ProScript TLS: Building a TLS 1.3 Implementation with a Verifiable Protocol Model

Karthikeyan Bhargavan
Nadim Kobeissi
Bruno Blanchet
miTLS and flexTLS

miTLS: reference implementation of TLS 1.0-1.2
- Written in F#, a .NET variant of OCaml
- Verified using a dependent type system (F7) + some crypto proofs in EasyCrypt
- Built to support proof from ground up
- See Antoine’s talk for new developments

flexTLS: specification-based testing for TLS
- Uses miTLS as a reference spec
- Good for experimenting with new features
- Easy to implement known attacks, find new ones
ProScript TLS

**Goal:** get developers to run light-weight analysis

- F#/F* too far a leap for many developers
- We want them to write their own tests and attacks
- But don’t break miTLS to make it easier to test with!

**A reference implementation of TLS in JavaScript**

- Protocol core written in a statically-typed, purely functional subset of JavaScript called ProScript
- Typing avoids common JavaScript pitfalls
- Model extraction & verification with ProVerif
- **Not a cryptographic proof!** Good for finding bugs.
ProScript TLS

Current implementation status
- Implements TLS 1.0-1.3 (1-RTT)
- RSA/DHE/ECDHE, AES-CBC/GCM
- Interoperates with NSS for TLS 1.3 for 1-RTT
- Interoperates with clients/servers for TLS 1.0-1.2

Current verification status
- An extracted model for the core of 1-RTT
- Verifies standard secrecy/authentication properties
- Ongoing work: 0-RTT, PSK, TLS 1.0-1.2
TLS 1.3 Protocol

Draft 11 specification
- 0-RTT with DH/PSK
- 0.5 RTT server data
- 0- or 1-RTT client auth

Complex key schedule
- Keys at derived at multiple stages
- Record keys change at different places
- Difficult to debug
(1) Client Auth + 0-RTT Data

Client C

Knows $sk_C, cert_S, g^s$

$\log_0$

Computes:

$\left( k_0^m, k_0^h, k_0^d \right) = \text{kdf}(g^{xs}, \log_0, g^s, cert_S)$

$\log'_0$

$\text{enc}_{k_0^h}(Certificate(cert_C))$

$\text{enc}_{k_0^h}(CertificateVerify(sign_{sk_C}(\log'_0)))$

$\text{enc}_{k_0^h}(\text{Finished}(\text{mac}_{k_0^m}(\log''_0)))$

$\text{enc}_{k_0^d}(Data(m_0))$

Server S

Knows $sk_S, s, cert_C$

$\log'_0$

Computes:

$\left( k_0^m, k_0^h, k_0^d \right) = \text{kdf}(g^{xs}, \log_0, g^s, cert_S)$

$\log'_0$

Client knows S’s semi-static key $g^s$

Client auth block

0-RTT data
(2) Server Auth + 0.5-RTT Data

Server Semi-Static

Server Auth Block

1.5-RTT Data
(3) Client Auth + 1-RTT Data

Client auth block (again)

- enc\(^{k_i^h}\) (Certificate\( cert_C\))
- enc\(^{k_i^h}\) (CertificateVerify\( \text{sign}^{sk_C}(\text{log}_5)\))
- enc\(^{k_i^h}\) (Finished\( \text{mac}^{k_i^m}(\text{log}_6)\))

 Computes \( rms, ems = kdf(g^{xs}, g^{xy}, \text{log}_7) \)

1-RTT Data

Conversation:
\( C \rightarrow S : m_0 \)
\( S \rightarrow C : m_1 \)
\( C \rightarrow S : m_2 \)
\( S \rightarrow C : m_3 \)

Composite Data Stream
TLS 1.3 in JavaScript

Handshake messages processed in flights

- Record layer collects message flights, then calls a purely functional callback function
- Callbacks return message flights to send back
- Some flights broken up to support key changes

