Unified Formal Verification of Security Requirements in the TLS Family*

Yongho Ko, Hoseok Kwon, and Ilsun You†

Dept. of Cyber Security, Kookmin University, Seoul, South Korea
{koyh0911, ilsunu}@gmail.com, hoseok1997@kookmin.ac.kr

Abstract

For the Transport Layer Security (TLS) protocols, TLS 1.2 and TLS 1.3 with various handshake options have been widely deployed and standardized by the Internet Engineering Task Force (IETF). Recently, the importance of rigorous formal verification has been emphasized, particularly in accordance with ISO/IEC 29128-1:2023, which defines the highest assurance level through unbounded security analysis. In this paper, we conduct a systematic and comprehensive verification of the TLS family, including TLS 1.2 and TLS 1.3 all of variants, using the formal verification tool ProVerif under unbounded adversarial settings. Our verification results demonstrate that the TLS family ensures confidentiality, integrity, authentication, and secure key exchange across all handshake modes. However, our analysis also reveals residual weaknesses: while Perfect Forward Secrecy (PFS) is guaranteed in (Elliptic Curve) Diffie-Hellman ((EC)DHE)-based modes, it is not preserved in Rivest-Shamir-Adleman (RSA), Pre-Shared Key (PSK)-only, or 0-RTT configurations. Moreover, all modes remain vulnerable to ClientHello replay at the initial handshake stage. Compared with earlier bounded analyses, our study provides stronger guarantees under ISO/IEC 29128-1:2023 unbounded assurance, while highlighting practical deployment risks that require mitigation.

1 Introduction

In recent years, Transport Layer Security (TLS) has become the de facto standard for securing communication across the Internet. According to the latest Secure Sockets Layer (SSL) Pulse statistics reported by the global cybersecurity company Qualys, nearly 99.9% of websites support TLS 1.2, while approximately 70.1% already support TLS 1.3 [1]. This rapid deployment underscores that TLS, in its different versions, is no longer optional but a fundamental component of modern network security.

The evolution from TLS 1.2 to TLS 1.3 reflects not only performance enhancements but also the growing need to address stronger security requirements such as Perfect Forward Secrecy (PFS), replay resistance, and robust mutual authentication. Despite the widespread adoption and standardization of TLS, rigorous formal evaluations under the most demanding verification conditions remain limited. Prior analyses frequently rely on bounded settings, which cannot fully capture adversarial capabilities in practice.

Against this backdrop, several academic efforts have attempted to formally analyze TLS protocols. Kohlweiss et al. [2] conducted one of the earliest constructive cryptography analyses of TLS 1.3 (draft-05), focusing on Pre-Shared Key (PSK) options; they identified weaknesses that later influenced the specification, but their study did not include Zero Round Trip Time (0-RTT) and did not achieve unbounded analysis. Around the same period, Cremers et al. [3] used

*Proceedings of the 9th International Conference on Mobile Internet Security (MobiSec'25), Article No. 65, December 16-18, 2025, Sapporo, Japan.

†Corresponding Author

the Tamarin prover to analyze TLS 1.3 draft-10, proving confidentiality, authentication, and PFS for the PSK option under unbounded settings, though replay attacks were not considered. Their later work [4] extended the analysis to TLS 1.3 draft-21, showing that all handshake modes satisfied unbounded properties, while replay resistance remained not fully verified.

Bhargavan et al. [5] employed ProVerif and CryptoVerif to verify TLS 1.2 and TLS 1.3 (draft-18) in an integrated Full Handshake (FH), confirming confidentiality, authentication, PFS, and replay resistance under unbounded analysis, but restricting coverage to the FH. Around the same time, Daassa et al. [6] applied AVISPA to TLS 1.2 and detected flaws such as the renegotiation attack (CVE-2009-3555 [7]) using mutation testing, yet without providing unbounded proofs. Blanchet et al. [8] later released reference ProVerif models for TLS 1.2 and TLS 1.3, enabling verification of confidentiality, integrity, and authentication; however, due to limited optimization, the verification was extremely slow and did not address PFS or replay resistance.

Subsequent works broadened the landscape from complementary angles. Dowling et al. [9] presented a computational reductionist proof in the multi-stage AKE (MSAKE) model for TLS 1.3, covering FH, PSK-Only, PSK-(Elliptic Curve) Diffie-Hellman ((EC)DHE), and 0-RTT, demonstrating confidentiality, authentication, PFS, and replay resistance—albeit outside an unbounded symbolic framework. Tran and Ogata [10] verified TLS 1.2 (Rivest-Shamir-Adleman (RSA) and PSK-Only) in CafeOBJ, confirming confidentiality and authentication under bounded analysis. More recently, Ko et al. [11] applied ProVerif to the AKMA protocol integrated with three TLS 1.3 PSK-based modes (PSK-only, PSK-(EC)DHE, and 0-RTT), proving confidentiality, integrity, mutual authentication, PFS, and replay resistance under unbounded analysis, although limited to PSK modes. Sardar et al. [12] focused on TLS 1.3 FH using ProVerif, modeling evidence freshness and server authentication to detect replay attacks under unbounded settings, but incurred high computational cost and considered only the FH.

A consolidated comparison of these studies—covering protocol/draft scope, verification tools, and achieved properties—is provided in Table 1. As summarized there, prior work either narrowed its attention to specific handshake modes or adopted bounded settings, leaving gaps in unbounded symbolic verification across the full TLS option space.

To address these gaps, this paper presents the first comprehensive unbounded ProVerif analysis of both TLS 1.2 and TLS 1.3 across all handshake modes. Through an optimized modeling approach, we systematically verify confidentiality, integrity, mutual authentication, secure key exchange, PFS, and replay resistance, while achieving significantly faster execution than prior efforts. These results reinforce TLS's essential role in securing the digital ecosystem. The contributions of this paper are as follows:

- We analyze and compare the limitations of partial and restricted verification approaches presented in prior studies.
- We provide the first systematic unbounded formal verification of all TLS 1.2 and TLS 1.3 handshake modes (FH, PSK-Only, PSK-(EC)DHE, and 0-RTT), validating core and advanced security requirements in accordance with ISO/IEC 29128-1:2023.

The remainder of this paper is organized as follows: Section 2 provides a comprehensive background analysis of TLS 1.2 and TLS 1.3. Section 3 presents the comparative analysis of these protocols through formal verification. Section 4 discusses the insights and implications drawn from the verification results. Finally, Section 5 concludes the paper and outlines directions for future research.

