FATT Chance: On the Robustness of Standalone and Hybrid ML-KEM Key Exchange in TLS 1.3

Part of the Nadim Kobeissi investigative series

Nadim Kobeissi

Symbolic Software, France

Abstract. Two post-quantum upgrades to TLS 1.3 are being standardized in parallel: a hybrid key exchange (already deployed) that runs an elliptic-curve Diffie–Hellman exchange alongside ML-KEM, and a standalone mode that uses ML-KEM on its own. The Internet-Draft draft-usama-tls-risks-of-mlkem points out that the machine-checked symbolic proofs of TLS 1.3 rely on the commutativity of Diffie–Hellman, which ML-KEM does not share: a key encapsulation mechanism is asymmetric, one endpoint generating a key pair and the other encapsulating against it. The existing proofs therefore no longer apply, a new one is needed, and the draft argues that hybrids should be preferred.

We supply that proof. We extend the reftls ProVerif models with a faithful, non-commutative KEM and analyze classical (EC)DHE, standalone ML-KEM, and the hybrid together, as unbounded concurrent sessions against a single active attacker free to break any cryptographic component.

The central result is a sharp and tight contrast in robustness: standalone ML-KEM is a single point of failure, secure only while ML-KEM itself is unbroken, whereas the hybrid stays secure as long as *either* of its components survives: an attacker must break both, in one session, to learn anything. This single point of failure reaches authentication as well as confidentiality: with the sole key-exchange secret exposed and no secret pre-shared key salting the key schedule, the server Finished message becomes forgeable, so a client can complete a handshake that no server completed, while the hybrid stays safe unless both components break.

The three modes also interoperate without ever confusing one another's keys, so migrating from (EC)DHE to a hybrid is a strict improvement. Two further experiments address the draft's remaining concerns: reusing an ML-KEM key forfeits the forward secrecy that an ephemeral key preserves, and a principal acting as both initiator and responder exposes no role-confusion attack arising from the asymmetry. At the symbolic level, and under stated assumptions, the analysis substantiates the draft's case for preferring hybrid key exchange.

Keywords: TLS · post-quantum cryptography · hybrid key exchange · formal verification

1 Introduction

It was a Tuesday, or maybe a Thursday; in this line of work the days blur together like a badly seeded PRNG. I was nursing a cold cup of coffee and an even colder open tab when the draft slid across my desk like a dame with a problem she couldn't solve alone. ML-KEM, it said. No commutativity. No role symmetry. The kind of asymmetry that gets a proof killed in a dark alley with $g^{xy} = g^{yx}$ scrawled on the wall in chalk. I'd seen the

E-mail: nadim@symbolic.software (Nadim Kobeissi)

reftls models before — good models, honest models, the kind that trusted Diffie–Hellman to commute the way you trust an old friend to pick up the phone. And now here was this KEM, walking in with one endpoint generating keys and the other encapsulating, breaking every assumption I'd ever made a living on.

I told myself I was done with formal verification. I'd said that before. But the draft had requested a FATT review, and a heavy request like that lands with a loud thud. I lit a fresh ProVerif instance, poured myself something stronger than coffee, and said the words every tired gumshoe says right before the third act goes sideways: *okay, fine, I'll take the case.*

There's a committee for this sort of thing — the FATT committee, they call it, the kind of acronym that sounds like a man falling down a flight of stairs — and I am not on it. Did I clear this review with them? Sweetheart, in this town you don't ask the committee for permission. You ask ProVerif, and ProVerif doesn't take meetings. So I ran the FATT review alone, alone with only an unbounded number of replicated sessions and the kind of moral clarity that only survives in people who say exactly what they're thinking in every single email they send keeping me company. Let the working group ratify it afterward. The dame with the problem didn't come to a committee. She came to me.

The migration of TLS 1.3 [Res25] to post-quantum key exchange is moving down two roads at once. *Hybrid* key exchange, specified in draft-ietf-tls-hybrid-design [SFG25], runs an elliptic-curve Diffie–Hellman exchange and an ML-KEM [Nat24] key encapsulation shoulder to shoulder and feeds the concatenation of their shared secrets into the key schedule; `X25519MLKEM768` is the face you see most often. *Standalone* ML-KEM, specified in draft-ietf-tls-mlkem [Con26], registers ML-KEM-512/768/1024 as TLS named groups and leans on the ML-KEM shared secret alone, in place of the (EC)DHE secret.

The two roads keep very different company[1] in the IETF TLS working group, and the standalone proposal in particular has collected its share of enemies.[2] It is against that backdrop that draft-usama-tls-risks-of-mlkem [Sar26] lays a precise technical observation on the table, not just a procedural grievance.[3] The existing machine-checked symbolic proofs of TLS 1.3, namely the reftls models of Bhargavan, Blanchet, and Kobeissi [BBK17], together with their forks and the attested-TLS extensions built on them, model Diffie–Hellman through an ideal exponentiation equipped with the commutativity equation $g^{xy} = g^{yx}$.

Seeing my own name, Kobeissi, surface in that nine-year-old citation gave me pause; I think back to those younger, more innocent years, when commutativity still felt like a promise the world would keep. But I've seen a lot of promises broken since those days.

Standalone ML-KEM swaps that symmetric key *exchange* for an asymmetric key *encapsulation*: only one endpoint generates the key pair (dk, ek) and the other encapsulates against the public ek, producing a ciphertext ct. No commutativity, no role symmetry, no alibi. The draft concludes, correctly, that this breaks the existing proofs as written, that a new proof is therefore required, and that this entails modeling KEMs in the tool. It further argues that hybrids should be *at least* as strong as their stronger component, so that standalone ML-KEM is a security regression next to the hybrid, and it requests a formal-analysis (FATT) review.

Just when I thought I was out, they pull me back in.

Contributions.

- We extend the reftls RFC 8446bis model with a faithful, *asymmetric* KEM primitive (Section 3). It carries no commutativity equation; the decapsulation rule is the only

[1] This is an understatement.
[2] This is also an understatement.
[3] For purely procedural objections, see `[REDACTED]`.

way to recover an encapsulated secret. This is exactly the modeling change [Sar26] identifies as necessary.

- We model all three key-agreement modes in one library and compose *unbounded sessions of all three concurrently*, over shared long-term keys and pre-shared keys, against a single active attacker (Section 4). Integration attacks that only appear when the modes coexist (cross-mode share confusion, reflection, or material reuse across modes) are therefore in scope, not excluded by analyzing each mode alone.
- We machine-check secrecy and authentication, letting the attacker weaken each component independently, and we add two *tightness controls* and two *reachability* checks so the guarantees are demonstrably necessary and non-vacuous (Sections 5–6).
- We confirm the robustness contrast: standalone ML-KEM is secure unless ML-KEM breaks (and the assumption is load-bearing), while the hybrid is secure unless *both* components break, tightly (Section 6). We additionally surface, and isolate with dedicated controls, an authentication caveat refining the draft's confidentiality-only statement (Finding 3).
- We model *key reuse* (Section 4.4): a long-lived ML-KEM keypair shared across unbounded standalone sessions, its decapsulation key compromised only after they complete. This substantiates the drafts' forward-secrecy rationale (Finding 4): an ephemeral key keeps past traffic secret under a compromised PSK, a reused-then-compromised key does not.
- We model *non-static roles* (Section 4.5): one principal running both the initiator and the responder over shared keys, under both a type-distinct and a deliberately confusable (raw) share encoding. The KEM asymmetry surfaces no role-confusion or unknown-key-share attack (Finding 5). This is the draft's stated *key point*, taken up here in a server-authenticated, one-initiator-per-session setting.

