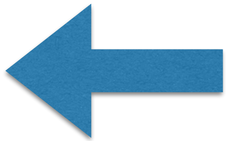
Automated Verification for Secure Messaging Protocols and their Implementations: A Symbolic and Computational Approach

Nadim Kobeissi — Karthikeyan Bhargavan — Bruno Blanchet
Presented by Nadim Kobeissi

2nd IEEE European Symposium on Security and Privacy
April 28 2017, Paris, France

About

- We are interested in the formal verification of web protocols.
- Protocols of current and previous interest:
  - Transport layer (TLS, QUIC). *(Oakland S&P 2017)*
  - Secure messaging (Signal Protocol, OTR, Telegram). 
  - Domain validation (ACME). (*Financial Cryptography 2017)*

Impactful Vulnerabilities in Today's World

- Our TLS findings in 2015 show that these are attacks that matter:
  - **Protocol logic flaws**: Tricking the protocol state machine into assuming a state too early (SMACK, Early CCS).
  - **Cryptographic design flaws**: Padding oracle attacks (POODLE), truncated hash attacks (SLOTH), factoring weak keys (FREAK).
  - **Implementation flaws**.

Lessons Learned so Far

- “Code first, specify later.”
- Testing cryptographic protocol implementations has been overlooked, with disastrous results.
- We need an approach towards verifying production code and more tools for today’s real-world cryptography developer.

Secure Messaging Today

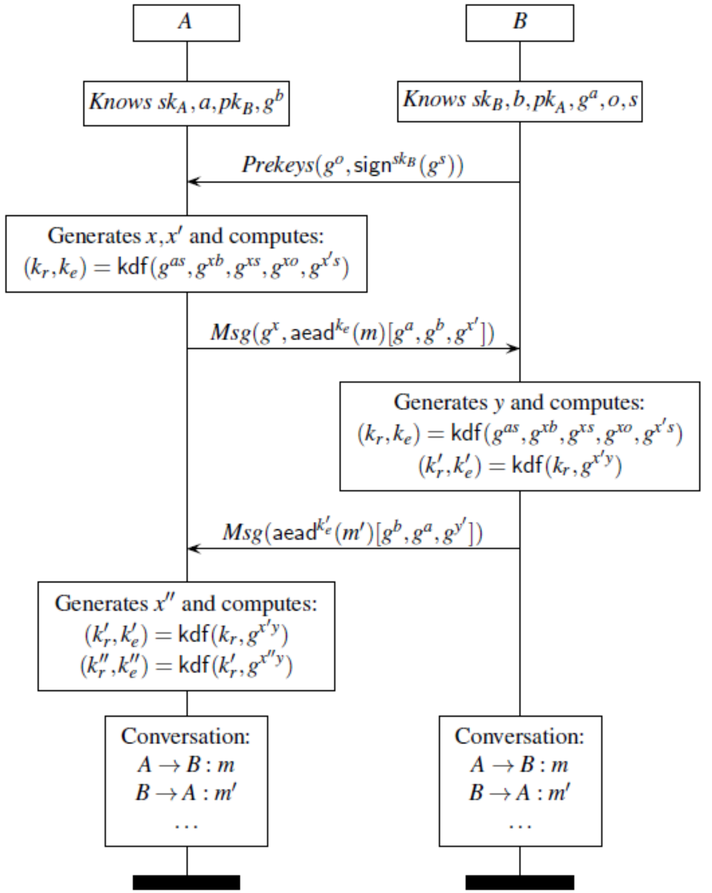
- WhatsApp, Telegram, Wire, Cryptocat…
  - Long-term conversations between “buddies” using multiple devices.
  - Secrecy, integrity, authenticity.
- Special context:
  - Asynchronous messaging (0-RTT AKE with added data).
  - Forward secrecy, future secrecy.