Table 1: Comparison of Formal Verification Studies on TLS 1.2 and TLS 1.3

#	Reference	TLS Version / Mode	Method	Verified Properties	Key Results	Limitations
1	Kohlweiss et al. (2016)	TLS 1.3 / PSK-Only, PSK-(EC)DHE	Constructive Cryptography	Key exchange security	Performed cryptographic proof for TLS 1.3.	Focused on a PSK option; unbounded properties not satisfied.
2	Cremers et al. (2016)	TLS 1.3 draft-10 / PSK-Only, PSK-(EC)DHE, 0-RTT	Tamarin	Confidentiality, PFS, Authentication	Verified TLS 1.3 PSK option under unbounded properties.	Replay attacks not analyzed; focused on PSK mode only.
3	Cremers et al. (2017)	TLS 1.3 draft-21 / All modes	Tamarin	Confidentiality, PFS, Authentication	Verified all TLS 1.3 options under unbounded settings.	Replay attack resistance not analyzed.
4	Bhargavan et al. (2017)	TLS 1.2 / RSA, TLS 1.3 / FH	ProVerif, CryptoVerif	Confidentiality, PFS, Authentication, Replay resistance	Verified integrated RSA, FH for TLS 1.2 and TLS 1.3 under unbounded properties.	Analysis limited to FH option.
5	Daassa et al. (2017)	TLS 1.2 / RSA	AVISPA	Confidentiality, Authentication	Successfully detected the Renegotiation Attack (CVE-2009-3555) using AVISPA.	Focus limited to specific attacks. Unbounded setting not explicitly verified.
6	Blanchet et al. (2018)	TLS 1.2 / RSA, TLS 1.3 / All options	ProVerif	Confidentiality, Integrity, Authentication	Provided ProVerif models for TLS 1.2 and all TLS 1.3 modes. Verified properties across handshakes.	Models not optimized, resulting in very slow verification; did not consider PFS or emerging attack vectors.
7	Dowling et al. (2021)	TLS 1.3 / FH, PSK-Only, PSK-(EC)DHE, 0-RTT	MSAKE	Confidentiality, PFS, Authentication, Replay resistance	Verified all TLS 1.3 options.	Unbounded properties not analyzed.
8	Tran & Ogata (2022)	TLS 1.2 / RSA	CafeOBJ	Confidentiality, Authentication	Verified main TLS options using automated formal tools.	Focused on RSA; unbounded properties not verified.
9	Ko et al. (2024)	TLS 1.3 / PSK-Only, PSK-(EC)DHE, 0-RTT	ProVerif	Confidentiality, Integrity, Mutual Authentication, PFS, Replay resistance	Verified AKMA protocol with three TLS 1.3 options under unbounded settings.	Focus limited to PSK mode.
10	Sardar et al. (2024)	TLS 1.3 / FH	ProVerif	Confidentiality, Integrity, Mutual Authentication, Replay resistance	Verified TLS 1.3 FH under unbounded settings, including replay attacks.	Focused on FH; ProVerif model not optimized, high resource usage.

2 Preliminaries

2.1 Notations

Table 2 shows abbreviations and notations to be used throughout this paper.

Table 2: Abbreviations and notations

	Meanings
PMS	pre-master secret
KDF	Key Derivation Function
AEAD	Authenticated Encryption with Associated Data
SAN	Subject Alternative Name
X,Y	ECDH(E) public Key
x,y	ECDH(E) private Key
Y_{next},y_{next}	ECDH(E) public, private key generated by AF for the next session
Y_{new},y_{new}	newly generated ECDH(E) public, private key from the previous session
ES	Early Secret
HS	Handshake Secret
MS	Master Secret
IKM	Input keying material
PRK	Pseudorandom key
TH	Transcript-Hash

2.2 TLS 1.2 RSA & DH

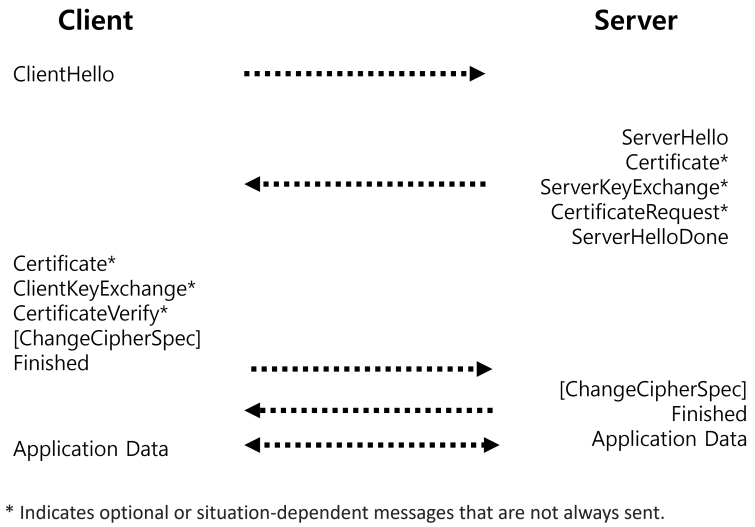

Figure 1: TLS 1.2 process

As shown in Figure 1, TLS 1.2 was designed to allow the selection of different key exchange methods depending on the implementation and operating environment, and this choice directly

affects the security properties and performance of the session. The two commonly used approaches were RSA key exchange and (EC)DHE-based key exchange.

The TLS 1.2 has the following two modes:

- **TLS 1.2 RSA** : TLS 1.2 RSA mode has a structure in which the client encrypts the session key it will send using the public key included in the server's certificate. Since this mode is structurally simple and completes with a single operation—encrypting the session key with the server's public key—it has relatively low computational complexity. It is suitable for environments that require high compatibility and fast handshakes, but it shows limitations in terms of long-term security.
- **TLS 1.2 (EC)DHE** : The TLS 1.2 (EC)DHE mode generates an ephemeral elliptic-curve key pair for each client session and performs a Diffie–Hellman exchange. Even if the long-term private key is exposed, the confidentiality of past messages is preserved. Because elliptic-curve keys provide equivalent or stronger security strength with shorter key lengths, they reduce message size and computational load, making them advantageous in terms of bandwidth and power efficiency. However, key exchange is somewhat more complex and may increase computational cost on the server side, and the implementation must adhere to recommended curves and secure random number generation.

2.3 TLS 1.3 FH & PSK-Option

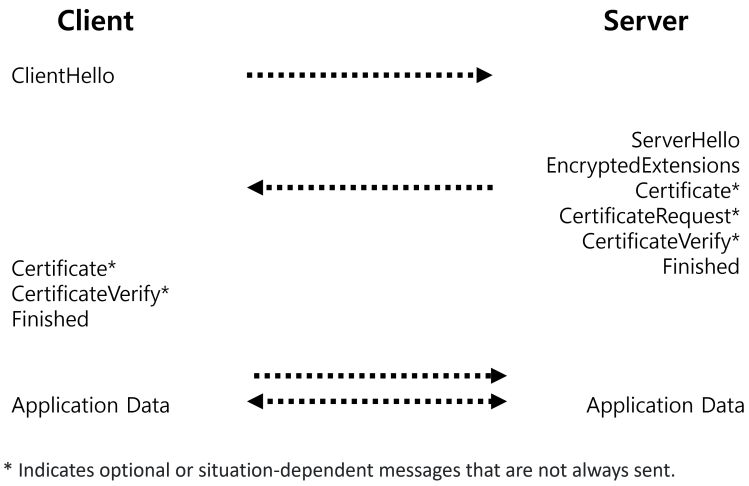

Figure 2: TLS 1.3 process

As shown in Figure 2, TLS 1.3 is divided into an FH, which establishes a secure session without relying on pre-shared keys for the initial connection, and PSK–based options for efficient setup and resumption. The protocol simultaneously achieves security and simplicity by removing legacy attack surfaces, mandating AEAD, strengthening transcript binding, and making (EC)DHE the default.

