About

• PROSECCO: Programming Securely with Cryptography.
• Team at INRIA Paris specializing in applied cryptography and formal verification.
• Also a consulting office: Symbolic Software.
• Goals:
  • Formally delineate the patterns in which cryptographic flaws occur across all the world’s important protocols.
  • Develop technologies to minimize these flaws occurring again in the future, based on what we’ve learned.
Goals of this Talk

• Motivate you with specific examples on what formal verification can bring to the web.

• Not a deep-dive into formal verification theory: the goal is to get you excited!

• Talk is based around my Ph.D. work for the past three years.
Technologies

- **Major projects:**
  - **ProVerif**: Automated protocol verification in the symbolic model.
    - Network execution under a Dolev-Yao attacker.
    - ProScript, TLS, Signal, ACME, Capsule, LDL...
  - **CryptoVerif**: Guided protocol verification with proofs in the computational model.
    - TLS, Signal, WireGuard...
  - **F***: ML programming language that lends itself to formal verification.
    - Dependent types, refinements, etc.
    - HACL* verified cryptography library, miTLS verified TLS implementation.
Cryptographic Web Applications

• Radical propulsion in market share:
  • Cryptocat: end-to-end encrypted chat with OTR (2011)
  • WhatsApp Web: end-to-end encrypted view into mobile device (2016)
  • Signal Desktop: Electron App (2017)
  • Skype: Electron App (2018)
Cryptocat: from buggy mess to formally verified secure messaging

• Popular chat software, rewritten in 2016. 30,000+ weekly users.

• Implements Signal protocol in ProScript, a purely functional subset of JavaScript we can automatically type check, translate and verify in ProVerif.
How does ProVerif Work?

1. Illustrate primitives and functions as perfect black boxes (symbolic model)
2. Emulate all protocol runs across a network with an active attacker.
3. Answer specific queries.
Signal Protocol: Overview

• Most widely used secure messaging protocol (WhatsApp, Wire, Allo, Messenger, Cryptocat, Signal...)

• Four-way Diffie-Hellman in AKE step.

• Complex key schedule for ratcheting between messages.

• Offers offline messaging (due to zero-round-trip AKE.)
Signal Protocol: Some Small Attacks

- Binding keys to wrong identity:
  - B may have used C’s ephemerals (unknown key share attack.)
- Attacker can exhaust B’s one-time keys.
- First message can be replayed:
  - No replay cache.
- Key compromise impersonation:
  - If both a and s are compromised.
- We found all attacks in ProVerif (2017).
Linking JavaScript Implementations to Verification Frameworks

• ProScript: evolution from Defensive JavaScript (Antoine Delignat-Lavaud, 2014) into a full language: subset of JavaScript -> ProVerif.

• Builds real apps that work!

• Working on producing groundwork for CryptoVerif models as well.
const RATCHET = {
    deriveSendKeys: function(them, myEphemeralKeyPriv) {
        const kShared = ProScript.crypto.DH25519(
            myEphemeralKeyPriv, them.ephemeralKey
        )
        const sendKeys = UTIL.HKDF(
            kShared, them.recvKeys[0], 'WhisperRatchet'
        )
        const kKeys = UTIL.HKDF(
            ProScript.crypto.HMACSHA256(sendKeys[1], '1'),
            Type_key.construct(),
            'WhisperMessageKeys'
        )
        return {
            sendKeys: sendKeys,
            kENC: kKeys[0]
        }
    },
}

let fun_deriveSendKeys(them:object_them, myEphemeralKeyPriv:key) =
    let kShared = ProScript.crypto.DH25519(
        myEphemeralKeyPriv, Object_them.get_ephemeralKey(them)
    )
    let sendKeys = fun_HKDF(
        kShared, Array_1.get_e_0(Object_them.get_recvKeys(them)),
        string_101)
    let kKeys = fun_HKDF(
        ProScript.crypto.HMACSHA256(Array_1.get_e_1(sendKeys),
        string_105),
        Type_key_construct(),
        string_108
    )
    in
    Object_112(
        Array_1.get_e_0(kKeys),
        sendKeys
    ).
Verification in ProVerif

- Define a top-level process.
- Define queries.
- Execute over a network with an active attacker.