Signal Protocol

Telegram Protocol (“MTProto”)

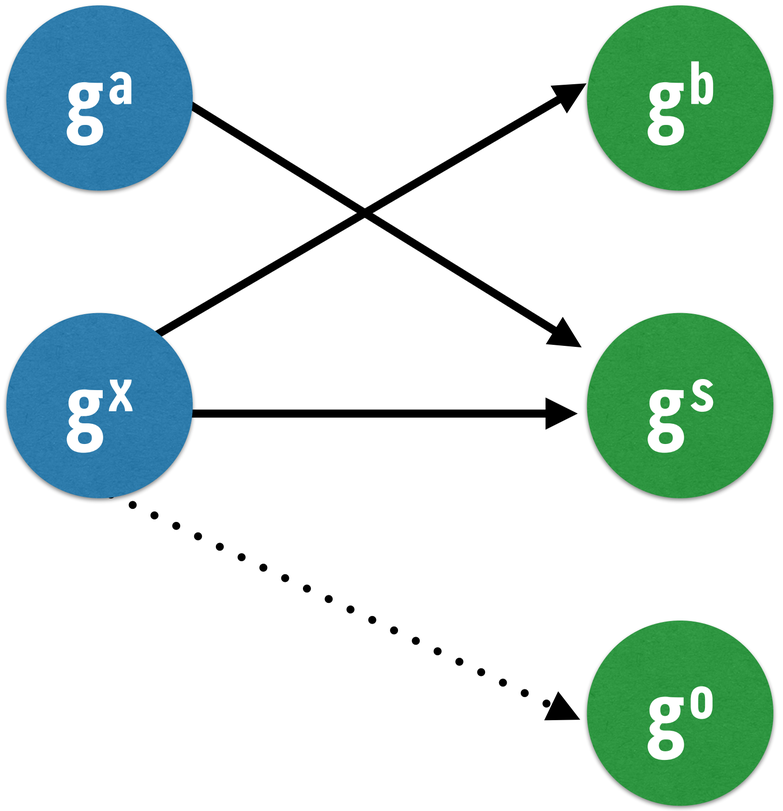
Signal Protocol: Overview

- Four-way Diffie-Hellman in AKE step.
- Offers offline messaging (due to zero-round-trip AKE.)
- Complex key schedule for ratcheting between messages.



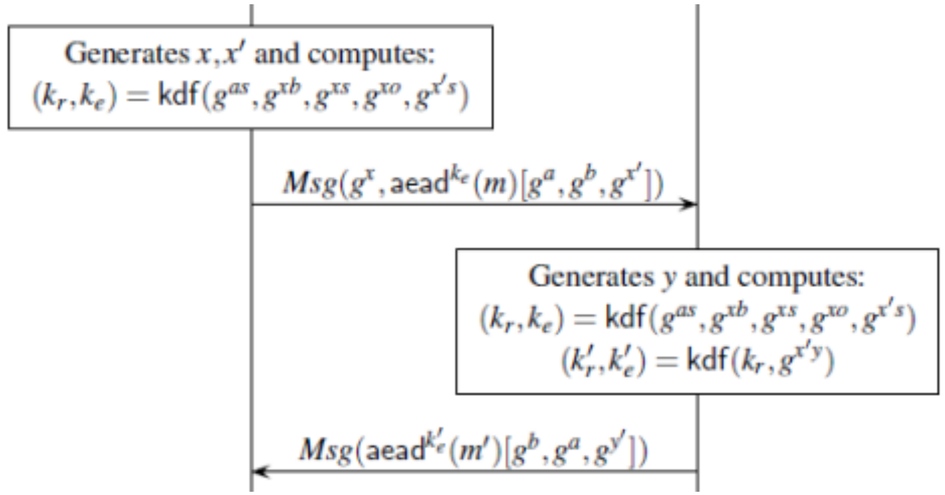
Signal 0-RTT: Asynchronous Messaging

- **4-way Diffie-Hellman:**
  - One ephemeral (g^s) is signed.
  - Usable even when g^0 is exhausted.



Signal Ratchet: Key Refresh

- Every message has a new ephemeral value g^x'.
- New keys derived from old keys + g^{yx}' (new x, old y).
- Key separation via multiple HKDFs.



ProScript: A Language for Protocol Implementation

- Aims to be the ideal language for reliably implementing and rapidly prototyping cryptographic protocols inside web applications.
- JavaScript is tricky but important: a restricted subset of JavaScript based on Defensive JavaScript (Delignat-Lavaud et. al).
- Automatic translation from real-world implementation code to symbolic models!

