Formally Verified Cryptographic Web Applications in WebAssembly

Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, Karthikeyan Bhargavan

Formally Verified Cryptographic Web Applications in WebAssembly

Jonathan Protzenko*, Benjamin Beurdouche†, Denis Merigoux† and Karthikeyan Bhargavan†
*Microsoft Research †Inria

***Abstract*—After suffering decades of high-profile attacks, the need for formal verification of security-critical software has never been clearer. Verification-oriented programming languages like F* are now being used to build high-assurance cryptographic libraries and implementations of standard protocols like TLS. In this paper, we seek to apply these verification techniques to modern Web applications, like WhatsApp, that embed sophisticated custom cryptographic components. The problem is that these components are often implemented in JavaScript, a language that is both hostile to cryptographic code and hard to reason about. So we instead target WebAssembly, a new instruction set that is supported by all major JavaScript runtimes.**

We present a new toolchain that compiles Low*, a low-level subset of the F* programming language, into WebAssembly. Unlike other WebAssembly compilers like Emscripten, our compilation pipeline is focused on compactness and auditability: we formalize the full translation rules in the paper and implement it in a few thousand lines of OCaml. Using this toolchain, we present two case studies. First, we build WHACL*, a WebAssembly version of the existing, verified HACL* cryptographic library. Then, we present LibSignal*, a brand new, verified implementation of the Signal protocol in WebAssembly, that can be readily used by messaging applications like WhatsApp, Skype, and Signal.

I. Introduction: Cryptographic Web Applications

Modern Web applications rely on a variety of cryptographic constructions and protocols to protect sensitive user data from a wide range of attacks. For the most part, applications can rely on standard builtin mechanisms. To protect against network attacks, client-server connections are typically encrypted using the Transport Layer Security (TLS) protocol, available in all Web servers, browsers, and application frameworks like iOS, Android, and Electron. To protect stored data, user devices and server databases are often encrypted by default.

However, many Web applications have specific security requirements that require custom cryptographic mechanisms. For example, popular password managers like LastPass [1] aim to synchronize a user's passwords across multiple devices and back them up on a server, without revealing these passwords to the server. So, the password database is always stored encrypted, with a key derived from a master passphrase known only to the user. If this design is correctly implemented, even a disgruntled employee or a coercive nation-state with full access to the LastPass server cannot obtain the stored passwords. A similar example is that of a cryptocurrency wallet, which needs to encrypt the wallet contents, as well as sign and verify currency transactions.

Secure messaging applications like WhatsApp and Skype use even more sophisticated mechanisms to provide strong guarantees against subtle attacks. For example, they provide

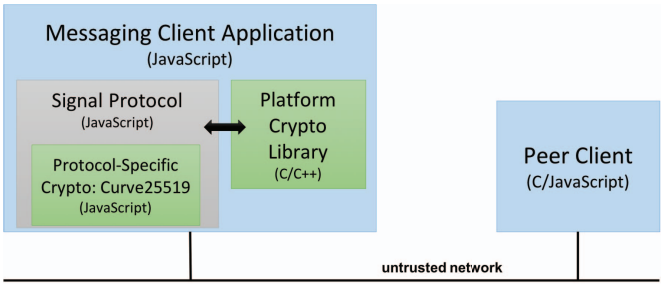

Fig. 1. Secure Messaging Web App Architecture: The application includes the official LibSignal library, which in turn uses the platform's crypto library, but also provides custom implementations for crypto primitives that are not available on all platforms. The security-critical components that we aim to verify are the core signal protocol and all the crypto code it relies on.

end-to-end security between clients, so that a compromised or coerced server cannot read or tamper with messages. They guarantee *forward secrecy*, so that even if one of the devices used in a conversation is compromised, messages sent before the compromise are still secret. They even provide *post-compromise security*, so that a compromised device can recover and continue to participate in a conversation. To obtain these guarantees, many messaging applications today rely on some variant of Signal, a cryptographic protocol designed by Moxie Marlinspike and Trevor Perrin [2], [3].

To provide a seamless experience to users, most Web applications are implemented for multiple platforms; e.g. native apps for iOS and Android, Electron apps that work on most desktop operating systems, installable browser extensions for specific browsers, or a website version accessible from any Web browser. Except for the native apps, these are all written in JavaScript. For example, most Signal-based messaging apps use the official LibSignal library, which has C, Java, and JavaScript versions. The desktop versions of WhatsApp and Skype use the JavaScript version, as depicted in Figure 1.

In this paper, we are concerned with the question of how we can gain higher assurance in the implementations of such cryptographic Web applications. The key novelty of our work is that we target WebAssembly rather than general JavaScript. We show how to build verified implementations of cryptographic primitives so that they can be deployed both within platform libraries (via a C implementation) and within pure JavaScript apps (via a WebAssembly implementation). We show how to build a verified implementation of the Signal protocol (as a WebAssembly module) and use it to develop a drop-in replacement for LibSignal-JavaScript.

WebAssembly. Introduced in 2017, WebAssembly [4] is a portable execution environment supported by all major browsers and Web application frameworks. It is designed to be an alternative to but interoperable with JavaScript.

WebAssembly defines a compact, portable instruction set for a stack-based machine. The language is made up of standard arithmetic, control-flow, and memory operators. The language only has four value types: floating-point and signed numbers, both 32-bit and 64-bit. Importantly, WebAssembly is *typed*, meaning that a well-typed WebAssembly program can be safely executed without fear of compromising the host machine (WebAssembly relies on the OS page protection mechanism to trap out-of-memory accesses). This allows applications to run independently and generally deterministically. WebAssembly applications also enjoy superior performance, since WebAssembly instructions can typically be mapped directly to platform-specific assembly.

Interaction with the rest of the environment, e.g. the browser or a JavaScript application, is done via an import mechanism, wherein each WebAssembly module declares a set of imports whose symbols are resolved when the compiled WebAssembly code is dynamically loaded into the browser. As such, WebAssembly is completely platform-agnostic (it is portable) but also Web-agnostic (there is no mention of the DOM or the Web in the specification).

This clean-slate design, endorsed by all major browsers, yields a language that has cleaner semantics, both on paper [4] and in mechanized rules [5]. As such, WebAssembly provides a better basis for reasoning about correctness than JavaScript, as one does not need to deal with a large semantics rife with corner cases [6], [7]. Indeed, analysis tools for WebAssembly are beginning to emerge. For example, CT-WebAssembly [8] aims to statically rule out some classes of side-channel violations by extending the WebAssembly semantics.

Our approach is to compile WebAssembly code from formally verified source code written in Low* [9], a subset of the F* programming language [10]. As far as we know, this is the first verification toolchain for WebAssembly that supports correctness, memory safety, and side-channel resistance.

Verified Crypto for WebAssembly. Programmers, when authoring Web applications, have very few options when it comes to efficient, trustworthy cryptographic libraries. When running within a browser-like environment, the W3C WebCrypto API [11] provides a limited choice of algorithms, while imposing the restriction that all code calling into WebCrypto must be asynchronous via the mandatory use of promises. This entails that WebAssembly code cannot call WebCrypto, since it does not support async functions. When running within a framework like Electron, programmers can use the crypto package, which calls OpenSSL under the hood and hence supports more algorithms, but requires trust in a large unverified library.

In both these scenarios, the main restriction is perhaps the lack of novel algorithms: for a new algorithm to be available, the W3C must adopt a new standard, and all browsers must implement it; or, OpenSSL must implement it, issue a release, and binaries must percolate to all target environments. For example, modern cryptographic standards such as Curve25519, Chacha20, Poly1305, SHA-3 or Argon2i are not available in WebCrypto or older versions of OpenSSL.

When an algorithm is not available on all platforms, Web developers rely on hand-written, unverified JavaScript implementations or compile such implementations from unverified C code via Emscripten. In addition to correctness questions, this JavaScript code is often vulnerable to new timing attacks. We aim to address this issue, by providing application authors with a verified crypto library that can be compiled to both C and WebAssembly: therefore, our library is readily available in both native and Web environments.

Verified Protocol Code in WebAssembly. Complex cryptographic protocols are hard to implement correctly, and correctness flaws (e.g. [12]) or memory-safety bugs (e.g. HeartBleed) in their code can result in devastating vulnerabilities. A number of previous works have shown how to verify cryptographic protocol implementations to prove the absence of some of these kinds of bugs. In particular, implementations of TLS in F# [13], C [14], and JavaScript [15] have been verified for correctness, memory safety, and cryptographic security. An implementation of a non-standard variant of Signal written in a subset of JavaScript was also verified for cryptographic security [16], but not for correctness.

We propose to build and verify a fully interoperable implementation of Signal in Low* for memory safety and functional correctness with respect to a high-level specification of the protocol in F*. We derive a formal model from this specification and verify its symbolic security using the protocol analyzer ProVerif [17]. We then compile our Low* code to WebAssembly and embed it within a modified version of LibSignal-JavaScript to obtain a drop-in replacement for LibSignal for use in JavaScript Web applications.

Contributions and Outline. Our contributions are threefold. First, we present the first verification and compilation toolchain targeting WebAssembly, along with its formalization and a compact auditable implementation. Second, we present WHACL*, the first high-assurance cryptographic library in WebAssembly, based on the existing HACL* library [18]. Third, we present LibSignal*, a novel verified implementation of the Signal protocol, that by virtue of our toolchain, enjoys compilation to both C and WebAssembly, making it a prime choice for application developers.

We next introduce our source language, F*, along with the target language, WebAssembly (II). Next, we formalize our toolchain, going through various intermediate languages to connect the semantics of our source and target (III). The following section demonstrates the applicability of our approach, by compiling an existing library, HACL*, to WebAssembly, and validating that the generated code enjoys side-channel resistance (IV). Finally, we introduce our novel Signal implementation Signal* (V) and explain its design and verification results.

II. Background: F* and WebAssembly

A. Verified Security Applications in F*

F* is a state-of-the-art verification-oriented programming language [10]. It is a functional programming language with dependent types and an effect system, and it relies on SMT-based automation to prove properties about programs using a weakest-precondition calculus. Once proven correct with regards to their specification, programs written in F* can be compiled to OCaml or F#. Recently [9], F* gained the ability to generate C code, as long as the run-time parts of the program are written in a low-level subset called Low*. This allows the programmer to use the full power of F* for proofs and verification and, relying on the fact that proofs are computationally irrelevant and hence erased, extract the remaining Low* code to C. This approach was successfully used by the HACL* [18] verified crypto library, and the resulting C code is currently used in the Firefox browser and Wireguard VPN. In the present work, we reuse the Low* subset as our source language when compiling to WebAssembly.

Writing specifications in F*. The programmer first writes a high-level specification for her program as a series of pure terminating functions in F*. To illustrate our methodology, we focus on the Curve25519 implementation found in HACL*, which is used as an example throughout this paper. Curve25519 extensively relies on arithmetic in the field of integers modulo the prime $2^{255} - 19$. The concise high-level specification of this field in F* is as follows:

```
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
```

The syntax of F* resembles F# and OCaml. Definitions are introduced using let; the syntax let f x y: Tot t defines a total (pure) function of two parameters x and y, which returns a value of type t while performing no side-effects. Total functions always terminate for all valid inputs; F* enforces this, and the programmer is sometimes required to provide a decreases clause to indicate to F* why the function terminates. Types may be annotated with a refinement between curly braces; for instance, the type elem above describes mathematical integers modulo prime. The backtick operator ` allows using a function as an infix operator for readability.

The definitions above form a specification. The set of field elements are defined as a type that refines mathematical integers (int) and two arithmetic operations on these elements (add, mul) are defined as pure terminating functions. These specifications can be *tested*, by extracting the code above to OCaml and running it on some test vectors as a sanity check. However, this specification is still quite far from a concrete low-level implementation.

Specifications can be layered. Building upon the field arithmetic above, we can define elliptic curve operations for Curve25519, culminating in a full specification for elliptic curve scalar multiplication.

Writing low-level code in Low*. Once equipped with a specification, the programmer can write an efficient stateful *implementation* in Low*, a subset of F*. She can then use the program verification capabilities of F* to show that the low-level implementation *matches* the high-level specification.

