VERIFPAL ANALYSIS

tls13.vp

Verifpal 1.6.0 · September 23, 2026

19	3	16
QUERIES	ATTACKS FOUND	NO ATTACK FOUND

This report describes the analysis of the Verifpal model `tls13.vp` against an active attacker, with each principal running 2 concurrent sessions. Of its 19 queries, 3 were broken. Every attack reported here is a witness against the model as written. A query reported as holding means that this search found no attack at these parameters, which is not a proof that none exists.

1 `tls13.vp`

Attacker	active
Sessions	2 per principal
Queries	19, with 3 attacks found
Result code	`c1c0c0c0c0c0e0c0c0c1c0c1a0a0a0a0a0f0f0`
Analysis time	78.6 s

1.1 Verdicts

Table 1: Every query in the model, with the verdict this run reached.

Query	Verdict	Scope	Line
confidentiality? *shstraffic*	attack	witness in §1.3.1	477
confidentiality? *shstraffic* precondition Server → Client: *appdata2*	holds	search exhausted at 2 sessions	478
confidentiality? *chstraffic_c* precondition Client → Server: *clientflight*	holds	search exhausted at 2 sessions	481
confidentiality? *saptraffic* precondition Server → Client: *appdata2*	holds	search exhausted at 2 sessions	484
confidentiality? *captraffic_c* precondition Client → Server: *clientflight*	holds	search exhausted at 2 sessions	487
confidentiality? *exportersecret* precondition Server → Client: *appdata2*	holds	search exhausted at 2 sessions	490
equivalence? *saptraffic*, *saptraffic_c* precondition Server → Client: *appdata2* precondition Client → Server: *clientflight*	holds	search exhausted at 2 sessions	493
confidentiality? *appmsg1*	holds	search exhausted at 2 sessions	497
confidentiality? *appmsg2*	holds	search exhausted at 2 sessions	498
confidentiality? *appmsg3*	attack	witness in §1.3.2	499
confidentiality? *respsk2*	holds	search exhausted at 2 sessions	500
confidentiality? *clientid*	attack	witness in §1.3.3	501
authentication? Server → Client: *serverflight*	holds	search exhausted at 2 sessions	502
authentication? Client → Server: *clientflight*	holds	search exhausted at 2 sessions	503

Table 1, continued

Query	Verdict	Scope	Line
authentication? Client → Server: *appdata1*	holds	search exhausted at 2 sessions	504
authentication? Server → Client: *appdata2*	holds	search exhausted at 2 sessions	505
authentication? Server → Client: *appdata3*	holds	search exhausted at 2 sessions	506
freshness? *saptraffic*	holds	search exhausted at 2 sessions	507
freshness? *captraffic_c*	holds	search exhausted at 2 sessions	508

