Some Simple Protocols:
Attacks and Proofs
Designing an RPC Protocol

**Goal:** protect the secrecy and authenticity of \( req \) and \( resp \)
Symmetric-Key RPC

Is this secure?

Initiator $I$

Initially Knows:
$k_{ir}$

Exchange:
$I \leftrightarrow R : req; resp$

Responder $R$

Initially Knows:
$k_{ir}$

Exchange:
$I \leftrightarrow R : req; resp$

$\text{aead}(k_{ir}, req)$

$\text{aead}(k_{ir}, resp)$
Reflection Attack on Symmetric-Key RPC

- Initiator $I$
  - Initially Knows: $k_{ir}$
  - Exchange: $I \leftrightarrow R : req; req$

- Attacker $O$
  - $\text{aead}(k_{ir}, \text{req})$
  - $\text{aead}(k_{ir}, \text{req})$

- Responder $R$
  - Initially Knows: $k_{ir}$

Attacker reflects request back to $I$
Fixing Symmetric-Key RPC

**Initiator** $I$

Initially Knows:

$k_{ir}$

Exchange:

$I \leftrightarrow R : req; resp$

**Responder** $R$

Initially Knows:

$k_{ir}$

Exchange:

$I \leftrightarrow R : req; resp$

Fix: Add message tags (or use different keys)

\[
\text{aead}(k_{ir}, 0\| req)
\]

\[
\text{aead}(k_{ir}, 1\| resp)
\]
Public-Key Encrypt-then-Sign RPC

Is this secure?
Re-signing Attack on Encrypt-then-Sign

Initiator $I$
- Initially Knows: $sk_i, pk_r$
- $\text{sign}(sk_i, \text{penc}(pk_r, 0||req))$

Attacker $O$
- Initially Knows: $pk_i, pk_r$
- $\text{sign}(sk_o, \text{penc}(pk_r, 0||req))$
- $\text{sign}(sk_r, \text{penc}(pk_o, 1||resp))$

Responder $R$
- Initially Knows: $sk_r, pk_i$

Knows Secret: $resp$

**Attacker** replaces I’s signature with his own, to obtain the secret $resp$
Fixing Encrypt-then-Sign

Fix: Add signer’s identity to inner encryption 
(secure composition)
Using Sign-then-Encrypt

Is this secure?
Identity Misbinding Attack on Sign-then-Encrypt

**Attacker** acting as a valid responder for $I$, re-encrypts request to $R$, causing an *identity mis-binding attack*
Response Correlation Attack on RPC

Attacker reorders responses. Works on all our RPC protocols.
Fixing Sign-then-Encrypt

**Initiator** $I$

Initially Knows: $sk_i, pk_r$

Exchange: $I \leftrightarrow R : req; resp$

**Responder** $R$

Initially Knows: $sk_r, pk_i$

Exchange: $I \leftrightarrow R : req; resp$

$$penc(pk_r, sign(sk_i, 0||pk_r||req))$$

$$penc(pk_i, sign(sk_r, 1||pk_i||req||resp))$$

**Fix:** add recipient identity and request to inner sig (secure composition)
Authenticated Key Exchange: STS

\[ g^x \]

\[ g^y, B, \text{SIG}_B(g^x, g^y) \]

\[ E, \text{SIG}_E(g^y, g^x) \]

Session Key = KDF(g^{xy})
Authenticated Key Exchange: ISO

Session Key = KDF($g^{xy}$)
Authenticated Key Exchange: ENC-STS

\[ g^x \]

\[ g^y, B, \{ \text{SIG}_B(g^x, g^y) \}_{K_s} \]

\[ A, \{ \text{SIG}_A(g^y, g^x) \}_{K_s} \]

Session Key \( K_s = \text{KDF}(g^{xy}) \)
Secure Messaging Today

**Common Goal:** End-to-End Security

**Relatively New Cryptographic Protocols**
- Signal, iMessage, Telegram MTProto, ...
- Use modern cryptographic constructions
- Subtly different security guarantees
- Implemented on desktops and phones

**Are they safe to use?**
- Often relied upon by high-risk populations
Our running example: Signal

- **Signal** powers WhatsApp, Messenger, Skype, Signal
  This means over 1 billion users
- Allows communicating asynchronously (trend)
- Relies on server with limited trust
- Generally trust-on-first-use
Our running example: Signal

- **Signal** powers WhatsApp, Messenger, Skype, Signal
  This means over 1 billion users
- Allows communicating **asynchronously** (trend)
- Relies on server with **limited trust**
- Generally **trust-on-first-use**

Let’s start by a quick overview of the protocol.
Alice \arrow{X3DH} X3DH \arrow{rk_0} Server

Bob
Alice

\[ rk_1, ck_1 \]

Diffie-Helman ratchet

Server

Bob
$m_1 + \text{ keys}$

"hey Bob"
Alice $\rightarrow$ symmetric key ratchet $\rightarrow$ Bob

$ck_2$
Alice ✈️ $m_2$ “where’s the secret stash” ✈️ Server ✈️ Bob
Diffie-Helman ratchet