The TLS 1.3 has the following four modes:

- **TLS 1.3 FH** : TLS 1.3 FH establishes a secure channel in 1-RTT without relying on PSKs, using ephemeral (EC)DHE and server (optionally client) certificates. PFS, downgrade

resistance, and key agreement/confirmation are satisfied as baseline guarantees, making it the default choice for high-assurance segments.

- **TLS 1.3 PSK-Only** : TLS 1.3 PSK-Only is a mode in which the client and server authenticate and derive keys based on a PSK, minimizing handshake latency and computational overhead. However, because it does not use per-session ephemeral key agreement, it does not provide PFS. It is therefore suitable for resumption or short-lived sessions in environments where PSKs are strictly generated, distributed, and rotated and the risk of key compromise is low.
- **TLS 1.3 PSK-(EC)DHE** : TLS 1.3 PSK-(EC)DHE augments mutual authentication via a PSK with ephemeral (EC)DHE, thereby achieving PFS. By combining the simplicity and efficiency of PSKs with the security of (EC)DHE, it provides a practical balance that delivers low latency and strong guaranties across a wide range of deployment scenarios.
- **TLS 1.3 0-RTT** : TLS 1.3 0-RTT uses a PSK derived from a prior session to allow the client to send application data before receiving the server's response, minimizing latency. While this immediacy improves performance, the early data can be subject to replay attacks and PFS guaranties are limited. Consequently, use of 0-RTT should be carefully governed by service-specific policies (e.g., anti-replay mechanisms, disallowing sensitive operations, strict lifetime management).

3 Security Analysis

In this section, we conduct a systematic security analysis of TLS 1.2 and TLS 1.3 using the formal verification tool ProVerif. The verification process follows the ISO/IEC 29128-1:2023 standard [13], enabling mathematical proofs of protocol correctness and the discovery of potential vulnerabilities. We employ ProVerif's "unbounded" verification capability—the strongest level defined by the standard—to achieve rigorous and comprehensive evaluation; ProVerif also supports diagnosis via automatically generated attack traces and graphs.

Our ProVerif modeling proceeds in three stages. First, we specify the declarations (types, constants, variables), channels, and security queries, as shown in Algorithm 1. Next, we modularize the handshake procedures as process macros by TLS option: TLS 1.2 RSA (Algorithm 2), TLS 1.2 (EC)DHE (Algorithm 3), TLS 1.3 FH (Algorithm 4), TLS 1.3 PSK-Only (Algorithm 5), TLS 1.3 PSK-(EC)DHE (Algorithm 6), and TLS 1.3 0-RTT (Algorithm 7). This modularization yields reusable, option-specific sub-processes that align with our event instrumentation and query set (Q1–Q7 in Algorithm 1). Finally, the main process coordinates cryptographic key setup and invokes the relevant sub-processes; we provide both a baseline composition "without PFS phase output" (Algorithm 8) and a variant "with PFS phase output" (Algorithm 9) to reflect different analysis contexts.

Algorithm 1 Declaration & Queries

```
(* Channel specification *)
free c: channel.
free sp: channel [private].
(* Type specification *)
type G, exponent, label, keyid, key.
(* Application Data specification *)
free App Data1, App Data2, App Data3: bitstring [private].
(* TLS 1.3 Label specification *)
const p_b_binder, c_e_traffic : label.
```

```
(* TLS 1.2 Queries specification *)
event S_STEP1_C_to_S(bitstring, bitstring).
event E_STEP1_C_to_S(bitstring, bitstring).
event T12_S_STEP2_S_to_C(spkey, bitstring).
event T12_E_STEP2_S_to_C(spkey, bitstring).
event S_STEP3_C_to_S(bitstring, bitstring).
event E_STEP3_C_to_S(bitstring, bitstring).
event S_STEP4_S_to_C(key, bitstring).
event E_STEP4_S_to_C(key, bitstring).

(* TLS 1.3 Queries specification *)
event S_STEP1_C_to_S(keyid, key, bitstring, bitstring, bitstring).
event E_STEP1_C_to_S(keyid, key, bitstring, bitstring, bitstring).
event T13_S_STEP2_S_to_C(keyid, key, label, bitstring).
event T13_E_STEP2_S_to_C(keyid, key, label, bitstring).
event S_STEP3_C_to_S(keyid, key, label, bitstring).
event E_STEP3_C_to_S(keyid, key, label, bitstring).
```

```
(* Security requirements verification *)
query kid: keyid, psk: key, rand: bitstring, binder: bitstring, ClientHello: bitstring;
Q1: inj-event(E_STEP1_C_to_S(kid, psk, rand, binder, ClientHello))
    ==> inj-event(S_STEP1_C_to_S(kid, psk, rand, binder, ClientHello)).
    query cert_pkS: spkey, s_key_exch: bitstring;
Q2: inj-event(T12_E_STEP2_S_to_C(cert_pks, s_key_exch))
    ==> inj-event(T12_S_STEP2_S_to_C(cert_pks, s_key_exch)).
query kid: keyid, k: key, lb_finished: label, aead_finished: bitstring;
Q3: inj-event(T13_E_STEP2_S_to_C(kid, k, lb_finished, aead_finished))
    ==> inj-event(T13_S_STEP2_S_to_C(kid, k, lb_finished, aead_finished)).
query kid: keyid, k: key, lb_finished: label, aead_finished: bitstring;
Q4: inj-event(E_STEP3_C_to_S(kid, k, lb_finished, aead_finished))
    ==> inj-event(S_STEP3_C_to_S(kid, k, lb_finished, aead_finished)).
(* Secrecy verification *)
Q5: query attacker(App Data1).
Q6: query attacker(App Data2).
Q7: query attacker(App Data3).
```