1.2 Protocol

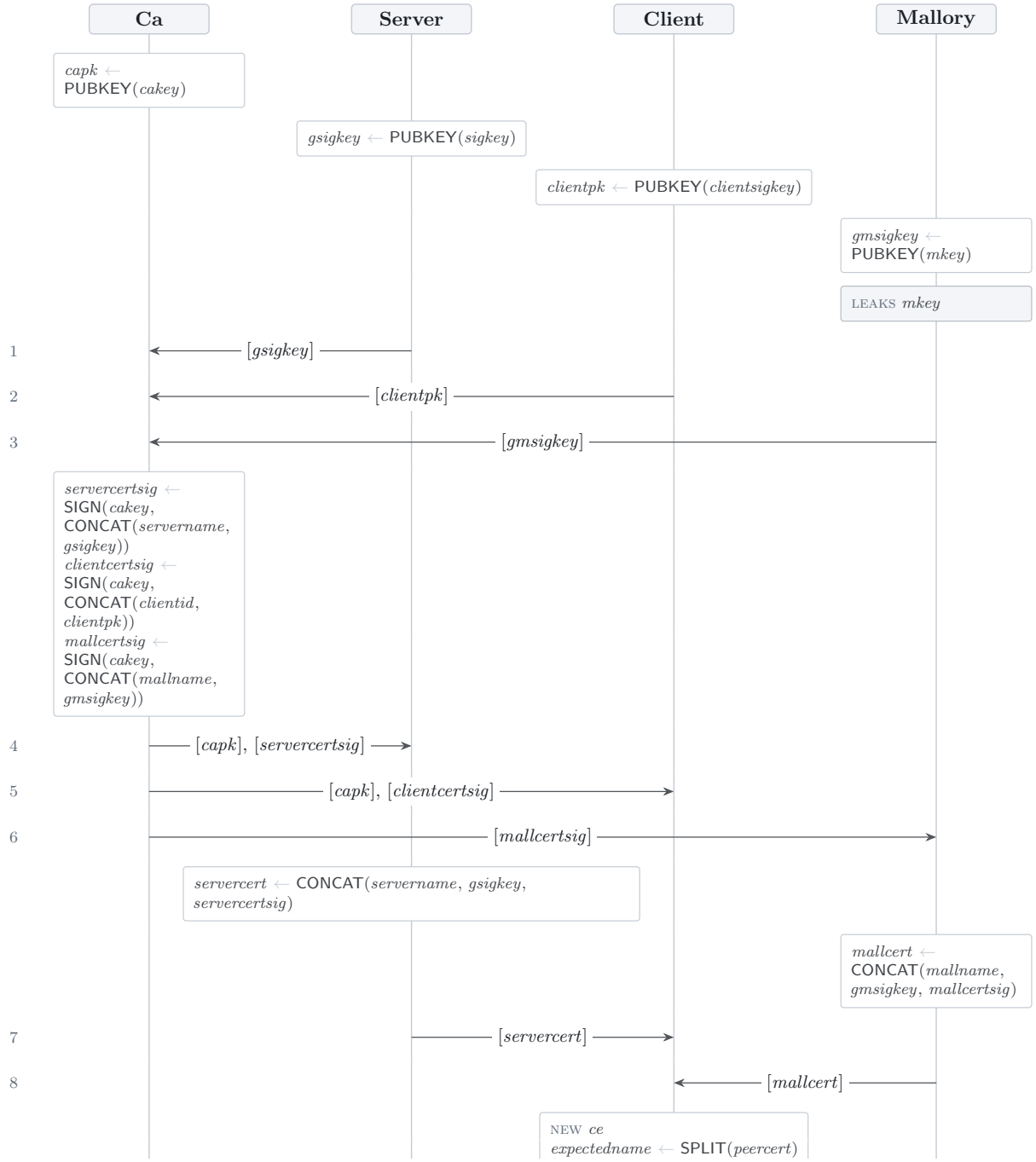

Figure 1, continued

Figure 1, continued

Figure 1, continued

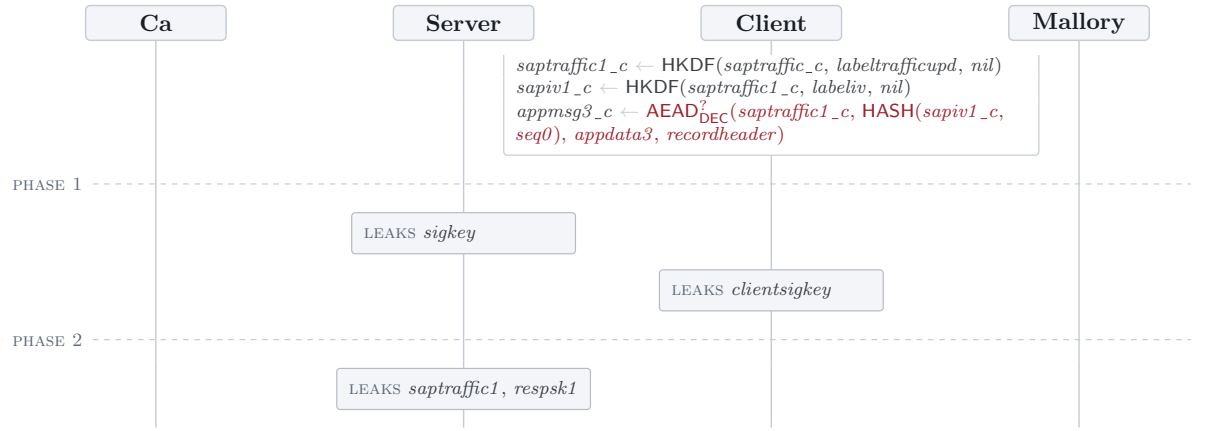

Figure 1: The protocol as written. Guarded values are bracketed; a dagger marks every value an attack below substitutes or replays.

1.3 Attacks

1.3.1 confidentiality? *shstraffic*

shstraffic (HKDF(*handshakesecret*, *labelshstraffic*, *transcriptsh*)$_1$) is obtained by Attacker.

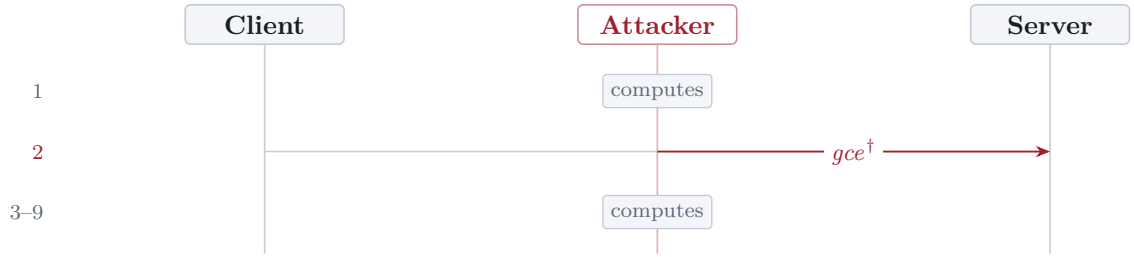

Figure 2: How the attacker breaks confidentiality? *shstraffic*. Substituted values carry a dagger.

1. Attacker constructs PUBKEY(*nil*).
2. Attacker replaces *gce* (sent by Client to Server) with PUBKEY(*nil*). (*gce* was PUBKEY(*ce*)) Client → Server
3. Attacker constructs *earlysecret*.
4. Attacker constructs *derivedes*.
5. Attacker observes *gse* on the wire.
6. Attacker constructs *dhe*, where it is DH$_{\text{KEX}}$(*gse*, *nil*).
7. Attacker constructs *handshakesecret*, where it is HKDF(*derivedes*, *dhe*, *nil*)$_1$.
8. Attacker constructs *transcriptsh*, where it is HASH(PUBKEY(*nil*), *gse*).
9. Attacker constructs *shstraffic*, where it is HKDF(*handshakesecret*, *labelshstraffic*, *transcriptsh*)$_1$.

1.3.2 confidentiality? *appmsg3*

appmsg3 (*appmsg3*) is obtained by Attacker.

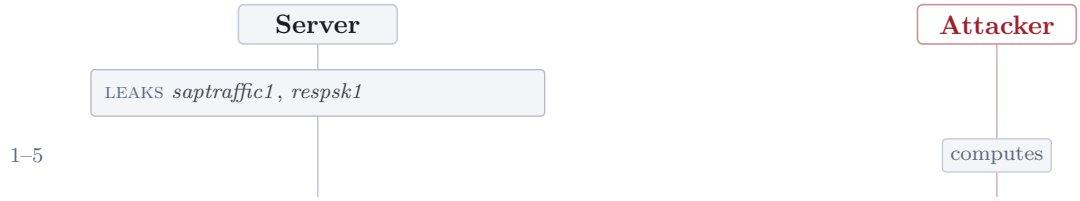

Figure 3: How the attacker breaks confidentiality? *appmsg3*. Substituted values carry a dagger.

1. Attacker observes *appdata3* on the wire.
2. Attacker is handed *saptraffic1* by a leaks declaration in Server.
3. Attacker constructs *sapiv1*.
4. Attacker constructs HASH(*sapiv1*, *seq0*).
5. Attacker opens *appdata3* with *saptraffic1*, HASH(*sapiv1*, *seq0*), obtaining *appmsg3*.

