Formally Verified Cryptographic Web Applications in WebAssembly

Jonathan Protzenko  Benjamin Beurdouche  Denis Merigoux  Karthikeyan Bhargavan

Microsoft Research & Inria (Prosecco)

May 6th 2019
Applications have to be deployed on a number of different platforms and architectures: desktop (Windows, Mac, Linux), website and smartphone application (Android and iOS). A popular solution is to write a Web application and bundle it using frameworks like Electron.
Applications have to be deployed on a number of different platforms and architectures: desktop (Windows, Mac, Linux), website and smartphone application (Android and iOS). A popular solution is to write a Web application and bundle it using frameworks like Electron.

Because of this, having good security means in practice ensuring good security using the technologies from the Web Stack.

Examples of cryptographic applications beyond TLS

• Storing encrypted data on servers (Lastpass).
• End-to-end encryption between devices (Whatsapp, Signal).
Applications have to be deployed on a number of different platforms and architectures: desktop (Windows, Mac, Linux), website and smartphone application (Android and iOS). A popular solution is to write a Web application and bundle it using frameworks like Electron.

Because of this, having good security means in practice ensuring good security using the technologies from the Web Stack.

How is security managed on the Web? Standard network protection is ensured by TLS and the HTTPS protocol. But what about higher-level security guarantees?
Applications have to be deployed on a number of different platforms and architectures: desktop (Windows, Mac, Linux), website and smartphone application (Android and iOS). A popular solution is to write a Web application and bundle it using frameworks like Electron.

Because of this, having good security means in practice ensuring good security using the technologies from the Web Stack.

How is security managed on the Web? Standard network protection is ensured by TLS and the HTTPS protocol. But what about higher-level security guarantees?

**Examples of cryptographic applications beyond TLS**

- Storing encrypted data on servers (Lastpass).
- End-to-end encryption between devices (WhatsApp, Signal).
Applications have to be deployed on a number of different platforms and architectures: desktop (Windows, Mac, Linux), website and smartphone application (Android and iOS). A popular solution is to write a Web application and bundle it using frameworks like Electron.

Because of this, having good security means in practice ensuring good security using the technologies from the Web Stack.

How is security managed on the Web? Standard network protection is ensured by TLS and the HTTPS protocol. But what about higher-level security guarantees?

Examples of cryptographic applications beyond TLS

- Storing encrypted data on servers (Lastpass).
- End-to-end encryption between devices (Whatsapp, Signal).
There is a need for cryptography beyond TLS, and at the application level. There are several ways to implement it.
There is a need for cryptography beyond TLS, and at the application level. There are several ways to implement it.

**Web cryptographic solutions**

- The WebCrypto API (fast, reliable, but only certain primitives)
- Custom Javascript (slow, not secure [BDLM14])
- asm.js (C compiled to Javascript)
- WebAssembly (new !)
Challenge: bringing verification to Web applications

Here is how Signal implements its cryptographic protocol:
Challenge: bringing verification to Web applications

Here is how Signal implements its cryptographic protocol:

How do we get verified software inside this architecture?
Choosing the toolchain

- **Specification language**: F* ? Gallina ?
- **Implementation language**: Low* ? Gallina ?
- **Intermediate language**: C ? OCaml ? Nothing ?
- **Web-compatible language**: Javascript ? WebAssembly ?
- **Machine code**: Offline
  - Online
Online toolchain: what is WebAssembly?

WebAssembly [Haa+17] is

- a low-level intermediate representation (or a macro-assembler);
- with structured control flow;
- written as an AST;
- architecture-independent;
- typechecked before execution;
- formally specified;
- memory-management-agnostic (it gives only a flat memory buffer);
- modular with a simple import-export semantic;
- interoperable with Javascript.
A WebAssembly function

(module
  (export "fib" (func $fib))
  (func $fib (param $n i32) (result i32)
    (if (i32.lt_s
         (get_local $n)
         (i32.const 2))
       (return (i32.const 1))
     )
    (return (i32.add
              (call $fib (i32.sub (get_local $n) (i32.const 2)))
              (call $fib (i32.sub (get_local $n) (i32.const 1)))
            ))
  )
)
Building a toolchain on top of WebAssembly

WebAssembly is better suited to cryptographic software than Javascript (machine arithmetic, manual memory management). It is the second best choice after using the WebCrypto API.
WebAssembly is better suited to cryptographic software than Javascript (machine arithmetic, manual memory management). It is the second best choice after using the WebCrypto API.

