Proving security of TLS 1.3 protocol

Lasse Letager Hansen (letager@cs.au.dk)

AARHUS
UNIVERSITY

March 31, 2025 - LogSem Seminar

Results of combining tools

We can prove security and correctness of a realistic implementation of protocols

Results of combining tools

We can prove security and correctness of a realistic implementation of protocols

This is achieved by using a combination of tools:

- Hax
  - SSProve: Security proof of key schedule
  - F*: runtime safety (panic freedom), correctness of serialization and parsing
  - ProVerif: authenticity and confidentiality guarantees
- libcrux: secure and efficient implementations of cryptographic primitives

TLS 1.3

Transport Layer Security:

- Used for client-server communication across a network
- prevents eavesdropping and tampering
- uses handshake protocol to decide ciphers and exchange keys

Related projects

- Project Everest: build and deploy formally verified implementations of HTTPS components (such as TLS)
- TLS 1.3 triage panel: checking status of formal analysis for proposed changes (requires updates or changes)
- Twin transition:
  - using formal methods
  - post quantum

What are State Separating Proofs (SSP)

We can construct cryptographic proofs modularly by

- deconstructing programs and protocols into packages
- compose packages in parallel and serial to get larger programs

What are State Separating Proofs (SSP)

We can construct cryptographic proofs modularly by

- deconstructing programs and protocols into packages
- compose packages in parallel and serial to get larger programs

To prove security we

- Construct games (pairs of packages) and show indistinguishability
- Combine a sequence of game hops, to go from real to ideal behavior

What are State Separating Proofs (SSP)

We can construct cryptographic proofs modularly by

- deconstructing programs and protocols into packages
- compose packages in parallel and serial to get larger programs

To prove security we

- Construct games (pairs of packages) and show indistinguishability
- Combine a sequence of game hops, to go from real to ideal behavior

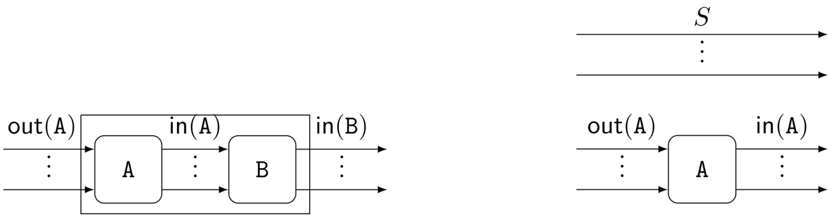

Figure: State Separation for Code-Based Game-Playing Proofs

What are State Separating Proofs (SSP)

Originating from

- the Everest project
- the Joy of Cryptography (book)

Also used for proofs by cryptographers

- Helps scale development and keep modularity

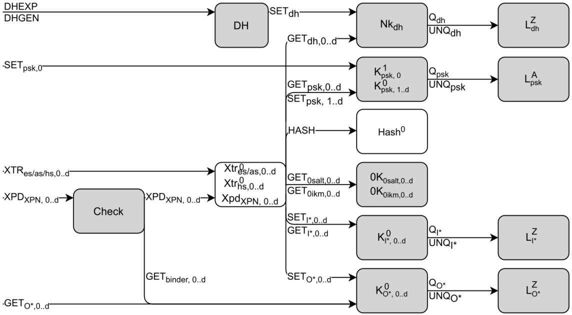

Figure: Image from "Key-schedule Security for the TLS 1.3 Standard"

What do we want to prove - Ideal protocol

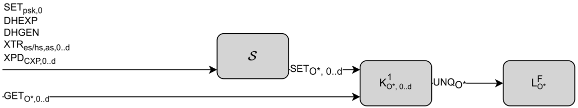

Figure: Image from "Key-schedule Security for the TLS 1.3 Standard"

From SSP to SSProve

From an existing informal proof, we construct a formal proof

- Write the code of the packages for each game hop
- Prove the correctness of composition of packages into games (Semi-automatic)
- Prove indistinguishability of each game
- Compose the games and show the advantage of an adversary is bounded

SSProve

- a foundational framework for modular cryptographic proofs in Coq
- a language with monadic state and probability
- game hopping style proofs in the computational model
- a program logic derived from the categorical Dijkstra monad framework

Differences from Paper proof

Indexing

- The Key Schedule is parameterized by a resumption bound (d)
- The Key Schedule Game (Gks) runs in rounds given by an index ℓ

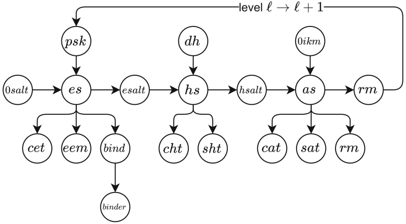

Figure: Image from "Key-schedule Security for the TLS 1.3 Standard"

- In each round we have a idealization order, grouping names in a sequence of steps

Differences from Paper proof

Wire Indexing

When constructing the protocol, we assign "wires". These are indexed by the bound, the round, and the level

$$wire^{b}_{t,d,\ell,n} = \texttt{start-offset} + n + \ell \cdot \#names + t \cdot (k+1) \cdot \#names$$