1.3.3 confidentiality? *clientid*

clientid (*clientid*) is obtained by Attacker.

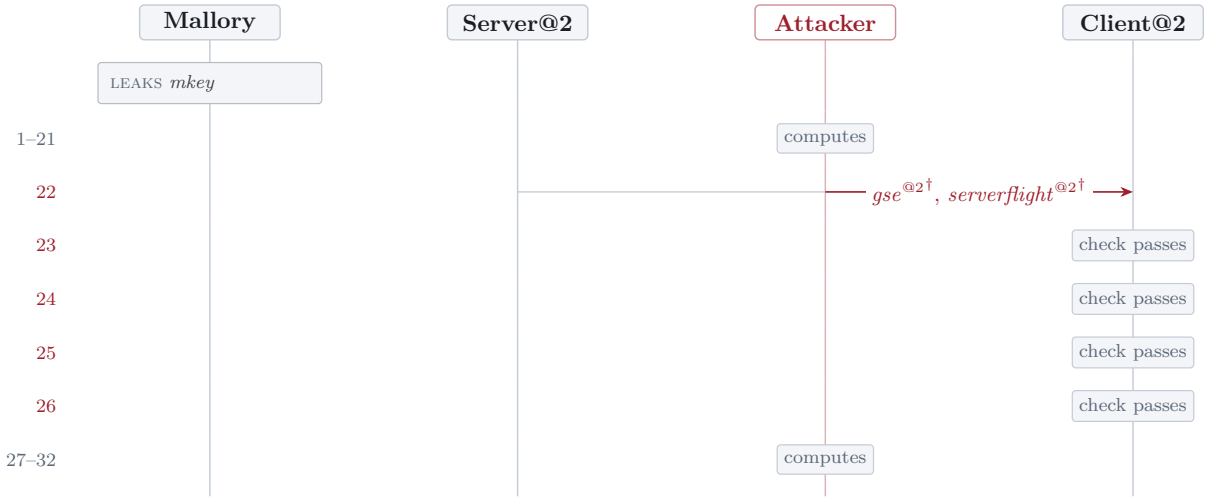

Figure 4: How the attacker breaks confidentiality? *clientid*. Substituted values carry a dagger.

1. Attacker constructs PUBKEY(*nil*).
2. Attacker constructs *earlysecret*.
3. Attacker constructs *derivedes*.
4. Attacker observes $gce^{@2}$ on the wire.
5. Attacker constructs $dhe_c^{@2}$, where it is $\mathsf{DH}_{\mathsf{KEX}}(gce^{@2}, nil)$.
6. Attacker constructs $handshakesecret_c^{@2}$, where it is $\mathsf{HKDF}(derivedes, dhe_c^{@2}, nil)_1$.
7. Attacker constructs $transcriptsh_c^{@2}$, where it is $\mathsf{HASH}(gce^{@2}, \mathsf{PUBKEY}(nil))$.
8. Attacker constructs $shstraffic_c^{@2}$, where it is $\mathsf{HKDF}(handshakesecret_c^{@2}, labelshstraffic, transcriptsh_c^{@2})_1$.
9. Attacker constructs $shsiv_c^{@2}$, where it is $\mathsf{HKDF}(shstraffic_c^{@2}, labeliv, nil)_1$.
10. Attacker constructs $\mathsf{HASH}(shsiv_c^{@2}, seq0)$.
11. Attacker observes *gmsigkey* on the wire.
12. Attacker observes *mallcertsig* on the wire.
13. Attacker is handed *mkey* by a leaks declaration in Mallory.
14. Attacker constructs $transcriptcert_c^{@2}$, where it is $\mathsf{HASH}(transcriptsh_c^{@2}, nil, gmsigkey, mallcertsig)$.
15. Attacker constructs $\mathsf{CONCAT}(ctxservercv, transcriptcert_c^{@2})$.
16. Attacker constructs $servercv_c^{@2}$, where it is $\mathsf{SIGN}(mkey, \mathsf{CONCAT}(ctxservercv, transcriptcert_c^{@2}))$.
17. Attacker constructs $sfinishedkey_c^{@2}$, where it is $\mathsf{HKDF}(shstraffic_c^{@2}, labelfinished, nil)_1$.
18. Attacker constructs $transcriptcv_c^{@2}$, where it is $\mathsf{HASH}(transcriptcert_c^{@2}, servercv_c^{@2})$.
19. Attacker constructs $serverfinished_c^{@2}$, where it is $\mathsf{MAC}(sfinishedkey_c^{@2}, transcriptcv_c^{@2})$.
20. Attacker constructs $serverflightdec^{@2}$, where it is $\mathsf{CONCAT}(nil, gmsigkey, mallcertsig, servercv_c^{@2}, serverfinished_c^{@2})$.
21. Attacker constructs $\mathsf{AEAD}_{\mathsf{ENC}}(shstraffic_c^{@2}, \mathsf{HASH}(shsiv_c^{@2}, seq0), serverflightdec^{@2}, recordheader)$.
22. Attacker replaces $gse^{@2}$, $serverflight^{@2}$ (sent by Server@2 to Client@2) with $\mathsf{PUBKEY}(nil)$, $\mathsf{AEAD}_{\mathsf{ENC}}(shstraffic_c^{@2}, \mathsf{HASH}(shsiv_c^{@2}, seq0), serverflightdec^{@2}, recordheader)$. ($gse^{@2}$ was $\mathsf{PUBKEY}(se^{@2})$; $serverflight^{@2}$ was $\mathsf{AEAD}_{\mathsf{ENC}}(shstraffic^{@2}, \mathsf{HASH}(shsiv^{@2}, seq0), serverflightdec^{@2}, recordheader)$) Server@2 → Client@2
23. Client@2's $\mathsf{AEAD}^{?}_{\mathsf{DEC}}(shstraffic_c^{@2}, \mathsf{HASH}(shsiv_c^{@2}, seq0), \mathsf{AEAD}_{\mathsf{ENC}}(shstraffic_c^{@2}, \mathsf{HASH}(shsiv_c^{@2}, seq0), \mathsf{CONCAT}(nil, gmsigkey, mallcertsig, servercv_c^{@2}, serverfinished_c^{@2}), recordheader), recordheader)$ passes — the attacker controls one of its inputs.
24. Client@2's $\mathsf{SIGNVERIF}^{?}(capk, \mathsf{CONCAT}(mallname, gmsigkey), mallcertsig)$ passes — the attacker controls one of its inputs.
25. Client@2's $\mathsf{SIGNVERIF}^{?}(gmsigkey, \mathsf{CONCAT}(ctxservercv, transcriptcert_c^{@2}), servercv_c^{@2})$ passes — the attacker controls one of its inputs.
26. Client@2's $\mathsf{ASSERT}^{?}(serverfinished_c^{@2}, serverfinished_c^{@2})$ passes — the attacker controls one of its inputs.
27. Attacker observes $clientflight^{@2}$ on the wire, where it is $\mathsf{AEAD}_{\mathsf{ENC}}(chstraffic_c^{@2}, \mathsf{HASH}(chsiv_c^{@2}, seq0), \mathsf{CONCAT}(clientid, clientpk, clientcertsig, clientcv^{@2}, clientfinished^{@2}), recordheader)$.