Field arithmetic in Curve25519 requires 256-bit integers which are not supported by generic CPUs and hence need to be encoded as arrays of 32-bit or 64-bit integers. Consequently, to implement Curve25519, we define a low-level representation of field elements called felem and stateful functions fadd and felem that operate on these felems:

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

let fadd (output a b: felem):
  Stack unit
    (requires (λ h0 → live_pointers h0 [output; a; b]
                    ∧ fadd_pre h0.[a] h0.[b])
    (ensures (λ h0 _ h1 → modifies_only output h0 h1
                        ∧ h1.[output] == add h0.[a] h0.[b]))
```

The code first defines the type felem, using the popular *unpacked* representation [19], as an array of five *limbs* of 64 bits each. To represent an integer modulo $2^{255} - 19$, each limb only need to use 51 bits, and so it has 13 extra bits that it can use to store pending carries that need to be propagated later. Delaying the carry propagation in this way is a common optimization in many Curve25519 implementations, but needs careful verification since it is also a leading cause of functional correctness bugs [20]–[22].

We then show the type of fadd (its code appears in III-B). The type uses the Stack annotation (instead of Tot) to indicate that the function is stateful, and that it allocates memory only on the stack, not on the heap. The function takes three array arguments: the operands a and b of the addition, and output, the destination array where the result is to be stored.

The pre-condition of the function (indicated by requires) demands that all three arrays must be *live* in the initial heap h0; i.e. they have been allocated (and not freed) and contain values of the expected type and length. Since this function does *not* perform carry propagation, the pre-condition also requires (in fadd_pre, elided) that there must be enough space left in each limb to avoid overflows when adding two limbs.

The post-condition (indicated by ensures) guarantees that once the function has executed, the resulting memory h1 at address output contains *exactly* the specification add applied to the values contained at addresses a and b in the initial memory h0. Furthermore, nothing except the array output has been modified between h0 and h1.

Verification with F*. Verification goes as follows. Seeing the definition of the function, F* computes a *weakest precondition* for it, then checks that this weakest precondition subsumes the requires/ensures annotation of the function. This involves discharging proof goals to the Z3 theorem prover. Once Z3 approves, the correctness meta-theorem of F* concludes that the function does meet its specification. Thereafter, at every

call site for this function F* will verify that the pre-condition is satisfied, and will then be able to use the post-condition to prove further properties.

The verification of fadd ensures that it is memory safe: it does not read or write from an unallocated memory region and it does not access arrays out-of-bounds. It also guarantees functional correctness with respect to a high-level specification add. As we shall discuss in IV-C, our model of machine integers (e.g. uint64) treats them as abstract secrets that cannot be compared or used as memory addresses. Hence, typechecking our code also guarantees a form of timing side-channel resistance called secret independence [9].

Compilation to C. To compile the verified code to C, it must be in Low*, a restricted subset of F* that is suitable for compilation to C. (The fadd function above is in Low*.)

A Low* program must verify against an F* model of the C stack and heap (indicated by the Stack annotation). In particular, it must not modify the structure of the stack or allocate in any previous stack frame or on the heap. Finally, Low* programs may not use certain language features, such as closures, and must essentially remain first-order. Programs that obey all these restrictions compile transparently to C, via KreMLin [9], a dedicated compiler, and do not need any runtime support. In short, Low* is a curated subset of C modeled in F*.

All the specifications and proof annotations are erased at compile-time: pre- and post-conditions, refinements all disappear, leaving only an efficient implementation to be executed, using stack allocations, machine integers, pointers and loops. The fadd function is a small, representative building block. HACL* builds implementations all the way to the elliptic curve scalar multiplication in Curve25519. The total amount of low-level code, including proof annotations, whitespace and comments, is about 10,000 lines of Low* code, for a resulting 700 lines of C code after compilation.

B. WebAssembly: a runtime environment for the Web

WebAssembly is the culmination of a series of experiments (NaCl, PNaCl, asm.js) whose goal was to enable Web developers to write high-performance assembly-like code that can be run within a browser. Now with WebAssembly, programmers can target a portable, compact, efficient binary format that is supported by Chrome, Firefox, Safari and Edge. For instance, Emscripten [23], a modified version of LLVM, can generate WebAssembly. The code is then loaded by a browser, JIT'd to machine code, and executed. This means that code written in, say, C, C++ or Rust, can now be run efficiently on the web.

The syntax of WebAssembly is shown in Figure 2. We use i for WebAssembly instructions and t for WebAssembly types. WebAssembly is a typed, expression language that reduces using an operand stack; each instruction has a function type that indicates the types of operands it consumes from the stack, and the type of operand it pushes onto the stack. For instance, if ℓ has type i32, then get_local ℓ has type $[] \rightarrow$ i32, i.e. it consumes nothing and pushes a 32-bit value on the stack.

f ::=		function
	func t_f local $\overrightarrow{\ell : t}$ $\overrightarrow{i}$	
i ::=		instruction
	if t_f $\overrightarrow{i}$ else $\overrightarrow{i}$	conditional
	call f	function call
	get_local ℓ	read local variable
	set_local ℓ	set local variable
	t.load	load from memory
	t.store	write to memory
	t.const k	push constant
	drop	drop operand
	loop $\overrightarrow{i}$	loop
	br_if	break-if-true
	t.binop o	binary arithmetic
t ::=		value type
	i32	32-bits integer
	i64	64-bits integer
t_f ::=		function type
	$\overrightarrow{t} \rightarrow t$	
o ::=		operator
	add, sub, div, ...	

Fig. 2. WebAssembly Syntax (selected constructs)

Similarly, t.store has type i32; $t \rightarrow []$, i.e. it consumes a 32-bit address, a value of type t, and pushes nothing onto the stack.

We omit from this presentation: n-ary return types for functions (currently not supported by any WebAssembly implementation); treatment of packed 8-bit and 16-bit integer arrays (supported by our implementation, elided for clarity).

This human-readable syntax maps onto a compact binary format. The programmer is not expected to directly write programs in WebAssembly; rather, WebAssembly was designed as a compilation target. Indeed, WebAssembly delivers performance: offline compilers generates better code than a JIT; compiling WebAssembly code introduces no runtime-overhead (no GC); the presence of 64-bit values and packed arrays enables more efficient arithmetic and memory locality.

WebAssembly also delivers better security. Previous works attempted to protect against the very loose, dynamic nature of JavaScript (extending prototypes, overloading getters, rebinding this, etc.) by either defining a "safe" subset [24], [25], or using a hardening compilation scheme [26], [27]. By contrast, none of the JavaScript semantics leak into WebAssembly, meaning that reasoning about a WebAssembly program within a larger context boils down to reasoning about the boundary between WebAssembly and JavaScript.

From a security standpoint, this is a substantial leap forward, but some issues still require attention. First, the boundary between WebAssembly and JavaScript needs to be carefully audited: the JavaScript code is responsible for setting up the WebAssembly memory and loading the WebAssembly modules. This code must use defensive techniques, e.g. make sure that the WebAssembly memory is suitably hidden behind a closure. Second, the whole module loading process needs to be reviewed, wherein one typically assumes that the network content distribution is trusted, and that the WebAssembly API

cannot be tampered with (e.g. Module.instantiate).

Using WebAssembly now. The flagship toolchain for compiling to WebAssembly is Emscripten [23], a compiler from C/C++ to JavaScript that combines LLVM and Binaryen, a WebAssembly-specific optimizer and code emitter. Using Emscripten, several large projects, such as the Unity and Unreal game engines, or the Qt Framework have been ported to WebAssembly. Recently, LLVM gained the ability to directly emit WebAssembly code without going through Binaryen; this has been used successfully by Rust and Mono.

Cryptographic libraries have been successfully ported to WebAssembly using Emscripten. The most popular one is libsodium, which owing to its relatively small size and simplicity (no plugins, no extensibility like OpenSSL) has successfully been compiled to both JavaScript and WebAssembly.

Issues with the current toolchain. The core issue with the current toolchain is both the complexity of the tooling involved and its lack of auditability. Trusting libsodium to be a correct cryptographic library for the web involves trusting, in order: that the C code is correct, something notoriously hard to achieve; that LLVM introduces no bugs; that the runtime system of Emscripten does not interfere with the rest of the code; that the Binaryen tool produces correct WebAssembly code; that none of these tools introduce side-channels; that the code is sufficiently protected against attackers.

In short, the trusted computing base (TCB) is very large. The source language, C, is difficult to reason about. Numerous tools intervene, each of which may be flawed in a different way. The final WebAssembly (and JavaScript) code, being subjected to so many transformations and optimizations, can neither be audited or related to the original source code.

III. From F* to WebAssembly

Seeing that WebAssembly represents a compelling compilation target for security-critical code on the web; seeing that F* is a prime language for writing security-critical code; we repurpose the Low*-to-C toolchain and present a verified compilation path from Low* to WebAssembly.

A. Overview of the toolchain

Protzenko *et.al.* [9] model the Low*-to-C compilation in three phases (Figure 3). The starting point is Explicitly Monadic F* [28]. First, the erasure of all computationally-irrelevant code yields a first-order program with relatively few constructs, which they model as λow*, a simply-typed lambda calculus with mutable arrays. Second, λow* programs are translated to C*, a statement language with stack frames built into its reduction semantics. Third, C* programs go to CLight, CompCert's internal frontend language for C [29].

Semantics preservation across these three steps is shown using a series of simulations. More importantly, this Low*-to-C pipeline ensures a degree of *side-channel* resistance, via type abstraction. This is achieved through *traces* of execution, which track memory access and branches. The side-channel resistance theorem states that if two programs verify against an abstract secret type; if these two programs only differ in their secret values; if the only functions that operate on secrets have secret-independent traces; then once compiled to Clight, the two programs reduce by producing the same result and emitting the same traces. In other words, if the same program operates on different secrets, the traces of execution are indistinguishable.

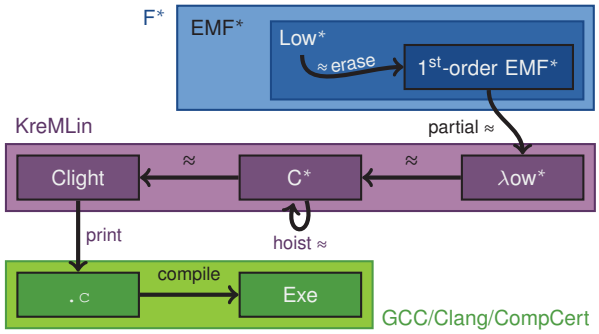

Fig. 3. The original Low*-to-C translation

We repurpose both the formalization and the implementation, and replace the λow* → C* → Clight toolchain with a new λow* → C♭ → WebAssembly translation. We provide a paper formalization in the present section and our implementation is now up and running as a new backend of the KreMLin compiler. (Following [9], we omit the handling of heap allocations, which are not used in our target applications.)

Why a custom toolchain. Using off-the-shelf tools, one can already compile Low* to C via KreMLin, then to WASM via Emscripten. As we mentioned earlier, this TCB is substantial, but in addition to the trust issue, there are technical reasons that justify a new pipeline to WASM.

First, C is ill-suited as an intermediary language. C is a statement language, where every local variable is potentially mutable and whose address can be taken; LLVM immediately tries to recover information that was naturally present in Low* but lost in translation to C, such as immutable local variables ("registers"), or an expression-based representation via a control-flow graph. Second, going through C via C* puts a burden on both the formalization and the implementation. On paper, this mandates the use of a nested stack of continuations for the operational semantics of C*. In KreMLin, this requires not only dedicated transformations to go to a statement language, but also forces KreMLin to be aware of C99 scopes and numerous other C details, such as undefined behaviors. In contrast, C♭, the intermediary language we use on the way to WebAssembly, is expression-based, has no C-specific concepts, and targets WebAssembly whose semantics have no undefined-behavior. As such, C♭ could be a natural compilation target for a suitable subset of OCaml, Haskell, or any other expression-based programming language.

B. Translating λow* to C♭

We explain our translation via an example: the implementation of the fadd function for Curve25519 (II). The function takes two arrays of five limbs each, adds up each limb pairwise

$$
\begin{aligned}
\tau &::= \mathsf{int32} \mid \mathsf{int64} \mid \mathsf{unit} \mid \{\overrightarrow{f = \tau}\} \mid \mathsf{buf}\ \tau \mid \alpha \\
v &::= x \mid g \mid k : \tau \mid () \mid \{\overrightarrow{f = v}\} \\
e &::= \mathsf{readbuf}\ e_1\ e_2 \mid \mathsf{writebuf}\ e_1\ e_2\ e_3 \mid \mathsf{newbuf}\ n\ (e_1 : \tau) \\
&\quad \mid \mathsf{subbuf}\ e_1\ e_2 \mid e.f \mid v \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\
&\quad \mid d\ \overrightarrow{e} \mid \mathsf{let}\ x : \tau = e_1\ \mathsf{in}\ e_2 \mid \{\overrightarrow{f = e}\} \mid e \oplus n \mid \mathsf{for}\ i \in [0; n)\ e \\
P &::= \cdot \mid \mathsf{let}\ d = \lambda \overrightarrow{y : \tau}.\ e_1 : \tau_1, P \mid \mathsf{let}\ g : \tau = e, P
\end{aligned}
$$

Fig. 4. λow* syntax

$$
\begin{aligned}
\hat{\tau} &::= \mathsf{int32} \mid \mathsf{int64} \mid \mathsf{unit} \mid \mathsf{pointer} \\
\hat{v} &::= \ell \mid g \mid k : \hat{\tau} \mid () \\
\hat{e} &::= \mathsf{read}_n\ \hat{e} \mid \mathsf{write}_n\ \hat{e}_1\ \hat{e}_2 \mid \mathsf{new}\ \hat{e} \mid \hat{e}_1 \oplus \hat{e}_2 \mid \ell := \hat{e} \mid \hat{v} \mid \hat{e}_1;\ \hat{e}_2 \\
&\quad \mid \mathsf{if}\ \hat{e}_1\ \mathsf{then}\ \hat{e}_2\ \mathsf{else}\ \hat{e}_3 : \hat{\tau} \mid \mathsf{for}\ \ell \in [0; n)\ \hat{e} \mid \hat{e}_1 \times \hat{e}_2 \mid \hat{e}_1 + \hat{e}_2 \mid d\ \overrightarrow{\hat{e}} \\
\hat{P} &::= \cdot \mid \mathsf{let}\ d = \lambda \overrightarrow{\ell : \hat{\tau}}.\overrightarrow{\ell : \hat{\tau}}, \hat{e} : \hat{\tau}, \hat{P} \mid \mathsf{let}\ g : \hat{\tau} = \hat{e}, \hat{P}
\end{aligned}
$$

Fig. 5. C♭ syntax

(using a for-loop) and stores the result in the output array. It comes with the precondition (elided) that the addition must not overflow, and therefore limb addition does not produce any carries. The loop has an invariant (elided) that guarantees that the final result matches the high-level specification of fadd.

