Verified Low-Level Programming Embedded in F*

JONATHAN PROTZENKO, Microsoft Research, USA
JEAN-KARIM ZINZINDOHOUÉ, INRIA Paris, France
ASEEM RASTOGI, Microsoft Research, USA
TAHINA RAMANANANDRO, Microsoft Research, USA
PENG WANG, MIT CSAIL, USA
SANTIAGO ZANELLA-BÉGUELIN, Microsoft Research, USA
ANTOINE DELIGNAT-LAVAUD, Microsoft Research, USA
CĂTĂLIN HRIŢCU, INRIA Paris, France
KARTHIKEYAN BHARGAVAN, INRIA Paris, France
CÉDRIC FOURNET, Microsoft Research, USA
NIKHIL SWAMY, Microsoft Research, USA

We present Low*, a language for low-level programming and verification, and its application to high-assurance optimized cryptographic libraries. Low* is a shallow embedding of a small, sequential, well-behaved subset of C in F*, a dependently-typed variant of ML aimed at program verification. Departing from ML, Low* does not involve any garbage collection or implicit heap allocation; instead, it has a structured memory model à la CompCert, and it provides the control required for writing efficient low-level security-critical code.

By virtue of typing, any Low* program is memory safe. In addition, the programmer can make full use of the verification power of F* to write high-level specifications and verify the functional correctness of Low* code using a combination of SMT automation and sophisticated manual proofs. At extraction time, specifications and proofs are erased, and the remaining code enjoys a predictable translation to C. We prove that this translation preserves semantics and side-channel resistance.

We provide a new compiler back-end from Low* to C and, to evaluate our approach, we implement and verify various cryptographic algorithms, constructions, and tools for a total of about 28,000 lines of code, specification and proof. We show that our Low* code delivers performance competitive with existing (unverified) C cryptographic libraries, suggesting our approach may be applicable to larger-scale low-level software.

CCS Concepts: • **Theory of computation → Hoare logic**; *Type theory*; • **Software and its engineering → Correctness**; **Software verification**; **Source code generation**; *Functional languages*; *Semantics*; *Compilers*;

Additional Key Words and Phrases: verified compilation, low-level programming, verified cryptography

ACM Reference Format:
Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified Low-Level Programming Embedded in F*. *Proc. ACM Program. Lang.* 1, ICFP, Article 17 (September 2017), 29 pages.
https://doi.org/10.1145/3110261

2475-1421/2017/9-ART17
https://doi.org/10.1145/3110261

1 INTRODUCTION

In the pursuit of high performance, cryptographic software widely deployed throughout the internet is still often subject to dangerous attacks [Int 2017; Dou 2017; Use 2017; Afek and Sharabani 2007; AlFardan and Paterson 2013; Bhargavan et al. 2014; Bhargavan and Leurent 2016; Böck 2016; Böck et al. 2016; Dobrovitski 2003; Duong and Rizzo 2011; Heartbleed 2014; Möller et al. 2014; Pincus and Baker 2004; Rizzo and Duong 2012; Smyth and Pironti 2014; Somorovsky 2016; Stevens et al. 2016; Święcki 2016; Szekeres et al. 2013; Wagner and Schneier 1996]. Recognizing a clear need, the programming language, verification, and applied cryptography communities are devoting significant efforts to develop implementations proven secure by construction against broad classes of these attacks.

Focusing on low-level attacks caused by violations of memory safety, several researchers have used high-level, type-safe programming languages to implement standard protocols such as Transport Layer Security (TLS). For example, Kaloper-Meršinjak et al. [2015] provide nqsbTLS, an implementation of TLS in OCaml, which by virtue of its type and memory safety is impervious to attacks (like Heartbleed [2014]) that exploit buffer overflows. Bhargavan et al. [2014] program miTLS in F#, also enjoying type and memory safety, but go further using a refinement type system to prove various higher-level security properties of the protocol. While this approach is attractive for its simplicity, to get acceptable performance, both nqsbTLS and miTLS link with fast, unsafe implementations of complex cryptographic algorithms, such as those provided by nocrypto [2017], an implementation that mixes C and OCaml, and libcrypto, a component of the widely used OpenSSL library [2017]. In the worst case, linking with vulnerable C code can void all the guarantees of the high-level code.