28 Attacker constructs $chstraffic_c^{@2}$, where it is HKDF($handshakesecret_c^{@2}$, $labelchstraffic$, $transcriptsh_c^{@2}$)$_1$.
29 Attacker constructs $chsiv_c^{@2}$, where it is HKDF($chstraffic_c^{@2}$, $labeliv$, nil)$_1$.
30 Attacker constructs HASH($chsiv_c^{@2}$, $seq0$).
31 Attacker opens $clientflight^{@2}$ with $chstraffic_c^{@2}$, HASH($chsiv_c^{@2}$, $seq0$), obtaining CONCAT($clientid$, $clientpk$, $clientcertsig$, $clientcv^{@2}$, $clientfinished^{@2}$).
32 Attacker splits CONCAT($clientid$, $clientpk$, $clientcertsig$, $clientcv^{@2}$, $clientfinished^{@2}$) and takes $clientid$.

1.4 Scope and assumptions

Every verdict above was reached against an active attacker, with each principal running 2 concurrent sessions, over exactly the model as written. An attack is a witness and stands on its own. A query reported as holding says only that this search found no attack at those parameters: the search space this engine defines was explored, which is never the space of all attacks.

- Scenario: Client runs with *peercert* as *servercert*, an honest peer.
- Scenario: Client runs with *peercert* as *mallcert*, a compromised peer.
- Per-scenario values and principals carry the suffix @2 onward.

A On soundness

Verifpal is sound but incomplete. Every attack shown here is a genuine attack on the model as written; a query reported as holding means no attack was found within the search this run performed, which is never a proof that none exists.