ProScript: A Language for Protocol Implementation

- Essentially, the subset manifests as a style:
  - **Typed**: ProScript's type checker infers the initial type of variables, constants, objects, functions and allows custom type definitions (eg. 32 byte array becomes a "key" type.)
  - **Immutable structures**: Extensible hash tables, objects and arrays are disallowed, and scoping is enforced.
  - **Purely functional**: We take advantage of JavaScript's naturally functional paradigms in order to encourage a programming style resembling the applied pi calculus.

The ProScript We Want

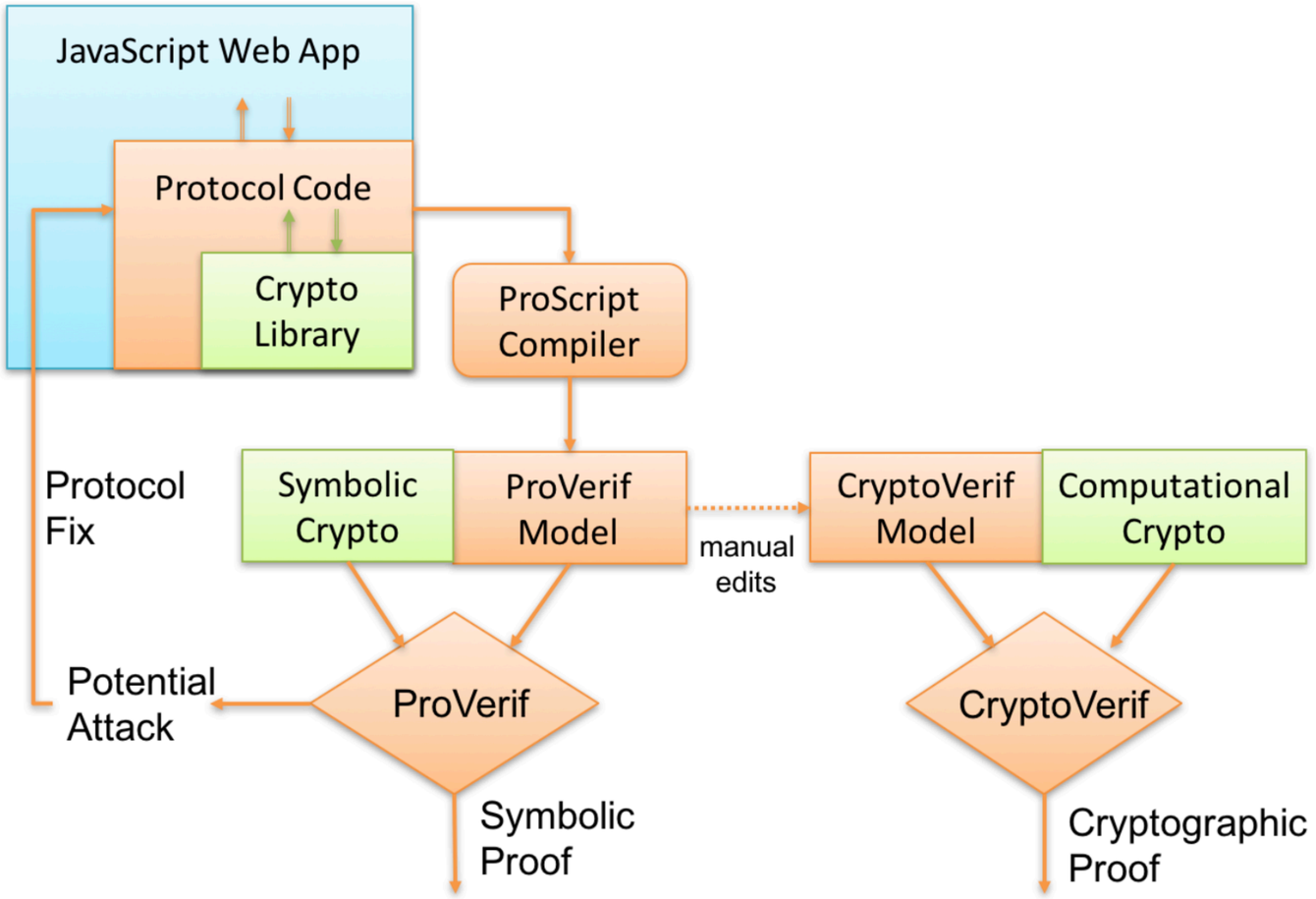
- A complete framework, not just an isolated subset, for the implementation and verification of cryptographic protocols, both:
  - **Automatic verification**: Efficient compilation into ProVerif (and in the future CryptoVerif) models that complete, that correctly model adversaries, and that accurately and intuitively reflect original code.
  - **Human verification**: Models that are human-readable to allow developers to extend them manually and to understand where and why a security guarantee is not achieved.

