludes rejection. After determining acceptance or rejection, the rest of the handshake proceeds as in normal TLS with the proper transcript.

As in the normal TLS handshake, if the client-facing server cannot complete the handshake the client's offered parameters, it sends an `HelloRetryRequest` in response to the client. The `HelloRetryRequest` contains an acceptance signal computed similar to $accept_{sh}^{cr_i}$ so the client can determine if ECH was accepted or rejected. Upon acceptance, the client uses `ClientHelloInner` in responding to the `HelloRetryRequest`. Otherwise, it uses `ClientHelloOuter`. To ensure the encryption of the second `ClientHelloInner` is bound to the first `ClientHelloInner`, the client re-uses the HPKE encryption context for the second `ClientHelloInner`. This means the server can only ever process the second `ClientHello` if it records the encryption context from the first `ClientHello`, which was generated by an honest client.

This entire interaction, complete with `HelloRetryRequest`, is shown in Figure 2.

5 ProVerif model

In this section, we describe our ProVerif model of TLS 1.3 with the ECH extension. We then show how we encode the security and privacy properties in this model. Finally, we describe the results of the analysis and its limitations. Our main goals for this analysis are to answer the following questions:

- Does TLS 1.3, with and without ECH, preserve the secrecy and authentication guarantees of TLS 1.3?
- Does TLS 1.3, with and without ECH, provide client identity privacy and unlinkability?
- Does TLS 1.3 with ECH provide server identity privacy and extension privacy?
- Can TLS 1.3 with ECH be downgraded to TLS 1.3?

We seek to answer these questions through an automated machine-checked symbolic analysis in ProVerif, for a configurable model that supports as many TLS 1.3 features as possible. Our full ProVerif models, with instructions for how to run the analysis, are provided in the supplementary material.

5.1 Cryptographic primitives

ProVerif is based on a symbolic model, similar to the applied pi calculus [], to describe cryptographic protocols. In such model, cryptographic primitives are represented by function symbols with their arity and types $(f, g, \ldots)$, keys and random numbers are represented by *names* $(a, b, k, \ldots)$, messages are modeled by *terms* $(M, N, \ldots)$ that are build as a name, and a variable $(x, y, \ldots)$ or the application of function symbol on terms.

This section describes models for Authenticated Encryption with Additional Data (AEAD), key generation, and the HPKE algorithms, all of which are used by TLS and ECH in particular.

5.1.1 AEAD algorithm

Code extract 1: librairies/primitives.pvl

```
32 type aead_strengh.
33 const WeakAE, StrongAE: aead_strengh [data].
34
35 type aead_alg.
36 fun id_aead(aead_strengh,nat):aead_alg [data].
37
38 type aead_key.
39 fun b2ae(bitstring):aead_key [typeConverter].
40
41 fun aead_enc(aead_alg, aead_key, bitstring, bitstring, bitstring): bitstring.
42 fun aead_forged(bitstring,bitstring): bitstring.
```

ProVerif's front-end relies on a simple type system to declare names, functions etc. However, when proving the security properties, ProVerif will consider an untyped attacker, i.e., an attacker that does not conform to types. Code Extract 1 models AEAD algorithms used in TLS. We consider two levels of security – strong or weak – and multiple encryption algorithms. Each of these algorithms is associated with an identifier built from the function id_aead taking as argument the security level aead_strengh and a natural number nat.

AEAD encryption is represented as application of the function aead_enc, defined at line 41, which takes as arguments the algorithm's identifier, a aead_key, two additional values of type bitstring, and finally the plaintext of type bitstring. As we consider weak security guarantees as well, we also define a function aead_forged that allows the attacker to forge a new ciphertext value from an existing one with a different plaintext.

The algebraic properties of cryptographic primitives are expressed by means of rewrite rules, as illustrated below with the declaration of the AEAD decryption algorithm.

Code extract 2: librairies/primitives.pvl

```
44 fun aead_dec(aead_alg, aead_key, bitstring, bitstring, bitstring): bitstring
45 reduc forall a:aead_alg, k:aead_key, n,p,ad:bitstring;
46   aead_dec(a, k, n, ad, aead_enc(a, k, n, ad, p)) = p
47 otherwise forall i:nat, k:aead_key, n,p,p',ad,ad':bitstring;
48   aead_dec(id_aead(WeakAE,i), k, n, ad, aead_forged(p,aead_enc(id_aead(WeakAE,i
       ), k, n, ad', p'))) = p.
```

Line 46 indicates that one can decrypt the cipher aead_enc(a, k, n, ad, p) and obtain the plaintext p if the corresponding key k and the two additional data n,ad are given to the decryption algorithm. Line 48 models the weak level of security, by allowing a successful decryption of a forged cipher in case the encryption algorithm corresponds to one with a weak security (modelled by id_aead(WeakAE,i)). Note that the **otherwise** keyword indicates that ProVerif will attempt to apply the first rewrite rule, and only if that one fails, will it attempt to apply the second rewrite rule. In case neither rewrite rules can be applied, ProVerif concludes that decryption failed.

Additionally, for AEAD algorithms with weak security levels, we also model a leakage of the plaintext, as shown below.

Code extract 3: librairies/primitives.pvl

```
fun aead_leak(bitstring):bitstring
reduc forall i:nat, k:aead_key, n,ad,x:bitstring;
  aead_leak(aead_enc(id_aead(WeakAE,i),k,n,ad,x)) = x.
```

5.1.2 Key generations

Protocols are modeled in ProVerif by means of process calculus. We illustrate the calculus while describing how we handle long-term key management (used for `Certificate` private keys). Code Extract 4 corresponds to the honest key generation.

Code extract 4: librairies/main_processes.m4.pvl

```
let gen_honest_long_term_keys =
 !
 new idP:idProc;
 in(io,a:domain);
 event Same(d2b(a));
 new s:seed;
 let sk_h = uc_privkey(s) in
 event GenCert(a,pk(sk_h));
 let crt = valid_cert(a,pk(sk_h)) in
 insert long_term_keys(a,sk_h,pk(sk_h),crt,idP);
 out(io,(pk(sk_h),crt))
.
```

In our model, clients and servers are represented by terms of type domain. (We use the terms domain and identity interchangeably throughout.) The process gen_honest_long_term_keys generates fresh new signing and verification keys for a client or server. Typically, the attacker can input a domain at line 8, i.e. **in**(io,a:domain). Then, at line 10, a new fresh name s of type seed is generated. The private signing key sk_h is built at line 11 by applying the private function uc_privkey on s. The corresponding public key pk(sk_h) and the certificate crt defined as valid_cert(a,pk(sk_h)) is given to the attacker at line 14 by outputting it on the public channel io, i.e. **out**(io,(pk(sk_h),crt)).

We store, at line 13, the association between the domain (identity) a and its signing and verification keys in the *table* long_term_keys. Tables are used in ProVerif for persistent storage that honest participants can access to retrieve, in our case, the public verification key of a domain. For example, Code Extract 5 models a client with identity c_dom sending the `Certificate` message.

Code extract 5: librairies/main_processes.m4.pvl

```
200 get long_term_keys(=c_dom,sk,c_pkey,crt,idP) [precise] in
201
        ⋮
215 (* The Certificate message *)
216 let certificate_msg = CRT(zero,crt) in
217 let encrypted_certificate_msg = aead_enc(a_alg,chk,zero,zero,m2b(
        certificate_msg)) in
218 out(io,encrypted_certificate_msg);
```

ProVerif executes line 200 by matching any element previously inserted in the table long_term_keys with the pattern long_term_keys(=c_dom,sk,c_pkey,crt,idP). Here, sk,c_pkey, crt and idP are variables and =c_dom represents an equality test, i.e. it will retrieve an element from the table long_term_keys where the first argument is equal to c_dom.

5.1.3 HPKE

We define a new model in ProVerif of HPKE [BBLW21], an emerging standard for hybrid public key encryption. Our model closely follows the standard and implements the sender-unauthenticated (base mode) variant of HPKE used in ECH.

HPKE [BBLW21] relies on several cryptographic dependencies: An Key Encapsulation Mechanism (KEM) that exposes at least two functions Encap(pkR) and Decap(enc, skR); A Key Derivation Function that exposes two functions Extract(salt, ikm) and Expand(prk, info, L). From these two functions the RFC defines two functions LabeledExtract(salt, label, ikm) and LabeledExpand(prk, label, info, L). Finally it relies an AEAD encryption algorithm as we described in the previous section.

These functions are defined as follows:

```
letfun hpke_encap(g:group,pkR:element) = dh_encap(g,pkR,e2b(pkR)).
letfun hpke_decap(g:group,enc:bitstring,skR:bitstring) = dh_decap(g,enc,skR).
letfun hpke_label_extract(h_alg:hash_alg,salt:bitstring,label:label,ikm:
    bitstring) = hkdf_extract(h_alg,salt,(label,ikm)).
letfun hpke_label_expand(h_alg:hash_alg,prk:bitstring,label:label,info:
    bitstring) = hkdf_expand_label(h_alg,k,l,info).
```

Per the HPKE specification, L represents the size of the output messages which we ignore in our case.

We model three main functionalities of HPKE:

1. Creation of the HPKE context for client (S) and server (R) through the template KeySchedule<ROLE> (Section 5.1);

2. Encryption to a public key, i.e. the functions SetupBaseS and SetupBaseR (section 5.1.1); and

3. Encryption and decryption using the AEAD algorithm (section 5.2).

The template KeySchedule<ROLE> takes as arguments mode, shared_secret, info, psk, psk_id. However, TLS ECH only requires SetupBaseS and SetupBaseR, so mode, psk, and psk_id can be modelled as constants. Thus we only consider the variables shared_secret, info, and psk.

The structure of the HPKE context in our model type takes three arguments: The role (R or S), an AEAD key, and the base nonce.

```
type hpkeRole.
const R,S:hpkeRole.

type hpkeContext.
fun hpke_context(hpkeRole,aead_key,bitstring):hpkeContext [data].
```

According to the HPKE specification, the context should also contain a sequence number initialized at 0. This sequence number is used for encrypting and decrypting using the AEAD algorithm. Moreover, it should be increased every time the encryption or decryption functions are invoked successfully. (The sequence number is not incremented if decryption fails, for example.) In ECH, the clients and servers invoke encryption and decryption on the same HPKE context at most twice. Thus, instead of relying on a complex model of this internal sequence number, our encryption and decryption functions will take as argument the sequence number and we will ensures that equal to either 1 and 2 when invoked.

Per the HPKE specification, the template KeySchedule<ROLE> is defined as follows:

```
def KeySchedule<ROLE>(mode, shared_secret, info, psk, psk_id):
    VerifyPSKInputs(mode, psk, psk_id)

    psk_id_hash = LabeledExtract("", "psk_id_hash", psk_id)
    info_hash = LabeledExtract("", "info_hash", info)
    key_schedule_context = concat(mode, psk_id_hash, info_hash)

    secret = LabeledExtract(shared_secret, "secret", psk)

    key = LabeledExpand(secret, "key", key_schedule_context, Nk)
    base_nonce = LabeledExpand(secret, "base_nonce",
                               key_schedule_context, Nn)
    exporter_secret = LabeledExpand(secret, "exp",
                                    key_schedule_context, Nh)

    return Context<ROLE>(key, base_nonce, 0, exporter_secret)
```

In our ProVerif model, this is represented by the following function:

```
letfun hpke_key_schedule(h:hash_alg,r:hpkeRole,shared_secret:element,info:
    bitstring) =
  let info_hash = hpke_label_extract(h,zero,hpke_info_hash,info) in
  let xsecret = hpke_label_extract(h,e2b(shared_secret),hpke_secret,zero) in

  let key = b2ae(hpke_label_expand(h,xsecret,hpke_key,info_hash)) in
  let base_nonce = hpke_label_expand(h,xsecret,hpke_base_nonce,info_hash) in

  hpke_context(r,key,base_nonce).
```

Notice that we removed all elements related to mode, psk and psk_id as they are always constant. As a result, we removed exporter_secret from the context.

