Formally Verified Cryptographic Web Applications in WebAssembly

Jonathan Protzenko
Benjamin Beurdouche
Denis Merigoux
Karthik Bhargavan

Microsoft Research
INRIA
INRIA
INRIA
The Web environment has become the *choice target* for deploying applications.

Think: websites, desktop apps (Electron), server apps (node.js), browser addons...

How about *security-sensitive* applications, such as: *password managers, secure messengers*?
Life is hard for secure web apps

Application developers are at a loss for secure toolchains targeting the Web runtime.

- custom cryptographic schemes
- ad-hoc protocols
- unverifiable app logic
- hostile target environment (JavaScript).

(Larger) **Claim:** the JavaScript toolchain is *inadequate* for Web-based security-sensitive applications.
An F* to WASM toolchain

We formalize a verified pipeline from Low* to WASM and implement it in the KreMLin compiler.
This work’s contributions

- A generic toolchain (formalization and implementation) to compile F* programs to WebAssembly
- The HACL* verified cryptographic library compiled to WebAssembly
- A formally verified implementation of Signal, in WebAssembly
  - Verified for functional correctness, memory safety, side-channel resistance and protocol security
  - No performance penalty; same API; ready to integrate
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 publishes keys to Bob through the server.
Alice

key bundle

Server

Bob
Alice \[ \text{X3DH} \]

\[ rk_0 \]

Server

Bob
Alice $\rightarrow$ Server $\rightarrow$ Bob

$rk_1, ck_1$

Diffie-Helman ratchet
Alice sends $m_1 + \text{keys}$ to the server, which then forwards it to Bob with the message "hey Bob".

- **Alice**: Sends $m_1 + \text{keys}$ to the server.
- **Server**: Receives $m_1 + \text{keys}$ and forwards it to Bob.
- **Bob**: Receives "hey Bob" from the server.
Alice

\[ ck_2 \]

symmetric key ratchet

Bob
Alice

m_2

where’s the secret stash

Server

Bob
Alice  

Server  

\[ m_1 + \text{keys} \]  

Bob
Alice

Server

Bob

Diffie-Helman ratchet

$rk_1, ck_1$
$m_1 = \text{“hey Bob”}$
symmetric key ratchet
\[ m_2 = \text{“where’s the secret stash”} \]
Diffie-Helman ratchet

$rk_2, ck_3$
Alice

Server

$m_3 + \text{keys}$

"it's at Oakland"

Bob
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:**

\[(i, g^i)\]

**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 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“)

Respander 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!
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.
Step 4: compile Low* to WebAssembly

A new, safe, widely supported target for fast, portable execution. Used primarily in web runtimes but not only.

- isolation guarantees
- basic type safety relying on an operand stack and structured control flow
- more compiler support every day: LLVM, emscripten, mono, etc.

Used for video games, AutoCad, large applications...
We formalize a verified pipeline from ProVerif to WASM and extend the KreMLin compiler with a WASM backend.
A direct route from Low* to WASM

We **formalize** the compilation from Low* to WASM.

A **simple** translation (WASM is an expression language) that **eliminates** complexity and fits in two paper pages.

Thanks to a new intermediary language in KreMLin, the compilations rules are **compact**, **auditable** and **simple**.
A direct route from Low* to WASM

We implement the compilation from Low* to WASM.

The implementation is carefully audited and follows the paper rules.

- 2,400 lines of OCaml code (total: 11,000)
- does not implement any sophisticated optimization
- very regular.

Consequence

A high-assurance compilation toolchain to WASM!
An indirect route from Low* to WASM

One reason we chose to implement our own toolchain...

Classic route (via Emscripten): Low* → C → WASM
  - massive TCB
  - no side-channel reasoning
  - requires KreMLin to deal with C semantics (un-necessary transformations)

With only 2,400 extra lines of OCaml, we have greater confidence.
What we prove

Thanks to a combination of techniques, we guarantee:

- **memory safety**, by virtue of Low*
- **functional correctness**, by virtue of the specifications
- absence of “classic” side-channel leaks, by construction and through a dedicated check

In short, we offer a library of **core building blocks** of the Signal protocol.

Session and state management, policies to discard old ratchets, etc. are **left to the JavaScript code** (need integration with the browser).
Integration

We pass the entire testsuite. The WASM memory is behind a closure (defensive). We offer the same API.