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>send_client_hello</td>
<td>recv_client_hello</td>
</tr>
<tr>
<td>recv_server_hello</td>
<td>send_server_finished</td>
</tr>
<tr>
<td>recv_server_finished</td>
<td>recv_client_finished</td>
</tr>
<tr>
<td>write_data</td>
<td>write_data</td>
</tr>
<tr>
<td>read_data</td>
<td>read_data</td>
</tr>
</tbody>
</table>
Add new features, implement attacks

- Often only need to edit one callback
- Demos for Skip, Freak, Sloth

Ongoing work: systematic testing for TLS 1.3 (a la SMACK)

```javascript
/* Test for the Skip-CCS attack by deleting the ServerCCS message */
const Skip_CCS_server_callbacks = {
    hs_recv_client_hello: server_callbacks.hs_recv_client_hello,
    hs_recv_client_ccs: server_callbacks.hs_recv_client_ccs,
    hs_recv_client_finished: function(msgs, cipherState) {
        let out_msgs = server_callbacks.hs_recv_client_finished(msgs, cipherState);
        cipherState.require_alert = true;
        return [out_msgs[1]];
    }
}

/* Test for the SLOTH attack by modifying the ServerKeyExchange message */
const Sloth_server_callbacks = {
    hs_recv_client_hello: function(msgs, cipherState) {
        let out_msgs = server_callbacks.hs_recv_client_hello(msgs, cipherState);
        let ske = out_msgs[2];
        ske.sig.hash_alg = formats.HA.md5;
        ske.sign(cipherState.cr, cipherState.sr, cipherState.pv);
        return out_msgs
    },
    hs_recv_client_ccs: server_callbacks.hs_recv_client_ccs,
    hs_recv_client_finished: server_callbacks.hs_recv_client_finished,
};
```
TLS 1.3 in ProVerif

Deconstruct TLS source

- **Trusted Typed DJS:** Crypto, message formats
- **Verified ProScript:** Core protocol code (1000 / 4500 loc total)
- **Untrusted JavaScript:** Application, network, connection handling

Extract, verify in ProVerif

- add attacker model, security goals, tinker, …
Analysis: Weaknesses in 0-RTT Client Auth
Security Goals:

- **Secrecy:** Data $m_0$ known only to $C$ and $S$
- **Forward secrecy:** even if $sk_S$ leaks even if $s$ leaks?
- **Authentication:** $C$ and $S$ agree on sender $pk_C$, receiver $pk_S$, and data $m_0$
Known Weaknesses in 0-RTT

0-RTT data is not forward secret if $s$ is leaked
- Similarly, 0-RTT PSK is not forward secret
- Appears in our model as an attack
- *Solution*: rotate semi-static keys and PSKs

0-RTT data + client auth is replayable
- Authentication makes replay worse
- *Solution*: API needs to forbid 0-RTT POSTs e.g.
- **New query attack vector**: replayed 0-RTT requests may be responded with by 0.5-RTT response
- *Example*: authenticated GET whose response length is sensitive; even off-path attacks possible!
Unknown Key Share (Draft 7.5)

In draft >= 8, 1-RTT server does not prove possession of $s$

In draft 7, auth context did not include $cert_S$
Key Compromise Impersonation

Client $C$

Knows $sk_C, cert_M, g^s$

Computes:
\[
ctx = h_0 | g^s
\]
\[
(k_0^m, k_0^h, k_0^d) = kdf(g^{xs}, ctx)
\]

\[
h_0 = ClientHello(n_c, g^x)
\]

\[
\text{enc}^{k_0^h}(h_1 = Certificate(cert_C))
\]

\[
log_1 = ctx | h_1
\]

\[
\text{enc}^{k_0^h}(h_2 = CertificateVerify(sign^{sk_c}(log_1)))
\]

\[
log_2 = log_1 | h_2
\]

\[
\text{enc}^{k_0^h}(Finished(mac^{k_0^h}(log_2)))
\]

\[
\text{enc}^{k_0^d}(Data(m_0))
\]