All model files and the complete ProVerif transcript are released alongside this report so the results can be reproduced (Section 7).

2 Background

2.1 The TLS 1.3 key schedule

TLS 1.3 [Res25] derives all of its traffic keys from a sequence of HKDF stages. Writing Extract and Expand for HKDF-Extract and HKDF-Expand-Label, the handshake proceeds through an Early Secret derived from any pre-shared key (PSK), a Handshake Secret

$$HS \;=\; \mathrm{Extract}\big(\mathrm{DeriveSecret}(ES, \text{“derived”}),\; ss_{\mathrm{kex}}\big),$$

and a Master Secret, from which the handshake- and application-traffic keys and the Finished MAC keys are expanded. The quantity ss_{kex} is the *key-exchange shared secret*: in classical TLS 1.3 it is the (EC)DHE value g^{xy}. Crucially, the secrecy and authenticity of everything downstream of HS depends on ss_{kex} being secret, *unless* the salt $\mathrm{DeriveSecret}(ES, \cdot)$ is itself secret because the PSK is. This salt/IKM interplay is the dual-PRF property and it is central to the robustness arguments below.

2.2 From key exchange to key encapsulation

A key encapsulation mechanism is a triple (KeyGen, Encaps, Decaps): KeyGen produces (dk, ek); $\mathsf{Encaps}(ek)$ produces a ciphertext ct and a shared secret ss; and $\mathsf{Decaps}(dk, ct) =$

ss. ML-KEM [Nat24] is an IND-CCA2 KEM. In TLS the roles are fixed by the 1-RTT message ordering (draft-ietf-tls-mlkem §4.2; draft-ietf-tls-hybrid-design §3.2): the client's `key_share` carries the public *ek* and the server's carries the ciphertext *ct*. The first mover has no peer KEM key to encapsulate against, so it must publish a fresh *ek*; hence *client* = *KeyGen+Decaps*, *server* = *Encaps*. The ML-KEM shared secret then takes the place of the (EC)DHE secret in the key schedule (draft-ietf-tls-mlkem §4.3). This is the asymmetry at the heart of the discussion: unlike g^x and g^y, which are interchangeable and combine commutatively, *ek* and *ct* play distinct, non-interchangeable roles. Two consequences of this asymmetry frame the later experiments. First, the 1-RTT flow lets a client reuse its `key_share` across handshakes, so both drafts require the KEM to remain secure under public-key reuse (IND-CCA2), accepting *limited forward secrecy* as the price (mlkem §5.2–5.3, hybrid §2, §6). Second, the risks draft's central concern is that roles are not static: an endpoint may initiate one connection and respond to another, and it asks whether the KEM's initiator/responder asymmetry then becomes visible in some property that Diffie–Hellman's symmetry kept hidden.

2.3 Hybrid key exchange

The hybrid construction (draft-ietf-tls-hybrid-design §3.3) runs both an (EC)DHE exchange and a KEM and sets $ss_{\text{kex}} = ss_1 \parallel ss_2$, the concatenation of the component secrets (order fixed by the named group), again in place of the single (EC)DHE secret. Security of this "concatenate-then-KDF" combiner reduces to the dual-PRF property of the hash [BBF+19]; a computational proof of the hybrid design in CryptoVerif is due to Blanchet and Jacomme [BJ24]. As the risks draft notes, the elliptic-curve half of the hybrid retains its symmetry: g^x and g^y are still exchanged and g^{xy} still computed commutatively. Only the ML-KEM half is asymmetric.

2.4 Symbolic analysis, ProVerif, and the reftls models

Symbolic (Dolev–Yao) analysis models messages as terms in an algebra and the attacker as the network, deriving new terms by applying public operations [BBB+21]. ProVerif [Bla16] decides reachability and correspondence (event-implication) queries over unbounded sessions by Horn-clause resolution. The reftls models [BBK17] apply this to TLS 1.3: they encode the handshake, the key schedule, and PSK/0-RTT/resumption, and prove secrecy and (injective) authentication. A distinctive feature, which we retain and extend, is the *weak-primitive methodology*: each primitive has a "strong" and a "weak" variant, the attacker chooses which a session negotiates, and security properties are stated as implications — "the key leaks *only if* some negotiated primitive was the weak one." The model already provides `WeakDH`, `WeakHash`, and `WeakAE`; our work adds `WeakKEM`. This style is what lets a single model express "secure unless X is broken" for each X, and, once several weak knobs coexist, "secure unless *this combination* is broken," which is precisely the hybrid robustness question.

3 Modeling the commutativity break

The risks draft pinpoints the proof dependency in the reftls library: Diffie–Hellman is idealized as an exponentiation with an explicit commutativity equation. Our model keeps it verbatim:

```
fun dh_ideal(element,bitstring):element.
equation forall x:bitstring, y:bitstring;
dh_ideal(dh_ideal(G,x),y) =
dh_ideal(dh_ideal(G,y),x).
```

This equation is what makes $g^{xy} = g^{yx}$ and is exactly the symmetry the draft says a KEM lacks. We model the KEM with no such equation:

```
type kem_alg.  const StrongKEM, WeakKEM: kem_alg.
type kem_priv. type kem_pub. type kem_ct.
fun kem_pk(kem_priv): kem_pub.

(* encapsulator chooses a fresh ss, binds it to a ciphertext under ek *)
fun kem_encap(kem_alg, kem_pub, bitstring): kem_ct.
fun kem_decap(kem_alg, kem_priv, kem_ct): bitstring
reduc forall a:kem_alg, dk:kem_priv, ss:bitstring;
kem_decap(a, dk, kem_encap(a, kem_pk(dk), ss)) = ss.

(* WeakKEM: broken primitive -- ss recoverable straight from ct *)
fun kem_leak(kem_ct): bitstring
reduc forall ek:kem_pub, ss:bitstring;
kem_leak(kem_encap(WeakKEM, ek, ss)) = ss.
```

Three properties make this faithful to the draft's concern. First, there is *no commutativity and no role symmetry*: the only public way to recover an encapsulated *ss* is to apply `kem_decap` with the matching private key *dk*, and *dk* links to *ek* only through the one-way constructor `kem_pk`. The keypair holder (client) and the encapsulator (server) are not interchangeable. Second, the primitive is *asymmetric in exactly the modeled message flow*: the client constructs *ek*, the server constructs *ct*, and these are different term constructors that cannot be confused. Third, the weak-primitive knob `WeakKEM` mirrors `WeakDH`: it exposes a `kem_leak` rule by which the attacker reads *ss* directly off the ciphertext, modeling a fully broken ML-KEM, while `StrongKEM` is the idealized IND-CCA KEM whose secrets are non-malleable and unrecoverable.