Algorithm 2 TLS 1.2 RSA option

Input: server_rsa_certificate_chain, roaming_context
Output: RSA secure channel
Step 1: ClientHello Generation
  $client\_random \leftarrow \{0,1\}^{256}$
  $ClientHello \leftarrow (tls\_1.2, client\_random,$ ciphersuites incl. TLS_RSA)
  Event $S\_STEP1\_C\_to\_S(client\_random, ClientHello)$
  Send $(ClientHello)$
  Event $E\_STEP1\_C\_to\_S(client\_random, ClientHello)$
Step 2: ServerHello and Certificate
  $server\_random \leftarrow \{0,1\}^{256}$
  $ServerHello \leftarrow (tls\_1.2, server\_random,$ ciphersuite: RSA)
  Event $S\_STEP2\_S\_to\_C(server\_random, ServerHello)$
  Send $(ServerHello, Certificate, ServerHelloDone)$
  Event $E\_STEP2\_S\_to\_C(server\_random, ServerHello)$
Step 3: Certificate Verification and Key Derivation
  Verify server RSA certificate chain against configured trust anchors (root CA store)
  Validate ID in server certificate SAN records
  $PMS \leftarrow$ Random(48) with first two bytes = $0x03\ 0x03$ ▷ TLS 1.2 version in PMS
  $EncryptedPMS \leftarrow RSAES\_PKCS1\_v1\_5.Encrypt(pk\_S, PMS)$
  $master\_secret \leftarrow PRF(PMS,$ "master secret", $client\_random \| server\_random)$
  $key\_block \leftarrow PRF(master\_secret,$ "key expansion", $server\_random \| client\_random)$
  $(tk_{c\_app}, tk_{s\_app}) \leftarrow derive\_traffic\_keys\_tls12(key\_block)$
  Event $S\_STEP3\_C\_to\_S(EncryptedPMS, Finished)$
  Send $([Certificate]_{\text{if mutual auth}}, ClientKeyExchange(EncryptedPMS), [CertificateVerify]_{\text{if provided}},$
  $ChangeCipherSpec, Finished)$
  Event $E\_STEP3\_C\_to\_S(EncryptedPMS, Finished)$
Step 4: Decryption and Session Completion
  $PMS \leftarrow RSAES\_PKCS1\_v1\_5.Decrypt(sk\_S, EncryptedPMS)$
  Recompute $master\_secret$, key_block, (tk_{c_app}, tk_{s_app})
  Verify client Finished; send $(ChangeCipherSpec, Finished)$
  Event $S\_STEP4\_S\_to\_C(tk_{s\_app}, Finished)$
  Event $E\_STEP4\_S\_to\_C(tk_{s\_app}, Finished)$

Algorithm 3 TLS 1.2 (EC)DHE option

Input: supported_groups, certificate_chain, roaming_context
Output: (EC)DHE secure channel with PFS
Step 1: ClientHello Generation
  $client\_random \leftarrow \{0,1\}^{256}$
  $ClientHello \leftarrow (tls\_1.2, client\_random, supported\_groups,$ ciphersuites incl. DHE)
  Event $S\_STEP1\_C\_to\_S(client\_random, ClientHello)$
  Send $(ClientHello)$
  Event $E\_STEP1\_C\_to\_S(client\_random, ClientHello)$
Step 2: ServerHello and (EC)DHE Parameters
  $server\_random \leftarrow \{0,1\}^{256}$
  Choose group $G \in supported\_groups$; $y \leftarrow \mathbb{Z}_q$, $server\_dhe\_share \leftarrow g^y$
  $ServerHello \leftarrow (tls\_1.2, server\_random,$ ciphersuite: DHE, $G)$
  $ServerKeyExchange \leftarrow (G, g, server\_dhe\_share, \sigma\_S)$ ▷ σ_S is signature over params
  Event $S\_STEP2\_S\_to\_C(server\_random, server\_dhe\_share, ServerHello)$
  Send $(ServerHello, Certificate, ServerKeyExchange, ServerHelloDone)$
  Event $E\_STEP2\_S\_to\_C(server\_random, server\_dhe\_share, ServerHello)$
Step 3: Certificate/Param Verification and Key Exchange
  Verify server certificate chain against configured trust anchors (root CA store)
  Validate ID in server certificate SAN records
  Verify σ_S over $(G, g, server\_dhe\_share, client\_random, server\_random)$
  $x \leftarrow \mathbb{Z}_q$, $client\_dhe\_share \leftarrow g^x$
  $PMS \leftarrow (server\_dhe\_share)^x$ ▷ premaster_secret from (EC)DHE
  $master\_secret \leftarrow PRF(PMS,$ "master secret", $client\_random \| server\_random)$
  $key\_block \leftarrow PRF(master\_secret,$ "key expansion", $server\_random \| client\_random)$
  $(tk_{c\_app}, tk_{s\_app}) \leftarrow derive\_traffic\_keys\_tls12(key\_block)$
  Event $S\_STEP3\_C\_to\_S(client\_dhe\_share, Finished)$
  Send ($[Certificate]_{\text{if mutual auth}}, ClientKeyExchange(client\_dhe\_share), [CertificateVerify]_{\text{if provided}},$
  $ChangeCipherSpec, Finished)$
  Event $E\_STEP3\_C\_to\_S(client\_dhe\_share, Finished)$
Step 4: Verification and Session Completion
  Verify client certificate (if present) against configured trust anchors (root CA store)
  $PMS \leftarrow (client\_dhe\_share)^y$; recompute $master\_secret$, key_block, (tk_{c_app}, tk_{s_app})
  Verify client Finished; send $(ChangeCipherSpec, Finished)$
  Event $S\_STEP4\_S\_to\_C(tk_{s\_app}, Finished)$
  Event $E\_STEP4\_S\_to\_C(tk_{s\_app}, Finished)$

Algorithm 4 TLS 1.3 FH option (corrected)

Input: supported_groups, certificate_chain, roaming_context
Output: FH secure channel with PFS
Step 1: ClientHello Generation
  $client\_random \leftarrow \{0,1\}^{256}$
  $x \leftarrow \mathbb{Z}_q$, $client\_key\_share \leftarrow g^x$
  $ClientHello \leftarrow (tls\_1.3, client\_random, client\_key\_share, supported\_groups, extensions)$
  Event $S\_STEP1\_C\_to\_S(session\_id, client\_key\_share, client\_random, ClientHello, extensions)$
  Send $(ClientHello)$
  Event $E\_STEP1\_C\_to\_S(session\_id, client\_key\_share, client\_random, ClientHello, extensions)$
Step 2: ServerHello and Extensions
  $y \leftarrow \mathbb{Z}_q$, $server\_key\_share \leftarrow g^y$
  $server\_random \leftarrow \{0,1\}^{256}$
  $ServerHello \leftarrow (tls\_1.3, server\_random, server\_key\_share, extensions)$
  $EncryptedExtensions \leftarrow (server\_parameters, negotiated\_extensions)$
  Event $S\_STEP2\_S\_to\_C(session\_id, server\_key\_share, \text{“}server\_hello'', ServerHello)$
  Send $(ServerHello, EncryptedExtensions, [CertificateRequest], Certificate, CertificateVerify, Finished)$
  Event $E\_STEP2\_S\_to\_C(server\_key\_share, \text{“}server\_hello'', ServerHello)$
Step 3: Certificate Verification and Key Exchange
  Verify server certificate chain against configured trust anchors (root CA store)
  Validate ID in server certificate SAN records
  $DHE\_shared \leftarrow (server\_key\_share)^x$ ▷ Client computes shared secret
  $ES \leftarrow HKDF.Extract(0,0)$ ▷ No PSK input
  $HS \leftarrow HKDF.Extract(DHE\_shared, derive\_secret(ES, \text{“}derived''))$
  $(tk_{c\_hs}, tk_{s\_hs}, tk_{c\_app}, tk_{s\_app}) \leftarrow derive\_traffic\_keys(HS, transcript)$
  Event $S\_STEP3\_C\_to\_S(session\_id, tk_{c\_app}, \text{“}client\_finished'', Finished)$
  Send $(Certificate, CertificateVerify, Finished)$
  Event $E\_STEP3\_C\_to\_S(session\_id, tk_{c\_app}, \text{“}client\_finished'', Finished)$
Step 4: Server Processing and Session Completion
  Verify client certificate chain (if requested) against trust anchor
  Validate ID in client certificate SAN records
  $DHE\_shared \leftarrow (client\_key\_share)^y$ ▷ Server computes shared secret
  Recompute handshake/application keys using DHE_shared
  Verify client Finished message, then send server Finished