PS2PV: ProScript to ProVerif

```
const sendMessage = function(msg, state) {
    const newPrivKey = Crypto.randomBytes(32)
    const newPubKey  = Crypto.DH25519(
        newPrivKey, [
            0x00, 0x00, 0x00, 0x00,
            0x00, 0x00, 0x00, 0x00,
            0x00, 0x00, 0x00, 0x00,
            0x00, 0x00, 0x00, 0x00,
            0x00, 0x00, 0x00, 0x00,
            0x00, 0x00, 0x00, 0x00,
            0x00, 0x00, 0x00, 0x00,
            0x00, 0x00, 0x00, 0x09,
        ]
    )
    const shared = Crypto.DH25519(
        newPrivKey, state.theirPubKey
    )
    return {
        data: {
            msg: Crypto.AESGCMEncrypt(shared, msg),
            pubKey: newPubKey
        },
        state: {
            myPrivKey: newPrivKey,
            myPubKey: newPubKey,
            theirPubKey: state.theirPubKey
        }
    }
```

```
letfun sendMessage(msg:bitstring, state:object_1) =
    new newPrivKey:key;
    let newPubKey = Crypto_DH25519(newPrivKey, key_9) in
    let shared = Crypto_DH25519(
        newPrivKey,
        Object_1_get_theirPubKey(state)
    ) in
    Object_8(
        Object_10(
            Crypto_AESGCMEncrypt(
                shared, msg
            ),
            newPubKey
        ),
        Object_1(
            newPrivKey,
            newPubKey,
            Object_1_get_theirPubKey(state)
        )
    ).
```

ProScript Verification Approach: Rapid Prototyping for Implementers



Signal Protocol: Verification Results

- We have implemented Signal Protocol (version 3) in ProVerif, where we are able to symbolically verify under an active Dolev-Yao attacker, with up to three messages between Alice, Bob and a compromised Mallory:
  - Secrecy, Authenticity,
  - Indistinguishability,
  - Forward/Future Secrecy,
  - Key Compromise Impersonation.

Signal Protocol: Verification Results

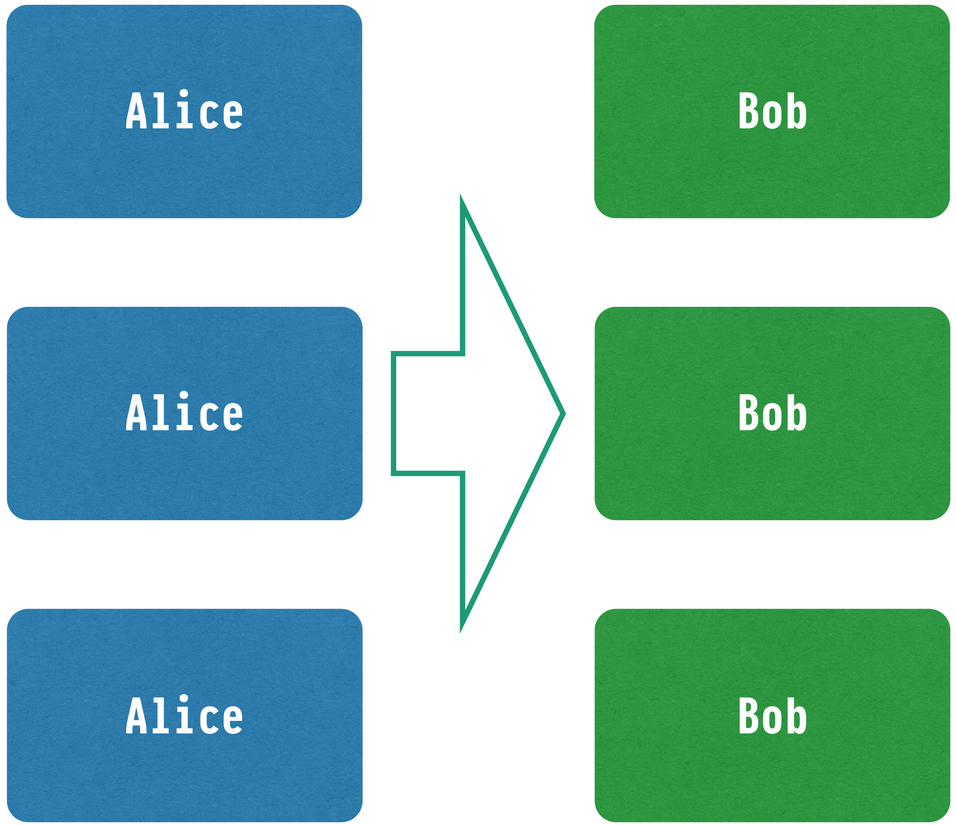
- **Secrecy**: The adversary cannot compute messages sent between Alice and Bob.
- **Indistinguishability**: The adversary cannot distinguish which of two messages is being sent.
- **Authenticity**: We define a query that creates a correspondence between send and receive events:
  - `event(Recv(a, b, m)) ==> event(Send(a, b, m)).`