In this paper, we aim to bridge the gap between high-level, safe-by-construction code, optimized for clarity and ease of verification, and low-level code exerting fine control over data representations and memory layout in order to achieve better performance. To this end, we introduce Low*, a domain-specific language for verified, efficient, low-level programming embedded within F* [Swamy et al. 2016], an ML-like language with dependent types designed for program verification. We use F* to prove functional correctness and security properties of high-level code. Where efficiency is paramount, we drop into its C-like Low* subset while still relying on the verification capabilities of F* to prove our code is memory safe, functionally correct, and secure.

We have applied Low* to program and verify a range of sequential low-level programs, including libraries for multi-precision arithmetic and buffers, and various cryptographic algorithms, constructions, and protocols built on top of them. Our experiments indicate that compiled Low* code yields performance on par with existing C code. This code can be used on its own, or used within existing software through the C ABI. In particular, our C code may be linked to F* programs compiled to OCaml, providing large speed-ups via its foreign-function interface (FFI) without compromising safety or security.

An Embedded DSL, Compiled Natively

Low* programs are a subset of F* programs: the programmer writes Low* code using regular F* syntax, against a library we provide that models a lower-level view of memory, akin to the structured memory layout of a well-defined C program (this is similar to the structured memory model of CompCert [Leroy 2009; Leroy et al. 2012], rather than the "big array of bytes" model systems programmers sometimes use). Low* programs interoperate naturally with other F* programs, and precise specifications of Low* and F* code are intermingled when proving properties of their combination. As usual in F*, programs are type-checked and compiled to OCaml for execution, after erasing their computationally irrelevant parts, such as proofs and specifications, using a

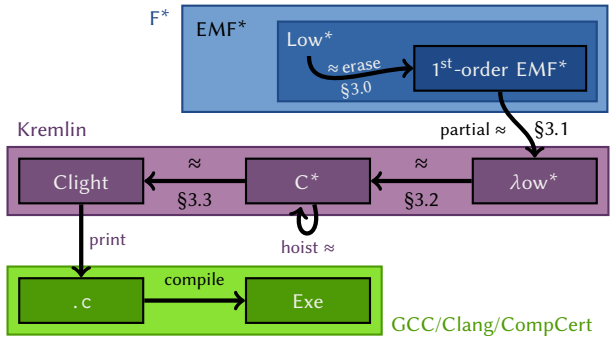

Fig. 1. Low* embedded in F*, compiled to C, with soundness and security guarantees (details in §3)

process similar to Coq's extraction mechanism [Letouzey 2002]. In particular, our memory-model library compiles to a simple, heap-based OCaml implementation.

Importantly, Low* programs have a second, equivalent but more efficient semantics via compilation to C, with the predictable performance that comes with a native implementation of their lower-level memory model. This compilation is implemented by KreMLin, a new compiler from the Low* subset of F* to C. Figure 1 illustrates the high-level design of Low* and its compilation to native code. Our main contributions are as follows:

Libraries for low-level programming within F (§2).* At its core, F* is a purely functional language to which effects like state are added programmatically using monads. In this work, we instantiate the state monad of F* to use a CompCert-like structured memory model that separates the stack and the heap, supporting bulk allocation and deallocation on the stack, and allocating and freeing individual blocks on the heap. Both the heap and the stack are further divided into disjoint logical regions, which enables us to manage the separation properties necessary for modular, stateful verification. On top of this, we program a library of C-style arrays and structs passed by reference, with support for pointer arithmetic and pointers to the interior of an array or a struct. By virtue of F* typing, our libraries and all their well-typed clients are guaranteed to be memory safe, e.g., they never access out-of-bounds or deallocated memory.