Algorithm 5 TLS 1.3 PSK-Only option

Input: PSK_{shared}, $PSK_{identity}$, roaming_context
Output: PSK-only secure channel
Step 1: ClientHello Generation
  $client\_random \leftarrow \{0,1\}^{256}$
  $ClientHello \leftarrow (tls\_1.3, client\_random, PSK_{identity}, psk\_ke)$
  $ES \leftarrow HKDF.Extract(PSK_{shared}, 0)$
  $binder\_key \leftarrow derive\_secret(ES, \text{“res\_binder”}, ClientHello)$
  $binder \leftarrow HMAC(binder\_key, hash(ClientHello))$
  Event $S\_STEP1\_C\_to\_S(PSK_{identity}, PSK_{shared}, client\_random, ClientHello, binder)$
  Send $(ClientHello, binder)$
  Event $E\_STEP1\_C\_to\_S(PSK_{identity}, PSK_{shared}, client\_random, ClientHello, binder)$
Step 2: ServerHello and Key Derivation
  Verify $binder$ using shared PSK_{shared}
  $server\_random \leftarrow \{0,1\}^{256}$
  $ServerHello \leftarrow (tls\_1.3, server\_random, PSK_{identity}, psk\_ke)$
  $HS \leftarrow HKDF.Extract(0, derive\_secret(ES, \text{“derived”}))$ ▷ No (EC)DHE input
  $(tk_{c\_hs}, tk_{s\_hs}, tk_{c\_app}, tk_{s\_app}) \leftarrow derive\_traffic\_keys(HS, transcript)$
  Event $S\_STEP2\_S\_to\_C(PSK_{identity}, tk_{s\_app}, \text{“server\_finished”}, ServerHello)$
  Send $(ServerHello, Finished)$
  Event $E\_STEP2\_S\_to\_C(PSK_{identity}, tk_{s\_app}, \text{“server\_finished”}, ServerHello)$
Step 3: Session Completion
  Verifies server Finished message
  Event $S\_STEP3\_C\_to\_S(PSK_{identity}, tk_{c\_app}, \text{“client\_finished”}, Finished)$
  Send client $Finished$
  Event $E\_STEP3\_C\_to\_S(PSK_{identity}, tk_{c\_app}, \text{“client\_finished”}, Finished)$

Algorithm 6 TLS 1.3 PSK-(EC)DHE option

Input: PSK_{shared}, $PSK_{identity}$, supported_groups
Output: PSK-(EC)DHE secure channel with PFS
Step 1: ClientHello with Key Share
  $client\_random \leftarrow \{0,1\}^{256}$
  $x \leftarrow \mathbb{Z}_q$, $client\_key\_share \leftarrow g^x$
  $ClientHello \leftarrow (tls\_1.3, client\_random, client\_key\_share, PSK_{identity}, psk\_dhe\_ke)$
  $ES \leftarrow HKDF.Extract(PSK_{shared}, 0)$
  $binder\_key \leftarrow derive\_secret(ES, \text{“res\_binder”}, ClientHello)$
  $binder \leftarrow HMAC(binder\_key, hash(ClientHello))$
  Event $S\_STEP1\_C\_to\_S(PSK_{identity}, client\_key\_share, client\_random, ClientHello, binder)$
  Send $(ClientHello, binder)$
  Event $E\_STEP1\_C\_to\_S(PSK_{identity}, client\_key\_share, client\_random, ClientHello, binder)$
Step 2: ServerHello with (EC)DHE
  Verify $binder$ using shared PSK_{shared}
  $y \leftarrow \mathbb{Z}_q$, $server\_key\_share \leftarrow g^y$
  $DHE\_shared \leftarrow (client\_key\_share)^y$
  $server\_random \leftarrow \{0,1\}^{256}$
  $ServerHello \leftarrow (tls\_1.3, server\_random, server\_key\_share, PSK_{identity}, psk\_dhe\_ke)$
  $HS \leftarrow HKDF.Extract(DHE\_shared, derive\_secret(ES, \text{“derived”}))$ ▷ Includes (EC)DHE
  $(tk_{c\_hs}, tk_{s\_hs}, tk_{c\_app}, tk_{s\_app}) \leftarrow derive\_traffic\_keys(HS, transcript)$
  Event $S\_STEP2\_S\_to\_C(PSK_{identity}, server\_key\_share, \text{“server\_finished”}, ServerHello)$
  Send $(ServerHello, Finished)$
  Event $E\_STEP2\_S\_to\_C(PSK_{identity}, server\_key\_share, \text{“server\_finished”}, ServerHello)$
Step 3: Key Computation and Session Completion
  $DHE\_shared \leftarrow (server\_key\_share)^x$
  Recompute handshake and application keys using DHE_shared
  Verify server Finished message, send client Finished
  Event $S\_STEP3\_C\_to\_S(PSK_{identity}, tk_{c\_app}, \text{“client\_finished”}, Finished)$
  Event $E\_STEP3\_C\_to\_S(PSK_{identity}, tk_{c\_app}, \text{“client\_finished”}, Finished)$

Algorithm 7 TLS 1.3 0-RTT option

Input: PSK_{shared}, $PSK_{identity}$, early_data_context
Output: 0-RTT channel with immediate data transmission
Step 1: Early Data Transmission
  $client\_random \leftarrow \{0,1\}^{256}$
  $x \leftarrow \mathbb{Z}_q$, $client\_key\_share \leftarrow g^x$
  $ClientHello \leftarrow (tls\_1.3, client\_random, early\_data, client\_key\_share, PSK_{identity}, psk\_dhe\_ke)$
  $ES \leftarrow HKDF.Extract(PSK_{shared}, 0)$
  $client\_early\_traffic\_secret \leftarrow derive\_secret(ES, \text{“c\_e\_traffic”}, ClientHello)$
  $early\_data\_encrypted \leftarrow AEAD.Encrypt(client\_early\_traffic\_secret, early\_data\_context)$
  $binder \leftarrow HMAC(derive\_secret(ES, \text{“res\_binder”}), hash(ClientHello))$
  Event $S\_STEP1\_C\_to\_S(PSK_{identity}, client\_early\_traffic\_secret, client\_random, ClientHello, early\_data\_encrypted)$
  Send $(ClientHello, binder, early\_data\_encrypted)$ ▷ 0-RTT data
  Event $E\_STEP1\_C\_to\_S(PSK_{identity}, client\_early\_traffic\_secret, client\_random, ClientHello, early\_data\_encrypted)$
Step 2: Early Data Processing
  Verify $binder$ using shared PSK_{shared}
  $client\_early\_traffic\_secret \leftarrow derive\_secret(ES, \text{“c\_e\_traffic”}, ClientHello)$
  $decrypted\_early\_data \leftarrow AEAD.Decrypt(client\_early\_traffic\_secret, early\_data\_encrypted)$
  Process early data immediately ▷ No handshake completion wait
  $y \leftarrow \mathbb{Z}_q$, $server\_key\_share \leftarrow g^y$
  Event $S\_STEP2\_S\_to\_C(PSK_{identity}, server\_key\_share, \text{“early\_data\_accept”}, decrypted\_early\_data)$
  Continue with standard PSK-(EC)DHE handshake completion
  Event $E\_STEP2\_S\_to\_C(PSK_{identity}, server\_key\_share, \text{“early\_data\_accept”}, decrypted\_early\_data)$
Step 3: Handshake Completion with End of Early Data
  Complete DHE key exchange as in Algorithm 6
  Server sends $EndOfEarlyData$ message
  Transition to handshake traffic keys, then application traffic keys
  Event $S\_STEP3\_C\_to\_S(PSK_{identity}, tk_{c\_app}, \text{“end\_early\_data”}, EndOfEarlyData)$
  Event $E\_STEP3\_C\_to\_S(PSK_{identity}, tk_{c\_app}, \text{“end\_early\_data”}, EndOfEarlyData)$