For the hybrid we deliberately retain `dh_ideal`'s commutativity for the elliptic-curve component, because — as the draft itself argues — that part *is* symmetric; only the ML-KEM component uses the asymmetric primitive. The model thus reflects the draft's own decomposition of the hybrid into a symmetric DH half and a complementary KEM half.

A raw KEM, for the role experiment. The type-distinct `kem_pub` and `kem_ct` above let the symbolic attacker tell an encapsulation key from a ciphertext for free. That is faithful to TLS's message framing (they ride in distinct `ClientHello`/`ServerHello` `key_share`s of different lengths), but it is also exactly the distinction one might suspect the asymmetry leans on. For the non-static-roles experiment (§6) we therefore add a *raw* KEM whose key and ciphertext are *both* `bitstring`s, carried under one share constructor, so the two are type-indistinguishable and the attacker may substitute one for the other:

```
fun kem_pk_r(kem_priv): bitstring.                          (* ek : bitstring *)
fun kem_encap_r(kem_alg, bitstring, bitstring): bitstring.  (* ct : bitstring *)
fun kem_decap_r(kem_alg, kem_priv, bitstring): bitstring
reduc forall a:kem_alg, dk:kem_priv, ss:bitstring;
kem_decap_r(a, dk, kem_encap_r(a, kem_pk_r(dk), ss)) = ss.
```

The decapsulation rule still reduces only on a genuine `kem_encap_r` output, so even when confusion is *expressible* it is the KEM's algebraic structure, not merely a type tag, that the experiment puts to the test.

4 ProVerif models for post-quantum TLS

A single ProVerif library carries all three key-agreement modes, and the top-level process composes them under replication:

```
process ( !Client13()     | !Server13()      (* classical (EC)DHE     *)
| !Client13_kem() | !Server13_kem()  (* standalone ML-KEM     *)
| !Client13_hyb() | !Server13_hyb()  (* hybrid                *)
| !longTermKeysProc() | !appData()
| !secrecyQuery() | !channelBindingQuery() )
```

Each replicated client/server instance negotiates a mode and its component algorithms as chosen by the environment, so *unbounded sessions of all three modes coexist*, drawing certificates and PSKs from the same tables, against a single attacker with the run of the network. This is a deliberately stronger setting than three isolated analyses: integration attacks (confusing a standalone *ct* with a hybrid *ct*, reflecting one mode's share into another, or reusing material across modes) can only manifest when the modes share an attacker and key material. The events each session emits carry the negotiated mode, so a query can still single out, say, "a hybrid session with a strong KEM" amid all the others.

The composition above is the *robustness* model. The two experiments that follow, key reuse and non-static roles, each need processes and a threat configuration that would *perturb* these robustness queries if folded in (a reused key rewrites the forward-secrecy story; the role processes reuse the standalone constructors and events), and a single combined run would also be needlessly heavy. We therefore keep three composition files over the one shared library: the robustness model here, and the two experiment models of §6. Isolation keeps the robustness results identical to the original validated run.

4.1 Three modes and the combiner

The key-exchange secret handed to the unchanged Handshake-Secret derivation differs only in ss_{kex}:

$$ss_{\text{kex}} = \begin{cases} g^{xy} & \text{classical,} \\ ss_{\text{kem}} & \text{standalone,} \\ (ss_{\text{kem}},\, g^{xy}) & \text{hybrid.} \end{cases}$$

The combiner is the existing key-schedule function, unchanged:

```
letfun kdf_hs(es:bitstring, e:bitstring) =
let extra = derive_secret(es, tls13_derived, empty_string) in
hkdf_extract(extra, e).
```

For the hybrid, $e = (ss_{\text{kem}}, g^{xy})$ is the ProVerif pair, modeling the concatenation; pairing is injective, so this is a faithful "concatenate-then-Extract" as long as concatenation is length-unambiguous. Because `hkdf_extract` is, symbolically, a free constructor, it is automatically one-way in *both* its salt and its input-keying-material arguments — the symbolic incarnation of the dual-PRF assumption [BBF+19] the hybrid combiner needs.

4.2 Role asymmetry in the processes

The processes realize the fixed roles of Section 2. In the standalone mode the client generates the keypair and decapsulates, while the server encapsulates a fresh secret:

```
(* Client13_kem *)                    (* Server13_kem *)
let (dk,ek) = kem_keygen() in         new ss_kem:bitstring;
(* ... send ek, receive ct ... *)      let ct = kem_encap(ka, ek, ss_kem) in
let kem_ss = kem_decap(ka, dk, ct) in (* ... send ct ... *)
let handshake_secret =                let handshake_secret =
kdf_hs(early_secret, kem_ss) in       kdf_hs(early_secret, ss_kem) in
```

The hybrid mode runs both a Diffie–Hellman exchange and a KEM, with the same client/server asymmetry on the KEM half:

```
(* Client13_hyb *)                          (* Server13_hyb *)
let (x,gx)  = dh_keygen(g) in               let (y,gy)  = dh_keygen(g) in
let (dk,ek) = kem_keygen() in               new ss_kem:bitstring;
(* ... send (gx,ek), receive (gy,ct) ... *) let ct = kem_encap(ka, ek, ss_kem) in
let gxy    = e2b(dh_exp(g,gy,x)) in          let gxy = e2b(dh_exp(g,gx,y)) in
let kem_ss = kem_decap(ka,dk,ct) in          let handshake_secret =
let handshake_secret =                           kdf_hs(early_secret,(ss_kem,gxy)) in
kdf_hs(early_secret,(kem_ss,gxy)) in
```

Agreement on ss_{kex} between honest peers follows by the decapsulation rule ($kem_ss = ss_kem$) and, for the hybrid, by DH commutativity (g^{xy} agrees); the two derive the same Handshake Secret.

4.3 Threat model and events

The negotiated mode and its component algorithms are environment-driven, so the attacker chooses, per session and independently, whether the Diffie–Hellman group is `StrongDH` or `WeakDH` and whether the KEM is `StrongKEM` or `WeakKEM` (and likewise the hash and AEAD). Independent selection of `WeakDH` and `WeakKEM` is what makes "secure unless both break" expressible. Each server emits an event recording the abstract negotiated mode,

```
type kex13.
fun Classical(group): kex13.  fun Standalone(kem_alg): kex13.
fun Hybrid(group, kem_alg): kex13.
event ServerKEXMode(random, random, pubkey, version, kex13).
```

so queries can name precisely the session class they constrain — e.g. `Standalone(WeakKEM)` or `Hybrid(WeakDH,WeakKEM)`. The existing `ServerChoosesKEX`, `ServerChoosesAE`, and `ServerChoosesHash` events continue to record the classical components.