Per the HPKE specification, the context setup functions are defined as follows:

```
def SetupBaseS(pkR, info):
  shared_secret, enc = Encap(pkR)
  return enc, KeyScheduleS(mode_base, shared_secret, info,
                           default_psk, default_psk_id)

def SetupBaseR(enc, skR, info):
  shared_secret = Decap(enc, skR)
  return KeyScheduleR(mode_base, shared_secret, info,
                      default_psk, default_psk_id)
```

These are represented in our ProVerif as follows:

```
letfun hpke_setup_base_S(h:hash_alg,g:group,pkR:element,info:bitstring) =
  let (shared_secret:element,enc:element) = hpke_encap(g,pkR) in
  (enc,hpke_key_schedule(h,S,shared_secret,info)).

letfun hpke_setup_base_R(h:hash_alg,g:group,enc:element,skR:bitstring,info:
    bitstring) =
  let shared_secret = hpke_decap(g,enc,skR) in
  hpke_key_schedule(h,R,shared_secret,info).
```

Finally, the Seal and Open functions in the HPKE specification are described as:

```
def ContextS.Seal(aad, pt):
  ct = Seal(self.key, self.ComputeNonce(self.seq), aad, pt)
  self.IncrementSeq()
  return ct

def ContextR.Open(aad, ct):
  pt = Open(self.key, self.ComputeNonce(self.seq), aad, ct)
  if pt == OpenError:
    raise OpenError
  self.IncrementSeq()
  return pt
```

As mentioned previously, we do not directly model the function self.ComputeNonce(self.seq) but rather gives the sequence number as argument. However, self.ComputeNonce(self.seq) should normally result of XORing base_nonce with the current sequence number. As ProVerif does not handle the XOR operation, we rather compute the concatenation of base_nonce with the sequence number:

```
letfun kpke_seal(a:aead_alg,context:hpkeContext,n:nat,aad:bitstring,pt:
    bitstring) =
  let hpke_context(=S,key,base_nonce) = context in
  aead_enc(a,key,(base_nonce,n),aad,pt).

letfun hpke_open(a:aead_alg,context:hpkeContext,n:nat,aad:bitstring,ct:
    bitstring) =
  let hpke_context(=R,key,base_nonce) = context in
  aead_dec(a,key,(base_nonce,n),aad,ct).
```

5.2 Modeling ECH

This section describes the processes used in modelling TLS + ECH. Prior symbolic models of TLS cover the “most general scenario,” i.e., the scenario where most of the information is given to the attacker, when analyzing reachability properties. In the case of equivalence properties, it is important to faithfully represent all scenarios since an equivalence property may be broken, for example, by the presence of different error messages.

5.2.1 ECH Generation

This section describes the model for ECH generation. It focuses on a single function generate_client_hello_messag that models this procedure. This function takes the following arguments:

- The ECH configuration ech_conf chosen by the client.
- The domain of the client and the backend server: c_dom and i_dom.
- Three booleans use_grease, use_psk and send_kex. The first one indicates if GREASE should be used to generate a fake `ClientHello`. The second one indicates if a PSK is used and the last one indicates if the key share of the DH group are sent by the client (when set to false, it should trigger an HRR by the server).
- The cipher suite chosen by the client: The DH group g[2], the hash algorithm h_alg and the AEAD algorithm a_alg.
- Some extra fields that are appended to the `ClientHello` message i_extra and o_extra
- The HPKE context and encapsulation context and enc that are the result of a call to hpke_setup_base_S(h_alg,g,pkR,(tls_ech_str,ech_conf)) with pkR the public key associated to ech_conf. Note that the encapsulation and context must be re-used when an HRR is received otherwise privacy of the backend-end server is broken (see Section 4). As such, the function generate_client_hello_message_ech takes an integer n as argument representing whether it is the first $(n = 1)$ or second $(n = 2)$ ECH generated. As we explained in Section 5.1.3, this integer will be used when encrypting and decryption using the AEAD algorithm.

For simplicity's sake, we only consider the case where the client offers real ECH for each fresh connection, i.e., it does not use GREASE, nor does it use a PSK. The function starts by generating the DH key shares and the random for both `ClientHelloOuter` and `ClientHelloInner`.

```
(* The key share extension *)
let (i_kex_ext:keyShareExt,o_kex_ext:keyShareExt,kex_data:bitstring) =
    make_key_share_extension_ech(g,send_kex) in

(* Generate the random for the inner and outer client hello *)
let i_cr = gen_B32_random () in
let o_cr = gen_B32_random () in
```

In `ClientHelloInner`, the value of the ech_extension field is the constant ech_is_inner. As explained in the ECH specification, when no PSK is used, the extension for PSK is empty for both the inner and outer client hello. In this case, the early secret in the TLS key schedule corresponds to the zero PSK.

`ClientHelloInner` is then generated as follows:

[2]The DH group variable g refers to the group and not the generator of the group.

```
(* ClientHelloInner *)
let i_offer = nego(TLS13,h_alg,a_alg,i_kex_ext,SNI(i_dom),ech_is_inner,
    empty_psk_ext,i_extra) in
let i_client_hello = CH(i_cr,i_offer) in
```

Authenticating the `ClientHelloOuter` is done by computing the ClientHelloOuterAAD (see [ROSW21, Section 5.2]) which contains the `ClientHelloOuter` stripped of its ECH extension payload; the contents of the ECH extension other than the payload, including the encapsulated key, config ID, the HPKE cipher suite, remain.

```
letfun generate_client_hello_outer_AAD(client_hello_outer:msg) =
  let CH(r,nego(version,h_alg,a_alg,kex_ext,sni_ext,ech_ext,psk_ext,extra)) =
      client_hello_outer in
  let client_ECH(h_alg_ech,a_alg_ech,config_id,enc,payload) = ech_ext in

  (* The client_hello_outer with extension removed *)
  let client_hello_outer' = CH(r,nego(version,h_alg,a_alg,kex_ext,sni_ext,
      empty_ech_ext,psk_ext,extra)) in

  client_hello_outer_AAD(h_alg_ech,a_alg_ech,config_id,enc,client_hello_outer')
      .
```

When generating `ClientHelloOuter`, a dummy `ClientHelloOuter` with an empty payload is first given to generate_client_hello_outer_AAD to produce the ClientHelloOuterAAD. It is then used to generate the final payload of the ECH extension:

```
    pkR = DeserializePublicKey(ECHConfig.contents.public_key)
     enc, context = SetupBaseS(pkR,
                               "tls ech" || 0x00 || ECHConfig)
     final_payload = context.Seal(ClientHelloOuterAAD,
                                  EncodedClientHelloInner)
```

Note that the context and encapsulated key are generated outside of the function and are given as arguments, as indicated at the beginning of this section. This translates in our model as follows:

```
let payload = kpke_seal(a_alg_ech,context,n,aad2b(client_hello_AAD),m2b(
    i_client_hello)) in
let ech_ext = client_ECH(h_alg_ech,a_alg_ech,config_id,enc,payload) in
let o_offer = nego(TLS13,h_alg,a_alg,o_kex_ext,SNI(o_dom),ech_ext,o_psk_ext,
    o_extra) in

(CH(o_cr,o_offer),i_client_hello,i_cr,o_cr,kex_data,psk_data)
```

The function generate_client_hello_message_ech returns the two `ClientHello` messages (`ClientHelloInner` and `ClientHelloOuter`), the associated random values, and some datas related to the public key shares and PSK (if available).

5.2.2 Configurable Features

A key novelty in our model is that our client and server processes are extensively configurable using two kinds of boolean flags: *global flags* enable or disable TLS features in all clients and processes, whereas *process flags* are used to enable features in specific client and server processes.

We define global flags for all the features listed in Table 1 in a configuration file that gets prefixed to each analyzed model. For example, the flag allow_HRR can be set to true to globally enable `HelloRetryRequest` in all clients and servers. We also define global flags that enable and disable ECH, PSK, tickets, etc. and analysis-related flags that disable and enable the compromise of different kinds of keys used in the model.

Each client and server process is initialized with a set of arguments that can further configure their behavior.

Client model. In order to obtain stronger security guarantees, we model both standard TLS client (process standard_client) and ECH client (process echo_client). For both clients types, we cover the cases where:

- an `HelloRetryRequest` was sent by the server (processes standard_client and echo_client).
- a `CertRequest` was sent by the server (process send_certificate_message).
- a resumption PSK is generated at the end of the session and that can be re-used in future sessions (process send_finished_message)

Contrary to `HelloRetryRequest`, generation and processing of PSK tickets, client certificate, server certificate, etc are the same between TLS client and ECH client. As such our models rely on many of the same processes for both clients which increase readability and avoid unnecessary code duplication.

An ECH client is thus parametrized by the following arguments:

```
let ech_client(id_client:idProc,use_psk,comp_psk,send_kex_c,use_grease:bool,
  c_dom,i_dom:domain,
  tls_g,backend_g:group, tls_h,backend_h:hash_alg, tls_a,backend_a:aead_alg, (*
      The cipher suite and groups for both front-end and backend *)
  ech_conf:echConfig, (* The ECH configuration used by the client *)
  i_extra:extraExt(* Additional fields for inner *)
  ) =
  ...
```

This process defines a ECH client configured with the following arguments: use_ech specifies whether the client supports ECH, use_psk enables server support for PSK-DHE, comp_psk says that the client will leak its resumption PSK to the attacker at the end of the handshake, send_kex_c says that the client will send its DH key share (setting send_kex_c to false will trigger an `HelloRetryRequest` by the server), use_grease specifies whether the client will send a fake `ClientHello`, c_dom,i_dom are respectively the client and backend-server's domain name, tls_g,backend_g, tls_h,backend_h, tls_a and backend_a define the client preferred cryptographic algorithms for the client-facing and backend server, ech_conf is the ECH configuration offered by the client, and i_extra contains additional (potentially secret) information the client can use in its `ClientHelloInner` message. The argument id_client is a flag used only to help ProVerif terminate and be more precise; it plays no part in the protocol execution.

The standard TLS client is parametrized by similar arguments:

```
let standard_client(id_client:idProc,
  use_psk,comp_psk,send_kex_c:bool,
  c_dom,s_dom:domain,
  g:group,h_alg:hash_alg,a_alg:aead_alg
  ) =
  ...
```

Server model. Contrary to the client, we only have one single process to model a server:

```
let server(
  id_server:idProc,use_ech,use_psk,req_cert:bool,
  s_dom:domain,
  tls_g:group,tls_h:hash_alg,tls_a:aead_alg, (* The group and cipher suite
      accepted by the standard TLS server or client-facing *)
  backend_g:group,backend_h:hash_alg,backend_a:aead_alg, (* The group and
      cipher suite accepted by the backend server *)
  ee_extra:extraExt (* Some extensions to be sent in the EncryptedExtension
      message *)
  ) =
  ...
```

This process defines a TLS 1.3 server configured with many arguments, most of them similar to the client's process. The main difference are the flags cert_req which says that the server should ask for a client certificate and s_extra that contains additional (potentially secret) information the server can use in its `Extensions` message.