```
let fadd (dst: felem) (a b: felem): Stack unit ... =
  let invariant = ... in
  C.Loops.for 0ul 5ul invariant (λ i → dst.(i) ← a.(i) + b.(i))
```

This function formally belongs to EMF*, the formal model for F* (Figure 3). The first transformation is erasure, which gets rid of the computationally-irrelevant parts of the program: this means removing the pre- and post-condition, as well as any mention of the loop invariant, which is relevant only for proofs. After erasure, this function belongs to λow*.

The λow* language. λow* is presented in Figure 4. λow* is a first-order lambda calculus, with recursion. It is equipped with stack-allocated buffers (arrays), which support: writebuf, readbuf, newbuf, and subbuf for pointer arithmetic. These operations take indices, lengths or offsets expressed in *array elements* (not bytes). λow* also supports structures, which can be passed around as values (as in C). Structures may be stored within an array, or may appear within another structure. They remain immutable; to pass a structure by reference, one has to place it within an array of size one. None of: in-place mutation of a field; taking the address of a field; flat (packed) arrays within structures are supported. This accurately matches what is presently implemented in Low* and the KreMLin compiler.

Base types are 32-bit and 64-bit integers; integer constants are annotated with their types. The type α stands for a secret type, which we discuss in the next section. For simplicity, the scope of a stack allocation is always the enclosing function declaration.

Looking at the fadd example above, the function belongs to Low* (after erasure) because: its signature is in the Stack effect, i.e. it verifies against the C-like memory model; it uses imperative mutable updates over pointers, i.e. the felem types and the ← operator; it uses the C loops library. As such, fadd can be successfully interpreted as the following λow* term:

$$
\begin{aligned}
&\mathsf{let}\ \mathit{fadd} = \lambda(\mathit{dst} : \mathsf{buf}\ \mathsf{int64})(a : \mathsf{buf}\ \mathsf{int64})(b : \mathsf{buf}\ \mathsf{int64}). \\
&\quad \mathsf{for}\ i \in [0; 5).\ \mathsf{writebuf}\ \mathit{dst}\ i\ (\mathsf{readbuf}\ a\ i + \mathsf{readbuf}\ b\ i)
\end{aligned}
$$

λow* enjoys typing preservation, but not subject reduction. Indeed, λow* programs are only guaranteed to terminate if they result from a well-typed F* program that performed verification in order to guarantee spatial and temporal safety. In the example above, the type system of λow* does *not* guarantee that the memory accesses are within bounds; this is only true because verification was performed over the original EMF* program.

The differences here compared to the original presentation [9] are as follows. First, we impose no syntactic constraints on λow*, i.e. we do not need to anticipate on the statement language by requiring that all writebuf operations be immediately under a let. Second, we do not model in-place mutable structures, something that remains, at the time of writing, unimplemented by the Low*/KreMLin toolchain. Third, we add a raw pointer addition $e \oplus n$ that appears only as a temporary technical device during the structure allocation transformation (below).

The C♭ language. C♭ (Figure 5) resembles λow*, but: i) eliminates structures altogether, ii) only retains a generic pointer type, iii) expresses all memory operations (pointer addition, write, reads, allocation) in terms of byte addresses, offsets and sizes, and iv) trades lexical scoping in favor of local names. As in WebAssembly, functions in C♭ declare the set of local mutable variables they introduce, including their parameters.

Translating from λow* to C♭ involves three key steps: ensuring that all structures have an address in memory; converting let-bindings into local variable assignments; laying out structures in memory.

1) Desugaring structure values. Structures are values in λow* but not in C♭. In order to compile these, we make sure every structure is allocated in memory, and enforce that only pointers to such structures are passed around. This is achieved via a mundane type-directed λow*-to-λow* transformation detailed in Figure 6. The first two rules change the calling-convention of functions to take pointers instead of structures; and to take a destination address instead of returning a structure. The next two rules enact the calling-convention changes at call-site, introducing an uninitialized buffer as a placeholder for the return value of f. The next rule ensures that let-bindings have pointer types instead of structure types. The last rule actually implements the allocation of structure literals in memory.

The auxiliary take_addr function propagates the address-taking operation down the control flow. When taking the address of sub-fields, a raw pointer addition, in bytes, is generated. Unspecified cases are ruled out either by typing or by the previous transformations.

This phase, after introducing suitable let-bindings (elided), establishes the following invariants: i) the only subexpressions

$$
\begin{array}{lcll}
\mathsf{let}\ d = \lambda y : \tau_1.\ e : \tau_2 & \rightsquigarrow & \mathsf{let}\ d = \lambda y : \mathsf{buf}\ \tau_1.\ [\mathsf{readbuf}\ y\ 0/y]e : \tau_2 & \text{if } \tau_1 \text{ is a struct type} \\
\mathsf{let}\ d = \lambda y : \tau_1.\ e : \tau_2 & \rightsquigarrow & \mathsf{let}\ d = \lambda y : \tau_1.\ \lambda r : \mathsf{buf}\ \tau_2.\ \mathsf{let}\ x : \tau_2 = e\ \mathsf{in}\ \mathsf{writebuf}\ r\ 0\ x : \mathsf{unit} & \text{if } \tau_2 \text{ is a struct type} \\
\\
f\ (e : \tau) & \rightsquigarrow & \mathsf{let}\ x : \mathsf{buf}\ \tau = \mathsf{newbuf}\ 1\ e\ \mathsf{in}\ f\ x & \text{if } \tau \text{ is a struct type} \\
(f\ e) : \tau & \rightsquigarrow & \mathsf{let}\ x : \mathsf{buf}\ \tau = \mathsf{newbuf}\ 1\ (\_ : \tau)\ \mathsf{in}\ f\ e\ x;\ \mathsf{readbuf}\ x\ 0 & \text{if } \tau \text{ is a struct type} \\
\\
\mathsf{let}\ x : \tau = e_1\ \mathsf{in}\ e_2 & \rightsquigarrow & \mathsf{let}\ x : \mathsf{buf}\ \tau = \mathsf{take\_addr}\ e_1\ \mathsf{in}\ [\mathsf{readbuf}\ x\ 0/x]e_2 & \text{if } \tau \text{ is a struct type} \\
\{\overrightarrow{f = e}\} \quad \text{(not under newbuf)} & \rightsquigarrow & \mathsf{let}\ x : \mathsf{buf}\ \{\overrightarrow{f = \tau}\} = \mathsf{newbuf}\ 1\ \{\overrightarrow{f = e}\}\ \mathsf{in}\ \mathsf{readbuf}\ x\ 0 & \text{if } \tau \text{ is a struct type} \\
\\
\mathsf{take\_addr}(\mathsf{readbuf}\ e\ n) & \rightsquigarrow & \mathsf{subbuf}\ e\ n & \\
\mathsf{take\_addr}((e : \overrightarrow{f : \tau}).f) & \rightsquigarrow & \mathsf{take\_addr}(e) \oplus \mathsf{offset}(\overrightarrow{f : \tau}, f) & \\
\mathsf{take\_addr}(\mathsf{let}\ x : \tau = e_1\ \mathsf{in}\ e_2) & \rightsquigarrow & \mathsf{let}\ x : \tau = e_1\ \mathsf{in}\ \mathsf{take\_addr}\ e_2 & \\
\mathsf{take\_addr}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) & \rightsquigarrow & \mathsf{if}\ e_1\ \mathsf{then}\ \mathsf{take\_addr}\ e_2\ \mathsf{else}\ \mathsf{take\_addr}\ e_3 &
\end{array}
$$

Fig. 6. Ensuring all structures have an address

$$
\begin{array}{lcll}
\mathsf{size\ int32} & = & 4 & \\
\mathsf{size\ unit} & = & 4 & \\
\mathsf{size\ int64} & = & 8 & \\
\mathsf{size\ buf}\ \tau & = & 4 & \\
\mathsf{size}\ \overrightarrow{f : \tau} & = & \mathsf{offset}\ (\overrightarrow{f : \tau}, f_n) + \mathsf{size}\ \tau_n & \\
\\
\mathsf{offset}\ (\overrightarrow{f : \tau}, f_0) & = & 0 & \\
\mathsf{offset}\ (\overrightarrow{f : \tau}, f_{i+1}) & = & \mathsf{align}(\mathsf{offset}\ (\overrightarrow{f : \tau}, f_i) + \mathsf{size}\ \tau_i, & \\
 & & \quad \mathsf{alignment}\ \tau_{i+1}) & \\
\\
\mathsf{alignment}(\overrightarrow{f : \tau}) & = & 8 & \\
\mathsf{alignment}(\tau) & = & \mathsf{size}\ \tau & \text{otherwise} \\
\\
\mathsf{align}(k, n) & = & k & \text{if } k \bmod n = 0 \\
\mathsf{align}(k, n) & = & k + n - (k \bmod n) & \text{otherwise}
\end{array}
$$

Fig. 7. Structure layout algorithm

that have structure types are of the form $\{\overrightarrow{f = e}\}$ or $\mathsf{readbuf}\ e\ n$ and ii) $\{\overrightarrow{f = e}\}$ appears exclusively as an argument to newbuf.

2) Assigning local variables. Transformation 1) above was performed within λow*. We now present the translation rules from λow* to C♭ (Figures 8 and 17). Our translation judgements from λow* to C♭ are of the form $G; V \vdash e : \tau \Rrightarrow e' : \tau' \dashv V'$. The translation takes G, a (fixed) map from λow* globals to C♭ globals; V, a mapping from λow* variables to C♭ locals; and $e : \tau$, a λow* expression. It returns $\hat{e} : \hat{\tau}$, the translated C♭ expression, and V', which extends V with the variable mappings allocated while translating e.

We leave the discussion of the WRITE* rules to the next paragraph, and now focus on the general translation mechanism and the handling of variables.

Since λow* is a lambda-calculus with a true notion of *value*, let-bound variables cannot be mutated, meaning that they can be trivially translated as C♭ local variables. We thus compile a λow* let-binding $\mathsf{let}\ x = e_1$ to a C♭ assignment $\ell := \hat{e}_1$ (rule LET). We chain the V environment throughout the premises, meaning that the rule produces an extended V'' that contains the additional $x \mapsto \ell, \hat{\tau}$ mapping. Translating a variable then boils down to a lookup in V (rule VAR).

The translation of top-level functions (rule FUNDECL) calls into the translation of expressions. The input variable map is pre-populated with bindings for the function parameters, and the output variable map generates extra bindings $\overrightarrow{y}$ for the locals that are now needed by that function.

3) Performing struct layout. Going from λow* to C♭, BUFWRITE and BUFNEW (Figure 8) call into an auxiliary writeB function, defined inductively via the rules WRITE*. This function performs the layout of structures in memory, relying on a set of mutually-defined functions (Figure 7): size computes the number of bytes occupied in memory by an element of a given type, and offset computes the offset in bytes of a field within a given structure. Fields within a structure are aligned on 64-bit boundaries (for nested structures) or on their intrinsic size (for integers), which WASM can later leverage.

We use writeB as follows. From BUFWRITE and BUFNEW, we convert a pair of a base pointer and an index into a byte address using size, then call $\mathsf{writeB}\ e_1\ e_2$ to issue a series of writes that will lay out e_2 at address e_1. Writing a base type is trivial (rule WRITEINT32). Recall that from the earlier desugaring, only two forms can appear as arguments to writebuf: writing a structure located at another address boils down to a memcpy operation (rule WRITEDEREF), while writing a literal involves recursively writing the individual fields at their respective offsets (rule WRITELITERAL).

The allocation of a buffer whose initial value is a struct type is desugared into the allocation of uninitialized memory followed by a series of writes in a loop (rule BUFNEW).

Translated example. After translation to C♭, the earlier fadd function now features four locals: three of type pointer for the function arguments, and one for the loop index; buffer operations take byte addresses and widths.

$$
\begin{array}{l}
\mathsf{let}\ \mathit{fadd} = \lambda(\ell_0, \ell_1, \ell_2 : \mathsf{pointer})(\ell_3 : \mathsf{int32}). \\
\quad \mathsf{for}\ \ell_3 \in [0; 5). \\
\qquad \mathsf{write}_8\ (\ell_0 + i \times 8)\ (\mathsf{read}_8\ (\ell_1 + i \times 8) + \mathsf{read}_8(\ell_2 + i \times 8))
\end{array}
$$

C. Translating C♭ to WebAssembly

The C♭ to WASM translation appears in Figure 9). A C♭ expression $\hat{e}$ compiles to a series of WASM instructions $\overrightarrow{i}$.

WRITE32 compiles a 4-byte write to WASM. WASM is a stack-based language, meaning we accumulate the arguments

$$\textsc{Let}\quad \frac{G; V \vdash e_1 : \tau_1 \Rrightarrow \hat{e}_1 : \hat{\tau}_1 \dashv V' \qquad \ell \text{ fresh} \qquad G; (x \mapsto \ell, \hat{\tau}_1) \cdot V' \vdash e_2 : \tau_2 \Rrightarrow \hat{e}_2 : \hat{\tau}_2 \dashv V''}{G; V \vdash \mathsf{let}\ x : \tau_1 = e_1\ \mathsf{in}\ e_2 : \tau_2 \Rrightarrow \ell := \hat{e}_1; \hat{e}_2 : \hat{\tau}_2 \dashv V''}$$

$$\textsc{FunDecl}\quad \frac{G; \overrightarrow{y \mapsto \ell, \hat{\tau}} \vdash e_1 : \tau_1 \Rrightarrow \hat{e}_1 : \hat{\tau}_1 \dashv \overrightarrow{x \mapsto \ell', \hat{\tau}'} \cdot \overrightarrow{y \mapsto \ell, \hat{\tau}}}{G \vdash \mathsf{let}\ d = \lambda \overrightarrow{y : \tau}.\ e_1 : \tau_1 \Rrightarrow \mathsf{let}\ d = \lambda \overrightarrow{\ell : \hat{\tau}}.\overrightarrow{\ell' : \hat{\tau}'}, \hat{e_1} : \hat{\tau}_1}$$

$$\textsc{Var}\quad \frac{V(x) = \ell, \tau}{G; V \vdash x \Rrightarrow \ell : \tau \dashv V} \qquad \textsc{BufWrite}\quad \frac{G; V \vdash \mathsf{writeB}\ (e_1 + e_2 \times \mathsf{size}\ \tau_1)\ e_3 \Rrightarrow \hat{e} \dashv V'}{G; V \vdash \mathsf{writebuf}\ (e_1 : \tau_1)\ e_2\ e_3 \Rrightarrow \hat{e} : \mathsf{unit} \dashv V'}$$

$$\textsc{WriteInt32}\quad \frac{G; V \vdash e_1 \Rrightarrow \hat{e}_1 \dashv V' \qquad G; V' \vdash e_2 \Rrightarrow \hat{e}_2 \dashv V''}{G; V \vdash \mathsf{writeB}\ e_1\ (e_2 : \mathsf{int32}) \Rrightarrow \mathsf{write}_4\ \hat{e}_1\ \hat{e}_2 \dashv V''}$$

$$\textsc{WriteLiteral}\quad \frac{G; V_i \vdash \mathsf{writeB}\ (e + \mathsf{offset}\ (\overrightarrow{f : \tau}, f_i))\ e_i \Rrightarrow \hat{e}_i \dashv V_{i+1}}{G; V_0 \vdash \mathsf{writeB}\ e\ (\{\overrightarrow{f = e : \tau}\}) \Rrightarrow \hat{e}_0; \ldots; \hat{e}_{n-1} \dashv V_n}$$

$$\textsc{WriteDeRef}\quad \frac{\ell \text{ fresh} \qquad V' = \ell, \mathsf{int32} \cdot V \qquad G; V \vdash v_i \Rrightarrow \hat{v}_i \dashv V \qquad \mathsf{memcpy}\ v_1\ v_2\ n = \mathsf{for}\ \ell \in [0; n)\ \mathsf{write}_1\ (v_1 + \ell)\ (\mathsf{read}_1\ (v_2 + \ell)\ 1)}{G; V \vdash \mathsf{writeB}\ v_1\ (\mathsf{readbuf}\ (v_2 : \tau_2)\ 0) \Rrightarrow \mathsf{memcpy}\ v_1\ v_2\ (\mathsf{size}\ \tau_2) \dashv V'}$$

$$\textsc{BufNew}\quad \frac{\ell, \ell' \text{ fresh} \qquad G; x \mapsto (\ell, \mathsf{int32}) \cdot y \mapsto (\ell', \mathsf{int32}) \cdot V \vdash \mathsf{writeB}\ (x + \mathsf{size}\ \tau \times y)\ v_1 \Rrightarrow \hat{e} \dashv V'}{G; V \vdash \mathsf{newbuf}\ n\ (v : \tau) \Rrightarrow \ell := \mathsf{new}\ (n \times \mathsf{size}\ \tau);\ \mathsf{for}\ \ell' \in [0; n)\ \hat{e};\ \ell \dashv V'}$$

Fig. 8. Translating from λow* to C♭ (selected rules)

to a function on the operand stack before issuing a call instruction: the sequence $\overrightarrow{i_1}$; $\overrightarrow{i_2}$ pushes two arguments on the operand stack, one for the 32-bit address, and one for the 32-bit value. The store instruction then consumes these two arguments.

By virtue of typing, this expression has type unit; for the translation to be valid, we must push a unit value on the operand stack, compiled as i32.const 0. A similar mechanism operates in FOR, where we drop the unit value pushed by the loop body on the operand stack (a loop growing the operand stack would be ill-typed in WebAssembly), and push it back after the loop has finished.

WebAssembly only offers a flat view of memory, but Low* programs are written against a memory stack where array allocations take place. We thus need to implement run-time memory management, the only non-trivial bit of our translation. Our implementation strategy is as follows. At address 0, the memory always contains the address of the top of the stack, which is initially 1. We provide three functions for run-time memory stack management.