Algorithm 8 Main Process without FS

```
(* Main Process *)
process
  new PSK;
  insert pre-shared_key_db(PSK);
  ( (!proc_C(PSK)) | (!proc_S(PSK)) )
```

Algorithm 9 Main Process with FS

```
(* Main Process *)
process
  new PSK;
  insert pre-shared_key_db(PSK);
  ( (!proc_C(PSK)) | (!proc_S(PSK)) | phase 1; out(c, (PSK)) )
```

During the verification process, each security requirement is explicitly mapped to corresponding ProVerif queries, as summarized in Table 3. Confidentiality of application data transmitted over public channels is validated using Q5–Q7, where the attacker predicate is applied to test for potential data leakage. Integrity of protocol messages is ensured through Q1–Q4, which employ inj-event relations to verify that each transmitted event corresponds to its legitimate reception.

Mutual authentication is likewise established using (Q2 or Q3) and Q4, as these queries guarantee that both client and server events are consistently matched. The same queries, in

combination with Q5–Q7, are also employed to validate secure key exchange, ensuring that the negotiated session keys are not leaked to the adversary. PFS is verified under long-term key compromise scenarios, combining the use of the `phase` function with Q5–Q7 to demonstrate that past session keys remain protected. Finally, replay defense is modeled through freshness checks using Q1–Q4, confirming that old messages cannot be re-used to impersonate legitimate parties.

Due to space limitations within this paper, detailed explanations of the ProVerif modeling are condensed; however, the complete and detailed models are publicly available on GitHub[1] for further review and replication. This structured mapping enables a systematic and comprehensive validation of confidentiality, integrity, mutual authentication, secure key exchange, PFS, and defense against replay attacks within the TLS 1.2 and TLS 1.3 protocols.

Security Requirement	**Verification query**
Confidentiality	Q5: attacker(App Data1) Q6: attacker(App Data2) Q7: attacker(App Data3)
Integrity	Q1: inj-event(E_STEP1_C_to_S) ==> inj-event(S_STEP1_C_to_S) Q2: inj-event(T12_E_STEP2_S_to_C) ==> inj-event(T12_S_STEP2_S_to_C) Q3: inj-event(T13_E_STEP3_C_to_S) ==> inj-event(T13_S_STEP3_C_to_S) Q4: inj-event(E_STEP3_C_to_S) ==> inj-event(S_STEP3_C_to_S)
Mutual Authentication	(Q2 or Q3) && Q4: inj-event(E_STEP3_C_to_S) ==> inj-event(S_STEP3_C_to_S)
Secure Key Exchange	(Q2 or Q3) && Q4: inj-event(E_STEP3_C_to_S) ==> inj-event(S_STEP3_C_to_S) && (Q5, Q6, Q7)
PFS	'phase' function && Q5: attacker(App Data1) Q6: attacker(App Data2) Q7: attacker(App Data3)
Defense against replay attack	Q1: inj-event(E_STEP1_C_to_S) ==> inj-event(S_STEP1_C_to_S) Q2: inj-event(T12_E_STEP2_S_to_C) ==> inj-event(T12_S_STEP2_S_to_C) Q3: inj-event(T13_E_STEP3_C_to_S) ==> inj-event(T13_S_STEP3_C_to_S) Q4: inj-event(E_STEP3_C_to_S) ==> inj-event(S_STEP3_C_to_S)

Table 3: Security requirements and verification queries

3.1 Queries

During the process, robust verification is performed at each communication step to ensure mutual authentication, integrity, and freshness using the inj-event function. If freshness is not maintained, replay attacks may occur. Integrity verification queries for messages are modeled as Q1, Q2, Q3, and Q4, with freshness verification conducted to counteract replay attacks. Furthermore, both (Q2 or Q3) and Q4 are employed to verify that mutual authentication and secure key exchange are achieved while also ensuring the freshness of the messages. For further validation of the secure key exchange, the queries Q5 through Q7 are applied to evaluate the negotiated session keys for potential leakage. Finally, the confidentiality of the application data transmitted over public channels is verified based on the queries Q5 through Q7. On the

[1]The full ProVerif models can be accessed at https://github.com/yonghoko/ProVerif/Unified-Formal-Verification-of-Security-Requirements-in-the-TLS-Family

other hand, PFS is validated in cases where the long-term key is compromised, as outlined in Algorithm 9 using the 'Phase' function. In the process macro section, all event functions used for verification are emphasized.

3.2 Verification Results

Figure 3 presents our verification results of TLS 1.2 and TLS 1.3 handshake modes. The analysis confirms that in all TLS options, the ClientHello phase is exposed to resource-exhaustion replay attacks, where an adversary can repeatedly resend initial messages to overload the server and induce excessive computational overhead. As illustrated in Figure 4, an attacker can capture or forge ClientHello messages and replay them to the server; the server processes each incoming ClientHello and allocates computational resources (e.g., key schedule computation, cookie/state allocation, or cryptographic operations), which in aggregate can lead to denial-of-service at the signaling or application level.

We further investigated the impact of long-term key compromise. The verification shows that PFS is broken in TLS 1.2 with RSA key exchange, TLS 1.3 PSK-Only, and TLS 1.3 0-RTT, since an adversary possessing the long-term secret can recover past session keys and undermine the confidentiality of previously established communications.

Table 4 summarizes these findings across all TLS handshake modes. The comparison highlights two key weaknesses: (i) every mode is vulnerable to ClientHello replay at the initial stage, and (ii) certain modes fail to guarantee PFS when long-term secrets are disclosed. Therefore, robust TLS deployments must incorporate replay detection mechanisms at the handshake initiation and adopt stronger key exchange methods that ensure PFS across all modes.

Verification summary(TLS 1.2 RSA):
- Query inj-event(E_STEP1_C_to_S(*rand,client_hello_1*)) ==> inj-event(S_STEP1_C_to_S(*rand,client_hello_1*)) is false.
- Query inj-event(E_STEP2_S_to_C(*cert_pkS,s_key_exch*)) ==> inj-event(S_STEP2_S_to_C(*cert_pkS,s_key_exch*)) is false.
- Query inj-event(E_STEP3_C_to_S(*c_ms,c_finished*)) ==> inj-event(S_STEP3_C_to_S(*c_ms,c_finished*)) is true.
- Query inj-event(E_STEP4_S_to_C(*s_ms,s_finished*)) ==> inj-event(S_STEP4_S_to_C(*s_ms,s_finished*)) is true.
- Query not attacker_bitstring_p1(c_app_data[]) is false.
- Query not attacker_bitstring_p1(s_app_data[]) is false.