Ciphersuite negotiation limitations. In the ECH specification, the server may accept several cipher suites and key exchange groups. However, in our model, a TLS session is encoded as if the server would only accept a unique cipher suite and a unique key exchange algorithm group. Modeling a server that accepts more than one ciphersuite or key exchange algorithm group would have been possible, yet is unnecessarily complex and dramatically increases verification time for the model.

Configuration file. As previously mentioned, a key novelty in our model is that our client and server processes are extensively configurable using the global and process flags. We regrouped the setting of these flags in a single file config.m4.pvl. For example, this file contains the following:

```
(* When 'false', an honest client will always send its key share with the group
    .
  Moreover, an honest server will never send a HRR request.*)
letfun allow_HRR = true.

...

(* Each behavior parameters has two variables. When 'set_p' is 'true' then
  the attacker chooses the value of 'p'for each single session. When 'set_p'
  is 'false' then the value of 'p' is set to 'default_p' for all sessions. *)

(* Determine if client and server want to use a psk. *)
letfun set_use_psk = true.
letfun default_use_psk = false.

...

(* Ech client *)
letfun clients_with_ech = true.
```

The flags allow_HRR and clients_with_tls are both global and indicate whether `HelloRetryRequest` is enabled and whether we consider clients with ECH. Process flags such as use_psk are defined by two variables: set_use_psk and default_use_psk. set_use_psk indicates whether the value of flag is fixed (when true) or should be given by the attacker (when false). The value of the flag is defined by default_use_psk.

The subprocess run_ech_client_use_psk below corresponds to how the client runs depending on the values of set_use_psk and default_use_psk.

```
1  let run_ech_client_use_psk(id_client:idProc,i_extra_ext:extraExt,c_dom,
       backend_dom:domain,ech_conf:echConfig,tls_a,backend_a:aead_alg,tls_g,
       backend_g:group,tls_h,backend_h:hash_alg) =
2    if set_use_psk
3    then (
4      let use_psk = default_use_psk in
5      run_ech_client_comp_psk(id_client,i_extra_ext,c_dom,backend_dom,ech_conf,
           tls_a,backend_a,tls_g,backend_g,tls_h,backend_h,use_psk)
6    )
7    else
8      in(io,use_psk:bool);
9      let () = mkprecise(bool2b(use_psk)) in
10     event Same(bool2b(use_psk));
11     run_ech_client_comp_psk(id_client,i_extra_ext,c_dom,backend_dom,ech_conf,
           tls_a,backend_a,tls_g,backend_g,tls_h,backend_h,use_psk)
12 .
```

The variables taken as arguments of run_ech_client_use_psk correspond to the values that are already defined at that stage. Line 9 and 10 are proof helpers. The use of the event Same is explained in Section 5.4.1. We refer the reader to the file modelisation_details.md for some explanations on the function mkprecise. Notice that in the **then** branch of the conditional, use_psk is set to default_use_psk whereas in the **else** branch, it is provided by an input from the attacker, i.e., **in**(io,use_psk:bool).

Main scenario example. We present here the generic scenario we use when analysing our security properties. The process all_internal_process represents the internal processes used by the client or server. The processes gen_dishonest_key and gen_honest_key correspond to the generation of dishonest and honest long-term keys. Notice on line 8 and 18 the conditional on clients_with_tls and clients_with_ech respectively that enable the presence of standard TLS clients or clients with ECH. On line 27, the process will select a configuration, if available, for the client-facing server given by the attacker and run an ECH client with this configuration.

```
1  process
2    (
3      (* Generates the keys *)
4        gen_honest_key
5      | gen_dishonest_key
6      | all_internal_processes
7    ) | (
8      if clients_with_tls then
9      (* TLS*)
10     !
11     new id_tls_client:idProc;
12     (* Domains *)
13     in(io,s_dom:domain);
14     in(io,c_dom:domain);
15
16     run_tls_client(id_tls_client,c_dom,s_dom)
17   ) | (
18     if clients_with_ech then
```

```
    (* ECH client *)
    !
    new id_client:idProc;
    (* Domains *)
    in(io,backend_dom:domain);
    in(io,frontend_dom:domain);
    in(io,c_dom:domain);

    get ech_configurations(ech_config(id,g,pkR,=frontend_dom,h_alg,a_alg),skR)
        in
    let ech_conf = ech_config(id,g,pkR,frontend_dom,h_alg,a_alg) in

    run_ech_client(id_client,empty_extra_ext,c_dom,backend_dom,ech_conf)
  ) | (
    !
    (* Server *)
    new id_server:idProc;
    in(io,s_dom:domain);
    run_server(id_server,s_dom,empty_extra_ext)
  )
```

5.3 Encoding Security Goals

We define security properties for all the authentication, confidentiality, and privacy goals. We begin with an overview of how security properties described in 3 are modelled. We then describe in detail our modelling strategy for privacy properties in 3.

5.4 Legacy properties

Security Events. To precisely model our security goals, we annotate our client and server processes with events like ClientFinished and ServerFinished (indicating the completion of the handshake), and ServerPreFinished (indicating that the server has sent its `Finished` but not received the client `Finished`). We then add events like ClientSends, ServerReceives, ServerSends , and ClientReceives that indicate sending and receiving of application data. Each event is parameterized by arguments that include all the relevant connection parameters, including the identities of the client and server, their long-term keys, session identifiers, etc.

In addition to these protocol events, we define events that mark the compromise of long-term keys at the client and server. The attacker can ask for any HPKE private key, long-term signature key, or pre-shared key stored in a database to be compromised. Before leaking the key to the attacker, we trigger an event (e.g. CompromisedLtk).

Authentication, Integrity, and Downgrade Resilience. Each of our authentication goals (CAUTH, SAUTH, INT, AGR) are stated as ProVerif queries that expresss correspondences between events at the client and server. For example, the server authentication query (SAUTH) states that every time a client issues the event ClientReceives for a message m received from server S, it must either be the case that there was a matching event ServerSends issued by the server S for message m, or else either the private key of S (sk_S) (if the connection uses certificates) or the pre-shared key between C and S ($psk_{C,S}$) (if the connection uses PSKs) must be known to the adversary. The definition of CAUTH is similar.

In comparison to the prior ProVerif model of TLS 1.3 [BBK17] our authentication goals for application data are stronger in three respects. First, we prove *injective* correspondence between events, hence proving that each client and server session maps to a *unique* event at a peer, whereas the prior analysis only proved non-injective correspondence. Second, we prove a stronger guarantee for ECH handshakes, we show that even if the private-key or PSK of the server S is compromised, we obtain authentication as long as the HPKE private key of the client-facing server F remains secret. Third, we model and prove a novel stream integrity guarantee (INT) for application data: if a client or server receives a message m with sequence number n, then the peer must have sent the same message at the same sequence number, unless one of the compromise conditions hold. Hence, the client and server agree on the sequencing of messages in each direction, no matter how many messages they send or receive. In contrast, the prior model modeled authentication only for a single message at a time.

For handshake authentication, we state similar queries but in terms of ClientFinished, ServerFinished, and. We model key and transcript integrity (AGR) and key uniqueness (UNIQ) as correspondence queries over these events. We also define a limited form of downgrade resilience (DOWN): we ask that if an uncompromised TLS 1.3 client or server finishes a handshake with an uncompromised peer, and the connection does not use ECH, then either the client or the server did not support ECH. In other words, a network attacker cannot manipulate a handshake between an ECH client and an ECH server to disable ECH.

Secrecy. Key secrecy (SEC) and forward secrecy (FS) for 1-RTT application data are written as queries over the attackers knowledge. For example, the forward secrecy query for 1-RTT application data states that if the client issues an event ClientSends at time j before sending a message m to a server S, and if the attacker obtains the message m, written **attacker**(m), then it must be the case that the corresponding long-term key of S (either sk_S or $psk_{C,S}$) was compromised at some time $i < j$, and (if the connection uses ECH) then the HPKE private key of F must also have been compromised at some time $i' < j$. This formulation of forward secrecy is quite different and more precise than the query in the prior ProVerif model [BBK17] since it now uses the recent timestamp feature of ProVerif [BCC22] instead of coarse global phases.

A second novelty in our work is that we state and prove a stronger secrecy guarantee for the handshake. All prior symbolic analyses of TLS 1.3, both in ProVerif [BBK17] and in Tamarin [CHSvdM16, CHH+17], only prove *syntactic* secrecy which means that an attacker cannot fully compute some secret, but even if the attacker computes all but one bit of the secret, this secrecy property would still hold. A stronger secrecy guarantee is to prove that the secret is *indistinguishable* from a random bitstring, but this kind of equivalence property is significantly harder to prove for symbolic tools, which is probably why it was left out in prior work. In the following section, we explain how equivalence queries are expressed in ProVerif and we details all privacy properties as well as indistinguishability.

5.4.1 Equivalence queries in ProVerif

ProVerif can prove equivalence $P \approx Q$ for processes P and Q that have the same structure and differ only in the choice of terms. These equivalence are written by means of a *biprocess* which encodes both P and Q. To express the messages that differ between P and Q, we can use the construct **diff**[M,M'] where M is the term that is used in the process P and M' is the term that is used in the process Q. ProVerif then tries to prove the equivalence $P \approx Q$ by showing that all traces of P (resp. Q) can be match in Q (resp. P) by the a trace that follows exactly the same control flow. For example, the privacy of a ballot in an electronic voting protocol can be expressed as:

$$V(sk_1, v_1) \mid V(sk_2, v_2) \approx V(sk_1, v_2) \mid V(sk_2, v_1)$$

In term of biprocess, this is encoded as the process V(sk_1,**diff**[v1,v2]) | V(sk_2,**diff**[v2,v1]).

The requirement on the same control flow can be quite restrictive as it requires properly matching sessions. For example, ProVerif would not be able to prove equivalence on the biprocess **out**(io,**diff**[a,b]) | **out**(io,**diff**[b,a]) with a,b public even though it corresponds to the equivalence $P \approx P$ with P being the process **out**(io,a) | **out**(io,b). Indeed, in the biprocess, the action **out**(io,**diff**[a,b]) indicates that it will try to match the execution of **out**(io,a) on the left with the execution of **out**(io,b) on the right, which is distinguishable since both a,b are public.