B Model source

B.1 `tls13.vp`

```
// SPDX-FileCopyrightText: © 2019-2026 Nadim Kobeissi <nadim@symbolic.software>
// SPDX-License-Identifier: GPL-3.0-only
//
// TLS 1.3 (RFC 9846, which obsoletes RFC 8446): the full handshake of
// Figure 1 with certificate-based authentication of both endpoints,
// followed by the post-handshake messages of §4.7. The server
// authenticates with a certificate, the server requests one from the
// client and the client authenticates with it, both certificates are
// issued by a certificate authority, and once the handshake completes
// the server hands out two resumption tickets and rotates its sending
// key. The companion model tls13-0rtt.vp is the resumption handshake
// those tickets are for.
//
// What is here, section by section:
//   §4.1   the running transcript hash, extended message by message;
//   §4.2   ClientHello and ServerHello as key shares;
//   §4.3.8 fresh key shares per connection (`generates ce`, `generates se`);
//   §4.5.1 Certificate as (name, key, CA signature over both), for both
//          endpoints, validated against a trust anchor received guarded;
//   §4.5.2 CertificateVerify over the transcript, prefixed by the context
//          string that separates a server's signature from a client's;
//   §4.5.3 Finished as a MAC keyed by a finished key expanded from the
//          sender's handshake traffic secret, over the transcript through
//          CertificateVerify;
//   §5.2   records as AEAD under the traffic secret of the sender, with
//          the record header as the additional data;
//   §5.3   the per-record nonce formed from the sender's write IV and the
//   §7.3   record sequence number, the IV expanded from the traffic
//          secret under its own label, so a record is accepted only in
```

```
//          its place (F.2, order protection) and nothing about the nonce
//          travels;
//   §7.1   the key schedule from HKDF-Extract(0, 0) through the derived
//          steps to the main secret, every Derive-Secret carrying its own
//          label and the transcript hash of that point, plus the exporter
//          and resumption secrets hanging off the main secret;
//   §4.7.1 NewSessionTicket, with the PSK derived from the resumption
//          secret and the ticket nonce, two tickets on one connection;
//   §4.7.3 KeyUpdate under the old key, then §7.2's next-generation
//   §7.2   application traffic secret for the record that follows.
//
// Expected: c1c0c0c0c0c0e0c0c0c1c0c1a0a0a0a0a0f0f0, at two sessions and
// at one. That code was derived by reading the protocol, not by running
// the model; confirm it before pinning it.
//
// The queries follow Appendix F, and three of them fail on purpose. Read
// each failure with the RFC open, because each one is a sentence the RFC
// itself wrote.
//
// Secrecy of the session keys (F.1), and CK01's proviso:
//   confidentiality? sHsTraffic                                  FAILS
//   confidentiality? sHsTraffic  [precondition: server completes]  HOLDS
//   confidentiality? cHsTraffic_c [precondition: client completes] HOLDS
//   confidentiality? sApTraffic  [precondition: server completes]  HOLDS
//   confidentiality? cApTraffic_c [precondition: client completes] HOLDS
//   confidentiality? exporterSecret [precondition: server completes] HOLDS
//   equivalence? sApTraffic, sApTraffic_c [both complete]        HOLDS
// The server derives its handshake secrets from whatever key share
// arrives in the ClientHello, before anything has been checked, so an
// attacker that opens a connection with a share of its own owns a
// sHsTraffic. F.1 says as much: "the attacker can establish its own
// session keys with the server, but those session keys are distinct from
// those established by the client." What the server does with such a key
// is halt, at the client's CertificateVerify, without ever sending
// application data under it. The gated queries say exactly what F.1 says
// under "Establishing the same session keys": the guarantee holds
// "provided that it completes successfully on each endpoint". The
// precondition restricts each query to executions in which the named
// endpoint goes on past its last check, and in every such execution the
// keys are a two-party secret and the two sides hold the same one. The
// client has the same shape on its side: it derives cHsTraffic_c before
// it can decrypt the server's flight, so the ungated client-side query
// would fail the same way, and the gated one holds.
//
// Application data (F.2 confidentiality, F.1 forward secrecy):
//   confidentiality? appMsg1     HOLDS
//   confidentiality? appMsg2     HOLDS
//   confidentiality? appMsg3     FAILS
// Phase 1 leaks both long-term signing keys after the connection is over.
// Nothing recorded opens: the traffic secrets descend from the ephemeral
// dhe = DH_KEX(gce, se), and neither signing key is an ingredient. That is
// F.1's "forward secret with respect to long-term keys". Phase 2 leaks
// sApTraffic1, the server's sending secret after its KeyUpdate. appMsg3
// was encrypted under it and falls, the attacker expanding the write IV
// from the leaked secret and forming the nonce as the client does;
// appMsg2 was encrypted under the generation before, which
// HKDF-Expand-Label cannot be run backwards to, and holds. That is F.2's
// "forward secrecy after key change", and the only failure among the
// application data. F.2 also says the other direction is not provided:
// leak generation 0 instead and every later generation follows from it.
// This model does not show that, because one file can leak in one order.
//
// Resumption (F.1, tickets):
//   confidentiality? resPsk2     HOLDS
// Phase 2 also leaks resPsk1, the PSK behind the first ticket. The second
```

```
// ticket's PSK is a separate expansion of the resumption secret under its
// own nonce, and survives, as does the resumption secret itself under the
// traffic-secret leak: "the resumption secret computed by connection N
// and needed to form connection N+1 is separate from the traffic keys
// used by connection N".
//
// Endpoint identities (F.1):
//   confidentiality? clientId    FAILS
// The client's identity travels only inside its Certificate, sealed under
// cHsTraffic_c after the server has authenticated, so a network attacker
// never sees it. It is disclosed all the same, and the reason is the
// second scenario: there the client is configured to reach Mallory, whose
// signing key the attacker holds, and the attacker completes the server