**Protocol bugs**: Key Compromise Impersonation. If Bob’s long-term secret and Bob’s signed pre-key is compromised, attacker can impersonate Alice to Bob.

**Implementation bugs**: missing HMAC check.
Verification in ProVerif

• We verify:
  • Confidentiality.
  • Authenticity.
  • Forward secrecy.
  • Future secrecy.
  • Indistinguishability.
  • Absence of replay attacks.

```plaintext
free secMsg1:bitstring [private].
free secMsg2:bitstring [private].
free secMsg3:bitstring [private].
query attacker(secMsg1).
event Send(key, key, bitstring).
event Recv(key, key, bitstring).
query a:key,b:key,m:bitstring; event(Recv(a, b, m)) ==> event(Send(a, b, m)).
query a:key,b:key,m:bitstring; event(Recv(a, b, m)).
query a:key,b:key,m:bitstring; event(Send(a, b, m)).

let Initiator(
  initiatorIdentityKey:object_keypair,
  initiatorSignedPreKey:object_keypair,
  initiatorPreKey:object_keypair,
  responderIdentityKeyPub:key,
  responderIdentityDHKeyPub:key
) =
  out(io, {
    Object_keypair_get_pub(initiatorSignedPreKey),
    ProScript_crypto_ED25519_signature(
      Type_key_toBitstring(Object_keypair_get_pub(initiatorSignedPreKey)),
      Object_keypair_get_priv(initiatorIdentityKey),
      Object_keypair_get_pub(initiatorIdentityKey)
    ),
    Object_keypair_get_pub(initiatorPreKey)
  });
in(io, {
  responderSignedPreKeyPub:key,
  responderSignedPreKeySignature:bitstring,
```
Verification in CryptoVerif

• While ProVerif is a **symbolic model verifier** which plays protocols against an active attacker model...

• CryptoVerif is a **computational model prover**, that can produce full game-based proofs in a probabilistic Turing machine model.

• **New**: CryptoVerif syntax unified with ProVerif, allows sharing model groundwork more quickly.
ProScript, ProVerif and CryptoVerif: Also Applied to TLS 1.3

• The same approach was used for TLS 1.3 (draft-18) (Oakland S&P ‘17 Distinguished Paper Award):
  • RefTLS: An implementation of TLS 1.3 draft18 integrated into the Brave web browser.
  • ProVerif models: hand-written and compiled from RefTLS core.
  • CryptoVerif models: similar approach to Signal paper.
ACME v1 was formally modeled in ProVerif with an attacker model inspired by trends in domain name validation in certbot as well as traditional domain validation mechanisms.

Attacks found on certificate issuance, which were reported and fixed in ACMEv2. (FC’17)
NoiseShaper: Pushing ProVerif to the max!

- NoiseShaper (work in progress) takes any Noise handshake pattern and creates a mechanized specification of it in ProVerif.
- Automatically infers a top-level process and security queries based on the pattern (this is really cool!)
- Can be upgraded to support CryptoVerif!
And now, towards the future!

• Let’s take a look at Cryptocat again.

• In 2016/2017, we did:
  • ProScript protocol core (Signal).
  • Translates and verifies in ProVerif.
  • Manually proven in CryptoVerif.
  • Trusted cryptographic core.

• In 2018+, can we do better?
  • The structure is there, but can we improve upon the individual components?
  • PROSECCO is working on tons of next-gen tech: F*, HACL*, HACSPEC, and more CryptoVerif!
Capsule: A Protocol for Secure Collaborative Document Editing

• Document is a hash chain of diffs.
• Access to document is obtained by sharing a simple ID.
• Users must prove knowledge of ID to participate in document.
• Primitives:
  • BLAKE2 for symmetric operations.
  • Ed25519 for signatures.
Capsule: Security Goals

- Participant List Integrity.
- Confidentiality.
- Integrity.
- Authentication.
- Transcript Consistency.
Capsule: Formally Verified

Protocol Level:
- Formally verified with ProVerif.
- Hand proof.

Implementation Level:
- HACL-WASM: First software to use HACL* in WebAssembly.
- Functional correctness, secure memory, no side channels.
ProVerif}

Trace: Capsule
Implementation, client, etc. is almost there.

Learn more:
https://symbolic.software/capsule/

ePrint paper (proceedings paper later this year):
https://eprint.iacr.org/2018/253
What is F*?