Conversation:
\[C \rightarrow S : m_0\]

Server $M$

Knows $sk_M, g^s, s$

Server $S$

Knows $sk_S, s$

Computes:
\[
ctx = h_0 | g^s
\]
\[
(k_0^m, k_0^h, k_0^d) = kdf(g^{xs}, ctx)
\]

Attacker knows $s$

Inject data after authentication
Long-Term Client Impersonation

If client ephemeral x is leaked, attacker can forward 0-RTT client auth forever

- Result of replay + ephemeral compromise
- 0-RTT client auth unintentionally creates a long-term delegated credential
- *Example*: suppose client certificate is on smartcard, but ephemeral is on a public machine and can leak.
- *Solution*: Replay detection?
If attacker knows $x$, it can reuse client’s auth block any number of times.
Summary: 0-RTT Client Auth

- 0-RTT Auth is replayable and amplifies attacks on 0.5-RTT responses
- 0-RTT Auth is not forward-secure if $s$ leaks
- 0-RTT Auth is vulnerable to KCI if $s$ leaks
- 0-RTT Auth leaks signature capability if $x$ leaks

Question:
 Shall we get rid of certificate-based auth in 0-RTT?
 Many of these problems do not seem to occur with PSK.
Analysis: Mixing PSK with Signatures
PSK in Draft 11

Multiple modes and key sources
- Static PSK vs. Resumption Master Secret
- PSK-DHE vs. Pure PSK
- 0-RTT and/or 1-RTT PSK ciphers

PSK + certificate-based authentication
- Allowed for 1-RTT client auth (Thyla’s talk)
- Maybe allowed in 0-RTT, but underspecified
- Not allowed in 1-RTT server auth (but should be?)
- How to correctly compose PSK + signatures?
Pure PSK + Server Signature

No Ephemerals

Server sig covers pskid, but not psk

Conversation:
$S \rightarrow \text{anon} : m_1$
Impersonating Servers over Pure PSK

- Suppose C has a PSK with pskid at M
- Suppose attacker M has a PSK with same pskid at S
- M can forward S’s signature to C
- C thinks it is talking to M, but is talking to S
- Similar attack to Cremers et al (Thyla’s talk)
- No ServerFinished to save us here

Impersonating 0-RTT Clients over Static PSK

- Attacker M synchronizes pskid over C-M and C-S
- M forwards C’s signature to S
Certificate does not authenticate PSK, although PSK does authenticate certificate

- In isolation, this sub-protocol does not guarantee compound authentication, resulting in attacks
- Handshake encryption under PSK does not help
Compound Authentication

Solution: add more PSK-related info to context
- E.g. add resumption session hash to $pskid$
- E.g. add $cert$ to $ctx$, enforce unique $pskid$ per $cert$

Alternative: switch CertificateVerify and Finished
- MAC still covers certificate
- Signature now covers a PSK-based MAC
- Generic mitigation for authentication sub-protocol

\[
\begin{align*}
  a_1 &= \text{Certificate}(certs) \\
  a_2 &= \text{Finished}(\text{mac}^{psk}(ctx \mid a_1)) \\
  a_3 &= \text{CertificateVerify}(\text{sign}^{skS}(ctx \mid a_1 \mid a_2))
\end{align*}
\]
Conclusions

Verifying models derived from code is effective
  • Keeps us honest + enables quick formal feedback

0-RTT Client Auth is fragile against compromise
  • Well-documented in spec; get rid of it?

PSK + signatures do not mix easily
  • Protocol can be redesigned to compose them well

0-RTT replay is a source of headaches
  • Replay detection for client randoms?

Key schedule can potentially be simplified
  • Symbolic analysis does not detect new attacks
Questions?

• *Coming Soon*: ProScript TLS on GitHub