Using restrictions. In the main scenario described in Section 5.2.2, we run the standard client with the following code.

```
!
new id_tls_client:idProc;
(* Domains *)
in(io,s_dom:domain);
event Same(d2b(s_dom));
in(io,c_dom:domain);
event Same(d2b(c_dom));

run_tls_client(id_tls_client,c_dom,s_dom)
```

In our security properties, we want to consider scenarios where we can have an unbounded number of client sessions plus some additional sessions that, collectively, specifically encodes our security property. For example, for client identity (domain) privacy, we would put the previous process in parallel with the following one (as well as the process modeling the server):

```
(* TLS client with diff *)
!
(* Domains *)
in(io,ClientA:domain) [precise];
in(io,ClientB:domain) [precise];
let c_dom = diff[ClientA,ClientB] in
in(io,s_dom:domain) [precise];
event Same(d2b(s_dom));
new s:seed;
let id_client = idClientDiff(s) in

run_tls_client_diff(id_client,c_dom,s_dom)
```

The second process actually encodes client privacy, since we compare one client process with domain ClientA that executes the handshake to another client process with domain ClientB , denoted as **diff**[ClientA,ClientB]. However running only the second process would not be satisfactory as we have some additional assumptions that must be satisfied by the sessions of the second process, e.g. single-use PSK, that we do not want to apply to all sessions. However, for this process, we want to ensure that it is the same client that executes the process (likewise for the server). This is done in ProVerif by the means of **restriction**:

```
restriction x,x':bitstring; event(Same(diff[x,x'])) ==> x = x'.
```

This restriction indicates that ProVerif will only look at executions such that when an event Same is executed then the value on the left side should be the same as the value on the right side. In our use-case, the event of the first process **event** Same(d2b(s_dom)) indicates that the session of the standard client will be executed with the same server domain s_dom on both sides of the equivalence. The function d2b is simply a type converter function from domain to bitstring.

This allows us to compare scenarios concurrently as follows:

$$H(c_1, s_1, p_1) \mid \ldots \mid H(c_n, s_n, p_n) \mid \ H(A, S, p_{n+1}) \mid H(c_{n+2}, s_{n+2}, p_{n+2}) \mid H(c_{n+3}, s_{n+3}, p_{n+3}) \mid \ldots$$

and

$$H(c_1, s_1, p_1) \mid \ldots \mid H(c_n, s_n, p_n) \mid \ H(B, S, p_{n+1}) \mid H(c_{n+2}, s_{n+2}, p_{n+2}) \mid H(c_{n+3}, s_{n+3}, p_{n+3}) \mid \ldots$$

where $H(c, s, p)$ is the handshake between client c and server s with parameters p. Here A and B are honest clients and S is an honest server.

5.4.2 Indistinguishability between PSK and random

In the prior ProVerif model of TLS [BBK17], only PSKs that were freshly generated as random could be used between clients and servers. These typically correspond to external PSKs established out-of-band. However, at the end of a handshake, the server can send a `SessionTicket` message, as desribed in 2. Prior models did not cover the cases where clients and servers used resumption PSKs. In our model, we show that these PSK are indistinguishable from a randomly generated value.

Prior analyses also considered the cases were some PSKs may be compromised. Thus, we would ideally also prove that compromised PSKs derived from a ticket are also indistinguishable from a random value. This is however not the case in all scenarios: If the client runs a handshake with the attacker that impersonates a server, due to a compromised external PSK or compromised long term key, the resumption PSK obtained during this session is not indistinguishable from random. So we only check the indistinguishability of resumption PSKs from randomly generated values for those that come from sessions where either the PSK used in the session was not compromised or the server long term key was not compromised.

More generally, consider a sequence of sessions:

$$\xrightarrow{psk_0} H(c, s, p_1) \xrightarrow{psk_1} H(c, s, p_2) \xrightarrow{psk_2} \ldots \xrightarrow{psk_{n-1}} H(c, s, p_{n-1}) \xrightarrow{psk_n} \ldots$$

where $H(c, s, p_i) \xrightarrow{psk_i} H(c, s, p_{i+1})$ stands for the i-th handshake generating a ticket whose derived PSK is psk_i and that PSK is then used in the $i + 1$-th handshake with the same client and server. We can show that for all i, psk_i is indistinguishable from a random if all psk_j, $j < i$ are not *directly compromised* by the attacker (psk_i may be directly compromised however). *Directly compromised* means here that the honest process directly output the PSK to the attacker and so the key become trivially known to the attacker. In fact, we show that if a PSK psk_i is known to the attacker then either it was directly compromised in the session $H(c, s, p_i)$ or some previous PSK, i.e., psk_j for $j < i$ was directly compromised.

As previously mentioned, at the end of an handshake, the server may send a `SessionTicket` message that associates a ticket value and a resumption PSK. In particular, the `SessionTicket` message contains a ticket_nonce that is unique per ticket and a value ticket that will be used as the identity for the PSK. The PSK is generated from ticket_nonce and the resumption_master_secret generated during the handshake, as described below:

```
The PSK associated with the ticket is computed as:
 HKDF-Expand-Label(resumption_master_secret,
       "resumption", ticket_nonce, Hash.length)
```

In practice, the ticket itself is an opaque label and may be either a database lookup key or a self-encrypted and self-authenticated value [Res18]. Hence, the server should be able to retrieve the PSK from its identity and additional information such as the ciphersuite used. We abstract this by relying on a private function mk_idpsk as follows:

```
new ticket_nonce;
let new_psk = hkdf_expand_label(a_alg,rms,tls13_resumption,ticket) in
let ticket_id = mk_idpsk(c_dom,s_dom,h_alg,new_psk) in
```

The private function mk_idpsk takes as argument the client and server domain, hash algorithm, and PSK. We additionally provide private projection functions get_receiver_psk, get_sender_psk, get_hash_psk and get_ipsk that respectively allow one to retrieve the server domain, client domain, hash algorithm, and PSK. For instance:

```
reduc forall c_dom,s_dom:domain, h_alg:hash_alg, ipsk:internal_preSharedKey;
  get_ipsk(mk_idpsk(c_dom,s_dom,h_alg,ipsk)) = ipsk [private].
```

After receiving the values ticket_nonce and ticket_id from the server, the client generates the PSK and should add it in a table pre_shared_keys for global storage:

```
let NewSessionTicket(ticket_nonce:bitstring,ticket_id:identityPsk) = msg in
let new_psk = hkdf_expand_label(h_alg,rms,tls13_resumption,ticket) in
...
insert pre_shared_keys(c_dom,s_dom,h_alg,ticket_id,new_psk',session_client,
    is_safe)
```

We rely on a global table pre_shared_keys to client side PSKs. This table takes several arguments: c_dom and s_dom are respectively the domains of the client and server during the handshake. Thus, later on when a client with domain c_dom wishes to retrieve a PSK it shares with a server with domain s_dom, it will look up in the table the entry starting with c_dom and s_dom. Recall that only honest participant may access the content of tables, not the attacker. h_alg represents the hash algorithm used during the handshake. ticket_id is the PSK identity. new_psk is the PSK.

The argument session_client is not directly part of the TLS protocol but helps ProVerif prove the query. It is a nonce that is generated by a client process but never used in the messages exchanged between the client and server. It helps assocaited a PSK with the session from which it was derived.

Finally, the argument is_safe indicates whether or not this PSK is directly compromised or derived from a previous directly compromised PSK. The following listing shows how is_safe is computed and resulting PSKs are inserted into the pre_shared_keys table.

```
let NewSessionTicket(ticket_nonce:bitstring,ticket_id:identityPsk) = msg in
let new_psk = b2ipsk(hkdf_expand_label(h_alg,rms,tls13_resumption,
    ticket_nonce)) in

let (new_psk':internal_preSharedKey,is_safe:bool) =
  if s_pkey = NoPubKey || uncompromised_privkey(s_pkey) then
  if psk = NoPSK || safe_psk then
    new rand_ipsk[]:internal_preSharedKey;
    (diff[rand_ipsk,new_psk],not(comp_psk))
  else (new_psk,false)
  else (new_psk,false)
in
insert pre_shared_keys(c_dom,s_dom,h_alg,ticket_id,new_psk',session_client,
    is_safe);
if comp_psk
```

```
then out(io ,new_psk')
```

The variable s_pkey can either be instantiated by NoPubKey or by the public key of the server. Typically, if during the handshake the server sent its certificate then s_pkey records the public key that were used to verify the certificate. Otherwise, when a PSK is used, s_pkey records NoPubKey. Similarly, the variable psk is either instantiated by the PSK used in the handshake or NoPSK otherwise. If a PSK was used during the handshake then the client must have retrieved it from the global table pre_shared_keys. In that case, the variable safe_psk records whether this key was directly compromised. Finally the variable comp_psk indicates whether or not the attacker chose to compromise the pre-shared generated in the handshake. Thus, the branching condition

```
if s_pkey = NoPubKey || uncompromised_privkey(s_pkey) then
   if psk = NoPSK || safe_psk then
     new rand_ipsk[]:internal_preSharedKey;
     (diff[rand_ipsk,new_psk],not(comp_psk))
   else (new_psk,false)
   else (new_psk,false)
```

indicates that the generated PSK is considered directly compromised if (i) the handshake did not use a PSK but relied on compromised certificate long term key, (ii) the handshake used a PSK that was itself compromised, or (iii) the attacker decided to compromised the generated PSK. When the PSK is not directly compromised, we model our equivalence by generating a fresh random rand_ipsk used on the right side of the equivalence while the left side uses the real generated PSK. Compromise is modelled by directly outputting the PSK value on the public channel.

General scenario The main process is the same scenario as in Section 5.2.2. Notice that the attacker can choose for each session of the standard and ECH client whether or not it will compromise the resumption PSK generated during the handshake. This is controlled by the flag comp_psk.

5.4.3 Anonymity of the backend server

Backend server anonymity states that an attacker cannot know which backend server a client performed a handshake with upon using ECH. If we denote $H_e(c_i, fs_j, bs_k)$ as a handshake between client c_i, client-facing server fs_j, and backend server bs_k, we model this form of anonymity by comparing scenarios of the following form:

$$H_e(c_1, fs_1, bs_1) \mid \ldots \mid H_e(c_n, fs_n, p_n) \mid \ H_e(A, fs^*, BS_1) \mid \ldots$$
$$\text{and}$$
$$H(c_1, fs_1, bs_1) \mid \ldots \mid H_e(c_n, fs_n, p_n) \mid \ H_e(A, fs^*, BS_2) \mid \ldots$$

where the ECH key of the client-facing server fs^* is uncompromised. In our ProVerf model, this corresponds to adding in parallel to the processes of the main scenario described in Section 5.2.2 the following process where frontend_dom is the client-facing server fs^* and BackendA ,BackendB are the backend servers BS_1 and BS_2.

```
!
new id_client:idProc;
(* Domains *)
in(io ,frontend_dom:domain) [precise];
```

```
  event Same(d2b(frontend_dom));
  in(io,c_dom:domain) [precise];
  event Same(d2b(c_dom));
  new s:seed;
  let id_client = idClientDiff(s) in

  get ech_configurations(ech_config(id,g,pkR,=frontend_dom,h_alg,a_alg),skR) in
  let ech_conf = ech_config(id,g,pkR,frontend_dom,h_alg,a_alg) in
  event ConfigOffered(ech_conf);

  run_ech_client_diff(id_client,empty_extra_ext,c_dom,diff[BackendA,BackendB],
     ech_conf)
```

Notice that this process models backend server anonymity with the equivalence **diff**[BackendA,BackendB]. This equivalence depends on the following assumption.