```
get_stack  =  func [] → i32 local []
              i32.const 0; i32.load
set_stack  =  func i32 → [] local i32, ..., i32
              i32.const 0; get_local ℓ; i32.store
grow_stack =  func i32 → i32 local i32, ..., i32
              call get_stack; get_local ℓ; i32.op+;
              call set_stack; call get_stack
```

Thus, allocating uninitialized memory on the memory stack merely amounts to a call to grow_stack (rule NEW). Functions save the top of the memory stack on top of the operand stack, then restore it before returning their value (rule FUNC).

Combining all these rules, the earlier fadd is compiled as shown in Figure 10.

Looking forward. This formalization serves as a succinct description of our compiler as well as a strong foundation for future theoretical developments, while subsequent sections demonstrate the applicability and usefulness of our approach. This is, we hope, only one of many future papers connecting state-of-the-art verification tools to WASM. As such, the present paper leaves many areas to be explored. In particular, we leave proofs for these translations to future work. The original formalization only provides paper proofs in the appendix [9]; since we target simpler and cleaner semantics (WASM instead of C), we believe the next ambitious result should be to perform a mechanical proof of our translation, leveraging recent formalizations of the WASM semantics [5].

IV. VERIFIED CRYPTOGRAPHY IN WEBASSEMBLY

We now describe the first application of our toolchain: WHACL*, a WebAssembly version of the (previously existing) verified HACL* crypto library [18]. Compiling such a large body of code demonstrates the viability of our toolchain approach. In addition, we validate the generated code using a new secret independence checker for WebAssembly.

A. The source: HACL*

HACL* is a verified library of cryptographic primitives that is implemented in Low* and compiled to C via KreMLin [18]. It includes implementations of Chacha20 and Salsa20, AES, GCM, Curve25519, Poly1305, Ed25519, HMAC-SHA256, and the SHA2 family. Hence, it provides a full implementation of the NaCl API [30] and many of the key cryptographic algorithms used in modern protocols like TLS 1.3 and Signal. HACL* code is currently used by the Firefox browser and WireGuard VPN, among others.

HACL* is a choice application for our toolchain: it implements many of the newer cryptographic algorithms that are supported by neither WebCrypto nor older versions of OpenSSL. Indeed, WebCrypto supports none of: the Salsa family; Poly1305; or any of the Curve25519 family of APIs [11]. In contrast, WebAssembly is already available for 81% [31] of the userbase, and this number is only going to increase.

Furthermore, developers now have access to a unified verified cryptographic library for both their C and Web-based

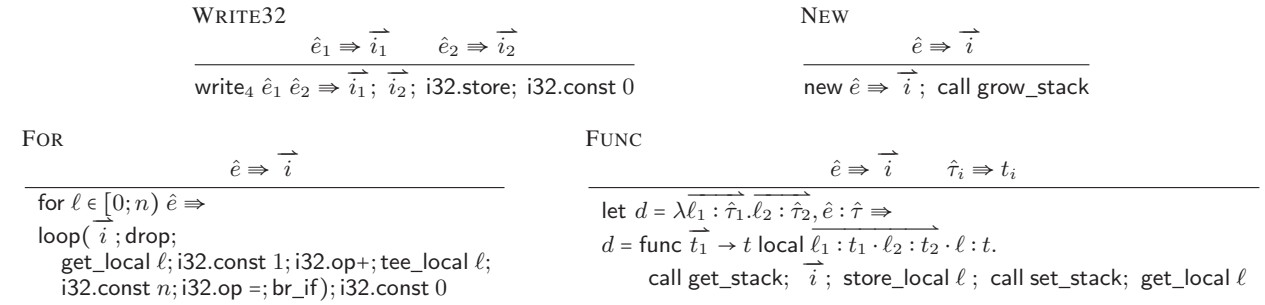

Fig. 9. Translating from C♭ to WebAssembly (selected rules)

```
fadd = func [int32; int32; int32] → []
  local [ℓ0, ℓ1, ℓ2 : int32; ℓ3 : int32; ℓ : int32].
  call get_stack; loop(
    // Push dst + 8*i on the stack
    get_local ℓ0; get_local ℓ3; i32.const 8; i32.binop*; i32.binop+
    // Load a + 8*i on the stack
    get_local ℓ1; get_local ℓ3; i32.const 8; i32.binop*; i32.binop+
    i64.load
    // Load b + 8*i on the stack (elided, same as above)
    // Add a.[i] and b.[i], store into dst.[i]
    i64.binop+; i64.store
    // Per the rules, return unit
    i32.const 0; drop
    // Increment i; break if i == 5
    get_local ℓ3; i32.const 1; i32.binop+; tee_local ℓ3
    i32.const 5; i32.op =; br_if
  ); i32.const 0
  store_local ℓ ; call set_stack; get_local ℓ
```

Fig. 10. Compilation of the fadd example to WebAssembly

applications; rather than dealing with two different toolchains, a single API is used for both worlds. This comes in contrast to applications that use various unverified versions of platform-specific crypto libraries through C, Java, or JavaScript APIs.

*B. The WebAssembly translation: WHACL**

We successfully compiled all the algorithms above to WebAssembly using KreMLin, along with their respective test suites, and dub the resulting library WHACL*, for WebHACL*, a novel contribution. All test vectors pass when the resulting WebAssembly code is run in a browser or in node.js, which serves as experimental validation for our compiler.

Once compiled to WebAssembly, there are several ways clients can leverage WHACL*. In a closed-world setting, the whole application can be written in Low*, meaning one compiles the entire client program with KreMLin in a single pass (which we do with Signal* in V). In this scenario, JavaScript only serves as an entry point, and the rest of the program execution happens solely within WebAssembly. KreMLin automatically generates boilerplate code to: load the WebAssembly modules; link them together, relying on JavaScript for only a few library functions (e.g. for debugging).

Primitive (blocksize, #rounds)	(A)	(B)	(C)
Curve25519 (1k)	0.83 s	0.15 s	4.05 s
Chacha20 (4kB, 100k)	1.86 s	1.74 s	6.62 s
Salsa20 (4kB, 100k)	1.55 s	2.24 s	5.52 s
Ed25519 sign (16kB, 1k)	3.01 s	0.27 s	15.6 s
Ed25519 verify (16kB, 1k)	3.07 s	0.24 s	15.6 s
Poly1305_32 (16kB, 10k)	0.27 s	0.19 s	_
Poly1305_64 (16kB, 10k)	1.93 s	0.19 s	11.5 s
SHA2_256 (16kB, 10k)	1.64 s	1.84 s	3.5 s
SHA2_512 (16kB, 10k)	1.16 s	1.21 s	3.2 s

Fig. 11. Performance evaluation of HACL*. (A) is HACL*/C, (B) is libsodium and (C) is WHACL*.

In an open-world setting, clients will want to use WHACL* from JavaScript. We rely on the KreMLin compiler to ensure that only the top-level API of WHACL* is exposed (via the exports mechanism of WebAssembly) to JavaScript. These top-level entry points abide by the restrictions of the WebAssembly-JavaScript FFI, and only use 32-bit integers (64-bit integers are not representable in JavaScript). Next, we automatically generate a small amount of glue code; this code is aware of the KreMLin compilation scheme, and takes JavaScript ArrayBuffers as input, copies their contents into the WebAssembly memory, calls the top-level entry point, and marshals back the data from the WebAssembly memory into a JavaScript value. We package the resulting code as a portable node.js module for easy distribution.

Performance. We benchmarked (Figure 11) and compared the performance of three WebAssembly cryptographic libraries: (A) HACL* compiled to C via KreMLin then compiled to WebAssembly via Emscripten, (B) libsodium compiled to WebAssembly via Emscripten, and (C) our KreMLin-compiled WHACL*. We measured, for each cryptographic primitive, the run time of batches of 1 to 100 thousands operations on a machine equipped with an Intel i7-8550U processor.

We first compare (A) and (B). When executed as a C library, HACL* is known to have comparable performance with libsodium (with assembly optimizations disabled) [18]. Consequently, when compiled with Emscripten, we find that the code for most HACL* primitives (A) is roughly as fast as the code from libsodium (B). However, we find that the generated WebAssembly code for Curve25519 and Ed25519 from HACL* (A) is 6-11x slower than the code from libsodium

(B). This is because HACL* relies on 128-bit arithmetic in these primitives, which is available on C compilers like gcc and clang, but in WebAssembly, 128-bit integers need to be encoded as a pair of 64-bit integers, which makes the resulting code very slow. Instead, libsodium switches to a hand-written 32-bit implementation, which is significantly faster. If and when HACL* adds a 32-bit implementation for these primitives, we expect the performance gap to disappear. This experience serves as a warning for naive compilations from high-performance C code to WebAssembly, irrespective of the compilation toolchain.

The more interesting comparison is between (C) and (A). Compared to the Emscripten route (A), our custom backend (C) generates code that is between 2.1x (SHA256) and 5.2x (Ed25519/verify) slower. This is a direct consequence of our emphasis on auditability and compactness: KreMLin closely follows the rules from Figure 8 and performs strictly no optimization besides inlining. As such, KreMLin, including the Wasm backend, amounts to 11,000 lines of OCaml (excluding whitespace and comments). Looking forward, we see two avenues for improving performance.

First, we believe a large amount of low-hanging fruits remain in KreMLin. For instance, a preliminary investigation reveals that the most egregious slowdown (Ed25519) is likely due to the use of 128-bit integers. Barring any optimizations, a 128-bit integer is always allocated as a two-word struct (including subexpressions), which in turn adds stack management overhead. We plan to optimize this away, something we speculate Emscripten already does.

Second, JIT compilers for Wasm are still relatively new [32], and also contain many low-hanging fruits. Right now, the Emscripten toolchain uses a WebAssembly-specific optimizer (Binaryen) that compensates for the limitations of browser JITs. We hope that whichever optimizations Binaryen deems necessary soon become part of browser JITs, which will help close the gap between (C) and (A).

C. Secret Independence in WebAssembly

When compiling verified source code in high-level programming language like F* (or C) to a low-level machine language like WebAssembly (or x86 assembly), a natural concern is whether the compiler preserves the security guarantees proved about source code. Verifying the compiler itself provides the strongest guarantees but is an ambitious project [29].

Manual review of the generated code and comprehensive testing can provide some assurance, and so indeed we extensively audit and test the WebAssembly generated from our compiler. However, testing can only find memory errors and correctness bugs. For cryptographic code, we are also concerned that some compiler optimizations may well introduce side-channel leaks even if they were not present in the source.

A Potential Timing Leak in Curve25519.js. We illustrate the problem with a real-world example taken from the Curve25519 code in LibSignal-JavaScript, which is compiled using Emscripten from C to JavaScript (*not* to WebAssembly). The source code includes an fadd function in C very similar to the one we showed in III. At the heart of this function is 64-bit integer addition, which a C compiler translates to some constant-time addition instruction on any modern platform.

Recall, however, that JavaScript has a single numeric type, IEEE-754 double precision floats, which can accurately represent 32-bit values but not 64-bit values. As such, JavaScript is a 32-bit target, so to compile fadd, Emscripten generates and uses the following 64-bit addition function in JavaScript:

```
function _i64Add(a, b, c, d) {
    /* x = a + b*2^32 ; y = c + d*2^32 ; result = l + h*2^32 */
    a = a|0; b = b|0; c = c|0; d = d|0;
    var l = 0, h = 0;
    l = (a + c)>>>0;
    // Add carry from low word to high word on overflow.
    h = (b + d + (((l>>>0) < (a>>>0))|0))>>>0;
    return ((tempRet0 = h,l|0)|0);
}
```

This function now has a potential side-channel leak, because of the (l>>>0) < (a>>>0) subterm, a direct comparison between l and a, one or both of which could be secret. Depending on how the JavaScript runtime executes this comparison, it may take different amounts of time for different inputs, hence leaking these secret values. These kinds of timing attacks are an actual concern for LibSignal-JavaScript, in that an attacker who can measure fine-grained running time (say from another JavaScript program running in parallel) may be able to obtain the long-term identity keys of the participants.

This exact timing leak does not occur in the WebAssembly output of Emscripten, since 64-bit addition is available in WebAssembly, but how do we know that other side-channels are not introduced by one of the many optimizations? This is a problem not just for Emscripten but for all optimizing compilers, and the state-of-the-art for side-channel analysis of cryptographic code is to check that the generated machine code preserves so-called "constant-time" behaviour [33], [34].

We propose to build a validation pass on the WebAssembly code generated from KreMLin to ensure that it preserves the side-channel guarantees proved for the Low* source code. To ensure that these guarantees are preserved all the way to machine code, we hope to eventually connect our toolchain to CT-Wasm [35], a new proposal that advocates for a notion of secrets directly built into the WebAssembly semantics.

Secrets in HACL*. HACL* code manipulates arrays of machine integers of various sizes and by default, HACL* treats all these machine integers as secret, representing them by an abstract type (which we model as α in λow*) defined in a secret integer library. The only public integer values in HACL* code are array lengths and indices.

The secret integer library offers a controlled subset of integer operations known to be constant-time, e.g. the library rules out division or direct comparisons on secret integers. Secret integers cannot be converted to public integers (although the reverse is allowed), and hence we cannot print a secret integer, or use it as an index into an array, or compare its value with another integer. This programming discipline

$$\textsc{Classify}\ \frac{C \vdash i : \pi}{C \vdash i : \sigma} \qquad \textsc{BinOpPub}\ \frac{o \text{ is constant-time}}{C \vdash t.\mathsf{binop}\ o : m\ m \to m}$$

$$\textsc{BinOpPriv}\ \frac{o \text{ is not constant-time}}{C \vdash t.\mathsf{binop}\ o : \pi\ \pi \to \pi} \qquad \textsc{Load}\ \frac{}{C \vdash t.\mathsf{load} : *\sigma\ \pi \to \sigma}$$

$$\textsc{Local}\ \frac{C(\ell) = m}{C \vdash \mathsf{get\_local}\ \ell : [] \to m}$$

$$\textsc{Cond}\ \frac{C \vdash \overrightarrow{i_1} : \overrightarrow{m} \to \pi \qquad C \vdash \overrightarrow{i_{\{2,3\}}} : \overrightarrow{m} \to \overrightarrow{m}}{C \vdash \mathsf{if}\ \overrightarrow{i_1}\ \mathsf{then}\ \overrightarrow{i_2}\ \mathsf{else}\ \overrightarrow{i_3} : \overrightarrow{m}\ \pi \to \overrightarrow{m}}$$

Fig. 12. Secret Independence Checker (selected rules)

guarantees a form of timing side-channel resistance called *secret independence* at the level of the Low* source [9].

Carrying this type-based information all the way to WebAssembly, we develop a checker that analyzes the generated WebAssembly code to ensure that secret independence is preserved, even though Low* secret integers are compiled to regular integers in WebAssembly. We observe that adding such a checker is only made possible by having a custom toolchain that allows us to propagate secrecy information from the source code to the generated WebAssembly. It would likely be much harder to apply the same analysis to arbitrary optimized WebAssembly generated by Emscripten.

We ran our analysis on the entire WHACL* library; the checker validated all of the generated WebAssembly code. We experimented with introducing deliberate bugs at various points throughout the toolchain, and were able to confirm that the checker declined to validate the resulting code.

Checking Secret Independence for WebAssembly. The rules for our secret independence checker are presented in Figure 12. We mimic the typing rules from the original WebAssembly presentation [4]: just like the typing judgement captures the effect of an instruction on the operand stack via a judgement $C \vdash i : \overrightarrow{t} \to \overrightarrow{t}$, our judgement $C \vdash i : \overrightarrow{m} \to \overrightarrow{m}$ captures the information-flow effect of an instruction on the operand stack.

The *context* C maps each local variable to either π (public) or σ (secret). The *mode* m is one of π, σ or $*\sigma$. The $*\sigma$ mode indicates a pointer to secret data, and embodies our hypothesis that all pointers point to secret data. (This assumption holds for the HACL* codebase, but we plan to include a more fine-grained memory analysis in future work.)

For brevity, Figure 12 omits administrative rules regarding sequential composition; empty sequences; and equivalence between $\overrightarrow{m}\ \overrightarrow{m_1} \to \overrightarrow{m}\ \overrightarrow{m_2}$ and $\overrightarrow{m_1} \to \overrightarrow{m_2}$. The mode of local variables is determined by the context C (rule Local). Constant time operations accept any mode m for their operands (rule BinOpPub); if needed, one can always classify data (rule Classify) to ensure that the operands to BinOpPub are homogeneous. For binary operations that are not constant-time (e.g. equality, division), the rules require that the operands be public. Conditionals always take a public value for the condition (rule Cond). For memory loads, the requirement is that the address be a pointer to secret data (always true of all addresses), and that the index be public data (rule Load).

In order to successfully validate a program, the checker needs to construct a context C that assigns modes to variables. For function arguments, this is done by examining the original λow* type for occurrences of α, i.e. secret types. For function locals, we use a simple bidirectional inference mechanism, which exploits the fact that i) our compilation scheme never re-uses a local variable slot for different modes and ii) classifications are explicit, i.e. the programmer needs to explicitly cast public integers to secret in HACL*.

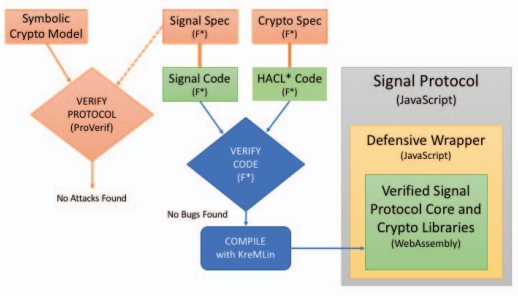

Fig. 13. LibSignal*: We write an F* specification for the Signal protocol and verify its security by transcribing it into a ProVerif model. We then write a Low* implementation of Signal and verify it against the spec using F*. We compile the code to WebAssembly, link it with WHACL* and embed both modules within a defensive JavaScript wrapper in LibSignal-JavaScript.

V. LibSignal*: Verified LibSignal in WebAssembly

As our main case study, we rewrite and verify the core protocol code of LibSignal in F*. We compile our implementation to WebAssembly and embed the resulting code back within LibSignal-JavaScript to obtain a high-assurance drop-in replacement for this popular library, which is currently used in the desktop versions of WhatsApp, Skype, and Signal.

Our Signal implementation is likely the first cryptographic protocol implementation to be compiled to WebAssembly, and is certainly the first to be verified for correctness, memory safety, and side-channel resistance. In particular, we carefully design a defensive API between our verified WebAssembly code and the outer LibSignal JavaScript code so that we can try to preserve some of the strong security guarantees of the Signal protocol, even against bugs in LibSignal.

Our methodology and it components are depicted in Figure 13. We first describe the Signal protocol and how we specify it in F*. Then, we detail our verified implementation in Low* and its integration into LibSignal-JavaScript.

A. An F* Specification for the Signal Protocol

Signal is a cryptographic protocol that allows two devices to exchange end-to-end encrypted messages via an untrusted server that is used only to store and forward encrypted data and public key material. Figures 15 and 16 depict the message flow and the main cryptographic computations in the protocol.

Asynchronous Session Initiation (X3DH). The first phase of the conversation (Figure 15) is sometimes called X3DH [2]. It consists of two messages that set up a bidirectional mutually-authenticated channel between an initiator I and responder R, who are identified by their long-term Diffie-Hellman *identity keys* $((i, g^i), (r, g^r))$.

A distinctive feature of X3DH, in comparison with classic channel-establishment protocols like TLS, is that it is *asynchronous* : I can start sending messages even if R is offline, as long as R has previously uploaded some public key material (called *prekeys*) to the messaging server. Hence, to begin the conversation with R, I must know R's public key g^r, and must have downloaded R's signed Diffie-Hellman prekey g^s (signed with r), and an optional one-time Diffie-Hellman prekey g^o. We assume that I knows the private key for its own public key g^i, and that R remembers the private keys r, s and o.

As depicted in Figure 15, I constructs the first session initiation message in three steps:

- **Initiate**: I generates a fresh Diffie-Hellman keypair (e, g^e) and uses e and its identity key i to compute 3 (or optionally 4) Diffie-Hellman shared secrets in combination with all the public keys it currently knows for R (g^r, g^s, g^o). It then puts the results together to derive an initial *root key* (rk_0) for the session.
- **SendRatchet**: I generates a second Diffie-Hellman keypair (x, g^x) and uses it to compute a Diffie-Hellman shared secret with g^s, which it then combines with rk_0 to obtain a new root key (rk_1) and a *sender chaining key* (ck_0^i) for message sent from I to R.
- **Encrypt**: I uses the sender chaining key to derive authenticated encryption keys (ek_0, mk_0) that it uses to encrypt its first message m_0 to R. It also derives a fresh sender chaining key (ck_1^i) for use in subsequent messages.

On receiving this message, the responder R performs the dual operations (**Respond**, **ReceiveRatchet**, **Decrypt**) to derive the same sequence of keys and decrypts the first message. At this point, we have established a unidirectional channel from I to R. To send messages back from R to I, R calls **SendRatchet** to initialize its own sender chaining key ck_0^r, and then calls **Encrypt** to encrypt messages to I.

At the end of the first two messages, both I and R have a session that consists of a *root key* (rk), two chaining keys, one in each direction (ck^i, ck^r), and ephemeral Diffie-Hellman keys (g^{x_0}, g^{y_0}) for each other.

Per-Message Key Update (Double Ratchet). In the second phase of the conversation (Figure 16), both I and R send sequences (or *flights*) of encrypted messages to each other. At the beginning of each flight, the sender calls **SendRatchet** to trigger a fresh Diffie-Hellman computation that mixes new key material into the root key. Then, for each message in the flight, the sender calls **Encrypt**, which updates the chaining key (and hence encryption keys) with each message. The receiver of the flight symmetrically calls **RecvRatchet** for each new flight, followed by **Decrypt** for each message.

This mechanism by which root keys, chaining keys, and encryption keys are continuously updated is called the Double Ratchet algorithm [3]. Updating the chaining key for every message provides a fine-grained form of *forward secrecy*: even if a device is compromised by a powerful adversary, the keys used to encrypt previous messages cannot be recovered. Updating the root key for every flight of message provides a form of *post-compromise security* [36]: if an adversary gains temporary control over a device and obtains all its keys, he can read and tamper with the next few messages in the current flight, but loses this ability as soon as a new flight of messages is sent or received by the device.

Specifying Signal in Pure F*. We formally specify the Signal protocol in a purely functional (and terminating) subset of F*. Appendix B shows the full F* module Spec.Signal.Core with the main functions of this specification: initiate, respond, ratchet, encrypt, and decrypt.

The main difference between these functions and the protocol operations in Figures 15 and 16 (except for the change of syntax) is that our F* code is purely functional and so it cannot generate fresh random values, such as ephemeral keys. Instead, each function is explicitly given as additional arguments all the fresh key material it may need. With this change, the code for sendRatchet and recvRatchet becomes the same, and is implemented as a single ratchet function.

Spec.Signal.Core in turn relies on two other specification modules: (1) Spec.Signal.Crypto specifies the cryptographic constructions used in Signal, by building on the formal crypto specs in HACL*; (2) Spec.Signal.Messages specifies serializers for protocol messages. For example, the function encrypt in Appendix B calls the hmac and hkdf3 functions to derive new keys, and calls aes_enc and mac_whisper_message to encrypt and then MAC the message; all these functions are from Spec.Signal.Crypto. To serialize the encrypted message before MACing, it calls serialize_whisper_message (from Spec.Signal.Messages).

Linking the F* specification to a security proof. Various aspects of the Signal protocol have been previously studied in a variety of cryptographic models, using both manual proofs [37]–[40] and automated tools [16]. One of our goals is to bridge the gap between these high-level security analyses and the concrete low-level details of how LibSignal is implemented and deployed in messaging applications today.

Although our Signal specification is written in the syntax of F*, it is quite similar to protocol models written in other formal languages. For example, we were able to easily transcribe our specification in the input language for the ProVerif symbolic protocol analyzer [17]. We analyzed the resulting model for all the security goals targeted by Signal: confidentiality, mutual authentication, forward secrecy, and post-compromise security. To simplify automatic verification, we proved these properties separately for the X3DH protocol and the subsequent Double-Ratchet phase, and we limited each flight to 2 messages. Our verification results for this model closely mirror previous results in [16], which used ProVerif to analyze a non-standard variant of Signal. Hence, our analysis serves both as a sanity

check on our specification, and as a confirmation that the expected security guarantees do hold for the standard version of Signal implemented in LibSignal.

B. Implementing Signal in Low*

An implementation of the Signal protocol needs to not just encode the protocol logic depicted in Figures 15 and 16 but also make choices on what cryptographic primitives to use, how to format messages, and how to provide a usable high-level API to a messaging application like WhatsApp. Since we aim to build a drop-in replacement for LibSignal-JavaScript, we mostly adopt the design decisions of that library.

Crypto Algorithms from HACL*. To implement message encryption, LibSignal uses a combination of AES-CBC and HMAC-SHA256 to implement a custom (but relatively standard) scheme for authenticated encryption with associated data (AEAD). To derive keys, LibSignal implements HKDF, again using HMAC-SHA256. For both AES-CBC and HMAC-SHA256, LibSignal-JavaScript relies on the WebCrypto API.

For Diffie-Hellman, LibSignal relies on the Curve25519 elliptic curve, and for signatures, it relies on a non-standard signature scheme called XEdDSA [41]. Neither of these primitives are available in WebCrypto. So, LibSignal-JavaScript includes a C implementation of these constructions, which is compiled to JavaScript using Emscripten. As discussed in Section IV-C, the resulting JavaScript is vulnerable to timing attacks. And even if Curve25519 is added to WebCrypto, XEdDSA is unlikely to be included in any standard API. Hence, high-assurance WebAssembly implementations for these primitives appear to be needed for LibSignal.

Most of these primitives were already available in HACL*, except for AES-CBC and XEdDSA. We extended HACL* with formal specifications and verified implementations for these primitives and compiled them to WebAssembly.

Formatting Messages using Protocol Buffers. To define its concrete message formats, LibSignal uses the ProtoBuf format, known for its compactness. Hence, LibSignal-JavaScript includes an efficient ProtoBuf parser and serializer written in JavaScript. Parsing protocol messages is an error-prone task, and verifying efficient parsers can be time-consuming. So instead, we treat the ProtoBuf library as untrusted code (under the control of the adversary) and reimplement a verified serializer for the one case in LibSignal where the security of the protocol relies on the message formatting.

When user messages are encrypted in LibSignal, they are first enciphered using AES-CBC, then the ciphertext is formatted with a ProtoBuf format called WhisperMessage, and the resulting message is HMACed for integrity. Consequently, the formatting of WhisperMessages becomes security-critical: if the ProtoBuf library has a bug in the serialization or parsing of these messages, an attacker may be able to bypass the HMAC and tamper with messages sent between devices.

We specify and implement a verified serializer for the WhisperMessage ProtoBuf format. This code includes generic serializing functions for variable-size integers (varint) and bytearrays (bytes), and a specialized function for converting a WhisperMessage into a sequence of bytes. This serializer is called during both message encryption and decryption. Notably, we do not implement a verified WhisperMessage parser, which would be significantly more complex. Instead, we require that the (unverified) application code at the recipient parses the encrypted message and call the decrypt function with the message components. To verify the MAC, our code re-serializes these components using our verified serializer. This design choice imposes a small performance penalty during decryption, but yields protocol code that is simpler and easier to verify.

Implementing the Core Protocol Functions. We closely followed our formal specification to reimplement the core functionality of LibSignal in Low*. The main difference is that our code is stateful: it reads and writes from arrays that are allocated by the caller, and it stores and modifies local variables and arrays on the stack. (In the compiled WebAssembly, all these arrays are allocated within the WASM memory.) We present in Appendix C. the Low* implementation for the ratchet function of the Signal Protocol. The full Low* codebase for Signal, including the ProtoBuf serializer and all protocol functions, consists of 3500 lines of code, compared to 570 lines of F* specifications.

We prove that our low-level code matches the high-level spec (functional correctness), and that it never reads and writes arrays out-of-bounds (memory safety). Furthermore, we prove secret independence for the whole protocol layer: our Signal code treats all inputs as secret and hence never branches on secret values or reads memory at secret indices. Note that the application code outside our verified core may well leak identity keys and message contents, but our proof guarantees that these leaks will not come from our protocol code.

A Wrapped WebAssembly Module for Signal. We compile our Signal code to WebAssembly functions where all inputs and outputs are expected to be allocated in the WASM memory. For instance, the initiate function, which in Low* takes five pointers and a boolean, is now a WASM export that wants five WASM addresses along with a 32-bit integer for the boolean. It returns an error code, also a 32-bit integer.