4.4 Key reuse

To study reuse we generate one ML-KEM keypair (dk, ek) *outside* the session replication and store it in a table; an unbounded number of standalone clients then draw the *same* keypair (the textbook ProVerif idiom for a long-lived key) rather than a fresh one. A holder process reveals dk in `phase 1`, after the phase-0 sessions complete, and raises `ReusedKEMKeyCompromised`; each reuse session protects a fresh payload `m_cr` under its application key, with an *attacker-supplied* nonce so that confidentiality genuinely rests on the key. The contrast with the ephemeral standalone client (a fresh dk per session, never revealed) is exactly the forward-secrecy question the drafts raise.

4.5 Non-static roles

To let one principal play both ends, the role experiment composes an initiator process and a responder process that draw from the *same* `longTermKeys` and `preSharedKeys` tables, so a single identity acts as initiator in some sessions and responder in others, under one attacker. Authentication is server-only (no client certificates, as in the base model), so the property we target is the initiator's authentication of, and agreement with, the responder. Each completed role emits an agreement event carrying the peer identity, the KEM shares, and the derived application key, so a query can ask both whether two opposite-role sessions that share a key agree on who is who (unknown-key-share) and whether every initiator that completes has a genuine matching responder (injective agreement). We run this under both share encodings of §3: the type-distinct `kem_pub`/`kem_ct` and the raw `bitstring` shares, the latter making cross-role substitution expressible.

Table 1: Faithfulness mapping from the drafts to the model.

Draft point	Source	Model treatment
client = KeyGen, server = Encaps	mlkem §4.2, hybrid §3.2	`kem_keygen`/`kem_decap` client; `kem_encap` server
ss replaces (EC)DHE	mlkem §4.3	`ss_kex = kem_ss` into `kdf_hs`
concatenated *ss* replaces (EC)DHE	hybrid §3.3	`ss_kex = (kem_ss, gxy)`
dual-PRF combiner	hybrid §6, [BBF+19]	idealized `hkdf_extract` (assumed)
IND-CCA, non-malleable	mlkem §5.2, hybrid §2	`StrongKEM` non-invertible constructor
KEM secure under key reuse	mlkem §5.2–5.3, hybrid §2	reused keypair; `dk` leaked in `phase 1` (§6)
DH retains commutativity	risks draft §"Technical Rationale"	`dh_ideal` kept for the DH half
shares are transcript-bound	mlkem §5.4, [CDM23]	`ek/ct/gx/gy` under sig+Finished
non-static roles surface asymmetry	risks draft §"Minimum Viable Modeling"	addressed, Phase 3 (§6)

4.6 Faithfulness to the drafts

Table 1 maps each load-bearing point of the two construction drafts (and the risks draft's framing) to its treatment in the model.

5 Security properties and queries

We verify the model's standard secrecy and authentication goals, generalized so that each names the weak-primitive conditions that excuse a violation. We abbreviate the event $\texttt{WeakOrCompromisedKey}(p)$ (the server's certificate key is attacker-known) as $\texttt{WeakKey}(p)$ and `CompromisedPreSharedKey` as `CompromisedPSK`, and we write ORTH for the orthogonal disjuncts, those about neither the key exchange nor the AEAD:

$$\text{ORTH} \;\equiv\; \big(\texttt{WeakKey}(p) \wedge (\mathit{psk}{=}\texttt{NoPSK} \vee \texttt{CompromisedPSK}(\mathit{psk}))\big) \;\vee\; \texttt{WeakHash},$$

keeping the classical key-exchange disjunct $\texttt{ServerChoosesKEX}(\texttt{DHE_13}(\texttt{WeakDH}, \cdot))$ separate. The handshake-key-secrecy and handshake-authentication queries carry exactly ORTH and this classical disjunct; the application-data queries additionally carry an AEAD disjunct (`ServerChoosesAE(WeakAE)` at 1-RTT, `ClientOffersAE(WeakAE)` at 0-RTT), and the server-side data query `ServerReceives` uses a *bare* PSK disjunct ($\mathit{psk}{=}\texttt{NoPSK} \vee \texttt{CompromisedPSK}$) instead of the `WeakKey`-conjoined form, because in this model the server does not authenticate the client by certificate.

Forward secrecy. The reftls model reveals the server's long-term signing key in a later protocol phase (`phase 1`, once the session has completed), and the secrecy queries are resolved against this post-compromise attacker (ProVerif's phase-1 attacker, written `attacker_p1` in the output). The confidentiality results below are therefore *forward*-secrecy statements: past application data stays secret even after the server's long-term key leaks, because confidentiality rests on the ephemeral (EC)DHE/KEM secret and not on the signing key. This is precisely where key-exchange robustness is load-bearing.

Secrecy. The headline confidentiality query states that the client's application-traffic key never leaks (event `ClientAEKeyLeaked`), and that the 1-RTT, client-to-server application payload `m_c` is not derivable by the (phase-1) attacker, *unless* a key-exchange or orthogonal weakness was negotiated:

$$\begin{aligned}\texttt{attacker}(\texttt{m_c}(\ldots)) \Longrightarrow\ & \texttt{ServerKEXMode}(\texttt{Standalone}(\texttt{WeakKEM})) \\ & \vee\ \texttt{ServerKEXMode}(\texttt{Hybrid}(\texttt{WeakDH}, \texttt{WeakKEM})) \\ & \vee\ \texttt{ServerChoosesKEX}(\texttt{DHE_13}(\texttt{WeakDH}, \cdot)) \ \vee\ \texttt{WeakAE} \ \vee\ \text{ORTH}.\end{aligned}$$

(The handshake-key query `ClientAEKeyLeaked` is identical but without the `WeakAE` disjunct.) The standalone disjunct names `WeakKEM` alone; the hybrid disjunct names the conjunction `Hybrid(WeakDH,WeakKEM)`, both components weak. That asymmetry between the two disjuncts is the formal statement of the robustness contrast, and the controls below check it is forced by the model and not merely assumed.

Authentication. The injective handshake-authentication query states that every client Finished matches a distinct, prior server Finished (event implication `ClientFinished` ⇒ `PreServerFinished`), unless a key-exchange or orthogonal weakness holds; the application-data authentication queries

`ClientReceives/ServerReceives` ⇒ `ServerSends/ClientSends`

are analogous, and the full query carries the same KEM disjuncts, for the reason explained in Finding 3.

Tightness controls. Two queries are identical to the secrecy query but each *omits* one KEM disjunct, and are expected to be FALSE:

T1 (standalone single point of failure) omits `Standalone(WeakKEM)` (keeping the hybrid disjunct). If false, a standalone session with a weak KEM leaks with no surviving excuse, so `WeakKEM` on its own breaks standalone.

T2 (hybrid needs both) omits `Hybrid(WeakDH,WeakKEM)` (keeping the standalone disjunct). If false, a hybrid session leaks through a path no remaining disjunct excuses, which, given the main query is TRUE, carrying *only* the both-weak hybrid disjunct, can only be the both-components-weak case.