Signal Protocol: Verification Results

- **Forward secrecy**: when are previous messages leaked?
  - In a single-message model: `out(io, responderIdentityKey)`
  - In a multiple message model, we can leak the entire state, or individual keys (latest ephemeral shared secret, etc.) (also how we test for **Future secrecy, which refers to future messages**.)

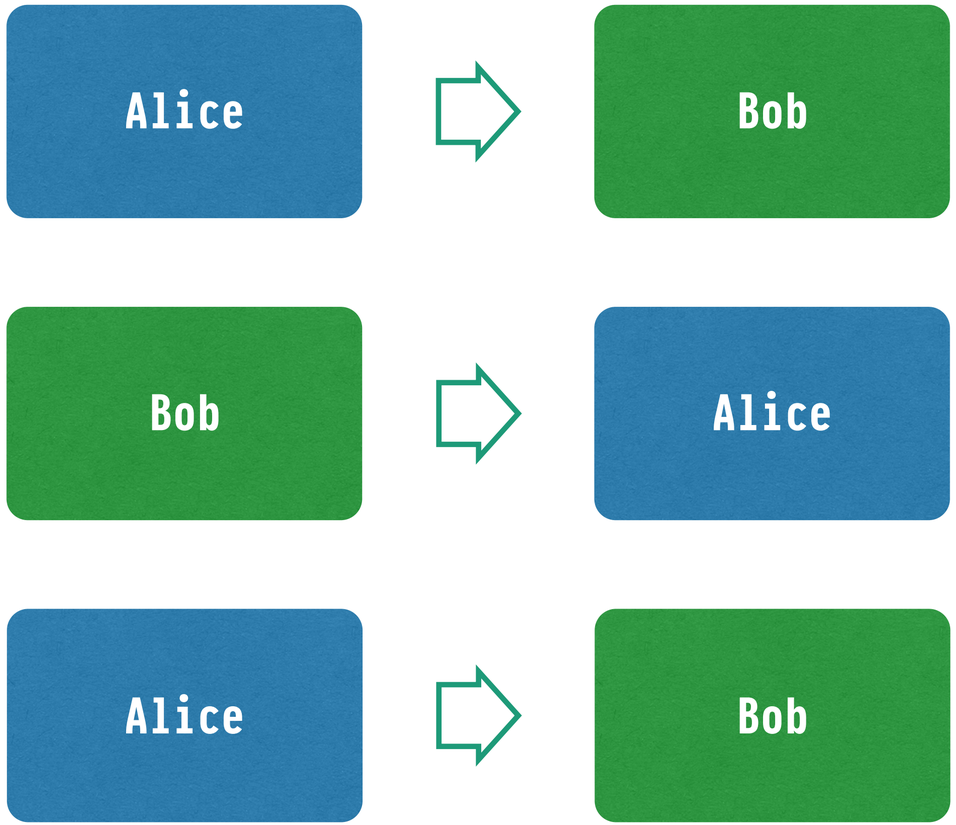
Signal Protocol: Verification Results

- We were able to distinguish refined scenarios:
  - Single-Flight Pattern:
    1. Alice cannot obtain fresh key share so just hashes forward.
    2. For **forward secrecy**, Alice must delete prior keys: can deletion be guaranteed?
    3. Message re-ordering? Memory volatility?
    4. **No future secrecy.**