Our verified toolchain should target it. Prosecco and Microsoft Research have already developed a toolchain from F* (Low*) to C to verify cryptographic primitives [Pro+17].
Building a toolchain on top of WebAssembly

WebAssembly is better suited to cryptographic software than Javascript (machine arithmetic, manual memory management). It is the second best choice after using the WebCrypto API.

Our verified toolchain should target it. Prosecco and Microsoft Research have already developed a toolchain from F* (Low*) to C to verify cryptographic primitives [Pro+17].

Problem

- Should we translate C to WebAssembly?
- Or directly from Low* to WebAssembly?
The case for a domain-specific compiler to WebAssembly

Going via Clight

+ Reusing existing toolchains (Low* to Clight and Emscripten)
  - No verified translation to WebAssembly (unless it’s added to CompCert...)
  - Formalization has to deal with C99 scopes and other C details
  - Loss of information (e.g. immutable local variables)
The case for a domain-specific compiler to WebAssembly

Going via Clight

+ Reusing existing toolchains (Low* to Clight and Emscripten)
  - No verified translation to WebAssembly (unless it’s added to CompCert…)
  - Formalization has to deal with C99 scopes and other C details
  - Loss of information (e.g. immutable local variables)

Custom intermediate language $C_b$

- Have to fork the existing Low* toolchain
+ $C_b$ is expression-based, no undefined behaviour
+ Simpler to formalize
+ Custom, stack-based memory management
From F* to WebAssembly

\[ F^* \] \rightarrow \lambda \omega^* \rightarrow C_\beta \rightarrow \text{WebAssembly} \rightarrow \text{Machine code} \\

[Pro+17] \rightarrow \text{To be presented} \rightarrow \text{To be presented} \rightarrow \text{Browser engine, [Haa+17]}

Offline

Online
F*

let prime = pow2 255 - 19

type elem = e:int{e >= 0 \ e < prime}

let add e1 e2 = (e1 + e2) \% prime

let mul e1 e2 = (e1 * e2) \% prime

let zero: elem = 0

let one: elem = 1
F*

let prime = pow2 255 - 19

let add e1 e2 = (e1 + e2) % prime
let mul e1 e2 = (e1 * e2) % prime

let zero: elem = 0
let one: elem = 1

Low*

type felem = p:uint64 p { length p = 5 }

let fadd (output a b: felem): Stack unit
  (requires (fun h0 -> live pointers h0 [output; a; b] /
     fadd_pre h0.[a] h0.[b])
  (ensures (fun h0 h1 -> modifies only output h0 h1 /
     h1.[output] == add h0.[a] h0.[b])))
\[ \tau ::= \text{int32} \mid \text{int64} \mid \text{unit} \mid \{ f = \tau \} \mid \text{buf } \tau \mid \alpha \]

\[ \nu ::= x \mid g \mid k : \tau \mid () \mid \{ f = \nu \} \]

\[ e ::= \text{readbuf } e_1 e_2 \mid \text{writebuf } e_1 e_2 e_3 \mid \text{newbuf } n \ (e_1 : \tau) \]

\[ \mid \text{subbuf } e_1 e_2 \mid e.f \mid \nu \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \]

\[ \mid \ell \overset{e}{\rightarrow} \mid \text{let } x : \tau = e_1 \text{ in } e_2 \mid \{ f = e \} \mid e \oplus n \mid \text{for } i \in [0; n) \ e \]

\[ P ::= \cdot \mid \text{let } d = \lambda y : \tau. \ e_1 : \tau_1, P \mid \text{let } g : \tau = e, P \]
\( \tau ::= \text{int32} | \text{int64} | \text{unit} | \{ f = \tau \} | \text{buf } \tau | \alpha \)

