Proving the Core Security Theorem for the TLS-1.3 Key Schedule

Karthikeyan Bhargavan[1], **Lasse Letager Hansen**[2] (letager@cs.au.dk),
Franziskus Kiefer[1], Jonas Schneider-Bensch[1], and Bas Spitters[2]

June 30, 2025 - NordiCrypt

[1]Cryspen
[2]Aarhus University

Results of combining tools

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

Results of combining tools

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

This is achieved by using a combination of tools:

- Hax (from Rust to)
  - SSProve: Security proof of key schedule in the computational model
  - F$\star$: runtime safety (panic freedom), correctness of serialization and parsing
  - ProVerif: authenticity and confidentiality guarantees in the symbolic model (or 'Dolev-Yao')
- libcrux: secure and efficient implementations of cryptographic primitives

Related projects

- Project Everest
  - build and deploy formally verified implementations of HTTPS components (such as TLS).
- TLS 1.3 triage panel
  - report if proposed changes break any existing formal analysis
- Twin transition:
  - using formal methods
  - post-quantum
- Signal, WireGuard, MLS

F⋆

- is a proof-oriented programming language
- enables dependent types with proof automation based on SMT solving
- is, in this work, used for showing
  - panic freedom: e.g. there are no overflow, division by zero, out-of-bounds errors, etc.
  - parsing correctness: all messages are parsed correctly based on the serialization scheme.
  - thus covering the main source of practical attacks

ProVerif

- is an automated tool for checking security protocols
- works in the symbolic model (or 'Dolev-Yao')
- automatically verifies security properties, such as confidentiality, integrity, and authenticity
- works on a protocol model written in terms of message-passing processes
- here we use the proofs of F$\star$ and SSProve in the computational model to cover assumptions.

What are State Separating Proofs (SSP)

We can construct cryptographic proofs modularly by

- deconstructing programs and protocols into packages (program fragments)
- 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 (program fragments)
- 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)

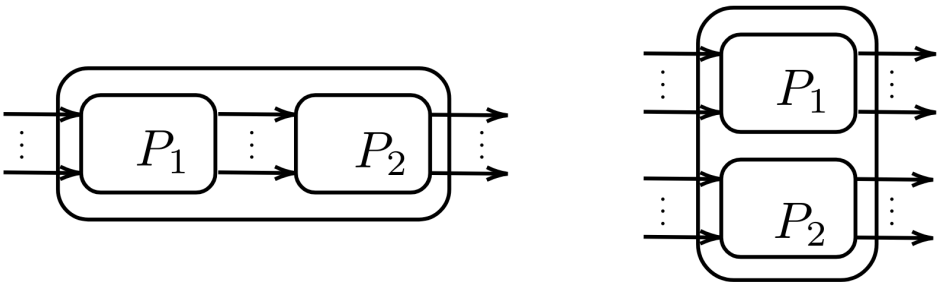

Figure: Image from "SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq"

What do we want to prove - protocol specification

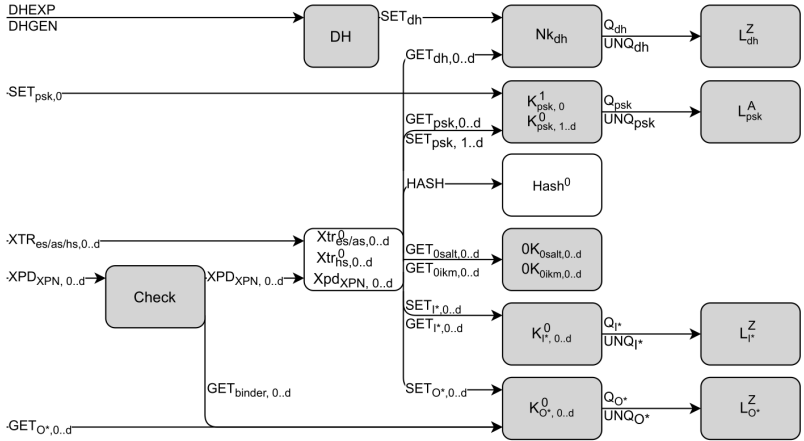

Figure: Image from "Key-schedule Security for the TLS 1.3 Standard"