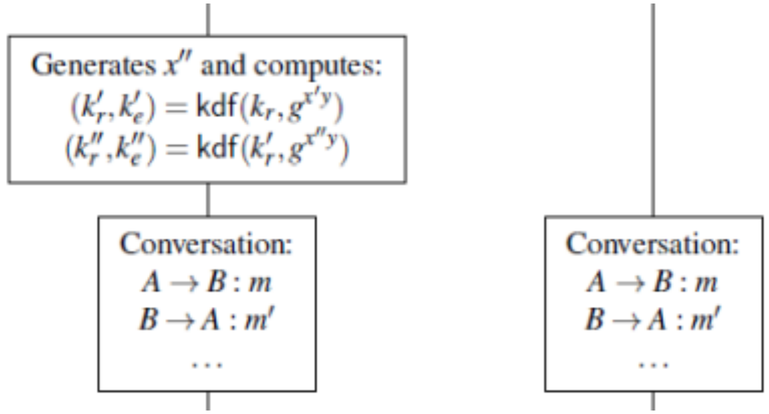
Signal Protocol: Verification Results

- We were able to distinguish refined scenarios:
  - Message-Response Pattern:
    1. Alice obtains fresh key material.
    2. **Forward secrecy** is maintained less dependently of physical factors (key deletion, network reliability).
    3. **Future secrecy.**



Signal Protocol: Discovered Attacks

- Replay attack on the first message in case of a one-time pre-key not being used (it is optional.)
  - Replay attacks are **not present** if the one-time pre-key is used.
- Key compromise impersonation: if Bob's signed pre-key is compromised, an attacker can force him to accept fake messages from Alice.



Signal Protocol: Verification Speeds

- Reasonably fast given the complexity of the protocol.
- All models were automatically generated from actual untouched Cryptocat implementation code using PS2PV.

- `authenticity-1-abm-oneway` : 00h. 24m. 51s.
- `authenticity-1-abm-twoway` : 26h. 10m. 08s.
- `authenticity-1-ab-oneway` : 00h. 04m. 07s.
- `authenticity-1-ab-twoway` : 00h. 09m. 25s.
- `forwardsecrecy-1-ab-oneway` : 00h. 06m. 14s.
- `forwardsecrecy-2-ab-oneway` : 00h. 14m. 10s.
- `forwardsecrecy-3-ab-oneway` : 00h. 46m. 14s.
- `futuresecrecy-3-ab-oneway.pv` : 00h. 44m. 25s.
- `indistinguishability-1-ab-oneway` : 01h. 51m. 17s.
- `kci-1-a-oneway` : 00h. 17m. 35s.
- `kci-1-b-oneway` : 00h. 05m. 59s.
- `secrecy-1-ab-oneway` : 00h. 03m. 30s.
- `secrecy-1-ab-twoway` : 00h. 07m. 06s.

How our Approach is Different

- Generating symbolic models from ProScript takes into account **protocol details whose relevance only appears at implementation**:
  - Need to keep different hash chains in Signal Protocol.
  - Trial decryption to deal with unreliable networks.
  - Verification of symbolic model says something about your **real-world implementation!**

Signal Protocol: Computational Proofs

- We begin from certain assumptions:
  - GDH Assumption on EC25519.
  - ED25519 signatures are unforgeable under CMA.
  - Different HKDF constructions constitute independent random oracles.
  - HMAC-SHA256 is a PRF.
  - AES-GCM is a secure AEAD secure against IND-CPA and INT-CTXT.