\( \nu ::= x | g | k : \tau | () | \{ f = \nu \} \)

\( e ::= \text{readbuf } e_1 \ e_2 | \text{writebuf } e_1 \ e_2 \ e_3 | \text{newbuf } n \ (e_1 : \tau) \)

\( | \ \text{subbuf } e_1 \ e_2 | e.f | \nu \ | \text{if } e_1 \ \text{then } e_2 \ \text{else } e_3 \)

\( | \ d \ \overrightarrow{e} \ | \ \text{let } x : \tau = e_1 \ \text{in } e_2 \ | \ \{ f = e \} \ | \ e \oplus n \ | \ \text{for } i \in [0; n) \ e \)

\( P ::= \cdot | \ \text{let } d = \lambda y : \tau. \ e_1 : \tau_1, P | \ \text{let } g : \tau = e, P \)

\[\begin{align*}
\hat{\tau} ::= \text{int32} | \text{int64} | \text{unit} | \text{pointer} \\
\hat{\nu} ::= \ell | g | k : \hat{\tau} | () \\
\hat{e} ::= \text{read}_n \hat{e} | \text{write}_n \hat{e}_1 \ \hat{e}_2 | \text{new} \hat{e} \ | \ \hat{e}_1 \oplus \hat{e}_2 | \ell ::= \hat{e} | \hat{\nu} | \hat{e}_1 ; \hat{e}_2 \\
| \ \text{if } \hat{e}_1 \ \text{then } \hat{e}_2 \ \text{else } \hat{e}_3 : \hat{\tau} | \ \text{for } \ell \in [0; n) \ \hat{e} \ | \ \hat{e}_1 \times \hat{e}_2 | \ \hat{e}_1 + \hat{e}_2 | \ d \ \overrightarrow{\hat{e}} \\
\hat{P} ::= \cdot | \ \text{let } d = \lambda \ell : \hat{\tau}. \ \ell : \hat{\tau}, \hat{e} : \hat{\tau}, \hat{P} | \ \text{let } g : \hat{\tau} = \hat{e}, \hat{P}
\end{align*}\]
\( \text{\(\lambda\)}\text{ow* to } C_b: \text{ desugaring structure values} \)

let \( d = \lambda y : \tau_1. \ e : \tau_2 \) \( \rightsquigarrow \) let \( d = \lambda y : \text{buf} \ \tau_1. \ \text{[readbuf} y \ 0/y] \ e : \tau_2 \)  
let \( d = \lambda y : \tau_1. \ e : \tau_2 \) \( \rightsquigarrow \) let \( d = \lambda y : \tau_1. \ \lambda r : \text{buf} \ \tau_2. \ \text{let} \ x : \tau_2 = e \ \text{in writebuf} \ r \ 0 \ x : \text{unit} \)

\( f \ (e : \tau) \) \( \rightsquigarrow \) let \( x : \text{buf} \ \tau = \text{newbuf} \ 1 \ e \ \text{in} \ f \ x \)  
\( (f \ e) : \tau \) \( \rightsquigarrow \) let \( x : \text{buf} \ \tau = \text{newbuf} \ 1 \ (_\ : \ \tau) \ \text{in} \ f \ e \ x; \ \text{readbuf} \ x \ 0 \)

let \( x : \tau = e_1 \ \text{in} \ e_2 \) \( \rightsquigarrow \) let \( x : \text{buf} \ \tau = \text{take_addr} \ e_1 \ \text{in} \ \text{[readbuf} x \ 0/x] \ e_2 \) 
\( \{ f = e \} \) \ (not\ under\ newbuf) \( \rightsquigarrow \) let \( x : \text{buf} \ \{ f = \tau \} = \text{newbuf} \ 1 \ \{ f = e \} \ \text{in} \ \text{readbuf} \ x \ 0 \)