Designing Low, a subset of F* easily compiled to C.* We intend to give Low* programmers precise control over the performance profile of the generated C code. As much as possible, we aim for the programmer to control even the syntactic structure of the C code, to facilitate its review by security experts unfamiliar with F*. As such, to a first approximation, Low* programs are F* programs well-typed in the state monad described above, which, after all their computationally irrelevant (ghost) parts have been erased, must meet several restrictions, as follows: the code (1) must be first order, to prevent the need to allocate closures in C; (2) must make any heap allocation explicit; (3) must not use any recursive datatype, since these would have to be compiled using additional indirections to C structs; and (4) must be monomorphic, since C does not support polymorphism directly. Importantly, Low* heavily leverages F*'s capabilities for partial evaluation, hence allowing the programmer to write high-level, reusable code that is normalized via meta-programming into the Low* subset before the restrictions are enforced. We emphasize that these restrictions apply only to computationally relevant code—proofs and specifications are free to use arbitrary higher-order, dependently typed F*, and very often they do.

A formal translation from Low to CompCert Clight (§3).* Justifying its dual interpretation as a subset of F* and a subset of C, we provide a formal model of Low*, called λow*, give a translation

from λow* to Clight [Blazy and Leroy 2009] and show that it preserves trace equivalence with respect to the original F* semantics. In addition to ensuring that the functional behavior of a program is preserved, our trace equivalence also guarantees that the compiler does not accidentally introduce side-channels due to memory access patterns (as would be the case without the restrictions above) at least until it reaches Clight, a useful sanity check for cryptographic code. Our theorems cover the translation of standalone λow* programs to C, proving that execution in C preserves the original F* semantics of the λow* program.

KreMLin, a compiler from Low to C (§4).* Our formal development guides the implementation of KreMLin, a new tool that emits C code from Low*. KreMLin is designed to emit well-formatted, idiomatic C code suitable for manual review. The resulting C programs can be compiled with CompCert for greatest assurance, and with mainstream C compilers, including GCC and Clang, for greatest performance. We have used KreMLin to extract to C the 20,000+ lines of Low* code we have written so far. After compilation, our verified standalone C libraries can be integrated within larger programs using standard means.

An empirical evaluation (§5). We present a few developments of efficient, verified, interoperable cryptographic libraries programmed in Low*.

(1) We provide HACL*, a "high-assurance crypto library" implementing and proving (in ~6,000 lines of Low*) several cryptographic algorithms, including the Poly1305 MAC [Bernstein 2005], the ChaCha20 cipher [Nir and Langley 2015], and multiplication on the Curve25519 elliptic curve [Bernstein 2006]. We package these algorithms to provide the popular NaCl API [Bernstein et al. 2012], yielding the first performant implementation of NaCl verified for memory safety and side-channel resistance, along with functional correctness proofs for its core components, including a verified bignum library customized for safe, fast cryptographic use (§5.1). Using this API, we build new standalone applications such as *PneuTube*, a new secure, asynchronous, file transfer application whose performance compares favorably with widely used, existing utilities like scp.

(2) Emphasizing the applicability of Low* for high-level, cryptographic security proofs on low-level code, we briefly describe its use in programming and proving (in ~14,000 lines of Low*) the Authenticated Encryption with Associated Data (AEAD) construction at the heart of the record layer of the new TLS 1.3 Internet Standard. We prove memory safety, functional correctness, and cryptographic security for its main ciphersuites, relying, where available, on verified implementations of these ciphersuites provided by HACL*. The C code extracted from our verified implementation is easily integrated within other applications, including, for example, an implementation in F* of TLS separately verified and compiled to OCaml (through OCaml's FFI).

Trusted computing base. To date, we have focused on designing and evaluating our methodology of programming and verifying low-level code shallowly embedded within a high-level programming language and proof assistant. We have yet to invest effort in minimizing the trusted computing base of our work, an effort we plan to expend now that we have evidence that our methodology is worthwhile. Currently, the trusted computing base of our verified libraries includes the implementation of the F* typechecker and the Z3 SMT solver [de Moura and Bjørner 2008]. Additionally, we trust the manual proofs of the metatheory relating the semantics of λow* to CompCert Clight. The KreMLin tool is informed by this metatheory, but is currently implemented in unverified OCaml, and is also trusted. Finally, we inherit the trust assumptions of the C compiler used to compile the code extracted from Low*.