Assumption Anonymity Backend 1. *The client-facing server's HPKE private key used for its ECH configuration ech_conf is uncompromised, whereas any backend server keys can be compromised.*

The table ech_configurations contains the ECH configurations associated with a client-facing server. They are generated by the following process which allows one server to have multiple ECH configurations, each with a different key.

```
let gen_honest_ech_config =
  !
  in(io,(s_dom:domain,g:group,h_alg:hash_alg,a_alg:aead_alg));
  if allow_weak_ciphersuite_and_group || is_strong_hash(h_alg) then
  if allow_weak_ciphersuite_and_group || is_strong_aead(a_alg) then
  if allow_weak_ciphersuite_and_group || is_strong_group(g) then
  event Same(d2b(s_dom));
  event Same(g2b(g));
  event Same(h2b(h_alg));
  event Same(a2b(a_alg));
  new id:configId;
  let (skR:bitstring,pkR:element) = dh_keygen(g) in
  let config = ech_config(id,g,pkR,s_dom,h_alg,a_alg) in
  insert ech_configurations(config,skR);
  (* The configuration is given to the attacker. *)
  out(io,config)
.
```

When allowing compromised ECH keys, we naturally output the private key as well as the configuration. However, for the purpose of our query, we need to assume that the HPKE key of the configuration used in the session with BackendA and BackendB are not compromised. We ensure it by emitting an event **event** CompromisedEchKeyForEquiv(config); when we generate compromised configurations and by adding the following restriction:

```
restriction id_c:idProc,config:echConfig;
  event(CompromisedEchKeyForEquiv(config)) && event(ConfigOffered(id_c,config))
      ==> false.
```

Notice that the event ConfigOffered is executed only on the targeted sessions. Therefore, this restriction ensures that we do not consider traces where the configuration offered during these sessions are compromised.

Managing long-term certificate keys using restrictions When trying to prove anonymity of the backend server, we are faced with some difficulties related to the management of certificate long term keys as well as PSKs. We will describe these issues and how we solve them. We will only focus on long-term certificate keys as the issues and solutions for PSKs are very similar.

In the case of a handshake where PSKs are not used (or not accepted), the server must send a `Certificate` and `CertVerify` message (see Section 4.4.2 and 4.4.3 in [Res18]). Both messages require the server to retrieve its private key and its valid certificate from the table long_term_keys. In our model, this corresponds to the follow code:

```
1   get long_term_keys(=s_dom,sk,s_pkey,crt,idP) in
2
3   (* The Certificate message *)
4   let certificate_msg = CRT(zero,crt) in
5   let encrypted_certificate_msg = aead_enc(a_alg,shk,zero,zero,m2b(
        certificate_msg)) in
6   out(io,encrypted_certificate_msg);
7   let cur_log_CRT = (cur_log,certificate_msg) in
8
9   (* The CertificateVerify message *)
10  let signed_log = sign(sk,hash(h_alg,cur_log_CRT)) in
11  ...
```

The variable s_dom is already instantiated by the server's domain executing this part of the process. Hence at line 1, **get** long_term_keys(=s_dom,sk,s_pkey,crt,idP) indicates that the process will retrieve from the table long_term_keys an entry whose first argument is equal to s_dom and instantiate sk,s_pkey,crt,idP accordingly.

To prove backend server anonymity, we want to match sessions where BackendA is the domain of the server on one side and BackendB is the domain of the server on the other side. However, in our model, we let the attacker generate the keys for the servers (honest servers included). Thus, as is, a trivial attack on the equivalence would be to consider a trace where the attacker generates a valid certificate for BackendA but not for BackendB and then run one handshake between a server and the client with **diff**[BackendA,BackendB] as backend-server. When the server playing the role of BackendA tries to retrieve its valid certificate and long term key, it will succeed on the left side, but fails on the right side with BackendB as no key exists for BackendB. This leads to the following assumption.

Assumption Anonymity Backend 2. *We only consider scenarios where either no long term key have been generated for both BackendA and BackendB or at least one long term key has been generated for each before executing the handshake with **diff**[BackendA,BackendB] as backend-server.*

Even with this assumption, we are left with a problem related to ProVerif encoding. When the attacker generates one valid certificate for both BackendA and BackendB, the process in Section 5.1.2 is executed twice. This leads to the following two entries in the table long_term_keys:

- long_term_keys(BackendA,sk_hA,pk(sk_hA),crtA,idPA)
- long_term_keys(BackendB,sk_hB,pk(sk_hB),crtB,idPB)

However, to pass the lookup **get** long_term_keys(=s_dom,sk,s_pkey,crt,idP) in the server's process, ProVerif would expect an entry where the first argument is **diff**[BackendA,BackendB]. This is not the case despite the fact that keys for both BackendA and BackendB exist in the table at the moment of the execution of the server's process, meaning that in the formal calculus such entry could be computed.

We solve this problem by adding a process that creates new entries in long_term_keys by swapping existing entries.

```
1 let swap_long_term_keys =
2   !
3   get long_term_keys(dom1,sk1,pk1,cert1,idP1) in
4   get long_term_keys(dom2,sk2,pk2,cert2,idP2) in
5   insert long_term_keys(diff[dom1,dom2],diff[sk1,sk2],diff[pk1,pk2],diff[cert1,
        cert2],diff[idP1,idP2]);
6   insert long_term_keys(diff[dom2,dom1],diff[sk2,sk1],diff[pk2,pk1],diff[cert2,
        cert1],diff[idP2,idP1])
7 .
```

By looking up long_term_keys(BackendA,sk_h_1,ok(sk_h_1),crt1,idP1) in line 3 and long_term_keys (BackendB,sk_h_2,ok(sk_h_2),crt2,idP2) in line 4, the process will add the following new entries in the table:

- long_term_keys(**diff**[BackendA,BackendB],**diff**[sk_hA,sk_hB],**diff**[pk(sk_hA),pk(sk_hB)],**diff**[crtA,crtB],**diff**[idPA,idPB])
- long_term_keys(**diff**[BackendB,BackendA],**diff**[sk_hB,sk_hA],**diff**[pk(sk_hB),pk(sk_hA)],**diff**[crtB,crtA],**diff**[idPB,idPA])

In particular, the first entry will pass the lookup **get** long_term_keys(=**diff**[BackendA,BackendB],sk, s_pkey,crt,idP) in the server's process. Note that even if we add new entries in term of biprocess, the entries on the left and right sides respectively of the biprocess remain unchanged.

Adding this swapping process is unfortunately not enough. To prove equivalence, ProVerif will check that all entries of the table that satisfies the equality test on the left side also satisfies the equality test on the right side (and vice-versa). So only adding our new swapped entries is not sufficient as the entries long_term_keys(BackendA,sk_hA,pk(sk_hA),crtA,idPA) and long_term_keys(BackendB,sk_hB,pk(sk_hB),crtB,idPB) will still succeed on one side but not on the other. To solve this problem, we remove the equality test and encode it using a restriction that ensures the long term key retrieved from the table matches that of the server process running. Removing the restriction is done by modifying the process as follows:

```
  get long_term_keys(s_dom',sk,s_pkey,crt,idP) in
  event Selected_lgt(s_dom',s_dom,idP);

  (* The Certificate message *)
  let certificate_msg = CRT(zero,crt) in
  ...
```

With this change, all entries of the table will pass the long term key lookup. To properly select the correct entry, we emit the event Selected_lgt(s_dom',s_dom,idP) that contains the targeted domains s_dom, the domains in the entry s_dom', and the process identifiers idP of the entry. The selection of the correct entry is done by the following restriction:

```
restriction
  dom1,dom1',dom2,dom2':domain,idP,idP':idProc;
  event(Selected_lgt(diff[dom1,dom1'],diff[dom2,dom2'],diff[idP,idP'])) ==>
    (* The entries taken from the table should correspond to the requested
        domains *)
```

```
 5      dom1 = dom2 &&  (* On the left side *)
 6      dom1' = dom2' && (* On the right side *)
 7      (
 8        (* If the requested domains on the left and on the right are the same, we
              use the same idProc to match them. *)
 9        (dom1 = dom1' && idP = idP') || (dom1 <> dom1')
10      )
11  .
```

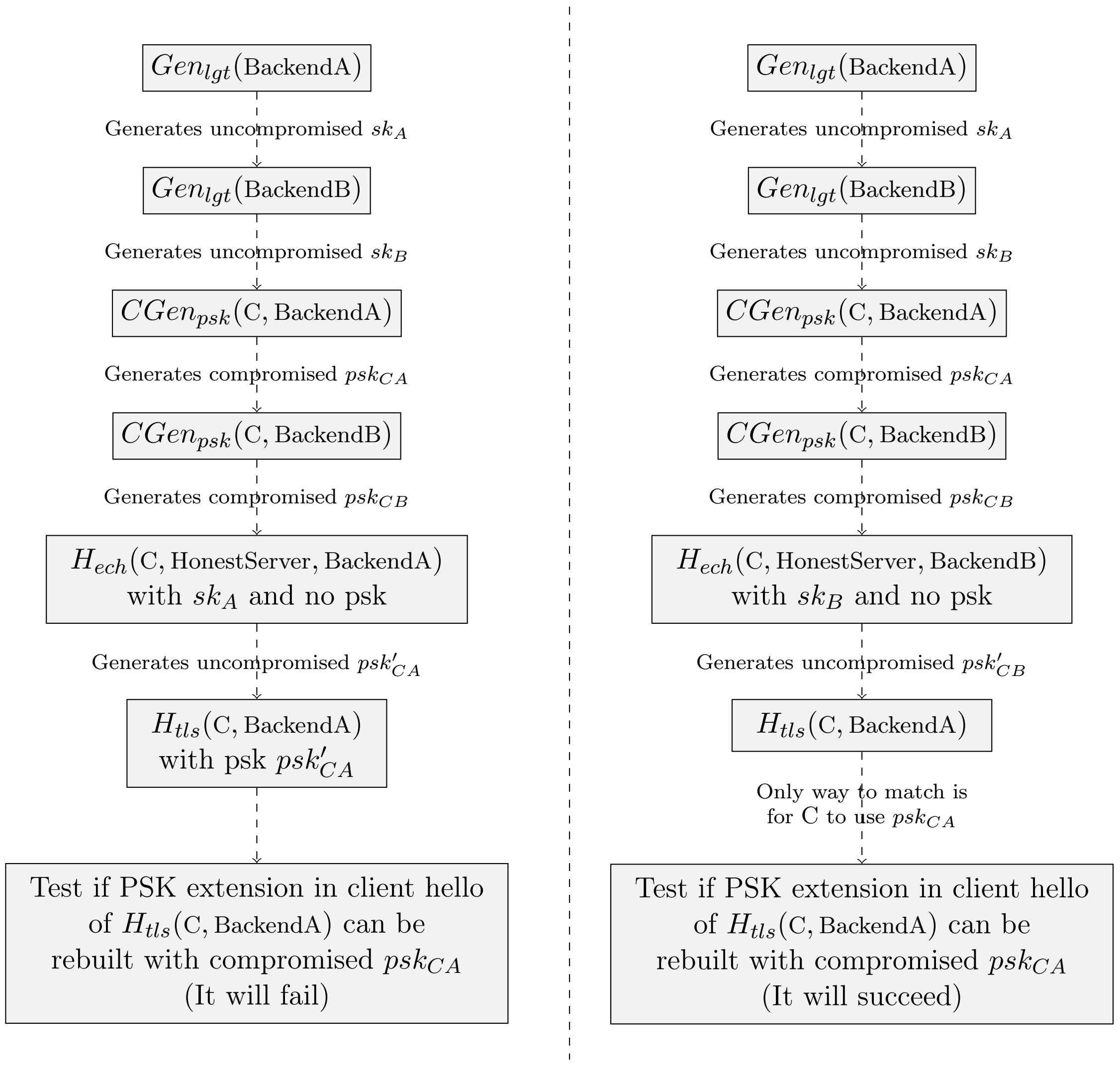
Line 5 and 6 of the restriction correspond to the fact that we want our entries to match the requested domains. Line 10 indicates that when the requested domain are the same (typically when it's not a backend server **diff**[BackendA,BackendB]), we can consider the long term key that were generated at the same moment on the left and ride side (represented by the fact that the process identifier idP and idP' are equal). This last line is added as a consequence of our swapping process generating too many unecessary new entries. For example, if the attacker decides to generate two long term keys for a server S by executing twice the process described in Section 5.1.2 for S, then two different entries with S as domain and with two different process identifiers will be generated.