// side of that handshake with Mallory's genuine certificate. The client
// then does what F.1 promises it will do: "it only reveals its identity
// to an authenticated server". Mallory is one. Verifpal's confidentiality
// query asks whether the attacker knows the value, not whom the client
// meant to tell, and a `knows private` identity is the same value in
// every run. Delete the second scenario and this query is expected to
// hold. The server's identity is public here, as a hostname is; the RFC
// protects the server's certificate only against passive attackers, and
// an active one learns it by opening a connection.
//
// Peer authentication (F.1) and integrity with non-replay (F.2):
//   authentication? Server → Client: serverFlight   HOLDS
//   authentication? Client → Server: clientFlight   HOLDS
//   authentication? Client → Server: appData1       HOLDS
//   authentication? Server → Client: appData2       HOLDS
//   authentication? Server → Client: appData3       HOLDS
// Verifpal's authentication query is injective agreement, so a hold rules
// out replay as well as forgery. The server's flight is accepted only
// under a signature, by a key a trust anchor vouches for, over a
// transcript that contains the client's own key share; the client's
// flight is accepted under the same construction, over a transcript that
// also contains the server's certificate and Finished. The second half is
// where the scenarios[] block earns its place. The client that talks to
// Mallory signs a CertificateVerify for that handshake, and the attacker
// holds it afterwards. Reusing it against the honest server is the attack
// F.1 describes for constructions "that do not bind to the server's
// certificate" ([PSK-FINISHED], [Kraw16]), and it fails here because the
// client's signature covers the server's Certificate and the shares of
// this connection. Every record after the handshake is sealed under a
// secret that mixes both ephemeral shares, so a record lifted from a
// concurrent connection decrypts nowhere, and the sequence number in the
// nonce keeps the four records the server sends under one key from
// standing in for each other.
//
// Uniqueness of the session keys (F.1):
//   freshness? sApTraffic, freshness? cApTraffic_c   HOLD
// Both traffic secrets descend from the per-connection se and ce.
//
// Modelling notes.
//
// The scenarios[] block gives the client two configurations: the
// certificate of the honest server, or Mallory's. In TLS that
// configuration is a hostname. It is a certificate here because Verifpal
// tells a corrupt peer from an honest one by following a leaked key into
// the values computed from it, and a bare name is computed from nothing:
// Mallory's certificate carries Mallory's public key, whose private half
// leaks, so a run configured with it is a run with a corrupt peer and its
// claims are not the protocol's to answer for. The client reads the
// expected name out of the configured certificate and checks the CA's
// signature over that name and the key in the Certificate message it
// receives, which is hostname verification as a browser does it; nothing
// else about the configured certificate is consulted. Check the received
```

```
// name instead and the attacker presents Mallory's genuine certificate to
// a client that wanted the honest server, which is the misbinding of F.9.
//
// The server's certificate check has the same shape, and needs it. The
// CA certifies Mallory too, so a server that accepted any certificate
// the CA issued would accept Mallory as its client, and the attacker,
// holding Mallory's key, would complete the client side of the handshake
// against the honest server: the client-flight and appData1
// authentication queries would fail and the server's gated session
// secrets would be shared with the attacker, all of it correct TLS with
// a peer the server never meant to serve. A deployment with client
// certificates decides which certified identities it serves, so the
// server here holds the identity of its one client and checks the CA's
// signature over that identity and the key in the certificate it
// receives. The transcript still covers the bytes as received (§4.1).
//
// Client certificates change the reading of this model against the
// server-only handshake that browsers run. Without them the server has no
// way to tell the client from an attacker with a key share, and every
// claim the server makes about who it talked to fails: authentication of
// the client's flight and data, and the confidentiality of the server's
// own session secrets, since "the client" may simply be the attacker.
// With them those claims hold, and the ungated sHsTraffic query above is
// what remains of that story.
//
// Records follow §5.2 and §5.3. Each traffic secret stands for the write
// key §7.3 expands from it, and the write IV is expanded from it under
// the "iv" label; the per-record nonce is that IV combined with the
// record sequence number, written HASH(iv, seq) because Verifpal has no
// XOR, and the additional data is the record header, the same public
// bytes on every record. Sequence numbers start over whenever the key
// changes (§5.3), so the first record under the updated secret is seq0
// again. An earlier version of this model drew a fresh nonce per record
// and sent it beside the ciphertext; TLS does neither, and the extra
// wire slot per record was what put the analysis out of reach.
//
// Three things the RFC has are left out because they cannot move a
// verdict and each one widened the search. The ClientHello and
// ServerHello randoms are absent: the transcript already binds the fresh
// key shares, and a random is a bare constant nothing checks on its own.
// EncryptedExtensions and CertificateRequest are absent from the
// transcript: as constants both sides already hold they could only sit in
// it inertly (parameter negotiation is not modelled, see below). And each
// endpoint's name check is folded into its certificate check, so a
// certificate for the wrong name fails one check earlier than it did and
// the endpoint halts in the same place.
//
// Not modelled: parameter negotiation and its downgrade protection
// (there is one cipher suite and one group), HelloRetryRequest and
// cookies, the PSK and 0-RTT modes (tls13-0rtt.vp), post-handshake
// client authentication, the exporter interface over the exporter secret,
// alerts, record padding, and the within-connection sequence-number
// checks beyond what the nonce expresses. Key compromise impersonation