Verification summary(TLS 1.2 DH):
- Query inj-event(E_STEP1_C_to_S(*rand,client_hello_1*)) ==> inj-event(S_STEP1_C_to_S(*rand,client_hello_1*)) is false.
- Query inj-event(E_STEP2_S_to_C(*cert_pkS,s_key_exch*)) ==> inj-event(S_STEP2_S_to_C(*cert_pkS,s_key_exch*)) is true.
- Query inj-event(E_STEP3_C_to_S(*c_ms,c_finished*)) ==> inj-event(S_STEP3_C_to_S(*c_ms,c_finished*)) is true.
- Query inj-event(E_STEP4_S_to_C(*s_ms,s_finished*)) ==> inj-event(S_STEP4_S_to_C(*s_ms,s_finished*)) is true.
- Query not attacker_bitstring_p1(c_app_data[]) is true.
- Query not attacker_bitstring_p1(s_app_data[]) is true.

Verification summary(TLS 1.3 FH):
- Query inj-event(E_STEP1_C_to_S(rand,client_hello_1)) ==> inj-event(S_STEP1_C_to_S(rand,client_hello_1)) is false.
- Query inj-event(E_STEP2_S_to_C(k_7,aead_finished)) ==> inj-event(S_STEP2_S_to_C(k_7,aead_finished)) is true.
- Query inj-event(E_STEP3_C_to_S(k_7,aead_finished)) ==> inj-event(S_STEP3_C_to_S(k_7,aead_finished)) is true.
- Query not attacker_bitstring(app_data[]) is true.

Verification summary(TLS 1.3 PSK-Only):
- Query inj-event(E_STEP1_C_to_S(rand,client_hello_1)) ==> inj-event(S_STEP1_C_to_S(rand,client_hello_1)) is false.
- Query inj-event(E_STEP2_S_to_C(k_7,aead_finished)) ==> inj-event(S_STEP2_S_to_C(k_7,aead_finished)) is true.
- Query inj-event(E_STEP3_C_to_S(k_7,aead_finished)) ==> inj-event(S_STEP3_C_to_S(k_7,aead_finished)) is true.
- Query not attacker_bitstring(app_data1[]) is false.
- Query not attacker_bitstring(app_data2[]) is false.

Verification summary(TLS 1.3 DH):
- Query inj-event(E_STEP1_C_to_S(rand,client_hello_1)) ==> inj-event(S_STEP1_C_to_S(rand,client_hello_1)) is false.
- Query inj-event(E_STEP2_S_to_C(k_7,aead_finished)) ==> inj-event(S_STEP2_S_to_C(k_7,aead_finished)) is true.
- Query inj-event(E_STEP3_C_to_S(k_7,aead_finished)) ==> inj-event(S_STEP3_C_to_S(k_7,aead_finished)) is true.
- Query not attacker_bitstring(app_data1[]) is true.
- Query not attacker_bitstring(app_data2[]) is true.

Verification summary(TLS 1.3 0-RTT):
- Query inj-event(E_STEP1_C_to_S(rand,client_hello_1)) ==> inj-event(S_STEP1_C_to_S(rand,client_hello_1)) is false.
- Query inj-event(E_STEP2_S_to_C(k_7,aead_finished)) ==> inj-event(S_STEP2_S_to_C(k_7,aead_finished)) is true.
- Query inj-event(E_STEP3_C_to_S(k_7,aead_finished)) ==> inj-event(S_STEP3_C_to_S(k_7,aead_finished)) is true.
- Query not attacker_bitstring(app_data1[]) is false.
- Query not attacker_bitstring(app_data2[]) is true.
- Query not attacker_bitstring(app_data3[]) is true.

Figure 3: Verification results of TLS Family

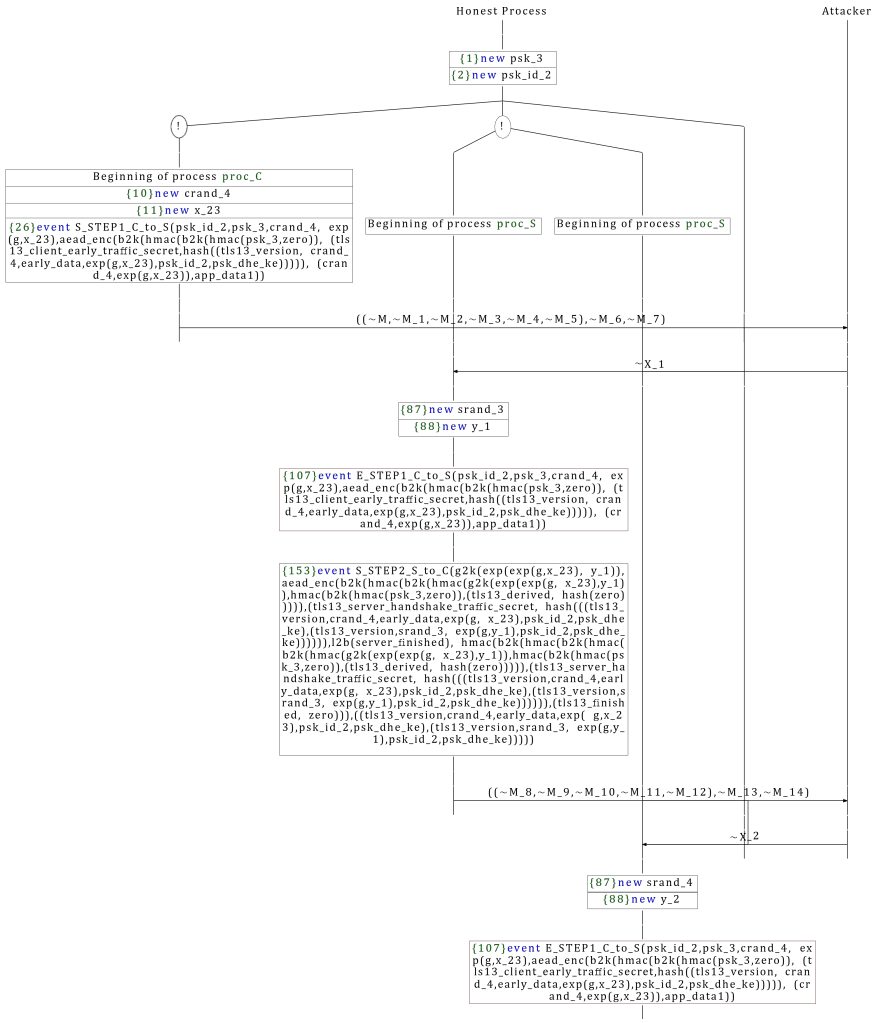

Figure 4: ClientHello replay attack process

Protocol	Co	In	MA	KE	PFS	RR
TLS 1.2 RSA	O	O	O	O	X	X
TLS 1.2 (EC)DHE	O	O	O	O	O	X
TLS 1.3 FH	O	O	O	O	O	X
TLS 1.3 PSK-Only	O	O	O	O	X	X
TLS 1.3 (EC)DHE	O	O	O	O	O	X
TLS 1.3 0-RTT	O	O	O	O	X	X

Co: Confidentiality; In: Integrity; MA: Mutual Authentication; KE: Key Exchange;
PFS: Perfect Forward Secrecy; RR: Replay resistance
O: Satisfied; X: Not satisfied;

Table 4: Comparison of TLS handshake modes with respect to security requirements

4 Discussion

Our verification results reveal several important implications for the design, deployment, and evaluation of TLS protocols.