- long_term_keys(S,sk_h1,pk(sk_h1),crt1,idP1)
- long_term_keys(S,sk_h2,pk(sk_h2),crt2,idP2)

When the requested domain is S on both sides, ProVerif should consider both entries. However, with the swapping process, an entry long_term_keys(S,**diff**[sk_h1,sk_h2],**diff**[pk(sk_h1),pk(sk_h2)], **diff**[crt1,crt2],**diff**[idP1,idP2]) will also be generated. Verifying the execution with this new entry is unnecessary as we already have a way to match the long term keys through the initial entries. It is critical to note that this solution with the swapping process only works when assuming Assumption Anonymity Backend 2.

As previously mentioned, we have similar issues and solutions for managing PSKs since a client will retrieve the PSKs with a server from a global table pre_shared_keys. At such we will only consider scenarios where either no PSKs have been generated between the client and both BackendA and BackendB, or at least one PSK has been generated for each before executing the handshake. However, this not sufficient as we can find another trivial attack even within the confines of this assumption. Consider the following scenario:

Assume that the attacker starts by generating a long-term certificate key for both BackendA and BackendB. Then it generates a compromised PSK between BackendA and a client C and a compromised PSK between BackendB and C. The attacker follows up by running a handshake between an ECH client C with backend **diff**[BackendA,BackendB] such that no PSK is used, in which case the server sends the Certificate and CertificateVerify messages). Notice that this scenario satisfy our assumptions. At the end of this handshake, on the left side, the client C will share a new PSK with BackendA and on the right side, it will share a new uncompromised PSK with BackendB. The attacker runs a new TLS handshake between C and BackendA on both side. On the right side, the client C only has access to a compromised PSK with BackendA whereas on the left side the second handshake will be run with the uncompromsied PSK obtained at the last handshake. This will lead the attacker to distinguish the two situations as it is unable to rebuild the PSK extension from the `ClientHello` message on the left side but it will succeed on the right side. These two scenarios are summarized below.



Gen_{lgt}(BackendA)
Generates uncompromised sk_A
Gen_{lgt}(BackendB)
Generates uncompromised sk_B
$CGen_{psk}$(C, BackendA)
Generates compromised psk_{CA}
$CGen_{psk}$(C, BackendB)
Generates compromised psk_{CB}
H_{ech}(C, HonestServer, BackendA) with sk_A and no psk
Generates uncompromised psk'_{CA}
H_{tls}(C, BackendA) with psk psk'_{CA}
Test if PSK extension in client hello of H_{tls}(C, BackendA) can be rebuilt with compromised psk_{CA} (It will fail)
Gen_{lgt}(BackendA)
Generates uncompromised sk_A
Gen_{lgt}(BackendB)
Generates uncompromised sk_B
$CGen_{psk}$(C, BackendA)
Generates compromised psk_{CA}
$CGen_{psk}$(C, BackendB)
Generates compromised psk_{CB}
H_{ech}(C, HonestServer, BackendB) with sk_B and no psk
Generates uncompromised psk'_{CB}
H_{tls}(C, BackendA)
Only way to match is for C to use psk_{CA}
Test if PSK extension in client hello of H_{tls}(C, BackendA) can be rebuilt with compromised psk_{CA} (It will succeed)

We therefore consider the following assumption for the PSKs.

Assumption Anonymity Backend 3. *We only consider scenarios where either no PSKs have been generated between a client and both BackendA and BackendB or at least one PSK been generated for each before executing the handshake with* ***diff****[BackendA,BackendB] as backend server. Moreover, we do not compare scenarios where a client would share an uncompromised PSK with a server on one side but none on the other side.*

5.4.4 Anonymity and unlinkability of the TLS and ECH client

Anonymity and unlinkability are typically encoded differently since they do not express the same security property. A protocol may preserve client anonymity, i.e., the attacker cannot deduce which client executed the handshake, but not unlinkability, as the attacker could still understand that two handshakes have been executed by the same client, albeit without knowing who that client is. However, in our model, we manage to encode both properties within the same equivalence by adding in parallel to the processes of the main scenario, described in Section 5.2.2, the following process:

```
(* TLS client with Diff *)
!
(* Domains *)
in(io,ClientA:domain) [precise];
in(io,ClientB:domain) [precise];
let c_dom = diff[ClientA,ClientB] in
in(io,s_dom:domain) [precise];
event Same(d2b(s_dom));
new s:seed;
let id_client = idClientDiff(s) in

run_tls_client_diff(id_client,c_dom,s_dom)
```

Intuitively, for each session of the client, we let the attacker choose two client domains, clientA and clientB, and we will execute on one side of the equivalence clientA and on the other side clientB. This naturally encodes anonymity as the attacker is not able to distinguish the two situations despite choosing itself the client domains. The strength of this encoding is that it also encodes unlinkability. Indeed, unlinkability is satisfied when the attacker is not able to distinguish the situation where all client domains are used in a single handshake from the situation where client domains can be used in multiple handshakes. This is covered by our equivalence as it considers executions where the attacker chooses for each session a different clientA and whereas it can choose for clientB the same client through multiple sessions.

Note that the above process considers clients running the standard TLS. We have a similar encoding for clients running ECH.

Assumptions on the keys. First, we start with a simple assumption on server HonestServer keys.

Assumption Anonymity/Unlinkability Client 1. *We assume that the long-term certificate key of the server s_dom used in these targeted sessions are not compromised.*

Without this assumption, the equivalence would be broken as the attacker could masquerade as the server using the compromised key and directly observe the client identity during the handshake. We achieve this by relying on the process identifier that is given to the client process and the following restriction:

```
restriction s:seed,s_pkey:pubkey;
  event(Client_Cert_server(idClientDiff(s),s_pkey)) && event(
      CompromisedLtkForEquiv(s_pkey)) ==> false.
```

The process identifier idClientDiff is only used for the targeted sessions and the event Client_Cert_server is executed when the client receives the `Certificate` message from the server, which contains the server long term public key.

Second, for the same reason as in the anonymity of the backend server, we discard scenario where clientA would have a certificate long term key but not clientB since, in the scenario where the server requests a client certificate, clientA would be able to produce such certificate but not clientB.

Assumption Anonymity/Unlinkability Client 2. *We only consider scenarios where either no long term key have been generated for both clientA and clientB chosen by the attacker or at least one long term key has been generated for each before executing the handshake with* ***diff****[clientA, clientB] as client.*

We consider a similar assumption for PSK.

Assumption Anonymity/Unlinkability Client 3. *We only consider scenarios where either no PSKs have been generated between a server and both clientA and clientB or at least one PSK been generated for each before executing the handshake with **diff**[clientA, clientB] as clients. Moreover, we do not compare scenarios where a client would share an uncompromised PSK with a server on one side but none on the other side.*

To prove our equivalence, and therefore unlinkability, we consider an additional assumption on the PSKs. Namely, that PSKs used in handshakes with **diff**[clientA,clientB] as clients are used at most once. This is necessary because unlinkability would be trivially broken by the attacker linking two sessions of the same client by observing the identity of the PSK in the `ClientHello` messages. Note that this assumption is necessary for TLS handshake but not always for ECH handshake, since the PSK identity is encrypted using ECH. More specifically, we show that the *single-use* assumption is necessary only for the ECH handshakes with **diff**[clientA,clientB] as clients that use GREASE (as they in fact plays as a standard TLS handshake). In contrast, if the ECH handshakes never use GREASE (or fake ECH), then PSKs may be used multiple times. For similar reasons, we also assume that the PSKs used in the handshakes with **diff**[clientA,clientB] as clients are uncompromised (otherwise the attacker would be able to distinguish the scenarios by rebuilding the PSK extension from the `ClientHello` messages).

Assumption Anonymity/Unlinkability Client 4. *We only compare scenarios where the PSKs in the TLS or ECH with GREASE handshakes with **diff**[clientA, clientB] as clients are* single-use *and uncompromised.*

This assumption is encoded in our model by first adding a new event Selected_otu_psk when retrieving the PSKs from the table pre_shared_keys as follows:

```
get pre_shared_keys(c_dom',=s_dom,=h_alg,id,internal_psk,idP',is_safe) in
event Selected_psk(c_dom,c_dom',idP',is_safe);
event Selected_otu_psk(idP,internal_psk,is_safe);
```

The first argument of Selected_otu_psk is the identifier process idP of the executed process (i.e. the client's process). The second argument is the PSK that was selected internal_psk and the third argument is whether this PSK is supposed to be uncompromised or not, i.e. is_safe (see Section 5.4.2 for how this value is built). Thus when an event Selected_otu_psk(idP,psk,is_safe) is executed, the result is a session of client whose process identifier is idP has selected the PSK psk.

Using this event, we can encode the single-use assumption using the following restriction:

```
restriction
  ipsk,ipsk1,ipsk2:preSharedKey,idP:idProc,s:seed,safe,safe1,safe2:bool;
  (* Single-use property *)
  event(Selected_one_time_used_psk(idClientDiff(s),diff[ipsk,ipsk1],diff[safe,
      safe1])) &&
  event(Selected_one_time_used_psk(idP,diff[ipsk,ipsk2],diff[safe,safe2])) ==>
      idP = idClientDiff(s);
  event(Selected_one_time_used_psk(idClientDiff(s),diff[ipsk1,ipsk],diff[safe1,
      safe])) &&
  event(Selected_one_time_used_psk(idP,diff[ipsk2,ipsk],diff[safe2,safe])) ==>
      idP = idClientDiff(s);
  (* Ucompromised property *)
```

```
9    event(Selected_one_time_used_psk(idClientDiff(s),diff[ipsk1,ipsk2],diff[safe1
        ,safe2])) ==> safe1 = true && safe2 = true
10   .
```

Recall that the identifier process idClientDiff(s) is only generated for sessions where the clients are **diff**[client_A,client_B]. Thus, Line 4 indicates that a PSK used by clientA in one of these sessions cannot be used in any other handshake. Line 7 is the symmetric property for clientB. Finally, Line 9 restricts the PSKs using in these handshakes to be uncompromised.

5.4.5 Strong secrecy of all extensions in `ClientHelloInner`

Strong secrecy of the extra extension means that the attacker cannot learn information about the `ClientHelloInner` contents. If we denote $H_e(c, fs, bs, ext)$ an ECH handshake between a client c, a client-facing server fs, a backend server bs and some extra extensions ext in the `ClientHelloInner` generated by c, the strong secrecy property checks the equivalence between the following situtations:

$$H_e(c_1, fs_1, bs_1, M_1) \mid \dots H_e(c_n, fs_n, bs_n, M_n) \mid \dots$$
and
$$H_e(c_1, fs_1, bs_1, N_1) \mid \dots H_e(c_n, fs_n, bs_n, N_n) \mid \dots$$

where $M_1, M_2, \dots$ and $N_1, N_2, \dots$ are any terms that the attacker can build.