\( \text{take_addr} \ (\text{readbuf} \ e \ n) \) \( \rightsquigarrow \) \( \text{subbuf} \ e \ n \)  
\( \text{take_addr} \ ((e : f : \tau).f) \) \( \rightsquigarrow \) \( \text{take_addr} \ (e) \oplus \text{offset} (f : \tau, f) \)  
\( \text{take_addr} \ (\text{let} \ x : \tau = e_1 \ \text{in} \ e_2) \) \( \rightsquigarrow \) let \( x : \tau = e_1 \ \text{in} \ \text{take_addr} \ e_2 \)  
\( \text{take_addr} \ (\text{if} \ e_1 \ \text{then} \ e_2 \ \text{else} \ e_3) \) \( \rightsquigarrow \) if \( e_1 \ \text{then} \ \text{take_addr} \ e_2 \ \text{else} \ \text{take_addr} \ e_3 \)
λow* to C♭: performing the struct layout

size int32 = 4
size unit = 4
size int64 = 8
size buf $\tau$ = 4
size $f : \tau$ = offset ($f : \tau, f_n$) + size $\tau_n$
offset ($f : \tau, f_0$) = 0
offset ($f : \tau, f_{i+1}$) = align(offset ($f : \tau, f_i$) + size $\tau_i$, alignment $\tau_{i+1}$)
alignment($f : \tau$) = 8
alignment($\tau$) = size $\tau$ otherwise
align($k, n$) = $k$ if $k \mod n = 0$
align($k, n$) = $k + n - (k \mod n)$ otherwise
\( \lambda \text{ow* to C}_\beta: \text{some rules I} \)

\[
\begin{align*}
\text{LET} & \quad G; V \vdash e_1 : \tau_1 \Rightarrow \hat{e}_1 : \hat{\tau}_1 \vdash V' \\
\ell \text{ fresh} & \quad G; (x \mapsto \ell, \hat{\tau}_1) \cdot V' \vdash e_2 : \tau_2 \Rightarrow \hat{e}_2 : \hat{\tau}_2 \vdash V'' \\
& \quad \frac{\text{FUNDECL}}{G; V \vdash \text{let } x : \tau_1 = e_1 \text{ in } e_2 : \tau_2 \Rightarrow \ell := \hat{e}_1; \hat{e}_2 : \hat{\tau}_2 \vdash V''} \\
\text{VAR} & \quad V(x) = \ell, \tau \\
& \quad \frac{\text{BUFWRITE}}{G; V \vdash \text{let } d = \lambda y : \tau. e_1 : \tau_1 \Rightarrow \text{let } d = \lambda \ell : \hat{\tau}. \ell' : \hat{\tau}' \mapsto \ell, \hat{\tau} \Rightarrow e_1 : \tau_1} \\
& \quad \frac{G; V \vdash \text{writeB} (e_1 + e_2 \times \text{size } \tau_1) e_3 \Rightarrow \hat{e} \vdash V'}{G; V \vdash \text{writebuf} (e_1 : \tau_1) e_2 e_3 \Rightarrow \hat{e} : \text{unit} \vdash V'}
\end{align*}
\]
\textbf{WriteInt32}
\[
G; V \vdash e_1 \Rightarrow \hat{e}_1 \rightarrow V^\prime \quad G; V^\prime \vdash e_2 \Rightarrow \hat{e}_2 \rightarrow V^{\prime\prime}
\]
\[
G; V \vdash \text{writeB } e_1 \ (e_2 : \text{int32}) \Rightarrow \text{write}_4 \ \hat{e}_1 \ \hat{e}_2 \rightarrow V^{\prime\prime}
\]

\textbf{WriteLiteral}
\[
G; V_i \vdash \text{writeB } (e + \text{offset } (\overrightarrow{f : \tau}, f_i)) \ e_i \Rightarrow \hat{e}_i \rightarrow V_{i+1}
\]
\[
G; V_0 \vdash \text{writeB } e \ (\{f = e : \tau\}) \Rightarrow \hat{e}_0; \ldots; \hat{e}_{n-1} \rightarrow V_n
\]