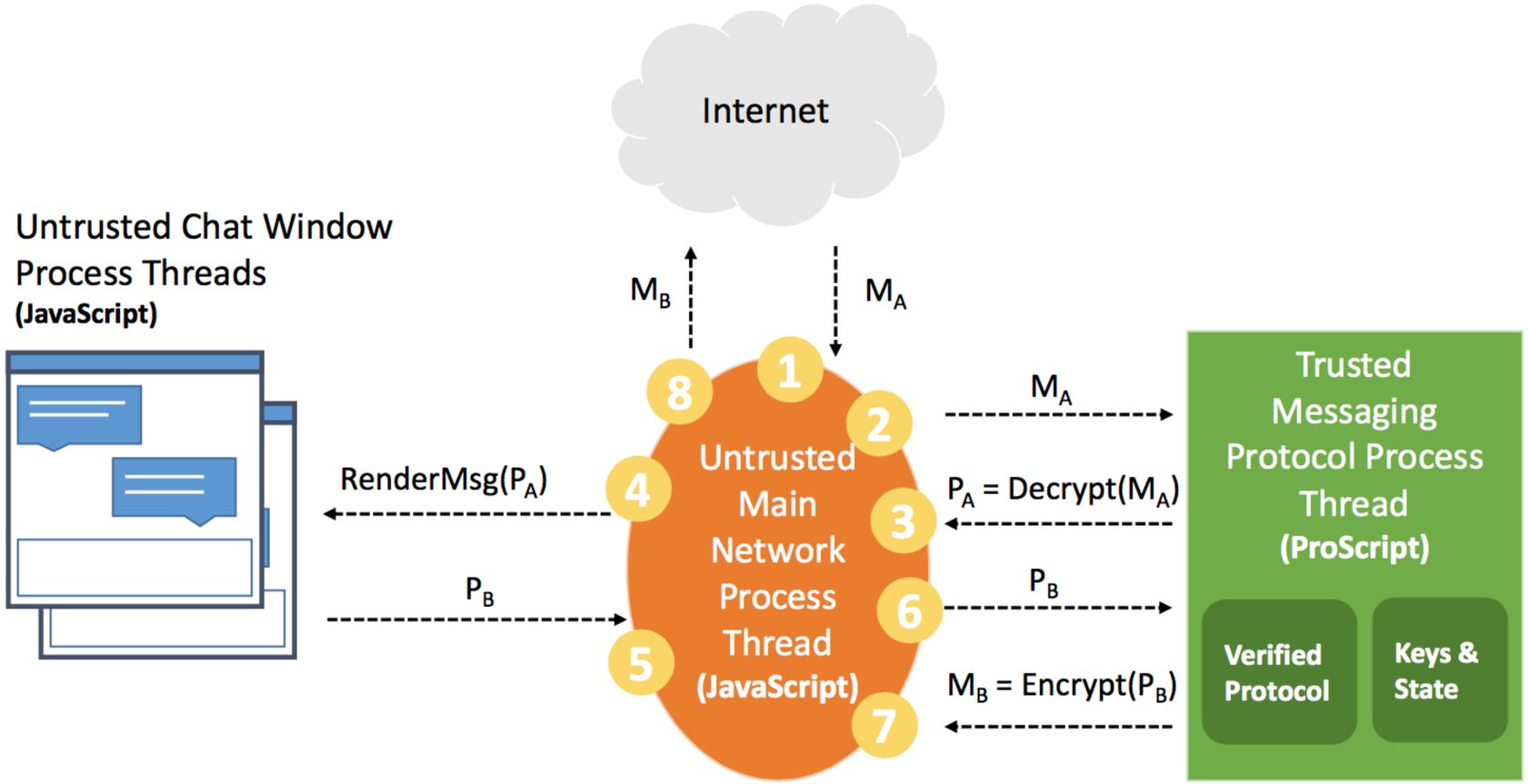
Signal Protocol: Computational Proofs

- From our assumptions, we produce proofs in the computational model using **CryptoVerif** (with one message). We prove:
  - Authenticity.
  - Absence of KCI (when long-term keys are compromised).
  - Indistinguishability.
  - Forward secrecy.
  - Replay Attack: We find the same attack from the symbolic analysis.