Given two wires and that $d \leq k$, we get no overlap if

- the names (n) differ
- the round indexes (ℓ) differ
- the wire types (t) differ
- the index of the other wire is before `start-offset`

Differences from Paper proof

Wire Indexing

When constructing the protocol, we assign "wires". These are indexed by the bound, the round, and the level

$$wire^{b}_{t,d,\ell,n} = \texttt{start-offset} + n + \ell \cdot \#names + t \cdot (k+1) \cdot \#names$$

Given two wires and that $d \leq k$, we get no overlap if

- the names (n) differ
- the round indexes (ℓ) differ
- the wire types (t) differ
- the index of the other wire is before `start-offset`

An artifact of verification, requiring disjointness and freshness of memory

- possibly made easier by an extension of SSProve using nominal sets

Differences from Paper proof

Composition order

The paper defines the Key Schedule as

$$G_{ks} = \bigcup_{\ell=0}^{d} G_{round_\ell}$$

$$G_{round_\ell} = \bigcup_{n \in N} P_{\ell,n}$$

Where d is a global/implicit argument.

Differences from Paper proof

Composition order

We defines the Key Schedule as

$$G_{ks} = \bigcup_{n \in N} G_{hierarchy\,n}$$

$$G_{hierarchy\,n} = \bigcup_{\ell=0}^{d} P_{\ell,n}$$

This seems to make the composition easier

- we only need to handle miss-alignment in the external cases e.g. with imports/exports

Differences from Paper proof

Composition order

We define all packages based on the horizontal and parallel constructions

$$Ks\ d\ N\ f_{\mathbb{B}} = \bigcup_{n \in N} \bigcup_{\ell=0}^{d} Key_{n,\ell}^{f_{\mathcal{B}}(n,\ell)}$$

$$Ls\ d\ N\ f_P = \bigcup_{n \in N} Log_n^{f_P(n)}$$

Generalize description of packages to bundles of similar interfaces

Differences from Paper proof

Assumption

We assume

- an implementation of a (secure) hashing algorithm
- that substituting Diffie-Hellman (DH) with a Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM) is still secure
  - Diffie-Hellman: the common standard for exchanging keys (weak to some forms of attack e.g. man-in-the-middle (MIM))
  - ML-KEM: post quantum secure key-exchange mechanism
- the implementation of the ML-KEM is secure

Application of Proof

Now we have a game proving security of a TLS-like key schedule.

- we instantiate the proof with an actual TLS-like implementation

Using the Hax framework, we

- translate the implementation to SSProve
- show equivalence between the translated code and real protocol (another game)

This gives us a (parameterized) security bound for the implementation.

Hax

- a subset of safe Rust with translations to proof assistants (F^*, Rocq, SSProve, ProVerif)
- executable specification in safe Rust
- used for writing specification and cryptographic implementation

Why Rust?

- memory safe
- ML-like type system
- as fast as C, industry grade
- used by cryptographers / software engineers

TLS Implementation

Key schedule implemented by

- Extract (XTR) and Expand (XPD) functions
- Parent name (PrntN) function

We instantiate Extract (XTR) and Expand (XPD) functions using

HMAC-based Extract-and-Expand Key Derivation Function (HKDF)

The Parent name function, defines the key derivation graph

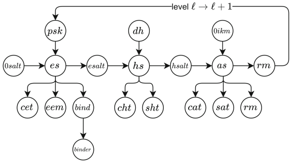

Figure: Image from "Key-schedule Security for the TLS 1.3 Standard"

TLS Implementation

Using Key schedule implementation for handshake

To implement the handshake protocol of TLS 1.3 we

- call XTR and XPD to step the graph
- bundle derivations in communication rounds
  - XTR_{ES}, $XPD_{\{CET,EEM,BIND,BINDER,ESALT\}}$
  - XTR_{HS}, $XPD_{\{CHT,SHT,HSALT\}}$
  - XTR_{AS}, $XPD_{\{CAT,SAT,RM,PSK\}}$
- Inject initial keys (PSK_0/no-PSK, 0_{IKM}, 0_{salt}, KEM)

TLS Implementation

Proofs help structure code

We use handles to separate the state from the keys.

- This adds (stronger) meta information to graph
  - Ensures that a given step, has the correct handle type
- This makes the code very modular and reusable (e.g. for MLS)

We use efficient and secure primitives from libcrux

- this ensures a realistic implementation, usable by even small/IOT devices

Security proof

Given all the parts above, we construct a sequence of game jumps

- instantiating proof
  - from implementation to real protocol
- modularize to enable SSP style proofs
  - from full real protocol to combination of modular parts
- idealizing parts
  - from real modular part to ideal modular part
- recombining parts
  - combining ideal parts, to get the full ideal protocol

Conclusion

- The cryptographic community is getting more interested in using formal methods
- SSP style of proofs invites modular and scalable implementations
  - proofs are re-usable
  - some work required to bundle and structure proof (somewhat automatable)
- Hax framework enables multi-tool verification effort, with a common reference implementation
- libcrux allows instantiation of primitives with a secure and efficient implementation