\textbf{WriteDeRef}
\[
\ell \ \text{fresh} \quad V^\prime = \ell, \text{int32} \cdot V \quad G; V \vdash v_i \Rightarrow \hat{v}_i \rightarrow V
\]
\[
\text{memcpy } v_1 \ v_2 \ n = \text{for } \ell \in [0; n) \ \text{write}_1 (v_1 + \ell) \ (\text{read}_1 (v_2 + \ell) \ 1)
\]
\[
G; V \vdash \text{writeB } v_1 \ (\text{readbuf } (v_2 : \tau_2) \ 0) \Rightarrow \text{memcpy } v_1 \ v_2 \ (\text{size } \tau_2) \rightarrow V^\prime
\]
BufNew

\[ \ell, \ell' \text{ fresh} \]

\[ G; x \mapsto (\ell, \text{int32}) \cdot y \mapsto (\ell', \text{int32}) \cdot V \vdash \text{writeB} (x + \text{size } \tau \times y) \ v_1 \Rightarrow \hat{e} \vdash \ V' \]

\[ G; V \vdash \text{newbuf } n \ (v : \tau) \Rightarrow \ell := \text{new } (n \times \text{size } \tau); \text{ for } \ell' \in [0; n) \ \hat{e}; \ \ell \vdash \ V' \]
We adopt a stack-based memory allocation scheme with a watermark at address 0.

\[
\begin{align*}
\text{get}\_\text{stack} & = \text{func } [\text{]} \rightarrow \text{i32 local } [\text{]} \\
& \quad \text{i32.const 0; i32.load} \\
\text{set}\_\text{stack} & = \text{func i32 } \rightarrow \text{[i32 local } \ell : \text{i32]} \\
& \quad \text{i32.const 0; get}\_\text{local } \ell ; \text{ i32.store} \\
\text{grow}\_\text{stack} & = \text{func i32 } \rightarrow \text{i32 local } \ell : \text{i32} \\
& \quad \text{call get}\_\text{stack; get}\_\text{local } \ell ; \text{ i32.op+;} \\
& \quad \text{call set}\_\text{stack; call get}\_\text{stack}
\end{align*}
\]
\[ \text{Write32} \]
\[ \hat{e}_1 \Rightarrow \overrightarrow{i}_1 \quad \hat{e}_2 \Rightarrow \overrightarrow{i}_2 \]
\[ \text{write4} \hat{e}_1 \hat{e}_2 \Rightarrow \overrightarrow{i}_1; \overrightarrow{i}_2; \text{i32.store; i32.const 0} \]

\[ \text{New} \]
\[ \hat{e} \Rightarrow \overrightarrow{i} \]
\[ \text{new} \hat{e} \Rightarrow \overrightarrow{i}; \text{call grow_stack} \]

\[ \text{FOR} \]
\[ \hat{e} \Rightarrow \overrightarrow{i} \]
\[ \text{for } \ell \in [0; n) \hat{e} \Rightarrow \]
\[ \text{loop}(\overrightarrow{i}; \text{drop;} \]
\[ \text{get_local } \ell; \text{i32.const 1; i32.op+; tee_local } \ell; \]
\[ \text{i32.const } n; \text{i32.op =; br_if); i32.const 0} \]
\textbf{Func}

\[ \hat{e} \Rightarrow i \quad \hat{\tau}_i \Rightarrow t_i \]

let \( d = \lambda \ell_1 : \hat{\tau}_1. \ell_2 : \hat{\tau}_2, \hat{e} : \hat{\tau} \Rightarrow \)

\( d = \text{func } t_1 \rightarrow t \text{ local } \ell_1 : t_1 \cdot \ell_2 : t_2 \cdot \ell : t. \)

\[ \text{call get\_stack; } i \; \text{; store\_local } \ell \; \text{; call set\_stack; get\_local } \ell \]
Example: compiled \texttt{fadd} function