Reachability (non-vacuity). Two reachability queries check that the standalone and hybrid server modes actually complete a handshake (`ServerKEXMode(Standalone(·))` and `ServerKEXMode(Hybrid(·,·))` are reachable). Without these a robustness implication could hold vacuously because its mode never runs.

Uniqueness. Four queries check that no two distinct completed sessions, across *all three* modes, agree on an application key, channel binding, resumption secret, or resumption PSK (the `Matching`· events never fire). These rule out cross-mode key confusion.

Reuse (Phase 1b). In the reuse model the forward-secrecy query asks that the reused-session payload `m_cr` stay secret unless a key-exchange or orthogonal weakness holds *or* the reused key is compromised (an extra `ReusedKEMKeyCompromised` disjunct). A tightness control drops that last disjunct, expected FALSE, and a reachability event confirms the reuse client actually completes.

Table 2: Security guarantees of the three TLS 1.3 key-agreement modes, side by side. A ✓ or ✗ holds under this paper's symbolic model and the stated assumptions; the right-hand column points to the Finding that establishes each robustness row. The **Hybrid** column is the only one without a ✗: it attains every guarantee that either pure mode does, which is the precise sense in which it *strictly dominates* both standalone ML-KEM and classical (EC)DHE.

Security guarantee	(EC)DHE	ML-KEM	Hybrid	Finding
Secure against a classical attacker	✓	✓	✓	—
Post-quantum secure†	✗	✓	✓	—
Secure if ML-KEM is broken	✓	✗	✓	1, 2
Secure if (EC)DHE is broken	✗	✓	✓	2
Key-confirmation authentication survives one key-exchange break‡	✗	✗	✓	3
Forward secrecy (ephemeral keys)§	✓	✓	✓	4
No role-confusion from KEM asymmetry	—	✓	✓	5

† A property of the construction, not established by the symbolic (Dolev–Yao) analysis, which models a classical attacker.
‡ Injective agreement (key confirmation), absent a secret PSK (Finding 3); the CertificateVerify signature still authenticates the server's identity. "One key-exchange break" means the sole component in a pure mode, either component in the hybrid.
§ With ephemeral keys; reusing an ML-KEM key trades this away (Finding 4).

Non-static roles (Phase 3). For each share encoding the role model adds two properties. The *unknown-key-share* query: if an initiator and a responder derive the same application key, they agree on the peer identity and on the KEM shares. The *injective agreement* query: every completing initiator matches a distinct responder, unless the certificate key, the hash, or the KEM is weak; a tightness control drops the `Standalone(WeakKEM)` disjunct. Reachability events confirm that both roles complete.

6 Results

We ran the combined model to completion. All thirty queries resolve exactly as the design intends; Table 3 summarizes them by category and the full verbatim transcript is reproduced in Appendix A. Table 2 first places the three key-agreement modes side by side on the security guarantees at stake; the hybrid is the only mode that attains every guarantee that either pure mode offers. We highlight three findings here; two further experiments, key reuse and non-static roles, each run separately over the shared library, follow in Sections 6.1–6.2.

Table 3: Verification outcomes for the combined classical + standalone + hybrid model (unbounded sessions, shared keys, one attacker). "⇒" abbreviates an event-implication query; `KEMdisj` abbreviates `Standalone(WeakKEM)` ∨ `Hybrid(WeakDH,WeakKEM)`.

Query	Result	Meaning
Generalized robustness (headline), expected TRUE		
`ClientAEKeyLeaked` ⇒ `KEMdisj` ∨ WeakDH ∨ ORTH	TRUE	1-RTT key secrecy
`attacker(m_c)` ⇒ `KEMdisj` ∨ WeakDH ∨ WeakAE ∨ ORTH	TRUE	1-RTT data secrecy
inj `ClientFinished` ⇒ `PreServerFinished` ∨ `KEMdisj` ∨ ···	TRUE	1-RTT handshake auth

Query	Result	Meaning
`ClientReceives` ⇒ `ServerSends` ∨ `KEMdisj` ∨ ···	TRUE	1-RTT data auth (C)
`ServerReceives` ⇒ `ClientSends` ∨ `KEMdisj` ∨ ···	TRUE	1-RTT data auth (S)
`attacker(m_c0)` ⇒ PSK ∨ WeakAE	TRUE	0-RTT data secrecy
`ServerReceives0` ⇒ `ClientSends0` ∨ PSK ∨ WeakAE	TRUE	0-RTT data auth
Tightness controls, expected FALSE		
T1: `attacker(m_c)` ⇒ (… ∨ `Hybrid(WeakDH,WeakKEM)`), *no* `Standalone`	FALSE	standalone SPOF
T2: `attacker(m_c)` ⇒ (… ∨ `Standalone(WeakKEM)`), *no* `Hybrid`	FALSE	hybrid needs both
Reachability / non-vacuity, expected FALSE *(= reachable)*		
`not ServerKEXMode(Standalone(·))`	FALSE	standalone runs
`not ServerKEXMode(Hybrid(·,·))`	FALSE	hybrid runs
Cross-mode uniqueness, expected TRUE		
`not MatchingAEKey`	TRUE	no key collision
`not MatchingChannelBinding`	TRUE	unique binding
`not MatchingResumptionSecret` / `...PSK`	TRUE	unique resumption
Necessity of each disjunct (partial queries), expected FALSE		
`ClientAEKeyLeaked` / inj `ClientFinished` with any single weak disjunct dropped	FALSE	each disjunct required
Legacy and liveness		
TLS 1.2 correspondences (no TLS 1.2 role composed)	TRUE	vacuous
`not` {`ServerFinished`,`ClientFinished`,…}(TLS13)	FALSE	handshake completes

Finding 1 (Standalone ML-KEM is a single point of failure)**.** *The secrecy query is* TRUE *carrying the disjunct* ***`Standalone(WeakKEM)`****: a standalone session's secrets leak only if its KEM is weak (or an orthogonal primitive is). Control* ***T1****, which removes that disjunct, is* FALSE*: with the "ML-KEM broken" assumption deleted from the statement, a standalone leak is no longer excused. Equivalently, the proof of standalone secrecy genuinely requires the ML-KEM-secure assumption. This is the symbolic counterpart of the draft's "secure unless ML-KEM is broken," and it confirms that, for standalone, breaking ML-KEM is sufficient to break the session.*

Finding 2 (The hybrid is secure unless both components break, tightly)**.** *The secrecy query is* TRUE *while carrying, for the hybrid, only the conjunctive disjunct* ***`Hybrid(WeakDH,WeakKEM)`*** *— there is no single-weak hybrid disjunct anywhere in the statement. Hence whenever a hybrid session's secrets leak, both its Diffie–Hellman and its ML-KEM components were weak; a hybrid session with one sound component does not leak. Control* ***T2****, which removes the both-weak disjunct, is* FALSE*, confirming that the both-weak path is genuine and reachable (so the property is not vacuous). Together: a hybrid session is secure unless both of its components are broken. Mechanistically, the Handshake Secret is* $\mathrm{Extract}(salt, (ss_{\mathrm{kem}}, g^{xy}))$*, and because symbolic* ***`hkdf_extract`*** *is one-way in its keying material, a single secret component of the pair keeps the output secret. This is the draft's "secure unless both ECDHE and ML-KEM are broken," machine-checked under concurrent composition with standalone and classical sessions.*

Findings 1 and 2 together substantiate, at the symbolic level, the draft's central comparison: the hybrid strictly dominates standalone ML-KEM in robustness, retaining security as long as either component survives, whereas standalone has no such fallback. The same run also exercises classical (EC)DHE sessions; with Finding 2 this gives the "(EC)DHE → hybrid is an improvement" direction, since a classical `WeakDH` session leaks (its `DHE_13(WeakDH)` disjunct is necessary) whereas a hybrid `(WeakDH,StrongKEM)` session does not.