- send prekey message A
- send message B
- receive message D
- receive message C
- send message E

Shuffled Signal Protocol Test Vectors as Alice

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>send prekey message A</td>
<td>58ms</td>
</tr>
<tr>
<td>send message B</td>
<td></td>
</tr>
<tr>
<td>receive message D</td>
<td></td>
</tr>
<tr>
<td>receive message C</td>
<td></td>
</tr>
<tr>
<td>send message E</td>
<td></td>
</tr>
</tbody>
</table>

Standard Signal Protocol Test Vectors as Bob

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>receive prekey message A</td>
<td>56ms</td>
</tr>
<tr>
<td>receive prekey message B</td>
<td></td>
</tr>
<tr>
<td>send message C</td>
<td></td>
</tr>
<tr>
<td>send message D</td>
<td></td>
</tr>
<tr>
<td>receive message E</td>
<td></td>
</tr>
</tbody>
</table>
Performance (1)

<table>
<thead>
<tr>
<th>Step</th>
<th>F*-WebAssembly</th>
<th>Vanilla Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>initiate/respond</td>
<td>61.6 ms</td>
<td>74.7 ms</td>
</tr>
<tr>
<td>Diffie-Hellman ratchet</td>
<td>21.7 ms</td>
<td>35.4 ms</td>
</tr>
<tr>
<td>symmetric key ratchet</td>
<td>2.19 ms</td>
<td>3.52 ms</td>
</tr>
</tbody>
</table>

Our implementation is faster on many operations than the original libsignal. (Reason: an asm.js version of curve25519).

For operations involving SHA and AES-CBC, hard to beat native crypto in WebCrypto.
## Performance (2)

<table>
<thead>
<tr>
<th>Primitive</th>
<th>HACL* → C → WASM via Emscripten</th>
<th>HACL* → WASM via KreMLin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chacha20 (4kB, 100k)</td>
<td>2.8 s</td>
<td>4.1 s</td>
</tr>
<tr>
<td>SHA2_256 (16kB, 10k)</td>
<td>1.8 s</td>
<td>3.5 s</td>
</tr>
<tr>
<td>SHA2_512 (16kB, 10k)</td>
<td>1.3 s</td>
<td>3.4 s</td>
</tr>
<tr>
<td>Poly1305_32 (16kB, 10k)</td>
<td>0.15 s</td>
<td>0.4 s</td>
</tr>
<tr>
<td>Curve25519 (1k)</td>
<td>0.7 s</td>
<td>2.5 s</td>
</tr>
<tr>
<td>Ed25519 sign (16kB, 1k)</td>
<td>3.0 s</td>
<td>10.0 s</td>
</tr>
<tr>
<td>Ed25519 verify (16kB, 1k)</td>
<td>3.0 s</td>
<td>10.0 s</td>
</tr>
</tbody>
</table>

- **simple** compilation scheme not always optimal
- 128-bit arithmetic destroys performance, need 32-bit versions
- low hanging fruits: see chacha20.
A general pattern any application in a Web context (desktop, server or browser)

Offers a solution for crypto libraries: new algorithms, custom schemes, absence of async, no legacy binaries

We built software: Signal* + Web-HACL* as a side effect

Please get in touch! https://signalstar.gforge.inria.fr/
Ye olde backuppe slides
Emscripten $\xrightarrow{\text{Low}^*} C$ (KreMLin): OK (ICFP’17)
$C \xrightarrow{\text{WASM}}$ (emscripten): low trust
Emscripten  Low* → C (KreMLin): OK (ICFP’17)
C → WASM (emscripten): low trust
KreMLin  Low* → WASM (KreMLin): OK (S&P’19)
(* Spec *)
let p = pow2 255 - 19

type elem = n:int { 0 <= n \< p }
let add (x y: elem): elem = (x + y) % p

(* Implem *)
type felem = p:uint64_p { length p = 5 }

let fadd (output a b: felem):
  Stack unit
  (requires (fun h0 -> live h0 output \/
    live h0 a \/
    live h0 b \/
    fadd_pre h0.[a] h0.[b])
  (ensures (fun h0 _ h1 ->
    modifies 1 output h0 h1 \/
    h1.[output] == add h0.[a] h0.[b])))
(* Spec *)
let p = pow2 255 - 19