Cryptocat: Software and Implementation

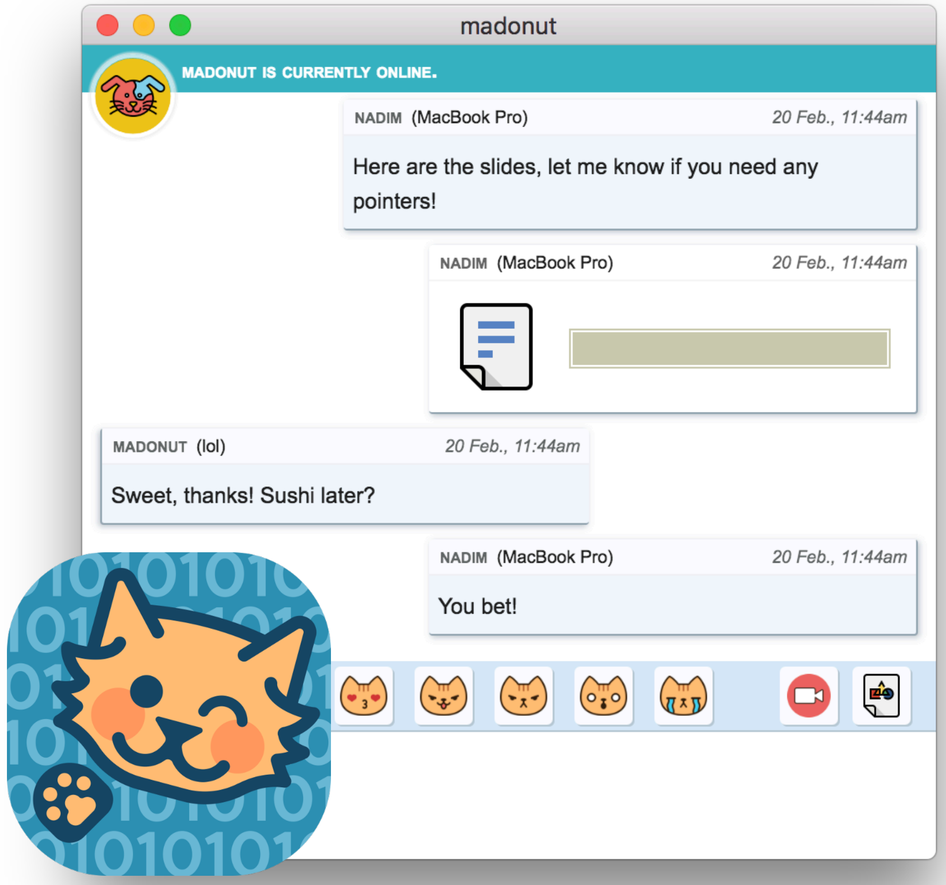
- Popular chat software, recently rewritten. 30,000+ weekly users.
- Implements Signal protocol in ProScript, a purely functional subset of JavaScript we can automatically type check, translate and verify in ProVerif.
- Why the rewrite?
  - AES-CTR nonce re-use.
  - Incorrect typing leading to weak Curve25519 private keys.
  - And more!

Cryptocat Architecture: Language-based + thread-based isolation



Cryptocat: Software and Implementation

- Works great!
- No performance or feature set hit due to verification.
- Send 200MB files, video messages, etc.



Caveats

- One consequence of using mechanized tools is that we are limited by their heuristics and expressiveness.
  - Limited to 2 or 3 messages depending on model.
  - In KCI, forward/future secrecy and indistinguishability models, Alice only plays the initiator role and Bob only the responder role, and they do not run the protocol with malicious principals.
  - Not to worry! The next presentation in this session, "*A Formal Security Analysis of the Signal Messaging Protocol*", covers what we miss (but we covered stuff they miss too!)
- PS2PV compiler buggy and experimental.
- We target only protocol-level analysis. JavaScript runtimes might be broken.
- Verification ≠ perfection.

Future Work

- Use compositional theorems to prove larger parts of the protocol. We do this for TLS 1.3 in our upcoming Oakland S&P 2017 paper, *Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate*.
  - Also uses ProVerif, ProScript, CryptoVerif.
- Develop verified translations to both ProVerif **and also CryptoVerif**.

Thank You

- With disciplined programming and some verification expertise, the systematic analysis of complex cryptographic web applications is now becoming practical.

- Paper, models, code and more: https://github.com/Inria-Prosecco/proscript-messaging/

- Cryptocat: https://crypto.cat