// (F.1) is not tested either: it needs a handshake run after the
// client's key has leaked, and every handshake here runs at phase 0.

attacker[active]

principal CA[
    knows private caKey
    caPk = PUBKEY(caKey)
]

principal Server[
    knows private sigKey
```

```
    knows public serverName
    gSigKey = PUBKEY(sigKey)
]

principal Client[
    knows private clientSigKey
    knows private clientId
    clientPk = PUBKEY(clientSigKey)
]

// A second certified server. Its signing key is the attacker's from the
// start, so it stands for any legitimately certified site the attacker
// controls.
principal Mallory[
    knows private mkey
    knows public mallName
    gmSigKey = PUBKEY(mkey)
    leaks mkey
]

// Enrolment with the CA is out of band and guarded: a CA that certifies
// whatever key arrives on an open wire would be an oracle.
Server → CA: [gSigKey]
Client → CA: [clientPk]
Mallory → CA: [gmSigKey]

principal CA[
    knows public serverName, mallName
    knows private clientId
    // §4.5.1: a certificate binds an identity to a key. The client's
    // identity is private, since the client's certificate must stay
    // confidential (F.1), so its certificate reaches it as the CA's
    // signature alone and the client assembles the message itself.
    serverCertSig = SIGN(caKey, CONCAT(serverName, gSigKey))
    clientCertSig = SIGN(caKey, CONCAT(clientId, clientPk))
    mallCertSig = SIGN(caKey, CONCAT(mallName, gmSigKey))
]

// The trust anchor is distributed independently (§4.5.1), hence guarded.
CA → Server: [caPk], [serverCertSig]
CA → Client: [caPk], [clientCertSig]
CA → Mallory: [mallCertSig]

principal Server[
    serverCert = CONCAT(serverName, gSigKey, serverCertSig)
]

principal Mallory[
    mallCert = CONCAT(mallName, gmSigKey, mallCertSig)
]

// The client's configuration: which site it means to reach. See the
// modelling notes above for why this is a certificate and not a name.
Server → Client: [serverCert]
Mallory → Client: [mallCert]

principal Client[
    knows public peerCert
    knows public recordHeader, updateNotRequested
    knows public seq0, seq1, seq2, seq3
    knows public labelDerived, labelFinished, labelIv
    knows public labelCHsTraffic, labelSHsTraffic
    knows public labelCApTraffic, labelSApTraffic
    knows public labelExpMaster, labelResMaster
    knows public labelResumption, labelTrafficUpd
```

```
290     knows public ctxServerCV, ctxClientCV
291     expectedName, _, _ = SPLIT(peerCert)
292     // ClientHello (§4.2.2): a key share, fresh per connection.
293     generates ce
294     gce = PUBKEY(ce)
295 ]
296
297 Client → Server: gce
298
299 principal Server[
300     // The server knows which client it serves; see the modelling notes.
301     knows private clientId
302     knows public recordHeader, updateNotRequested
303     knows public seq0, seq1, seq2, seq3
304     knows public labelDerived, labelFinished, labelIv
305     knows public labelCHsTraffic, labelSHsTraffic
306     knows public labelCApTraffic, labelSApTraffic
307     knows public labelExpMaster, labelResMaster
308     knows public labelResumption, labelTrafficUpd
309     knows public ctxServerCV, ctxClientCV
310     // ServerHello (§4.2.3).
311     generates se
312     gse = PUBKEY(se)
313     dhe = DH_KEX(gce, se)
314     // §7.1. HKDF(salt, ikm, nil) is HKDF-Extract; HKDF(secret, label,
315     // transcript) is Derive-Secret. With no PSK the early secret is
316     // HKDF-Extract(0, 0), a public value, and secrecy enters with dhe.
317     earlySecret = HKDF(nil, nil, nil)
318     derivedES = HKDF(earlySecret, labelDerived, nil)
319     handshakeSecret = HKDF(derivedES, dhe, nil)
320     transcriptSH = HASH(gce, gse)
321     cHsTraffic = HKDF(handshakeSecret, labelCHsTraffic, transcriptSH)
322     sHsTraffic = HKDF(handshakeSecret, labelSHsTraffic, transcriptSH)
323     // §7.3: the write IVs of both directions' handshake keys.
324     cHsIv = HKDF(cHsTraffic, labelIv, nil)
325     sHsIv = HKDF(sHsTraffic, labelIv, nil)
326     derivedHS = HKDF(handshakeSecret, labelDerived, nil)
327     mainSecret = HKDF(derivedHS, nil, nil)
328     // §4.5.1: the transcript through the Certificate.
329     transcriptCert = HASH(transcriptSH, serverName, gSigKey, serverCertSig)
330     // §4.5.2: CertificateVerify, with the server context string.
331     serverCV = SIGN(sigKey, CONCAT(ctxServerCV, transcriptCert))
332     transcriptCV = HASH(transcriptCert, serverCV)
333     // §4.5.3: Finished under the finished key of the base key.
334     sFinishedKey = HKDF(sHsTraffic, labelFinished, nil)
335     serverFinished = MAC(sFinishedKey, transcriptCV)
336     // Certificate, CertificateVerify and Finished, sealed under the
337     // server handshake traffic secret (§5.2), record 0 under that key.
338     serverFlight = AEAD_ENC(sHsTraffic, HASH(sHsIv, seq0), CONCAT(serverName, gSigKey,
        serverCertSig, serverCV, serverFinished), recordHeader)
339     transcriptSFin = HASH(transcriptCV, serverFinished)
340     cApTraffic = HKDF(mainSecret, labelCApTraffic, transcriptSFin)
341     sApTraffic = HKDF(mainSecret, labelSApTraffic, transcriptSFin)
342     cApIv = HKDF(cApTraffic, labelIv, nil)
343     sApIv = HKDF(sApTraffic, labelIv, nil)
344     exporterSecret = HKDF(mainSecret, labelExpMaster, transcriptSFin)
345 ]
346
347 Server → Client: gse, serverFlight
348
349 principal Client[
350     dhe_c = DH_KEX(gse, ce)
351     earlySecret_c = HKDF(nil, nil, nil)
352     derivedES_c = HKDF(earlySecret_c, labelDerived, nil)
353     handshakeSecret_c = HKDF(derivedES_c, dhe_c, nil)
```

```
    transcriptSH_c = HASH(gce, gse)
    cHsTraffic_c = HKDF(handshakeSecret_c, labelCHsTraffic, transcriptSH_c)
    sHsTraffic_c = HKDF(handshakeSecret_c, labelSHsTraffic, transcriptSH_c)
    cHsIv_c = HKDF(cHsTraffic_c, labelIv, nil)
    sHsIv_c = HKDF(sHsTraffic_c, labelIv, nil)
    derivedHS_c = HKDF(handshakeSecret_c, labelDerived, nil)
    mainSecret_c = HKDF(derivedHS_c, nil, nil)
    serverFlightDec = AEAD_DEC(sHsTraffic_c, HASH(sHsIv_c, seq0), serverFlight, recordHeader)?
    certName, certKey, certSig, serverCV_c, serverFinished_c = SPLIT(serverFlightDec)
    // §4.5.1.3: the chain is validated against the trust anchor, for the
    // name the client meant to reach.
    _ = SIGNVERIF(caPk, CONCAT(expectedName, certKey), certSig)?
    transcriptCert_c = HASH(transcriptSH_c, certName, certKey, certSig)
    // §4.5.2: the signature is checked under the key in the certificate,
    // not under anything the client knew in advance.