```
(type 34 (func (param i32 i32 i32 i32 i32 i32) (result i32)))
(func 34 (type 34) (local ...))
(export "Signal_Impl_Core_initiate" (func 34))
```

However, typical JavaScript applications like LibSignal-JavaScript would use JavaScript arrays and records to pass around session state, ephemeral key material, and parsed messages. To properly embed our WebAssembly code within a JavaScript application, we automatically generate a wrapper module in JavaScript that provides functions to translate back and forth between the two views by encoding and decoding JavaScript ArrayBuffers in the WASM memory. For example, the JavaScript wrapper code for calling a WebAssembly function that expects a list of buffer objects is in Figure 14.

Extending the Protocol Module to Full LibSignal*. LibSignal encapsulates all the protocol functionality within a

```
function callWith(f) {
    // Saves the stack pointer value before the function call
    var m32 = new Uint32Array(FStarSignal.Kremlin.mem.buffer);
    var sp = m32[0];
    // Calls the function
    var ret = f();
    // Restores thae value of the stack pointer
    m32[0] = sp;
    return ret;
}

function callWithBuffers(args, func) {
  callWith(() => {
    // Allocates arguments in the Wasm memory and grows the stack pointer
    var pointers = args.map((arg) => grow(new Uint8Array(arg)));
    // Calls the function with the pointers to the allocated zones
    var result = func(pointers);
    for (var i = 0; i < args.length; i++) {
      // Copying the contents back to Javascript
      args[i].set(read_memory(pointers[i], args[i].byteLength));
    }
    return result;
  });
}