First, the confirmation that all TLS 1.2 and TLS 1.3 handshake modes are susceptible to resource-exhaustion replay attacks at the ClientHello stage underscores the practical significance of freshness validation in early handshake messages. Although the cryptographic primitives underlying TLS remain secure, the absence of strict replay detection mechanisms in the initial phase exposes servers to denial-of-service conditions. This finding highlights the necessity of practical countermeasures such as cookie-based retries, rate-limiting strategies, or early binding of session contexts to mitigate signaling-level replay attacks.

Second, our analysis shows that PFS is not uniformly guaranteed across all handshake modes. Specifically, TLS 1.2 with RSA key exchange, TLS 1.3 PSK-Only, and TLS 1.3 0-RTT fail to preserve PFS when long-term secrets are compromised. This observation emphasizes that the security assurances of TLS depend strongly on the choice of handshake mode. From a deployment perspective, it is therefore critical to deprecate legacy RSA-based TLS 1.2 configurations and restrict the use of PSK-Only and 0-RTT modes in high-security environments where PFS is a strict requirement.

Third, by performing unbounded verification in line with ISO/IEC 29128-1:2023, this study provides a stronger assurance guarantee than previous bounded analyses or partial protocol evaluations. The ability to establish confidentiality, integrity, mutual authentication, secure key exchange, and PFS under unbounded adversarial models elevates the confidence in TLS security proofs, moving beyond the limitations of prior work that considered only restricted cases.

Finally, the optimization of ProVerif models introduced in this study represents a practical contribution for future security evaluations. While prior TLS models often suffered from impractically long execution times, our optimized approach significantly reduces verification overhead. This improvement demonstrates that unbounded formal verification of complex protocols such as TLS is not only theoretically possible but also practically achievable, opening the door for broader adoption of unbounded verification techniques in both academic and industrial settings.

In summary, these findings collectively stress the dual importance of careful mode selection and practical replay mitigation strategies in TLS deployments, while also illustrating the value of unbounded and efficient formal verification for establishing strong and reliable protocol assurances.

5 Conclusion

This paper presented a comprehensive unbounded formal verification of all handshake modes of TLS 1.2 and TLS 1.3, covering FH, PSK-Only, PSK-(EC)DHE, and 0-RTT. By modeling the protocols in ProVerif and mapping each security requirement to precise verification queries, we systematically evaluated confidentiality, integrity, mutual authentication, secure key exchange, PFS, and defense against replay attacks in accordance with ISO/IEC 29128-1:2023 unbounded assurance.

Our analysis revealed two significant findings. First, all handshake options are vulnerable to resource-exhaustion replay attacks at the ClientHello stage, highlighting the need for practical replay detection mechanisms and freshness validation in early handshake messages. Second, PFS is not consistently preserved across all modes: TLS 1.2 with RSA key exchange, TLS 1.3

PSK-Only, and TLS 1.3 0-RTT *early data* fail to maintain PFS under long-term key compromise. However, in TLS 1.3 0-RTT, post-handshake application data does achieve PFS when (EC)DHE is negotiated. These results emphasize that TLS security guarantees depend heavily on mode selection, and careful deployment policies are essential in practice. Moreover, TLS 1.2 RSA key exchange, although historically deployed, is now deprecated in modern IETF standards and browsers due to its lack of forward secrecy, further underscoring the need to avoid legacy configurations [14].

Despite these contributions, our study has certain limitations. The models do not incorporate network-level dynamics such as packet loss, latency, or traffic analysis, and they focus solely on conventional cryptographic primitives.

As a promising direction, future work will extend our verification framework to TLS 1.3 environments incorporating PQC, thereby providing unbounded assurance against both classical and quantum adversaries.

6 Acknowledgments

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2022-0-00103, Development of security verification technology against vulnerabilities to assure IoT/IIoT device safety, 100%).

References

[1] Qualys SSL Labs. Ssl pulse: Survey of the ssl/tls deployment on the internet. https://www.ssllabs.com/ssl-pulse/, 2025. Accessed: 2025-06.

[2] Markulf Kohlweiss, Cas Cremers, Mark Manulis, et al. A constructive cryptography analysis of tls 1.3 draft-05. In *Proceedings of EUROCRYPT Workshops*, 2016. Part of early TLS 1.3 draft analysis.

[3] Cas Cremers, Marko Horvat, Jonathan Hoyland, 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 (S&P)*, pages 470–485, 2016.

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

[5] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the tls 1.3 standard candidate. In *IEEE Symposium on Security and Privacy (S&P)*, pages 483–503, 2017.

[6] H. Daassa, F. Zarai, and A. Belghith. Tls protocol verification for securing e-commerce websites. *Journal of Internet Banking and Commerce*, 22(2):1–10, 2017.

[7] MITRE. Cve-2009-3555: Tls protocol session renegotiation security vulnerability. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3555, 2009. Accessed: 2025-09-22.

[8] Bruno Blanchet et al. Proverif models of tls 1.2 and tls 1.3. https://github.com/ProVerif/TLS-verification, 2018. Accessed: 2025-09-22.

[9] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the tls 1.3 handshake protocol. *Journal of Cryptology*, 34(2):1–69, 2021.

[10] H. Tran and K. Ogata. Formal verification of tls 1.2 by automatically generating proof scores. *Computers & Security*, 113:102545, 2022.

[11] Y. Ko, H. Kwon, B. Kim, and I. You. Toward an era of secure 5g convergence applications: Formal security verification of 3gpp akma with tls 1.3 psk option. *Applied Sciences*, 14(11):1152, 2024.

[12] A. Sardar, M. Siddiqi, M. Tariq, and H. Kim. Towards validation of tls 1.3 formal model and vulnerabilities in intel's ra-tls protocol. *IEEE Access*, 12:121134–121150, 2024.

[13] ISO. ISO/IEC 29128-1:2023 Information security, cybersecurity and privacy protection — Verification of cryptographic protocols — Part 1: Framework. Technical report, the International Organization for Standardization, March 2023. https://standards.iteh.ai/catalog/standards/iso/5a8c7c4d-434f-4816-a4e0-b33660dd311c/iso-iec-29128-1-2023.

[14] Richard Barnes, Benjamin Beurdouche, Christopher A. Wood, and Eric Rescorla. Deprecating obsolete key exchange methods in (d)tls 1.2. Internet-Draft draft-ietf-tls-deprecate-obsolete-kex-02, Internet Engineering Task Force, October 2024. Work in Progress.