$rk_1, ck_1$
$m_1 = \text{“hey Bob”}$
symmetric key ratchet

${ck}_2$
Alice

Server

$m_2 = \text{“where’s the secret stash”}$

Bob
The image illustrates a protocol involving Alice, Bob, and a Server, using the Diffie-Hellman ratchet. Alice sends a message to the Server, which then communicates with Bob. The Diffie-Hellman ratchet ensures secure communication, with Alice's key symbolized as $r_k$ and Bob's key as $c_k$. The diagram shows the setup and the flow of the communication.
Alice

Server

Bob

$m_3 + \text{keys}$

“it’s at Oakland”
Signal: a recap

- the protocol is sophisticated
- X3DH for session initiation
- double-ratchet for asynchronous communications, forward secrecy and post-compromise security
- involves non-trivial cryptography (X25519, etc.)

https://signal.org/docs/
Step 1: a protocol specification

Written in ProVerif (symbolic model). Builds on previous work (Euro S&P’17).

Guarantees integrity, confidentiality, forward secrecy, post-compromise security.

Initiator I

Prior Knowledge: 

\[(i, g^i)\]

Responder R

Prior Knowledge: 

\[(r, g^r), (s, g^s), (o, g^o)\]

\[
\text{Initiate}(i, g^r, g^s, g^o) \rightarrow (rk_0): \]

\[
\begin{align*}
generate & (e, g^e) \\
dh_0 & = 0x\text{FF} \mid g^{si} \mid g^{re} \mid g^{se} || g^{oe} \\
rk_0 & = \text{HKDF}(dh_0, 0x00^{32}, \text{"WhisperText"})
\end{align*}
\]
Step 1: a protocol specification

Written in ProVerif (symbolic model). Builds on previous work (Euro S&P’17).

Guarantees integrity, confidentiality, forward secrecy, post-compromise security.

Initiator I

Prior Knowledge: 
\[ (i, g^i) \]

Initiate \((i, g^r, g^s[, g^o]) \) → \((rk_0)\):

- generate \((e, g^e)\)
- \(dh_0 = 0xFF \| g^{si} \| g^{re} \| g^{se} \| g^{oe}\)
- \(rk_0 = HKDF(dh_0, 0x00^{32}, "WhisperText")\)

Responder R

Prior Knowledge: 
\[ (r, g^r), (s, g^s[, (o, g^o)]) \]
Step 2: transcribe specifications to F*

An ML-like language with support for program verification via SMT automation.

- Specifications include more detail than ProVerif (e.g. tags)
- Currently manual; hope to automate it
- Specifications extract to OCaml, for tests – not suitable for implementations!
...transcribed to an F* spec ...

```fsharp
let initiate' 
    (our_identity_priv_key: privkey) (* i *)
    (our_onetime_priv_key: privkey) (* e *)
    (their_identity_pub_key: pubkey) (* g' *)
    (their_signed_pub_key: pubkey) (* g^s *)
    (their_onetime_pub_key: option pubkey) (* g^o, optional *)
  : Tot (lbytes 32) =
      (* output: rk_0 *)

let dh1 = dh our_identity_priv_key their_signed_pub_key in
let dh2 = dh our_onetime_priv_key their_identity_pub_key in
let dh3 = dh our_onetime_priv_key their_signed_pub_key in
let shared_secret =
    match their_onetime_pub_key with
    | None -> ff @| dh1 @| dh2 @| dh3
    | Some their_onetime_pub_key ->
        let dh4 = dh our_onetime_priv_key their_onetime_pub_key in
        ff @| dh1 @| dh2 @| dh3 @| dh4

in
let res = hkdf1 shared_secret zz label_WhisperText in
res
```
...implemented in Low*

```ocaml
val initiate': output: lbuffer uint8 (size 32) ->
    our_identity_priv_key: privkey_p ->
    our_onetime_priv_key: privkey_p ->
    their_identity_pub_key: pubkey_p ->
    their_signed_pub_key: pubkey_p ->
    their_onetime_pub_key: pubkey_p ->
    defined_their_onetime_pub_key: bool ->
  Stack unit
    (requires (fun h -> live h output /
       disjoint output our_identity_priv_key /
       ... (* more disjointness *)))))
  (ensures (fun h0 _ h1 -> modifies1 output h0 h1 /
    (* THE IMPLEMENTATION MATCHES THE SPEC *))
    h1.[output] == Spec.Signal.Core.initiate'
    h0.[our_identity_priv_key] h0.[our_onetime_priv_key] h0.[their_identity_pub_key] h0.[their_signed_pub_key]
    (if defined_their_onetime_pub_key h0.[their_onetime_pub_key] then
        Some(h0.[their_onetime_pub_key])
      else
        None)))
```
Step 3a: implement cryptography

We use HACL* for the cryptographic primitives.

HACL* has been integrated in Firefox, WireGuard, mbedTLS, etc.

Now available on the Web!

Generally useful:

• fills the gap for custom or new primitives (not in WebCrypto or Node)
• a solution for code that needs synchronous APIs
• avoid legacy libraries (OpenSSL on Node).
Step 3b: implement Signal core

We implement all the core operations of the Signal protocol in Low*.

Low* is a low-level subset of F* that compiles to C using the KreMLin compiler.

Low* has been used by HACL*, EverCrypt, Merkle Trees, libquiccrypto.

Now a verified implementation of Signal in C and WebAssembly.