// Example of function using the combinator
function FStarGenerateKeyPair() {
  var keyPair = {privKey: new ArrayBuffer(32), pubKey: new ArrayBuffer(33)};
  callWithBuffers(
    [keyPair.privKey, keyPair.pubKey],
    function([privKeyPtr, pubKeyPtr]
  ) {
    FStarSignal.Module.Signal_Impl_Core_generate_key_pair(
      privKeyPtr, pubKeyPtr
    );
  })
  return keyPair;
}
```

Fig. 14. JavaScript wrapper for calling a WebAssembly function func that expects a list of ArrayBuffer objects encoded in the WebAssembly memory.

small set of JavaScript functions that provide a simple session-based API to the user application. At any point, the user may ask LibSignal to either (a) initiate a new session, or (b) to respond to a session request it has received, or (c) to encrypt a message for a session, or (d) to decrypt a message received for a session. In addition, periodically, the application may ask LibSignal to generate signed and onetime prekeys.

The code for these functions needs to manage a session data structure, and load and store it from long-term storage; it needs to implement message formats, handle message loss and retransmission, and respond gracefully to a variety of errors. In LibSignal-JavaScript, all this code is interleaved with the core cryptographic protocol code for Signal. We carefully refactored the JavaScript code to separate out the protocol code as a separate module, and then replaced this module with our verified WebAssembly implementation. Hence, we obtain a modified LibSignal* library that meets the same API as LibSignal-JavaScript, but uses a verified WebAssembly code for both the protocol operations and the cryptographic library.

Protecting Signal against JavaScript bugs. In LibSignal, the application stores the device's long-term private identity key, but all other session secrets, including the Diffie-Hellman private keys, session root keys, chaining keys, and pending message encryption keys, are stored locally by LibSignal using the web storage API. Although the user application is not entrusted with this data, due to the inherent lack of isolation in JavaScript, any bug in the JavaScript code of LibSignal, the user application, or any of the modules they depend on may leak these session secrets, breaking the guarantees of Signal.

So what security guarantees can we expect to preserve when we embed our verified WebAssembly code within an unverified application like LibSignal-JavaScript? The isolation guarantees of WebAssembly mean that bugs in the surrounding JavaScript cannot affect the functional behavior of our verified code. However, the JavaScript still has access to the WebAssembly memory, and so any bug can still corrupt or leak all our protocol secrets.

To protect against such bugs, we write our JavaScript wrapper in a *defensive* style: it hides the WASM memory in a closure that only reveals a functional API to the rest of LibSignal. Since all the cryptographic functionality is implemented as WebAssembly modules loaded within this wrapper, short-term session secrets never need to leave this wrapper. The only secrets that remain outside our wrapper are the long-term identity keys and medium-term prekeys. These keys are needed for session setup, but once the first two messages have been exchanged, even a bug that reveals all these long-term and medium-term keys to the adversary will not reveal the messaging keys. Hence, our defensive design tries to preserve Signal's forward secrecy guarantees even against buggy software components. However, note that this protection is partial and unverified; to fully protect against malicious JavaScript applications, we would need further defensive measures, like those proposed in prior work [27], [42], [43].

Evaluation. LibSignal-JavaScript comes with a comprehensive browser-based test-suite. We ran these tests on our modified LibSignal implementation to verify that our verified code interoperates correctly with the rest of LibSignal. This demonstrates that our implementation can be used as a drop-in replacement for LibSignal-JavaScript in applications like WhatsApp, Skype and Signal. The performance of our code is roughly the same as unmodified LibSignal: any speed improvements we may anticipate by using WebAssembly is offset by the overhead of encoding and decoding data structures between JavaScript and WebAssembly. Furthermore, our code has to use WebAssembly implementations even for cryptographic algorithms like AES-CBC and HMAC-SHA256 for which fast native implementations are available in the WebCrypto API but as async functions that cannot be called from WebAssembly.

Our modified LibSignal is a useful proof-of-concept applicable to real-world cryptographic applications deployed today. However, a principled approach when building new Web applications would be to design the application with clean WebAssembly-friendly APIs between the JavaScript and verified WebAssembly components. We advocate that the WebCrypto API be extended to cover more modern cryptographic primitives, that it also provide a synchronous API usable from WebAssembly, and that mainstream browsers use verified crypto code in C or assembly [44]–[48] to implement this API. When verified native crypto is unavailable, applications can fall back to verified WASM crypto libraries like WHACL*.

REFERENCES

[1] "The lastpass password manager." [Online]. Available: https://www.lastpass.com/how-lastpass-works

[2] M. Marlinspike and T. Perrin, "The x3dh key agreement protocol," 2016, https://signal.org/docs/specifications/x3dh/.

[3] T. Perrin and M. Marlinspike, "The double ratchet algorithm," 2016, https://signal.org/docs/specifications/doubleratchet/.

[4] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017, pp. 185–200.

[5] C. Watt, "Mechanising and verifying the webassembly specification," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, 2018, pp. 53–65.

[6] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The essence of javascript," in *European conference on Object-oriented programming*. Springer, 2010, pp. 126–150.

[7] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith, "A trusted mechanised javascript specification," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2014, pp. 87–100.

[8] J. Renner, S. Cauligi, and D. Stefan, "Constant-time webassembly," 2018, https://cseweb.ucsd.edu/~dstefan/pubs/renner:2018:ct-wasm.pdf.

[9] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, and N. Swamy, "Verified low-level programming embedded in f*," *Proceedings of the ACM on Programming Languages*, vol. 1, no. ICFP, pp. 17:1–17:29, Aug. 2017.

[10] N. Swamy, C. Hricu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin, "Dependent types and multi-monadic effects in F*," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2016, pp. 256–270.

[11] "Web cryptography api." [Online]. Available: https://www.w3.org/TR/WebCryptoAPI

[12] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of TLS," in *IEEE Symposium on Security and Privacy (Oakland)*, 2015, pp. 535–552.

[13] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub, "Implementing TLS with verified cryptographic security," in *IEEE Symposium on Security and Privacy (Oakland)*, 2013, pp. 445–459.

[14] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Z. Béguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue, "Implementing and proving the TLS 1.3 record layer," in *IEEE Symposium on Security and Privacy (Oakland)*, 2017, pp. 463–482.

[15] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *IEEE Symposium on Security and Privacy (Oakland)*, 2017, pp. 483–502.

[16] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach," in *2nd IEEE European Symposium on Security and Privacy (EuroSP)*, 2017, pp. 435–450.

[17] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and proverif," *Foundations and Trends in Privacy and Security*, vol. 1, no. 1-2, pp. 1–135, Oct. 2016.

[18] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A verified modern cryptographic library," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, ser. CCS '17, 2017, pp. 1789–1806.

[19] D. J. Bernstein, "Curve25519: new diffie-hellman speed records," in *Public Key Cryptography-PKC 2006*. Springer, 2006, pp. 207–228.

[20] D. J. Bernstein, B. Van Gastel, W. Janssen, T. Lange, P. Schwabe, and S. Smetsers, "Tweetnacl: A crypto library in 100 tweets," in *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*. Springer, 2014, pp. 64–83.

[21] D. Benjamin, "poly1305-x86.pl produces incorrect output," https://mta.openssl.org/pipermail/openssl-dev/2016-March/006161, 2016.

[22] H. Böck, "Wrong results with Poly1305 functions," https://mta.openssl.org/pipermail/openssl-dev/2016-March/006413, 2016.

[23] A. Zakai, "Emscripten: An llvm-to-javascript compiler," in *ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA)*, 2011, pp. 301–312.

[24] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra, "Automated analysis of security-critical javascript apis," in *IEEE Symposium on Security and Privacy (Oakland)*, 2011, pp. 363–378.

[25] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis, "Defensive javascript," in *Foundations of Security Analysis and Design VII*. Springer, 2014, pp. 88–123.

[26] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits, "Fully abstract compilation to javascript," in *ACM SIGPLAN Notices*, vol. 48, no. 1. ACM, 2013, pp. 371–384.

[27] N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P. Strub, and G. M. Bierman, "Gradual typing embedded securely in javascript," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2014, pp. 425–438.

[28] D. Ahman, C. Hriţcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy, "Dijkstra monads for free," in *ACM SIGPLAN Notices*, vol. 52, no. 1. ACM, 2017, pp. 515–529.

[29] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[30] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*. Springer, 2012, pp. 159–176.

[31] "Can i use: Webassembly." [Online]. Available: https://caniuse.com/#feat=wasm

[32] A. Jangda, B. Powers, A. Guha, and E. Berger, "Mind the gap: Analyzing the performance of webassembly vs. native code," 2019.

[33] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *USENIX Security Symposium*, 2016, pp. 53–70.

[34] G. Barthe, B. Grégoire, and V. Laporte, "Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time"," in *IEEE Computer Security Foundations Symposium (CSF)*, 2018, pp. 328–343.

[35] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "Ct-wasm: Type-driven secure cryptography for the web ecosystem," *arXiv preprint arXiv:1808.01348*, 2018.

[36] K. Cohn-Gordon, C. J. F. Cremers, and L. Garratt, "On post-compromise security," in *IEEE 29th Computer Security Foundations Symposium (CSF)*, 2016, pp. 164–178.

[37] K. Cohn-Gordon, C. J. F. Cremers, B. Dowling, L. Garratt, and D. Stebila, "A formal security analysis of the signal messaging protocol," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017, pp. 451–466.

[38] M. Bellare, A. C. Singh, J. Jaeger, M. Nyayapati, and I. Stepanovs, "Ratcheted encryption and key exchange: The security of messaging," in *CRYPTO*, 2017, pp. 619–650.

[39] J. Jaeger and I. Stepanovs, "Optimal channel security against fine-grained state compromise: The safety of messaging," in *CRYPTO*, Cham, 2018, pp. 33–62.

[40] B. Poettering and P. Rösler, "Towards bidirectional ratcheted key exchange," in *CRYPTO*, Cham, 2018, pp. 3–32.

[41] T. Perrin, "The xeddsa and vxeddsa signature schemes," 2017, https://signal.org/docs/specifications/xeddsa/.

[42] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis, "Language-based defenses against untrusted browser origins," in *Proceedings of the 22th USENIX Security Symposium*, 2013, pp. 653–670.

[43] ——, *Defensive JavaScript*, 2014, pp. 88–123.

[44] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel, "Verified correctness and security of openssl HMAC," in *USENIX Security Symposium*, 2015, pp. 207–221.

[45] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P. Strub, "Jasmin: High-assurance and high-speed cryptography," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 1807–1823.

[46] A. W. Appel, "Verification of a cryptographic primitive: Sha-256," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 37, no. 2, p. 7, 2015.

[47] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson, "Vale: Verifying high-performance cryptographic assembly code," in *Proceedings of the USENIX Security Symposium*, Aug. 2017.

[48] A. Tomb, "Automated verification of real-world cryptographic implementations," *IEEE Security and Privacy*, vol. 14, no. 6, pp. 26–33, 2016.

APPENDIX

A. THE SIGNAL CRYPTOGRAPHIC PROTOCOL

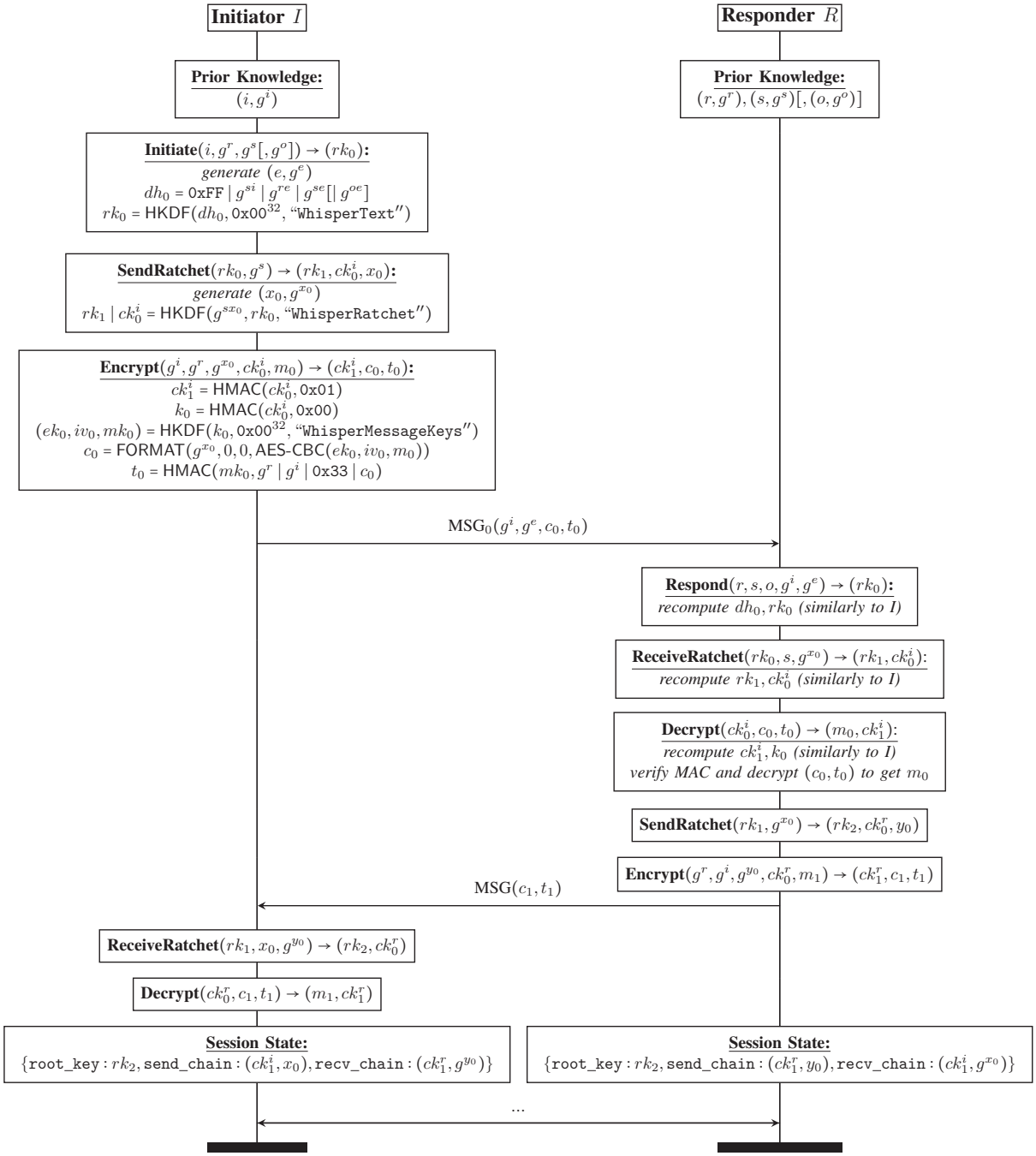

Fig. 15. Signal Protocol (first two messages). These messages set up a bidirectional mutually authenticated channel between I and R, using a series of Diffie-Hellman operations. Each message carries a payload. This protocol is sometimes called X3DH. The figure does not show the (out-of-band) prekey message in which R delivers (g^s, g^o) to I (via the server) and I verifies R's ED25519 signature on g^s.

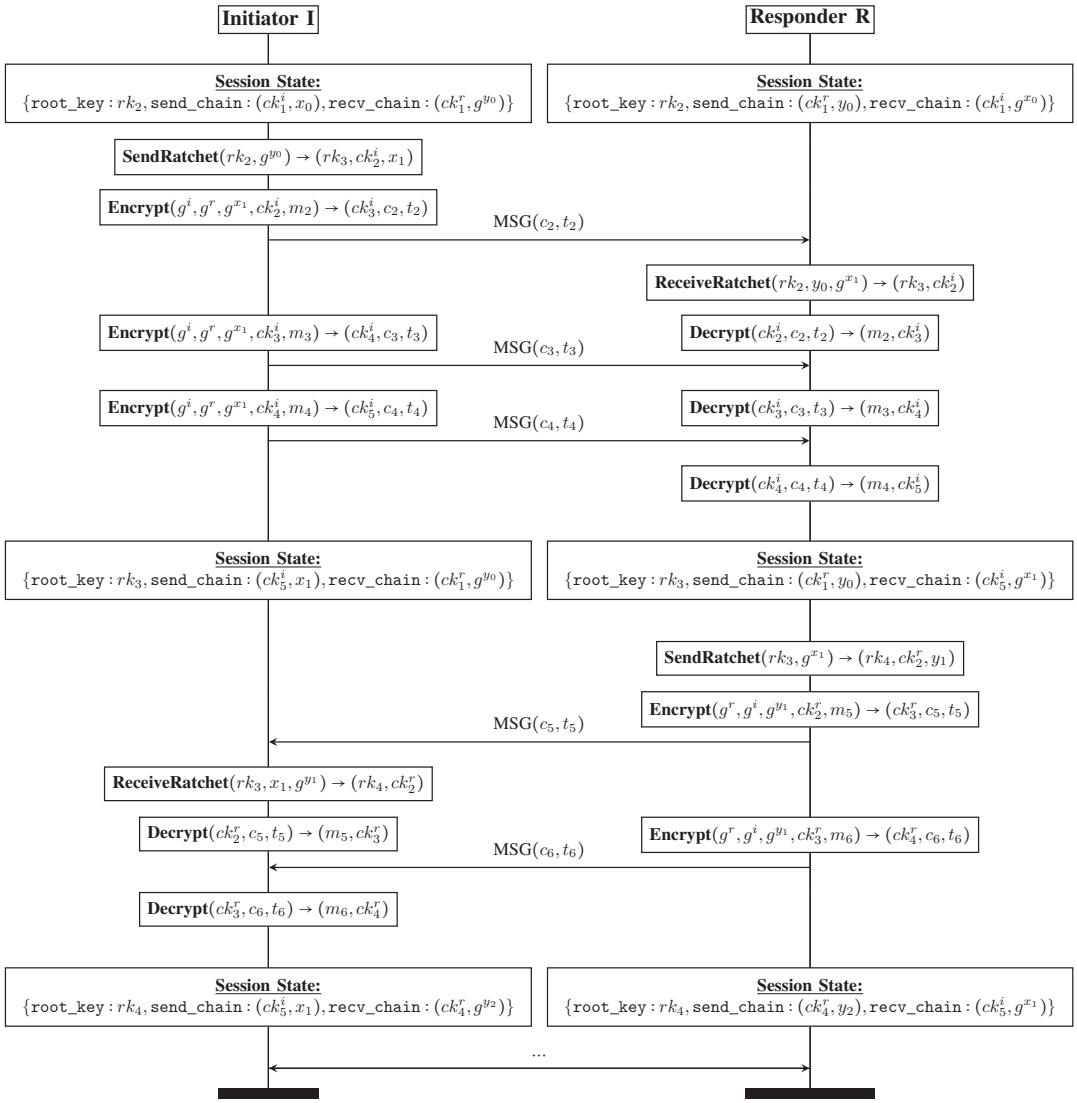

Fig. 16. Signal Protocol (secure messaging). Once the channel is set up, I and S can send flights of messages to each other in any order. The first message of each flight carries a fresh Diffie-Hellman key which is mixed into the root key. Subsequent messages in each flight advance the sender's chaining key. This protocol is sometimes a Double Ratchet protocol.

B. F* Functional Specification for Core Signal Protocol