Finding 3 (A broken sole key-exchange factor also breaks handshake authentication)**.** *The injective handshake-authentication query is* TRUE *carrying the KEM disjuncts, and three dedicated authentication-tightness controls confirm those disjuncts are load-bearing: removing from the query, in turn, only* `Standalone(WeakKEM)`*, only* `Hybrid(WeakDH,WeakKEM)`*, or only the classical* `DHE_13(WeakDH)` *disjunct each yields* FALSE *(run separately over the same model; see the artifact). These are the authentication analogues of the secrecy controls* ***T1/T2****, and the mechanism is the dual of* ***T1****. When the sole key-exchange secret is public (a weak KEM in standalone, or weak DH in classical) and the key schedule is not salted by a secret PSK (a fresh, non-resumption handshake, or a compromised PSK), the Handshake Secret is public. The server* Finished *MAC key is derived from that same Handshake Secret (the server handshake-traffic secret's* `finished` *key, RFC 8446bis §4.4.4), so it too is public and the server* Finished *becomes forgeable — and RFC 8446bis makes the* Finished *"essential for providing authentication of the handshake." The witnessing trace uses a strong hash and an honest certificate key (with a compromised PSK), so it is neither a hash collision nor a signature forgery.*

Precisely, what fails is key confirmation (injective agreement), not peer-identity authentication. The CertificateVerify *signature, taken under the honest signing key over the transcript through the server* Certificate *— and hence over the KEM* `key_share` *— still ties the session to the server's identity, so the client's view of whom it is talking to remains correct. What the attacker defeats is the guarantee that a server actually completed a matching session: knowing the Handshake Secret, it drives the client to* ClientFinished *with a forged server* Finished *while no server reaches the corresponding* PreServerFinished*. This is the AKE matching-conversation property that TLS 1.3 vests in the* Finished*, and it degrades exactly like confidentiality.*

Two caveats keep this in proportion. First, it is not specific to ML-KEM: the classical `DHE_13(WeakDH)` *control is* FALSE *for the same reason, and the original* reftls *model already carries the* `WeakDH` *disjunct on its authentication query — standalone ML-KEM merely inherits, as the sole key-exchange factor, the well-known TLS 1.3 dependency of* Finished*-based authentication on key-exchange secrecy (absent a secure PSK). Second, the hybrid resists it: the Handshake Secret stays secret unless both components break (Finding 2), so the* `Hybrid(WeakDH,WeakKEM)` *control being* FALSE *confirms that here too both components must fail. The non-static-roles experiment of Finding 5 runs an analogous control on its agreement query, and it is likewise* FALSE*.*

Finally, the four uniqueness queries are TRUE and the two reachability queries are FALSE (= reachable): no two completed sessions across the three modes ever derive the same application key, channel binding, or resumption secret (no cross-mode confusion), and both new modes do run, so Findings 1–2 are non-vacuous.

6.1 Key reuse: forward secrecy, traded away

We ran the reuse model (§4.4) separately; Table 4 records the outcomes.

Finding 4 (Reuse trades away forward secrecy)**.** *The reuse forward-secrecy query is* TRUE *carrying a* `ReusedKEMKeyCompromised` *disjunct, and the tightness control that drops it is* FALSE*. Concretely, a strong-KEM session whose reused decapsulation key is later*

compromised, together with a compromised PSK, leaks its past application data. The ephemeral standalone query (the setting of Finding 1) carries no such disjunct and stays secret under the very same PSK compromise: a fresh decapsulation key, never stored, gives forward secrecy that the leaked long-term key cannot undo. So an ephemeral ML-KEM key buys forward secrecy that a reused key trades away. This is the symbolic counterpart of the drafts' insistence on IND-CCA2 and their warning of limited forward secrecy under reuse (mlkem §5.2–5.3, hybrid §2, §6).

6.2 Non-static roles: the asymmetry stays invisible

We ran the role model (§4.5) separately, under both share encodings; Table 4 records the outcomes.

Finding 5 (No role-confusion from the KEM asymmetry)**.** *Under both the type-distinct and the raw share encodings the unknown-key-share query is* TRUE*: no two opposite-role sessions ever derive the same application key while disagreeing on identity or shares, even under the raw encoding, where the attacker may hand an encapsulation key where a ciphertext is expected (the decapsulation rule simply rejects it). The injective agreement, the initiator's authentication of the responder, is* TRUE *unless a component is weak; its* `WeakKEM` *disjunct is load-bearing (the minus-*`WeakKEM` *control is* FALSE*), but the trace it admits is the same weak-key-exchange-with-compromised-PSK* Finished*-forgery as classical (EC)DHE, the mechanism of Finding 3: the attack uses a strong hash, ruling out a signature collision, so it is not a confusion born of the asymmetry. In this server-authenticated, one-initiator-per-session setting, then, the KEM's initiator/responder asymmetry surfaces no new role-confusion or unknown-key-share attack. The draft's central worry does not, here, materialize as a distinct vulnerability at the symbolic level: an informative negative answer.*

Table 4: Experiment outcomes: key reuse (Phase 1b) and non-static roles (Phase 3), each run separately over the shared library; abbreviations as in §5.

Query	Result	Meaning
Key reuse (Phase 1b)		
`attacker(m_cr)` ⇒ `ReusedKEMKeyCompromised` ∨ `Standalone(WeakKEM)` ∨ WeakAE ∨ ORTH	TRUE	reuse secret unless key compromised
same, *minus* `ReusedKEMKeyCompromised`	FALSE	reuse trades away FS
`not ReuseClientDone`	FALSE	reuse client runs
Non-static roles (Phase 3), typed shares and raw shares		
UKS: same key ⇒ agree on identity and shares (typed; raw)	TRUE	no unknown-key-share
inj `InitiatorAgree` ⇒ inj `ResponderAgree` ∨ WeakKey ∨ `Standalone(WeakKEM)` ∨ WeakHash	TRUE	initiator authenticates responder
same, *minus* `Standalone(WeakKEM)`	FALSE	WeakKEM tight (as classical WeakDH)
`not ResponderAgree` / `...Raw`	FALSE	both roles run

7 Reproducibility

The analysis extends the public reftls RFC 8446bis model. One shared library, `tls-lib-pq.pvl`, adds the KEM primitives (type-distinct and raw), the three key-exchange modes, the reuse and role processes, and the new events. Three query files compose it into the three models of this paper: `tls13-pq.pv` (robustness, the thirty queries of Table 3), `tls13-pq-reuse.pv` (Phase 1b), and `tls13-pq-roles.pv` (Phase 3); the experiments are kept separate so they do not perturb the robustness queries (§4). A supplementary file, `tls13-pq-authtight.pv`, carries the three authentication-tightness controls of Finding 3, each dropping one key-exchange disjunct from the authentication query. Each is checked with ProVerif [Bla16], e.g.

```
proverif -lib tls-lib-pq tls13-pq.pv
```

The model files are available in the reftls repository at https://github.com/symbolicsoft/reftls, and the complete transcripts (resolving as in Tables 3–4 and Appendix A) are archived alongside them.