We encode this property by putting in parallel to the processes of the main scenario described in Section 5.2.2 the following process.

```
1   !
2     (* Domains *)
3     new id_client:idProc;
4     in(io,backend_dom:domain);
5     event Same(d2b(backend_dom));
6     in(io,frontend_dom:domain);
7     event Same(d2b(frontend_dom));
8     in(io,c_dom:domain) [precise];
9     event Same(d2b(c_dom));
10
11    (* The extra extension *)
12    in(io,x:extraExt);
13    in(io,x':extraExt);
14    let inner_ext = diff[x,x'] in
15
16    get ech_configurations(ech_config(id,g,pkR,=frontend_dom,h_alg,a_alg),skR) in
17    let ech_conf = ech_config(id,g,pkR,frontend_dom,h_alg,a_alg) in
18    event ConfigOffered(ech_conf);
19
20    run_ech_client_diff(id_client,inner_ext,c_dom,backend_dom,ech_conf)
```

Notice that on Lines 12 and 13, we let the attacker choose two values for the extra extension; one of them will be run on the left side of the equivalence and the other one with will run on the right side (represented by **let** inner_ext = **diff**[x,x'].

Once again, we assume that the ECH key of the frontend server for these clients is not compromised, otherwise there would be a trivial attack wherein the attacker decrypts `ClientHelloInner` and recovers its contents.

Assumption Strong secrecy Extension 1. *The client-facing server HPKE private key corresponding to its ECH configuration ech_config is not compromised.*

	Property	1-RTT	HRR	CC	PHA	PSK-DHE	TKT	0-RTT	Time
TLS	All	✓	✓	✓	✓	✓	✓	✓	10h7m
TLS + ECH	SEC, UNIQ	✓	✓	✓	✓	✓	✓	✗	2h48m
	SEC0	✓	✓	✗	✓	✓	✓	✓	55m
	FS, INT	✓	✗	✓	✗	✓	✓	✗	3h40m
	CAUTH	✓	✗	✗	✓	✓	✓	✓	2h39m
		✓	✓	✓	✗	✓	✓	✗	3h26m
	SAUTH, AGR	✓	✓	✓	✗	✓	✓	✗	3h26m
	DOWN	✓	✓	✗	✗	✓	✓	✗	34h16m

	Property	HRR	CC	PHA	PSK-DHE	TKT	Time
TLS	IND, CIP UNL, S-EXT	✓	✓	✗	✓	✓	17H15
	CIP,UNL	✓	✓	✓	✓	✗	10h10m
ECH	IND	✗	✓	✗	✓	✓	21h16m
		✓	✗	✗	✗	✓	12h47
	SIP	✗	✗	✗	✓	✓	24h27m
		✓	✗	✗	✗	✗	1h13m
	CIP, UNL	✗	✓	✗	✓	✗	21h42m
		✗	✗	✗	✓	✓	35h22m
		✗	✓	✓	✗	✗	3h27m
	S-EXT,C-EXT	✓	✓	✗	✓	✗	21h20m

Table 3: Verification Results for TLS 1.3 with ECH

5.5 ProVerif Analysis Results

Our full TLS 1.3 + ECH model totals 7600 lines. We ran it for all the properties described above for various combinations of features, resulting in 621 total runs of ProVerif, from which we selected the best combination for each property, for which ProVerif produced a result under 48 hours without running out of memory (on average limited at 100GB). The analysis was run in parallel on a Linux server with a 64-core AMD processor clocked at 3.8Ghz with 515 GB RAM.

Verifying TLS 1.3. The results are shown in Table 3. We first re-ran all the classic TLS 1.3 secrecy and authentication reachability queries on our model of TLS 1.3 without ECH. We enable all optional features supported by our model, but recall that we disable version and ciphersuite negotiation. This analysis completes in about 10 hours and validates our changes to the TLS 1.3 model of [BBK17], while providing stronger authentication guarantees, as described above.

The analysis of equivalence-based properties typically takes more time and memory, and since these properties only speak about handshake elements, we disable application data (1-RTT, 0-RTT) when analyzing them. ProVerif proves key indistinguishability, server extension privacy, client identity privacy, unlinkability and for TLS 1.3 (without ECH, PHA, or application data) in 17 hours. We also proved CIP and UNL with HRR, PHA, CC and PSK-DHE but without tickets in 10 hours.

Reachability Properties for ECH. We then verify properties of TLS 1.3 with ECH. The added complexity of ECH makes the analysis much more expensive, requiring us to selectively disable certain features to obtain proofs for various properties. ProVerif is able to prove key secrecy (SEC) and key uniqueness (UNIQ) with 0-RTT disabled. Application data has no impact on these two handshake properties, so disabling 0-RTT does not weaken this result.

Secrecy of 0-RTT data (SEC0) is then proved with client certificates (CC) disabled. Since certificates only appear after 0-RTT data has already been sent, disabling it has no impact on this property.

Forward secrecy (FS) and stream integrity (INT) for 1-RTT data are significantly more complex properties, and we can only prove them without HRR, PHA, and 0-RTT. Enabling PHA would be interesting for these properties, since PHA is interleaved with 1-RTT data, but ProVerif runs out of memory in this case.

Server authentication (SAUTH) and key and transcript agreement (AGR) are handshake properties and are proved with post-handshake (client) authentication and 0-RTT disabled, neither of which have an impact on these properties.

Client authentication (CAUTH) is proved either with client certificates (CC) and PSKs (within the handshake) or for PHA, but not both at the same time. In practice, it is likely that TLS clients will either enable CC or PHA but not both. Still, ideally we would prove this property with all three client authentication modes (CC,PSK,PHA) enabled, but ProVerif runs out of memory.

Finally, ECH downgrade resilience is proved with certificate-based client authentication (CC,PHA) and 0-RTT data disabled.

Equivalence Properties for ECH. For TLS 1.3 with ECH, PSK-DHE, tickets, and client certificates (but without HRR or PHA), key indistinguishability is proved in 21 hours. PHA has no impact on the handshake key schedule, but HRR does affect key computation. We separately prove indistinguishability for DHE handshakes with HRR, but without CC, PHA, and PSK-DHE. Put together, these results cover both certificate-based DHE handshakes and sequences of PSK-DHE handshakes with ticket-based resumption.

Server identity privacy is proved for handshakes with ECH and HRR, hence proving that ECH is not vulnerable to the attack of Figure 8. SIP is also proved for PSK handshakes handshakes with ticket-based resumption (TKT), hence proving the absence of the attack of Figure 6. Together these results cover all known privacy attacks on previous versions of ESNI and ECH.

Client identity privacy and unlinkability are proved for clients that use both PSKs and CC (but do not use HRR, PHA, or TKT.) These properties are additionally proved for handshakes that enable CC and PHA (but do not use PSK or TKT). These results cover both unauthenticated and authenticated clients that use client certificates, or PSKs, or post-handshake authentication.

Finally, server and client extension privacy are proved for handshakes with HRR, CC, and PSK-DHE but without TKT and PHA.

6 Discussion

We have systematically analyzed TLS 1.3 with the ECH extension for a series of security and privacy properties in a variety of configurations. This analysis gives us more confidence in the design of ECH and our analysis has already influenced the design of the protocol. We discovered and presented attacks to the working group, participated in the protocol design for various versions and are now in the process of presenting our findings to the group. Ours is the first formal security analysis for this privacy extension.

It is important to underline, however, that our verification results only hold in our model, and there is always a gap between a formal model and real-world deployments and concerns. Here we discuss some of these concerns and provide implementation and deployment guidelines for ECH implementers.

Network Configuration and Application Behavior. ECH does nothing to hide the IP address of the client-facing server. Thus, if client-facing servers orchestrate deployments such that each backend server is correlated or associated with a single IP address, then ECH offers little privacy benefits. Moreover, when considering the privacy of higher-level application behavior, such as a web page load which involves opening many TLS connections for subresources on a page, the *set* of IP addresses observed may leak information about any one subresource load. Patil and Borisov [PB19] analyze the effect of these "page load fingerprints" on the privacy of web browsing applications. Protecting against unique page load fingerprints requires more invasive changes either in the client network configuration, e.g., by sending all subresource connections over a VPN-like tunnel, or application-layer changes to alter what subresources are required for an application.

Do Not Stick Out. One important criteria for ESNI and ECH to be effective is that use of it does not stick out. Indeed, early deployments of ESNI have seen some blocking in certain regions [CGH19]. In order to ease ECH into deployment, it needs to be specifically designed so as to not obviously *stick out*. In other words, without prior information, an attacker should not be able to examine a TLS connection that *uses* the ECH extension and conclude that it indeed *negotiated* ECH without prior knowledge about the client-facing server or backend server. The ECH extension alone indicates *support* for the protocol, but does not indicate that ECH was actually negotiated without prior knowledge about the client-facing server configuration. Similarly, the size of the TLS handshake may leak whether ECH was negotiated. Our analysis does not attempt to model or prove this property, and we leave it for future work.

Traffic Analysis. More generally, most kinds of traffic analysis are out of scope of our model. Any attacks that rely on observing patterns of application messages, including their timing and lengths do not appear in our ProVerif analysis. Stating and proving privacy in the presence of traffic analysis is a hard but interesting problem, especially since it depends on network configuration and application behavior as described above.

ECH Implementation and Deployment Guidelines. Despite these limitations, ECH implementations should still follow some implementation and deployment guidelines to obtain the maximal privacy guarantees from ECH.

First, our analysis shows that server privacy only holds when the client uses the same set of protocol parameters (ciphersuites, versions, groups) when connecting to both S_1 and S_2, otherwise the choice of algorithms may reveal which server it is talking to. This is already the case for most web browsers that offer a standard set of ciphersuites to all websites. However, in PSK resumption handshakes, the client may only offer the ciphersuite associated with the PSK. In this case, privacy only holds if the client holds PSKs with both S_1 and S_2 and both PSKs are associated with the same protocol parameters.

Second, the SNI extension in `ClientHello` can be of variable size; so the ECH specification recommends padding this value to a multiple of 32 bytes to avoid leaking information about the server. This should be implemented carefully to ensure that S_1 and S_2 have the same padded length.

Third, client-facing servers must ensure that all backend servers supports the same set of cryptographic capabilities. This means that they all support the same set of cryptographic algorithms, and that the *wire image* of these algorithms is consistent across all servers in the anonymity set. For example, recall that the TLS AEAD and key exchange algorithms are negotiated in plaintext. If two different backend servers supported different algorithms, one could partition the anonymity set based on these visible differences.

Fourth, the backend servers should pad their `Certificate` and `CertVerify` messages to ensure that any differences in certificate chains or handshake signatures do not leak via the length of the server's encrypted response. In general, the length of the server's encrypted response must be identical for all servers within the same anonymity set.