type elem = n:int { 0 <= n \&\& n < p }

let add (x y: elem): elem = (x + y) % p

(* Impl *)
type felem = p:uint64_p { length p = 5 }

let fadd (output a b: felem):
    Stack unit
    (requires (fun h0 -> live h0 output \&\&
                live h0 a \&\& live h0 b \&\&
                fadd_pre h0.[a] h0.[b])
    (ensures (fun h0 _ h1 ->
                modifies_1 output h0 h1 \&\&
                h1.[output] == add h0.[a] h0.[b]))

concise specification
(* Spec *)

let p = pow2 255 - 19

type elem = n: int { 0 < n < p }

let add (x y: elem) = (x + y) % p

(* Implem *)

type felem = p:uint64_p { length p = 5 }

let fadd (output a b: felem):
  Stack unit
  (requires (fun h0 -> live h0 output)
   live h0 a /
   live h0 b /
   fadd_pre h0.[a] h0.[b])
  (ensures (fun h0 _ h1 ->
    modifies_1 output h0 h1 /
    h1.[output] == add h0.[a] h0.[b])))
(*) Spec *)
let p = pow2 255 - 19

type elem = n:int { 0 <= n \&\& n < p }
let add (x y: elem): elem = (x + y) % p

(*) Implem *)

type felem = p:uint64_p { length p = 5 }

let fadd (output a b: felem):
  Stack unit
  (requires (fun h0 -> live h0 output \/
            live h0 a \&\& live h0 b \/
            fadd_pre h0.[a] h0.[b])
   (ensures (fun h0 _ h1 ->
             modifies_1 output h0 h1 \/
             h1.[output] == add h0.[a] h0.[b])))

memory safety
(* Spec *)
let p = pow2 255 - 19

type elem = n:int { 0 <= n /
 n < p }
let add (x y: elem): elem = (x + y) % p

(* Implm *)

type felem = p:uint64_p { length p = 5 }

let fadd (output a b: felem):
  Stack unit
  (requires (fun h0 -> live h0 output /
    live h0 a /
    live h0 b /
    fadd_pre h0.[a] h0.[b])
  (ensures (fun h0 _ h1 ->
    modifies_1 output h0 h1 /
    h1.[output] == add h0.[a] h0.[b])))
(* Spec *)
let p = pow2 255 - 19

let add (x y: elem): elem = (x + y) % p

(* Impl *)
type felem = p:uint64_p { length p = 5 }

let fadd (output a b: felem):
  Stack unit
  (requires (fun h0 -> live h0 output /
    live h0 a /
    live h0 b /
    fadd_pre h0.[a] h0.[b])
  (ensures (fun h0 _ h1 ->
    modifies_1 output h0 h1 /
    h1.[output] == add h0.[a] h0.[b])))
\( fadd = \text{func} \ [\text{int32}; \text{int32}; \text{int32}] \rightarrow [] \)

\text{local} \ [\ell_0, \ell_1, \ell_2 : \text{int32}; \ell_3 : \text{int32}; \ell : \text{int32}] .

\text{call get_stack; loop(}

// Push dst + 8*i on the stack
get_local \ell_0; get_local \ell_3; \text{i32.const 8; i32.binop*; i32.binop+}

// Load a + 8*i on the stack
get_local \ell_1; get_local \ell_3; \text{i32.const 8; i32.binop*; i32.binop+}
\text{i64.load}

// Load b + 8*i on the stack (elided, same as above)
// Add a.[i] and b.[i], store into dst.[i]
\text{i64.binop+; i64.store}

// Per the rules, return unit
\text{i32.const 0; drop}

// Increment i; break if i == 5
get_local \ell_3; \text{i32.const 1; i32.binop+; tee_local \ell_3}
\text{i32.const 5; i32.op =; br_if}
);
\text{i32.const 0}
\text{store_local \ell ; call set_stack; get_local \ell}
...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^r *)
  (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*

```haskell
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 /
                 ... (* more liveness *) /
                 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 then
             Some(h0.[their_onetime_pub_key])
            else
             None)))
```

J. Protzenko et al. — MSR + INRIA Verified Cryptographic Web Applications in WASM May. 22\textsuperscript{st}, 2019 22 / 22