• F* is a general-purpose functional programming language aimed at formal verification.

• It combines SMT-based verification (Z3) with an ML syntax to build amazing things:
  • **Project Everest**: full verified HTTPS stack in F* (Microsoft Research, INRIA.)
  • **HACL***: Fastest cryptographic library for most primitives, with side-channel resistance, functional correctness and memory safety.
  • **Much more**.

• Using the KreMLin compiler, it can be compiled to:
  • OCaml (default,)
  • Verified C code (from Low* subset, passes CompCert)
  • WebAssembly (work in progress.)
Data structures in TLS are implemented in RFCs using an informal “RFC language” of structs and enums, resulting in inconsistent, buggy implementations.

QuackDucky parses that language and generates full parsers/serializers in verified F*!

A single component to eliminate an entire class of bugs.
Project Everest: the QuackyDucky F* Parser Generator

• Data structures in TLS are implemented in RFCs using an informal “RFC language” of structs and enums, resulting in inconsistent, buggy implementations.

• QuackyDucky parses that language and generates full parsers/serializers in verified F*!

• A single component to eliminate an entire class of bugs.
**HACL-WASM: F* Primitives in WebAssembly**

1. **HACL**: a cryptographic library written in F*.
2. **Low***: a subset of F* we can compile to C.
3. **Kremlin**: a Low* to C compiler.
4. **Kremlin**: now also a Low* to WASM compiler.
5. **HACL-WASM!**
   - Native 64-bit operations: useful for Ed25519, Blake2b, etc.
   - Maintain constant-time and functional correctness properties.
- HACL-WASM gives us perhaps the most high-assurance cryptographic primitives for the web.
- Can we pair this with a protocol implementation from F*?
- Integration: not only Capsule, but Signal? Skype???
Related Work: HACL* in Mozilla Firefox 57+

- Mozilla’s TLS implementation now uses formally verified primitives!
- Compiled from F* to C using KreMLin (passes CompCert):
  - Curve25519 (first to be added)
  - Then Chacha20/Poly1305, SHA2...
- HACL* library by Zinzindohoue et al.
- Implemented in Firefox by B. Beurdouche (PROSECCO) and Mozilla team.
Ledger Design Language

- Public Ledger (and blockchain) designs are extremely legitimate and the space right now is very credible.
Ledger Design Language

- Public Ledger (and blockchain) designs are extremely legitimate and the space right now is very credible.
- Nobody designing public ledgers has any idea what they’re doing.
Ledger Design Language

- LDL is a modeling language for describing public ledgers.
- LDL compiler produces two outputs:
  - **ProVerif model**: verify the ledger design as a protocol executed over a network with an active attacker.
  - **F* client/server API**: prototype your ledger immediately with a verified secure implementation!
Ledger Design Language

• Using a simple syntax, we describe block types and properties, and use actor declarations to specify who does what.

• Result is a simple language that lets you reason about your ledger design as well as prototype it in the real world without risk.

• Upcoming publication at IEEE EuroS&P SPIDA 2018 (end of this month!)
Conclusion: So Many Ways to Bring Formal Verification to the Web

• Formally Verified Secure Messaging and TLS 1.3 Prototyping and Let’s Encrypt Improvements with ProScript, ProVerif and CryptoVerif, bringing automated protocol verification to web applications.

• Automated Mechanized Specifications and Attacker/Query Generation for an Entire Protocol Framework, via NoiseShaper which turns any Noise handshake pattern into smart answers to good questions.

• Secure Collaborative Document Editing with Formally Verified Protocol and Components, using ProVerif for protocol modeling and HACL-WASM for the most secure way to implement cryptographic primitives for the web yet.

• Formally Verified Parser Generation Directly from Spec, using the QuackyDucky to generate parsers in F* directly from TLS 1.2 and 1.3 specs.

• Secure Cryptographic Primitives Across Various Platforms, by designing HACL* in F* and compiling it to target C and WASM while preserving speed and advanced security guarantees!

• Designing New Domain-Specific Languages to Bring Much-Needed Rigor to “Wild West” Areas, with Ledger Design Language for modeling public ledger designs into ProVerif models and F* client/server APIs.
Conclusion: So Many Ways to Bring Formal Verification to the Web

- Personal website: https://nadim.computer
- Symbolic Software: https://symbolic.software