\[ fadd = \texttt{func [int32; int32; int32] \rightarrow []} \]
\[
\text{local } [\ell_0, \ell_1, \ell_2 : \text{int32}; \ell_3 : \text{int32}; \ell : \text{int32}].
\]
\[
call \text{get\_stack; loop(}
\]
\[
    \text{\quad// Push dst + 8*i on the stack}
\]
\[
    \text{get\_local } \ell_0; \text{get\_local } \ell_3; \text{i32.const 8; i32.binop*; i32.binop+}
\]
\[
    \text{\quad// Load a + 8*i on the stack}
\]
\[
    \text{get\_local } \ell_1; \text{get\_local } \ell_3; \text{i32.const 8; i32.binop*; i32.binop+}
\]
\[
    \text{i64.load}
\]
\[
    \text{\quad// Load b + 8*i on the stack (elided, same as above)}
\]
\[
    \text{\quad// Add a.[i] and b.[i], store into dst.[i]}
\]
\[
    \text{i64.binop+; i64.store}
\]
\[
    \text{\quad// Per the rules, return unit}
\]
\[
    \text{i32.const 0; drop}
\]
\[
    \text{\quad// Increment i; break if i == 5}
\]
\[
    \text{get\_local } \ell_3; \text{i32.const 1; i32.binop+; tee\_local } \ell_3
\]
\[
    \text{i32.const 5; i32.op =; br\_if}
\]
\[
); \text{i32.const 0}
\]
\[
\text{store\_local } \ell; \text{call set\_stack; get\_local } \ell
\]
The compiler, KreMLin, is 11,000 LOC. The translation is implemented following this formalization, and is designed to be auditable.

We left as future work the task of replicating and adapting the translation correctness of [Pro+17] from $\lambda$ow* to Clight:

**Lemma**

Let $P$ be a $\lambda$ow* program and $e$ be a $\lambda$ow* entry point expression, and assume that they compile: $\downarrow (P) = \hat{P}$ for some $C^*$ program $\hat{P}$ and $\downarrow (e) = \overline{s} \hat{e}$ for some $C^*$ list of statements $\overline{s}$ and expression $\hat{e}$.

Let $V$ be a mapping of local variables containing the initial values of secrets. Then, the $C^*$ program $\hat{P}$ terminates with trace $\ell$ and return value $v$, i.e.,

$\hat{P} \leftarrow ([], V, \overline{s}; \text{return } \hat{e}) \xrightarrow{\ell, *} ([], V', \text{return } v)$ if, and only if, so does the $\lambda$ow* program:

$P \leftarrow ([\}, e[V]) \xrightarrow{\ell, *} ([\}, H', v)$; and similarly for divergence.
Future work: secret independence theorem

**Theorem**
From [Pro+17], proven for the translation from λow* to Clight: given

1. a program well-typed against a secret interface, $\Gamma_s$, i.e., $\Gamma_s, \Gamma_P; \Sigma; \Gamma \vdash (H, e) : \tau$,
2. a well-typed implementation of the $\Gamma_s$ interface, $\Gamma_s; \Sigma; \cdot \vdash_{\Delta} P_s$, such that $P_s$ is equivalent modulo secrets,
3. a pair $(\rho_1, \rho_2)$ of well-typed substitutions for $\Gamma$,

then either:

1. both programs cannot reduce further, i.e. $P_s, P \vdash (H, e)[\rho_1] \not\rightarrow$ and $P_s, P \vdash (H, e)[\rho_2] \not\rightarrow$, or
2. both programs make progress with the same trace, i.e. there exists $\Sigma' \supseteq \Sigma, \Gamma' \supseteq \Gamma, H', e'$, a pair $(\rho'_1, \rho'_2)$ of well-typed substitutions for $\Gamma'$, and a trace $\ell$ such that

   i) $P_s, P \vdash (H, e)[\rho_1] \rightarrow^+_{\ell} (H', e')[\rho'_1]$ and $P_s, P \vdash (H, e)[\rho_2] \rightarrow^+_{\ell} (H', e')[\rho'_2]$, and

ii) $\Gamma_s, \Gamma_P; \Sigma'; \Gamma' \vdash (H', e') : \tau$
Translation validation for secret independence

**Classify**
\[
\begin{align*}
C \vdash i : \pi & \\
C \vdash i : \sigma & \\
\end{align*}
\]

**BinOpPub**
\[
\begin{align*}
o & \text{ is constant-time} \\
C \vdash t.\text{binop } o : m \rightarrow m & \\
\end{align*}
\]

**BinOpPriv**
\[
\begin{align*}
o & \text{ is not constant-time} \\
C \vdash t.\text{binop } o : \pi \rightarrow \pi & \\
\end{align*}
\]