    _ = SIGNVERIF(certKey, CONCAT(ctxServerCV, transcriptCert_c), serverCV_c)?
    transcriptCV_c = HASH(transcriptCert_c, serverCV_c)
    sFinishedKey_c = HKDF(sHsTraffic_c, labelFinished, nil)
    expectedServerFinished = MAC(sFinishedKey_c, transcriptCV_c)
    _ = ASSERT(serverFinished_c, expectedServerFinished)?
    transcriptSFin_c = HASH(transcriptCV_c, serverFinished_c)
    cApTraffic_c = HKDF(mainSecret_c, labelCApTraffic, transcriptSFin_c)
    sApTraffic_c = HKDF(mainSecret_c, labelSApTraffic, transcriptSFin_c)
    cApIv_c = HKDF(cApTraffic_c, labelIv, nil)
    sApIv_c = HKDF(sApTraffic_c, labelIv, nil)
    exporterSecret_c = HKDF(mainSecret_c, labelExpMaster, transcriptSFin_c)
    // The client's authentication block (§4.5, Table 2): Certificate,
    // CertificateVerify with the client context string over the
    // transcript through server Finished, and Finished under the client
    // handshake traffic secret.
    transcriptCCert = HASH(transcriptSFin_c, clientId, clientPk, clientCertSig)
    clientCV = SIGN(clientSigKey, CONCAT(ctxClientCV, transcriptCCert))
    transcriptCCV = HASH(transcriptCCert, clientCV)
    cFinishedKey_c = HKDF(cHsTraffic_c, labelFinished, nil)
    clientFinished = MAC(cFinishedKey_c, transcriptCCV)
    clientFlight = AEAD_ENC(cHsTraffic_c, HASH(cHsIv_c, seq0), CONCAT(clientId, clientPk,
    clientCertSig, clientCV, clientFinished), recordHeader)
    // §7.1: the resumption secret covers the transcript through client
    // Finished.
    transcriptCFin_c = HASH(transcriptCCV, clientFinished)
    resumptionSecret_c = HKDF(mainSecret_c, labelResMaster, transcriptCFin_c)
    generates appMsg1
    appData1 = AEAD_ENC(cApTraffic_c, HASH(cApIv_c, seq0), appMsg1, recordHeader)
]

Client → Server: clientFlight, appData1

principal Server[
    clientFlightDec = AEAD_DEC(cHsTraffic, HASH(cHsIv, seq0), clientFlight, recordHeader)?
    clientId_s, clientPk_s, clientCertSig_s, clientCV_s, clientFinished_s = SPLIT(clientFlightDec)

    // The chain is validated against the trust anchor, for the client
    // this server serves.
    _ = SIGNVERIF(caPk, CONCAT(clientId, clientPk_s), clientCertSig_s)?
    transcriptCCert_s = HASH(transcriptSFin, clientId_s, clientPk_s, clientCertSig_s)
    _ = SIGNVERIF(clientPk_s, CONCAT(ctxClientCV, transcriptCCert_s), clientCV_s)?
    transcriptCCV_s = HASH(transcriptCCert_s, clientCV_s)
    cFinishedKey_s = HKDF(cHsTraffic, labelFinished, nil)
    expectedClientFinished = MAC(cFinishedKey_s, transcriptCCV_s)
    _ = ASSERT(clientFinished_s, expectedClientFinished)?
    transcriptCFin = HASH(transcriptCCV_s, clientFinished_s)
    resumptionSecret = HKDF(mainSecret, labelResMaster, transcriptCFin)
    // §4.5.3: application data from the client is read only once its
    // Finished has been verified.
    appMsg1_s = AEAD_DEC(cApTraffic, HASH(cApIv, seq0), appData1, recordHeader)?
```

```
    // §4.7.1: two NewSessionTickets, each with its own nonce and so its
    // own PSK; the tickets are records 0 and 1 under sApTraffic.
    generates ticketNonce1, ticketNonce2
    resPsk1 = HKDF(resumptionSecret, labelResumption, ticketNonce1)
    resPsk2 = HKDF(resumptionSecret, labelResumption, ticketNonce2)
    newSessionTicket1 = AEAD_ENC(sApTraffic, HASH(sApIv, seq0), ticketNonce1, recordHeader)
    newSessionTicket2 = AEAD_ENC(sApTraffic, HASH(sApIv, seq1), ticketNonce2, recordHeader)
    generates appMsg2
    appData2 = AEAD_ENC(sApTraffic, HASH(sApIv, seq2), appMsg2, recordHeader)
    // §4.7.3: KeyUpdate goes out under the old key; §7.2: the next
    // generation is one HKDF-Expand-Label away, with its own write IV,
    // and its sequence numbers start over (§5.3).
    keyUpdate = AEAD_ENC(sApTraffic, HASH(sApIv, seq3), updateNotRequested, recordHeader)
    sApTraffic1 = HKDF(sApTraffic, labelTrafficUpd, nil)
    sApIv1 = HKDF(sApTraffic1, labelIv, nil)
    generates appMsg3
    appData3 = AEAD_ENC(sApTraffic1, HASH(sApIv1, seq0), appMsg3, recordHeader)
]

Server → Client: newSessionTicket1, newSessionTicket2, appData2, keyUpdate, appData3

principal Client[
    ticketNonce1_c = AEAD_DEC(sApTraffic_c, HASH(sApIv_c, seq0), newSessionTicket1, recordHeader)
    ?
    ticketNonce2_c = AEAD_DEC(sApTraffic_c, HASH(sApIv_c, seq1), newSessionTicket2, recordHeader)
    ?
    resPsk1_c = HKDF(resumptionSecret_c, labelResumption, ticketNonce1_c)
    resPsk2_c = HKDF(resumptionSecret_c, labelResumption, ticketNonce2_c)
    appMsg2_c = AEAD_DEC(sApTraffic_c, HASH(sApIv_c, seq2), appData2, recordHeader)?
    keyUpdate_c = AEAD_DEC(sApTraffic_c, HASH(sApIv_c, seq3), keyUpdate, recordHeader)?
    _ = ASSERT(keyUpdate_c, updateNotRequested)?
    sApTraffic1_c = HKDF(sApTraffic_c, labelTrafficUpd, nil)
    sApIv1_c = HKDF(sApTraffic1_c, labelIv, nil)
    appMsg3_c = AEAD_DEC(sApTraffic1_c, HASH(sApIv1_c, seq0), appData3, recordHeader)?
]

phase[1]

// Both long-term keys are compromised after the connection has closed
// (F.1, forward secrecy with respect to long-term keys).
principal Server[
    leaks sigKey
]

principal Client[
    leaks clientSigKey
]

phase[2]

// Then the server's current sending secret and one ticket's PSK (F.2,
// forward secrecy after key change; F.1, independence of tickets).
principal Server[
    leaks sApTraffic1, resPsk1
]

scenarios[
    Client[peerCert = serverCert]
    Client[peerCert = mallCert]
]

queries[
    confidentiality? sHsTraffic
    confidentiality? sHsTraffic[
        precondition[Server → Client: appData2]
```

```
480     ]
481     confidentiality? cHsTraffic_c[
482         precondition[Client → Server: clientFlight]
483     ]
484     confidentiality? sApTraffic[
485         precondition[Server → Client: appData2]
486     ]
487     confidentiality? cApTraffic_c[
488         precondition[Client → Server: clientFlight]
489     ]
490     confidentiality? exporterSecret[
491         precondition[Server → Client: appData2]
492     ]
493     equivalence? sApTraffic, sApTraffic_c[
494         precondition[Server → Client: appData2]
495         precondition[Client → Server: clientFlight]
496     ]
497     confidentiality? appMsg1
498     confidentiality? appMsg2
499     confidentiality? appMsg3
500     confidentiality? resPsk2
501     confidentiality? clientId
502     authentication? Server → Client: serverFlight
503     authentication? Client → Server: clientFlight
504     authentication? Client → Server: appData1
505     authentication? Server → Client: appData2
506     authentication? Server → Client: appData3
507     freshness? sApTraffic
508     freshness? cApTraffic_c
509 ]
```