8 Conclusion

draft-usama-tls-risks-of-mlkem raised a simple-sounding question: the machine-checked symbolic proofs of TLS 1.3 lean on Diffie–Hellman commutativity, and standalone ML-KEM does not commute, so can the integration of ML-KEM be given the same scrutiny anyway, and can the often-asserted advantage of hybrids be made precise rather than merely argued? The answer is yes on both counts. A faithful, non-commutative KEM turns out to be a modest extension of the reftls model, and once classical (EC)DHE, standalone ML-KEM, and the hybrid all run side by side under one attacker, the comparison the draft wanted falls out as machine-checked theorems. In plain terms, here is what the analysis found.

1. *Standalone ML-KEM is all-or-nothing* (Finding 1). A standalone session is secure exactly as long as ML-KEM itself is secure, and not one assumption longer. Break ML-KEM and the session falls with it; there is no second line of defense.

2. *The hybrid hedges that bet* (Finding 2). Running ML-KEM alongside elliptic-curve Diffie–Hellman, the hybrid stays secure as long as *either* half survives. An attacker has to break both, in the same session, to learn anything; defeating only the post-quantum half, or only the classical half, buys nothing.

3. *A broken key exchange undermines authentication, not just secrecy* (Finding 3). When the sole key-exchange secret becomes public and no secret pre-shared key is in play, the server's Finished message becomes forgeable. The certificate signature still authenticates the server's identity, but the Finished is what confirms a matching, completed session — so the client can be driven to accept a handshake that no server completed: a loss of key confirmation (injective agreement), not merely of confidentiality. This is not peculiar to ML-KEM; a broken classical Diffie–Hellman does the same. But standalone inherits it as its single point of failure, and the hybrid, once more, resists it.

4. *Reusing a key costs forward secrecy* (Finding 4). An ephemeral ML-KEM key, discarded after one handshake, keeps past traffic safe even if long-term secrets later leak. A reused key that is eventually compromised hands that past traffic over. This is the concrete price of the key reuse that IND-CCA2 exists to make safe.

5. *The asymmetry scare did not materialize* (Finding 5). We let a single principal act as both initiator and responder over shared keys, and even handed the attacker a deliberately confusable wire encoding, looking for the role-confusion the draft feared the KEM's asymmetry might introduce. None appeared: no unknown-key-share, no reflection, just a clean negative answer to the draft's central concern.

Taken together, these say something sharper than "hybrids are nice to have." The hybrid neutralizes every weakness standalone ML-KEM carries, both the confidentiality single point of failure and the authentication (key-confirmation) failure that rides along with it, by demanding that *both* of its components fail at once; and moving from today's classical (EC)DHE to a hybrid never gives up ground, because the classical guarantee survives intact inside the combination. That is the precise sense in which the hybrid *strictly dominates* standalone.

None of this says standalone ML-KEM is broken. Under its stated assumption, that ML-KEM holds, it is secure under our models. The argument against it is one of robustness, not of any present-day attack: it stakes everything on a single, relatively young assumption, where the hybrid keeps a mature fallback in reserve. For a deployer who cannot afford to be wrong about lattice cryptography for the lifetime of the data being protected, that distinction is the whole game. The two caveats bound the picture honestly: reuse is a real and quantifiable cost rather than a catastrophe, and the role-asymmetry worry, while genuinely the draft's headline concern, did not surface as a distinct vulnerability in the server-authenticated, one-initiator-per-session setting analyzed here.

References

[BBB+21] Manuel Barbosa, Gilles Barthe, Karthikeyan Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. In *2021 IEEE Symposium on Security and Privacy (S&P)*, pages 777–795. IEEE, 2021. doi:10.1109/SP40001.2021.00008.

[BBF+19] Nina Bindel, Jacqueline Brendel, Marc Fischlin, Brian Goncalves, and Douglas Stebila. Hybrid key encapsulation mechanisms and authenticated key exchange. In *Post-Quantum Cryptography (PQCrypto 2019)*, volume 11505 of *LNCS*, pages 206–226. Springer, 2019. doi:10.1007/978-3-030-25510-7_12.

[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 (S&P)*, pages 483–502. IEEE Computer Society, 2017. doi:10.1109/SP.2017.26.

[BJ24] Bruno Blanchet and Charlie Jacomme. Post-quantum sound CryptoVerif and verification of hybrid TLS and SSH key-exchanges. In *2024 IEEE 37th Computer Security Foundations Symposium (CSF)*. IEEE, 2024. URL: https://ieeexplore.ieee.org/document/10664427/.

[Bla16] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1–2):1–135, 2016. doi:10.1561/3300000004.

[CDM23] Cas Cremers, Alexander Dax, and Niklas Medinger. Keeping up with the KEMs: Stronger security notions for KEMs and automated analysis of KEM-based protocols. Cryptology ePrint Archive, Paper 2023/1933, 2023. URL: https://eprint.iacr.org/2023/1933.

[Con26] Deirdre Connolly. ML-KEM post-quantum key agreement for TLS 1.3. Internet-Draft draft-ietf-tls-mlkem-07, Internet Engineering Task Force, 2026. Work in Progress. URL: https://datatracker.ietf.org/doc/draft-ietf-tls-mlkem/07/.

[Nat24] National Institute of Standards and Technology. Module-lattice-based key-encapsulation mechanism standard. FIPS PUB 203, 2024. doi:10.6028/NIST.FIPS.203.

[Res25] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. Internet-Draft draft-ietf-tls-rfc8446bis-14, Internet Engineering Task Force, September 2025. Work in Progress. URL: https://datatracker.ietf.org/doc/draft-ietf-tls-rfc8446bis/14/.

[Sar26] Muhammad Usama Sardar. Potential Risks of Standalone ML-KEM in TLS 1.3. Internet-Draft draft-usama-tls-risks-of-mlkem-02, Internet Engineering Task Force, June 2026. Work in Progress. URL: https://datatracker.ietf.org/doc/draft-usama-tls-risks-of-mlkem/02/.