**Load**
\[
\begin{align*}
C \vdash t.\text{load} : \ast \sigma \rightarrow \sigma & \\
\end{align*}
\]

**Local**
\[
\begin{align*}
C(\ell) = m & \\
C \vdash \text{get\_local } \ell : [] \rightarrow m & \\
\end{align*}
\]

**Cond**
\[
\begin{align*}
C \vdash \textbf{if } i_1 \textbf{ then } i_2 \textbf{ else } i_3 : m \rightarrow \pi & \\
\end{align*}
\]
<table>
<thead>
<tr>
<th>Primitive (blocksize, #rounds)</th>
<th>HACL*</th>
<th>libsodium</th>
<th>WHACL*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Curve25519 (1k)</td>
<td>0.83 s</td>
<td>0.15 s</td>
<td>4.05 s</td>
</tr>
<tr>
<td>Chacha20 (4kB, 100k)</td>
<td>1.86 s</td>
<td>1.74 s</td>
<td>6.62 s</td>
</tr>
<tr>
<td>Salsa20 (4kB, 100k)</td>
<td>1.55 s</td>
<td>2.24 s</td>
<td>5.52 s</td>
</tr>
<tr>
<td>Ed25519 sign (16kB, 1k)</td>
<td>3.01 s</td>
<td>0.27 s</td>
<td>15.6 s</td>
</tr>
<tr>
<td>Ed25519 verify (16kB, 1k)</td>
<td>3.07 s</td>
<td>0.24 s</td>
<td>15.6 s</td>
</tr>
<tr>
<td>Poly1305_32 (16kB, 10k)</td>
<td>0.27 s</td>
<td>0.19 s</td>
<td>_</td>
</tr>
<tr>
<td>Poly1305_64 (16kB, 10k)</td>
<td>1.93 s</td>
<td>0.19 s</td>
<td>11.5 s</td>
</tr>
<tr>
<td>SHA2_256 (16kB, 10k)</td>
<td>1.64 s</td>
<td>1.84 s</td>
<td>3.5 s</td>
</tr>
<tr>
<td>SHA2_512 (16kB, 10k)</td>
<td>1.16 s</td>
<td>1.21 s</td>
<td>3.2 s</td>
</tr>
</tbody>
</table>
Application: the Signal protocol

- Symbolic Crypto Model
- Signal Spec (F*)
- Crypto Spec (F*)
- Signal Code (F*)
- HACL* Code (F*)

Signal Protocol (JavaScript)

- Defensive Wrapper (JavaScript)
  - Verified Signal Protocol Core and Crypto Libraries (WebAssembly)

VERIFY PROTOCOL (ProVerif)
- No Attacks Found

VERIFY CODE (F*)
- No Bugs Found

COMPILE with KreMLin
Conclusion

WebAssembly is the best available target for verified applications on the Web. Because it is small and simple, it is also a good compilation target for domain-specific languages. [Wat+19] proposes to extend it with secretness concepts.
WebAssembly is the best available target for verified applications on the Web. Because it is small and simple, it is also a good compilation target for domain-specific languages. [Wat+19] proposes to extend it with secretness concepts.

Adapting the proof from [Pro+17] is the most direct path to improve our confidence in the correctness of the translation to WebAssembly. However, the Trusted Code Base is quite big with the F* compiler and Kremlin. A WebAssembly backend for CompCert could be a significant contribution towards a certified Web-compatible toolchain.
WebAssembly is the best available target for verified applications on the Web. Because it is small and simple, it is also a good compilation target for domain-specific languages. [Wat+19] proposes to extend it with secretness concepts.

Adapting the proof from [Pro+17] is the most direct path to improve our confidence in the correctness of the translation to WebAssembly. However, the Trusted Code Base is quite big with the F* compiler and Kremlin. A WebAssembly backend for CompCert could be a significant contribution towards a certified Web-compatible toolchain.

The Signal case study demonstrate the ability of the F* and Pro/CryptoVerif ecosystem to handle the proof of non-trivial properties (including functional correctness and security) of a large codebase, that can be extracted with a high confidence to portable code.