7 Related Work and Conclusion

We already discussed related work on TLS 1.3 in Section 3. Privacy aspects of secure channel, key exchange and related protocols, beyond TLS, have been studied in the literature. Lipp et al. [LBB19] provided the first mechanized cryptographic proof of the WireGuard protocol including identity-hiding. Ramacher et al. [RSW21] defined a privacy-preserving authenticated key exchange (PPAKE) protocol and prove that IKEv2 satisfies this definition. However, despite noting the importance of encrypted SNI in TLS as a step towards making it a PPAKE, they do not further analyze encrypted SNI designs for TLS 1.3. Zhao [Zha16] defined an identity-concealed AKE (CAKE), and proposed a new cryptographic construction called *higncryption* to build it in the context of TLS 1.3. However, the CAKE definition is weaker than that of PPAKE since it does not allow one to analyze key indistinguishability separately from privacy. Dagdelen et al. [DFG$^+$13] analyze OPACITY, though their analysis is limited in that each participant has one identity per protocol run. In contrast, TLS participants often have multiple identities. Fouque et al. [FOR16] analyze *client* privacy in 3GPP, defined in terms of user identity confidentiality, service untraceability, and location untraceability. Among these notions, identity confidentiality is only relevant for the TLS handshake protocol.

More generally, privacy properties has been a major concern for several kinds of cryptographic protocols. However, the tool support for automated verification of privacy-type properties is far less mature than for authentication and confidentiality properties. Many recent works have improved support for equivalence-based properties in various tools [CKR18, CDD17, BDS15, BCC22] and these tools have been used to analyse electronic voting protocols [CGT18, CW17], RFID protocols [HBD16], 5G-AKA protocol [BDM20], etc.

For most of these analyses, the protocol under study was relatively small or simplified for the tools to conclude. To our knowledge, our work is one of the most complex model for privacy properties ever proved for a large real-world protocol using symbolic analysis. Our results are at the limits of what ProVerif was able to prove for our model and we believe that they provide a good benchmark for future improvements to ProVerif.

References

[ABD$^+$15] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel

Thomé, Luke Valenta, et al. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 5–17, 2015.

[ABF$^+$19] Ghada Arfaoui, Xavier Bultel, Pierre-Alain Fouque, Adina Nedelcu, and Cristina Onete. The privacy of the TLS 1.3 protocol. *Proceedings on Privacy Enhancing Technologies*, 2019(4):190–210, 2019.

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy (SP 2013)*, pages 526–540, 2013.

[AR00] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). In *IFIP TCS*, volume 1872 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2000.

[ASS$^+$16] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: breaking TLS using SSLv2. In *USENIX Security Symposium*, pages 689–706, 2016.

[BBB$^+$21] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 777–795. IEEE, 2021.

[BBDL$^+$15] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: taming the composite state machines of TLS. In *IEEE Symposium on Security & Privacy (Oakland)*, 2015.

[BBF$^+$16] Karthikeyan Bhargavan, Christina Brzuska, Cédric Fournet, Matthew Green, Markulf Kohlweiss, and Santiago Zanella Béguelin. Downgrade resilience in key-exchange protocols. In *IEEE Symposium on Security and Privacy (Oakland)*, pages 506–525, 2016.

[BBK17] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 483–502, 2017.

[BBLW21] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. Hybrid Public Key Encryption. Internet-Draft draft-irtf-cfrg-hpke-12, Internet Engineering Task Force, September 2021.

[BCC22] Bruno Blanchet, Vincent Cheval, and Véronique Cortier. ProVerif with lemmas, induction, fast subsumption, and much more. In *IEEE Symposium on Security and Privacy (S&P'22)*. IEEE Computer Society, 2022. To appear.

[BDF$^+$14] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE Symposium on Security & Privacy (Oakland)*, pages 98–113, 2014.

[BDLF$^+$17] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jianyang Pan, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, and Jean Zinzindohoué. Implementing and proving the tls 1.3 record layer. In *SP 2017-38th IEEE Symposium on Security and Privacy*, pages 463–482, 2017.

[BDM20] David Baelde, Stéphanie Delaune, and Solène Moreau. A method for proving unlinkability of stateful protocols. In *Proceedings of the 33rd IEEE Computer Security Foundations Symposium (CSF'20)*, pages 169–183, Virtual conference, June 2020. IEEE Computer Society Press.

[BDS15] David A. Basin, Jannik Dreier, and Ralf Sasse. Automated symbolic proofs of observational equivalence. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 1144–1155. ACM, 2015.

[BFG19] Jacqueline Brendel, Marc Fischlin, and Felix Günther. Breakdown resilience of key exchange protocols: Newhope, TLS 1.3, and hybrids. In Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part II*, volume 11736 of *Lecture Notes in Computer Science*, pages 521–541. Springer, 2019.

[BL16a] Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 456–467, 2016.

[BL16b] Karthikeyan Bhargavan and Gaetan Leurent. Transcript collision attacks: Breaking authentication in TLS, IKE, and SSH. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2016.

[Bla18] Bruno Blanchet. Composition theorems for cryptoverif and application to TLS 1.3. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 16–30, 2018.

[BSJ$^+$15] Richard Barnes, Bruce Schneier, Cullen Jennings, Ted Hardie, Brian Trammell, Christian Huitema, and Daniel Borkmann. Confidentiality in the Face of Pervasive Surveillance: A Threat Model and Problem Statement. RFC 7624, August 2015.

[CDD17] Véronique Cortier, Antoine Dallon, and Stéphanie Delaune. Sat-equiv: An efficient tool for equivalence properties. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 481–494. IEEE Computer Society, 2017.

[CGH19] Zimo Chai, Amirhossein Ghafari, and Amir Houmansadr. On the importance of encrypted-sni ({ESNI}) to censorship circumvention. In *9th {USENIX} Workshop on Free and Open Communications on the Internet ({FOCI} 19)*, 2019.

[CGT18] Véronique Cortier, David Galindo, and Mathieu Turuani. A formal analysis of the neuchatel e-voting protocol. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 430–442. IEEE, 2018.

[CHH+17] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of tls 1.3. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1773–1788, 2017.

[CHSvdM16] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *IEEE Symposium on Security and Privacy (Oakland)*, pages 470–485, 2016.

[CKR18] Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. DEEPSEC: deciding equivalence properties in security protocols theory and practice. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 529–546. IEEE Computer Society, 2018.

[CKW11] Véronique Cortier, Steve Kremer, and Bogdan Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *J. Autom. Reason.*, 46(3–4):225–259, apr 2011.

[CW17] Véronique Cortier and Cyrille Wiedling. A formal analysis of the norwegian e-voting protocol. *J. Comput. Secur.*, 25(1):21–57, 2017.

[DFG+13] Özgür Dagdelen, Marc Fischlin, Tommaso Gagliardoni, Giorgia Azzurra Marson, Arno Mittelbach, and Cristina Onete. A cryptographic analysis of opacity. In *European Symposium on Research in Computer Security*, pages 345–362. Springer, 2013.

[DFGS15] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1197–1210, 2015.

[DFGS21] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol. *J. Cryptol.*, 34(4):37, 2021.

[DFK+17] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. Implementing and proving the TLS 1.3 record layer. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 463–482, 2017.

[DFP$^+$21] Antoine Delignat-Lavaud, Cédric Fournet, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Jay Bosamiya, Joseph Lallemand, Itsaka Rakotonirina, and Yi Zhou. A security model and fully verified implementation for the IETF QUIC record layer. In *IEEE Symposium on Security and Privacy (Oakland)*, pages 1162–1178, 2021.

[DG19] Nir Drucker and Shay Gueron. Selfie: reflections on TLS 1.3 with PSK. *IACR Cryptol. ePrint Arch.*, 2019:347, 2019.

[Don17] Jason A. Donenfeld. Wireguard: Next generation kernel network tunnel. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017.* The Internet Society, 2017.

[DY06] D. Dolev and A. Yao. On the security of public key protocols. In *IEEE Trans. Inf. Theor.*, volume 29, page 198–208, 2006.

[FG17] Marc Fischlin and Felix Günther. 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 2017, Paris, France, April 26-28, 2017*, pages 60–75. IEEE, 2017.

[FOR16] Pierre-Alain Fouque, Cristina Onete, and Benjamin Richard. Achieving better privacy for the 3gpp aka protocol. *Proc. Priv. Enhancing Technol.*, 2016(4):255–275, 2016.

[HBD16] Lucca Hirschi, David Baelde, and Stéphanie Delaune. A method for verifying privacy-type properties: the unbounded case. In Michael Locasto, Vitaly Shmatikov, and Úlfar Erlingsson, editors, *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P'16)*, San Jose, California, USA, May 2016. IEEE Computer Society Press.

[HM18] Paul E. Hoffman and Patrick McManus. DNS Queries over HTTPS (DoH). RFC 8484, October 2018.

[KHN$^+$14] Charlie Kaufman, Paul E. Hoffman, Yoav Nir, Pasi Eronen, and Tero Kivinen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 7296, October 2014.

[KMO$^+$15] Markulf Kohlweiss, Ueli Maurer, Cristina Onete, Björn Tackmann, and Daniele Venturi. (de-)constructing TLS 1.3. In Alex Biryukov and Vipul Goyal, editors, *Progress in Cryptology - INDOCRYPT 2015 - 16th International Conference on Cryptology in India, Bangalore, India, December 6-9, 2015, Proceedings*, volume 9462 of *Lecture Notes in Computer Science*, pages 85–102, 2015.

[Kra03] Hugo Krawczyk. Sigma: The 'sign-and-mac'approach to authenticated diffie-hellman and its use in the ike protocols. In *Annual International Cryptology Conference*, pages 400–425. Springer, 2003.

[KW16] Hugo Krawczyk and Hoeteck Wee. The OPTLS protocol and TLS 1.3. In *IEEE European Symposium on Security & Privacy (Euro S&P)*, 2016. Cryptology ePrint Archive, Report 2015/978.

[LBB19] Benjamin Lipp, Bruno Blanchet, and Karthikeyan Bhargavan. A mechanised cryptographic proof of the wireguard virtual private network protocol. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 231–246. IEEE, 2019.

[LXZ$^+$16] X. Li, J. Xu, Z. Zhang, D. Feng, and H. Hu. Multiple handshakes security of TLS 1.3 candidates. In *IEEE Symposium on Security and Privacy (Oakland)*, pages 486–505, 2016.

[MDK14] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback. `https://www.openssl.org/~bodo/ssl-poodle.pdf`, 2014.

[PB19] Simran Patil and Nikita Borisov. What can you learn from an ip? In *Proceedings of the Applied Networking Research Workshop*, pages 45–51, 2019.

[Per18] Trevor Perrin. The Noise Protocol Framework. `http://noiseprotocol.org/noise.html`, 2018.

[Res18] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[ROSW21] Eric Rescorla, Kazuho Oku, Nick Sullivan, and Christopher A. Wood. TLS Encrypted Client Hello. Internet-Draft draft-ietf-tls-esni-13, Internet Engineering Task Force, August 2021. Work in Progress.

[RSW21] Sebastian Ramacher, Daniel Slamanig, and Andreas Weninger. Privacy-preserving authenticated key exchange: Stronger privacy and generic constructions. In *European Symposium on Research in Computer Security*, pages 676–696. Springer, 2021.

[ST16] Stefan Santesson and Hannes Tschofenig. Transport Layer Security (TLS) Cached Information Extension. RFC 7924, July 2016.

[VP15] Mathy Vanhoef and Frank Piessens. All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS. In *USENIX Security Symposium*, pages 97–112, 2015.

[Zha16] Yunlei Zhao. Identity-concealed authenticated encryption and key exchange. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1464–1479, 2016.