[SFG25] Douglas Stebila, Scott Fluhrer, and Shay Gueron. Hybrid key exchange in TLS 1.3. Internet-Draft draft-ietf-tls-hybrid-design-16, Internet Engineering Task Force, 2025. Work in Progress. URL: https://datatracker.ietf.org/doc/draft-ietf-tls-hybrid-design/16/.

A Complete verification summary

The following is an *abbreviated* rendering of the ProVerif verification summary for the combined model; the unedited transcript, with ProVerif's freshly-generated variable names (e.g. `cr_23`) and its phase-1 attacker predicate `attacker_p1`, is in the artifact. For readability we shorten event names here: `WeakKey` abbreviates `WeakOrCompromisedKey`, `PSK` abbreviates the condition psk=`NoPSK` $\vee$ `CompromisedPreSharedKey`(psk), and bound variables are elided. Recall that for the generalized queries TRUE is the desired outcome (the violation implies a named weakness), for the tightness controls **T1**/**T2** and the reachability queries FALSE is desired (= a path exists / the mode is reachable), and the partial queries are FALSE by construction (they omit a required disjunct).

```
Query event(ClientFinished(TLS12,...)) ==> event(ServerFinished(TLS12,...))
is true.                                  (* TLS 1.2 role not composed: vacuous *)
Query event(ClientFinished(TLS13,...)) ==> event(PreServerFinished(TLS13,...))
is false.                                 (* bare auth, no excuses: expected false *)
Query not event(ServerFinished(TLS12,...))      is true.   (* unreachable: no TLS 1.2 *)
Query not event(ClientFinished(TLS12,...))      is true.   (* unreachable: no TLS 1.2 *)
Query not event(ServerFinished(TLS13,...))      is false.  (* reachable: handshake runs *)
Query not event(PreServerFinished(TLS13,...))   is false.  (* reachable *)
Query not event(ClientFinished(TLS13,...))      is false.  (* reachable *)

(* Partial robustness queries -- each omits a required disjunct: expected false *)
Query ClientAEKeyLeaked ==> (WeakKey & PSK) || DHE_13(WeakDH)            is false.
Query ClientAEKeyLeaked ==> (WeakKey & PSK) || WeakHash                  is false.
Query ClientAEKeyLeaked ==> DHE_13(WeakDH)  || WeakHash                  is false.
Query inj ClientFinished ==> inj PreServerFinished || (WeakKey & PSK)    is false.
Query inj ClientFinished ==> inj PreServerFinished || (WeakKey & PSK)
|| DHE_13(WeakDH)    is false.
Query inj ClientFinished ==> inj PreServerFinished || DHE_13(WeakDH)
|| WeakHash          is false.
Query inj ClientFinished ==> inj PreServerFinished                       is false.

(* Cross-mode uniqueness: expected true *)
Query not event(MatchingChannelBinding(...))    is true.
Query not event(MatchingResumptionSecret(...))  is true.
Query not event(MatchingResumptionPSK(...))     is true.
Query not event(ResumptionPSKDerived(...))      is false.  (* reachable *)
Query not event(MatchingAEKey(...))             is true.

(* Generalized robustness (headline): expected true *)
Query ClientAEKeyLeaked ==> Standalone(WeakKEM) || Hybrid(WeakDH,WeakKEM)
|| (WeakKey & PSK) || DHE_13(WeakDH) || WeakHash      is true.
Query inj ClientFinished ==> Standalone(WeakKEM) || Hybrid(WeakDH,WeakKEM)
|| inj PreServerFinished || (WeakKey & PSK)
|| DHE_13(WeakDH) || WeakHash                      is true.
Query attacker(m_c) ==> Standalone(WeakKEM) || Hybrid(WeakDH,WeakKEM)
|| (WeakKey & PSK) || WeakAE || DHE_13(WeakDH) || WeakHash is true.
Query attacker(m_c0) ==> PSK || ClientOffersAE(WeakAE)                          is true.
Query ClientReceives ==> Standalone(WeakKEM) || Hybrid(WeakDH,WeakKEM)
|| ServerSends || (WeakKey & PSK) || WeakAE
|| DHE_13(WeakDH) || WeakHash                              is true.
Query ServerReceives ==> Standalone(WeakKEM) || Hybrid(WeakDH,WeakKEM)
|| ClientSends || PSK || WeakAE
|| DHE_13(WeakDH) || WeakHash                              is true.
Query ServerReceives0 ==> ClientSends0 || PSK || ClientOffersAE(WeakAE)       is true.

(* Tightness controls: expected false *)
Query attacker(m_c) ==> (WeakKey & PSK) || WeakAE || WeakHash || DHE_13(WeakDH)
|| Hybrid(WeakDH,WeakKEM)            is false.  (* T1: standalone SPOF *)
Query attacker(m_c) ==> (WeakKey & PSK) || WeakAE || WeakHash || DHE_13(WeakDH)
|| Standalone(WeakKEM)               is false.  (* T2: hybrid needs both *)

(* Mode reachability: expected false (= reachable) *)
Query not event(ServerKEXMode(...,Standalone(a)))    is false.
Query not event(ServerKEXMode(...,Hybrid(g,a)))      is false.
```

The reuse (Phase 1b) and role (Phase 3) experiments, run separately over the same library, give the following (abbreviated as above).

```
(* Phase 1b -- key reuse (tls13-pq-reuse.pv) *)
Query not event(ReuseClientDone(...))                              is false. (* reuse client runs *)
Query attacker_p1(m_cr) ==> ReusedKEMKeyCompromised || Standalone(WeakKEM)
|| (WeakKey & PSK) || WeakAE || WeakHash  is true.
Query attacker_p1(m_cr) ==> Standalone(WeakKEM)
|| (WeakKey & PSK) || WeakAE || WeakHash  is false. (* reuse trades away FS *)

(* Phase 3 -- non-static roles (tls13-pq-roles.pv), typed shares *)
Query not event(ResponderAgree(...))                               is false. (* responder runs *)
Query InitiatorAgree(p,ek,ct,k) && ResponderAgree(p',ek',ct',k)
==> p=p' && ek=ek' && ct=ct'                                        is true.  (* no UKS *)
Query inj InitiatorAgree(p,ek,ct,k) ==> inj ResponderAgree(p,ek,ct,k)
|| WeakKey(p) || Standalone(WeakKEM) || WeakHash           is true.  (* initiator authenticates resp. *)
Query inj InitiatorAgree(p,ek,ct,k) ==> inj ResponderAgree(p,ek,ct,k)
|| WeakKey(p) || WeakHash                                          is false. (* WeakKEM tight, as classical WeakDH
*)

(* Phase 3 -- raw shares (ek,ct both bitstring; confusion expressible) *)
Query not event(ResponderAgreeRaw(...))                            is false.
Query InitiatorAgreeRaw(...) && ResponderAgreeRaw(...), same key
==> agree on p, ek, ct                                             is true.  (* no UKS even raw *)
Query inj InitiatorAgreeRaw(...) ==> inj ResponderAgreeRaw(...)
|| WeakKey(p) || Standalone(WeakKEM) || WeakHash           is true.
```