```
module Spec.Signal.Core

open Lib.IntTypes
open Lib.ByteSequence
open Lib.Sequence
open Spec.Signal.Crypto
open Spec.Signal.Messages

#set-options "-z3rlimit␣50"
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 *)
    : 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 = ff @| dh1 @| dh2 @| dh3 in
  let shared_secret =
    match their_onetime_pub_key with
    | None → shared_secret
    | Some their_onetime_pub_key →
        let dh4 = dh our_onetime_priv_key
                    their_onetime_pub_key in
        shared_secret @| dh4 in
  hkdf1 shared_secret zz label_WhisperText

let respond
    (our_identity_priv_key: privkey) (* r *)
    (our_signed_priv_key: privkey) (* s *)
    (our_onetime_priv_key: option privkey) (* o, optional *)
    (their_identity_pub_key: pubkey) (* g^i *)
    (their_onetime_pub_key: pubkey) (* g^e *)
    : lbytes 32 = (* output: rk_0 *)

  let dh1 = dh our_signed_priv_key their_identity_pub_key in
  let dh2 = dh our_identity_priv_key their_onetime_pub_key in
  let dh3 = dh our_signed_priv_key their_onetime_pub_key in
  let shared_secret = ff @| dh2 @| dh1 @| dh3 in
  let shared_secret =
    match our_onetime_priv_key with
    | None → shared_secret
    | Some our_onetime_priv_key →
        let dh4 = dh our_onetime_priv_key
                    their_onetime_pub_key in
        shared_secret @| dh4 in
  hkdf1 shared_secret zz label_WhisperText

let ratchet
    (root_key:key) (* rk_j *)
    (our_ephemeral_priv_key:privkey) (* x_j *)
    (their_ephemeral_pub_key:pubkey) (* g^{y_j} *)
    : (key & key) = (* output: rk_{j+1}, ck_{j+1,0} *)

  let shared_secret = dh our_ephemeral_priv_key
                        their_ephemeral_pub_key in
  let keys = hkdf2 shared_secret
                   root_key label_WhisperRatchet in
  let root_key' = sub keys 0 32 in
  let chain_key = sub keys 32 32 in
  (root_key', chain_key)

let encrypt
    (our_identity_pub_key:pubkey) (* g^i or g^r *)
    (their_identity_pub_key:pubkey) (* g^r or g^i *)
    (chain_key:key) (* ck_j *)
    (our_ephemeral_pub_key:pubkey) (* g^x *)
    (prev_counter:size_nat) (* previous k *)
    (counter:size_nat) (* current j *)
    (plaintext:plain_bytes) (* message m_j *)
    : (cipher_bytes & lbytes 8 & key) = (* output: c_j, t_j, ck_{j+1} *)

  let msg_key = hmac chain_key zero in
  let chain_key' = hmac chain_key one in
  let keys = hkdf3 msg_key zz label_WhisperMessageKeys in
  let enc_key = sub keys 0 32 in
  let enc_iv = sub keys 32 16 in
  let mac_key = sub keys 64 32 in
  let ciphertext = aes_enc enc_key enc_iv plaintext in
  let whisper_msg =
      serialize_whisper_message our_ephemeral_pub_key
                prev_counter counter ciphertext in
  let tag8 =
      mac_whisper_msg mac_key their_identity_pub_key
                our_identity_pub_key whisper_msg in
  (ciphertext, tag8, chain_key')

let decrypt
    (our_identity_pub_key:pubkey) (* g^i or g^r *)
    (their_identity_pub_key:pubkey) (* g^r or g^i *)
    (chain_key:key) (* ck_j *)
    (their_ephemeral_pub_key:pubkey) (* g^y *)
    (prev_counter:size_nat) (* prev msg number k *)
    (counter:size_nat) (* current msg number j *)
    (ciphertext:cipher_bytes) (* ciphertext c_j *)
    (tag8:lbytes 8) (* tag t_j *)
    : option (plain_bytes & key) = (* outputs: m_j, ck_{j+1} *)

  let len = length ciphertext in
  let ciphertext = to_lseq ciphertext in
  let msg_key = hmac chain_key zero in
  let chain_key' = hmac chain_key one in
  let keys = hkdf3 msg_key zz label_WhisperMessageKeys in
  let enc_key = sub keys 0 32 in
  let enc_iv = sub keys 32 16 in
  let mac_key = sub keys 64 32 in
  let whisper_msg =
      serialize_whisper_message their_ephemeral_pub_key
                prev_counter counter ciphertext in
  let exp_tag8 =
      mac_whisper_msg mac_key our_identity_pub_key
                their_identity_pub_key whisper_msg in
  let plain = aes_dec enc_key enc_iv ciphertext in
  if equal_bytes tag8 exp_tag8
  then Some (plain,chain_key')
  else None
```

This snippet is pure F* code. It relies on the specific cryptographic constructions of the Signal protocol (in Spec.Signal.Crypto), such as hmac and hkdf{1,2,3}, as well as the message serialization primitives (in Spec.Signal.Messages). See §V. Other helpers include: zz (32 zero bytes), ff (32 0xff bytes), the label_* string constants used in Signal and @| (byte sequence concatenation).

C. Low* Implementation for the Ratchet function from Signal Protocol

```
val ratchet:
    output_keys: uint8_p { length output_keys = 64 }
  → root_key:key_p
  → our_ephemeral_priv_key:privkey_p
  → their_ephemeral_pub_key:pubkey_p →
  Stack unit
    (requires (λ h0 → live_pointers h0
                        [output_keys; root_key;
                         our_ephemeral_priv_key;
                         their_ephemeral_pub_key]
                    ∧ disjoint_from output_keys
                        [root_key; our_ephemeral_priv_key;
                         their_ephemeral_pub_key]))
    (ensures (λ h0 _ h1 → modifies_only output_keys h0 h1
                    ∧ let (root_key',chain_key') =
                          Spec.Signal.Core.ratchet
                            h0.[| root_key |]
                            h0.[| our_ephemeral_priv_key |]
                            h0.[| their_ephemeral_pub_key |]
                      h1.[| output_keys |] == root_key' @| chain_key'))
```

```
let ratchet
    output_keys root_key
    our_ephemeral_priv_key their_ephemeral_pub_key
=
  push_frame();
  let shared_secret = create 32ul (u8 0) in
  dh shared_secret
     our_ephemeral_priv_key their_ephemeral_pub_key;
  hkdf2 output_keys
        shared_secret 32ul root_key
        const_label_WhisperRatchet 14ul;
  pop_frame()
```

On the left is the *type declaration* (i.e. prototype) of our Low* implemetation of the ratchet function. The function takes four arguments: the first argument contains an output buffer (i.e. a mutable array) called output_keys containing two concatenated keys of 32 bytes each; the rest of the arguments are three input buffers. The function has no return value, and is declared as having a Stack effect, which means that it only allocates memory on the stack.

The pre-condition of the function, stated in the requires clause, requires that all the input and output buffers must be live in the heap when the function is called, and all input buffers must be disjoint from the output buffer. The post-condition of ratchet, stated in the ensures clause, guarantees that the function only modifies the output buffer, and that the output value of output_keys in the heap when the function returns matches the specification of ratchet in Spec.Signal.Core.

Hence, this type declaration provides a full memory safety and functional correctness specification for ratchet. Moreover, all buffers are declared to contain secret bytes (uint8), so the type declaration also requires that the code for ratchet be secret independent, or "constant-time", with respect to the (potentially secret) contents of these buffers.

The verified Low* code for ratchet is shown on the right. It closely matches the F* specification of ratchet in Spec.Signal.Core. The main difference is that the Low* code needs to allocate a temporary buffer to hold the shared_secret: so the function calls push_frame to create a new stack frame, create to allocate the buffer, and pop_frame when exiting the function.

D. Complete translation rules from λow* to Cb

IfThenElse

$$\frac{G;V \vdash e_1 : \mathsf{bool} \Rrightarrow \hat{e}_1 : \mathsf{bool} \dashv V' \quad G;V' \vdash e_2 : \tau \Rrightarrow \hat{e}_2 : \hat{\tau} \dashv V'' \quad G;V'' \vdash e_3 : \tau \Rrightarrow \hat{e}_3 : \hat{\tau} \dashv V'''}{G;V \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau \Rrightarrow \mathsf{if}\ \hat{e}_1\ \mathsf{then}\ \hat{e}_2\ \mathsf{else}\ \hat{e}_3 : \hat{\tau} \dashv V'''}$$

BufRead

$$\frac{G;V \vdash e_1 : \mathsf{buf}\ \tau \Rrightarrow \hat{e}_1 : \mathsf{pointer} \dashv V' \quad G;V' \vdash e_2 : \mathsf{int32} \Rrightarrow \hat{e}_2 : \mathsf{int32} \dashv V'' \quad \mathsf{size}(\tau) = n}{G;V \vdash \mathsf{readbuf}\ e_1\ e_2 : \tau \Rrightarrow \mathsf{read}_n(\hat{e}_1 + n \times \hat{e}_2) : \hat{\tau} \dashv V''}$$

BufSub

$$\frac{G;V \vdash e_1 : \mathsf{buf}\ \tau \Rrightarrow \hat{e}_1 : \mathsf{pointer} \dashv V' \quad G;V' \vdash e_2 : \mathsf{int32} \Rrightarrow \hat{e}_2 : \mathsf{int32} \dashv V'' \quad \mathsf{size}(\tau) = n}{G;V \vdash \mathsf{subbuf}\ e_1\ e_2 : \mathsf{buf}\ \tau \Rrightarrow \hat{e}_1 + n \times \hat{e}_2 : \mathsf{pointer} \dashv V''}$$

Field

$$\frac{G;V \vdash e : \mathsf{buf}\ \tau \Rrightarrow \hat{e} : \mathsf{pointer} \dashv V' \quad \mathsf{offset}(\tau, f) = k \qquad \mathsf{size}(\tau_f) = n}{G;V \vdash (\mathsf{readbuf}\ e\ 0).f : \tau_f \Rrightarrow \mathsf{read}_n(\hat{e} + k) : \hat{\tau}_f \dashv V'}$$

PointerAdd

$$\frac{G;V \vdash e : \mathsf{buf}\ \tau \Rrightarrow \hat{e} : \mathsf{pointer} \dashv V'}{G;V \vdash e \oplus n : \mathsf{buf}\ \tau \Rrightarrow \hat{e} + n : \mathsf{pointer} \dashv V'}$$

FunCall

$$\frac{G;V \vdash e : \tau_1 \Rrightarrow \hat{e} : \hat{\tau}_1 \dashv V'}{G;V \vdash d\ e : \tau_2 \Rrightarrow d\ \hat{e} : \hat{\tau}_2 \dashv V'}$$

Unit

$$\frac{}{G;V \vdash () : \mathsf{unit} \Rrightarrow () : \mathsf{unit} \dashv V}$$

Constant

$$\frac{}{G;V \vdash k : \tau \Rrightarrow k : \hat{\tau} \dashv V}$$

Global

$$\frac{g \in G}{G;V \vdash g : \tau \Rrightarrow g : \hat{\tau} \dashv V}$$

ForLoop

$$\frac{G;(i \mapsto \ell, \mathsf{int32}) \cdot V \vdash e : \mathsf{unit} \Rrightarrow \hat{e} : \mathsf{unit} \dashv V' \qquad \ell\ \text{fresh}}{G;V \vdash \mathsf{for}\ i \in [0;n)\ e : \mathsf{unit} \Rrightarrow \mathsf{for}\ \ell \in [0;n)\ \hat{e} : \mathsf{unit} \dashv V'}$$

Fig. 17. Translating from λow* to Cb (remaining rules). Some notes: Field: the type τ_f can only be a non-struct type per our invariant (III-B, 1).