What do we want to prove - ideal protocol

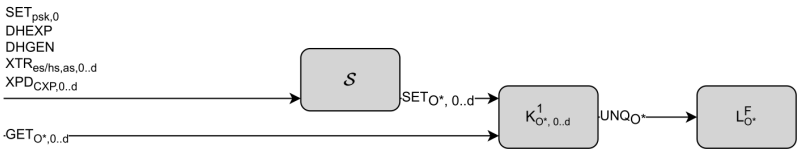

Figure: Image from "Key-schedule Security for the TLS 1.3 Standard"

What are State Separating Proofs (SSP)

Originating from

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

Using SSP for proofs

- helps scale development and keep modularity
- keeps formulation clear and consistent

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 the composition of packages into games (semi-automatic)
- prove indistinguishability of each game
- compose the games to show the advantage of an adversary is bounded

Rocq and Interactive Theorem Proving (ITP)

- Interactive theorem proving is an extension of automated theorem proving that allows one to check any mathematical proof.
- popular for critical software (cryptography),
- widely used in the programming language community,
- and gaining traction with mathematicians.

SSProve

- a framework for modular cryptographic proofs (e.g. SSP) in the Rocq proof assistant
- foundational, e.g. fully formalized and specified
- an imperative language with state and probability
- game hopping style proofs in the computational model

Differences from paper proof

Indexing

- The key schedule is parameterized by a resumption bound (d), the game runs in rounds given by an index ℓ

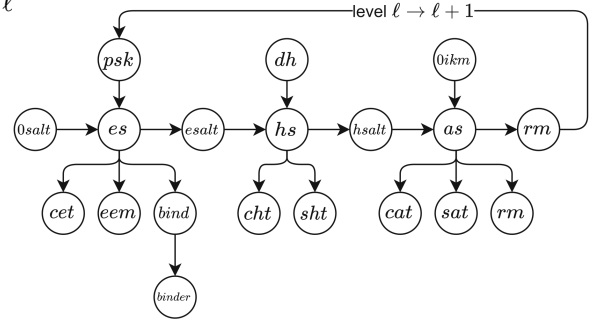

Figure: Image from "Key-schedule Security for the TLS 1.3 Standard"

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

Differences from paper proof

Wire Indexing

When constructing the protocol, we assign "wires". Similarly to channels in the UC model. These are indexed by the bound, the round, and the level

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

Differences from paper proof

Wire Indexing

When constructing the protocol, we assign "wires". Similarly to channels in the UC model. These are indexed by the bound, the round, and the level

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

Given two wires, they do not 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

Composition order

We define all packages based on the horizontal and parallel constructions

- can give a simpler proof due to a different composition order

In the verification, we need to be precise about variable renaming and memory separation

- e.g. requiring disjointness and freshness of memory
- can be simplified by vizualization and automation tools
  - CryptoZoo
  - SSBee (`https://github.com/sspverif/sspverif/`)
  - ProofFrog (`https://prooffrog.github.io/`)

Cryptographic Assumptions

Our cryptographic assumptions

- 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
- the implementation of the ML-KEM is (post-quantum) secure
  - we use an implementation of a hybrid KEM, which has been shown to be panic free
  - ML-KEM has been shown to be secure against store-now-decrypt-later in a quantum consistent model in EasyCrypt.

Application of Proof

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

- we instantiate the proof with our TLS 1.3 implementation

Using the Hax framework, we

- translate the implementation to SSProve

For SSProve, we

- show equivalence between the translated code and protocol spec (another game)
- application of game hopping for software verification

This gives us a security bound for the actual implementation

Hax

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

Why Rust?

- memory safe
- ML-like type system
- as fast as C, industry grade
- popular among cryptographic engineers

TLS Implementation

Using Key schedule implementation for handshake

To implement the handshake protocol of TLS 1.3 we

- define deriviation graph
- call XTR and XPD to step the graph
  - instantiated by HMAC-based Extract-and-Expand Key Derivation Function (HKDF)
- 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
- 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 protocol specification
- modularize to enable SSP style proofs
  - from full protocol specification 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

- Hax facilitates developing high-assurance cryptographic software
- Confluence of ideas from formal verification and cryptography
- SSP style of proofs invites modular, reusable and scalable implementations
- libcrux allows instantiation of primitives with a secure and efficient implementation
- Hax framework enables a multi-tool verification effort with a common reference implementation
- Increasing interest in collaborative efforts e.g. the "Crypto Proof ladder"