Supplementary materials. First, we provide, in the appendix, the hand proofs of the theorems described in §3. The present paper is focused on the metatheory and tools; we also authored a companion paper [Bhargavan et al. 2017] that describes the cryptographic model we used for the

```
let chacha20
  (len: uint32{len ≤ blocklen})
  (output: bytes{len = output.length})
  (key: keyBytes)
  (nonce: nonceBytes{disjoint [output; key; nonce]})
  (counter: uint32) : Stack unit
  (requires (λ m0 → output ∈ m0 ∧ key ∈ m0 ∧ nonce ∈ m0))
  (ensures (λ m0 _m1 → modifies1 output m0 m1 ∧
      m1.[output] ==
      Seq.prefix len (Spec.chacha20 m0.[key] m0.[nonce]) counter))) =
    push_frame ();
    let state = Buffer.create 0ul 32ul in
    let block = Buffer.sub state 16ul 16ul in
    chacha20_init block key nonce counter;
    chacha20_update output state len;
    pop_frame ()
```

```
void chacha20 (
  uint32_t len,
  uint8_t *output,
  uint8_t *key,
  uint8_t *nonce,
  uint32_t counter)


{
  uint32_t state[32] = { 0 };
  uint32_t *block = state + 16;
  chacha20_init(block, key, nonce, counter);
  chacha20_update(output, state, len);
}
```

Fig. 2. A snippet from ChaCha20 in Low* (left) and its C compilation (right)

record layer of TLS 1.3. Finally, we have an ongoing submission of a paper focused on our HACL* library [Zinzindohoué et al. 2017], where we describe in greater detail our proof techniques for reusing the bignum formalization across different algorithms and implementations, and provide a substantial performance evaluation.

We also offer numerous software artifacts. Our tool KreMLin [Protzenko 2017] is actively developed on GitHub, and so is HACL* [Zinzindohoué et al. 2017]. Most of the Low* libraries live in the F* repository, also on GitHub. The integration of HACL* within miTLS is also available on GitHub. For convenience, we offer a regularly-updated Docker image of Project Everest [Microsoft Research and INRIA 2016], which bundles together F*, miTLS, HACL*, KreMLin. One may fetch it via `docker pull projecteverest/everest`. The Docker image contains a `README.md` with an overview of the proofs and the code.

2 A LOW* TUTORIAL

At the core of Low* is a library for programming with structures and arrays manually allocated on the stack or the heap (§2.2). Memory safety demands reasoning about the extents and liveness of these objects, while functional correctness and security may require reasoning about their contents. Our library provides specifications to allow client code to be proven safe, correct and secure, while KreMLin compiles such verified client code to C.

We illustrate the design of Low* using several examples from our codebase. We show the ChaCha20 stream cipher [Nir and Langley 2015], focusing on memory safety (§2.1), and the Poly1305 MAC [Bernstein 2005], focusing on functional correctness. (§2.3). Going beyond functional correctness, we explain how we prove a combination of ChaCha20 and Poly1305 cryptographically secure (§2.4). Throughout, we point out key benefits of our approach, notably our use of dependently typed metaprogramming to work at a relatively high-level of abstraction at little performance cost.

2.1 A First Example: the ChaCha20 Stream Cipher

Figure 2 shows code snippets for the core function of ChaCha20 [Nir and Langley 2015], a modern stream cipher widely used for fast symmetric encryption. This function computes a block of pseudo-random bytes, usable for encryption, for example, by XORing them with a plaintext message. On the left is our Low* code; on the right its compilation to C. The function takes as arguments an

output length and buffer, and some input key, nonce, and counter. It allocates 32 words of auxiliary, contiguous state on the stack; then it calls a function to initialize the cipher block from the inputs (passing an interior pointer to the state); and finally it calls another function that computes a cipher block and copies its first len bytes to the output buffer.

Aside from the erased specifications at lines 7–10, the C code is in one-to-one correspondence with its Low* counterpart. These specifications capture the safe memory usage of chacha20. (Their syntax is explained next, in §2.2.) For each argument passed by reference, and for the auxiliary state, they keep track of their liveness and size. They also capture its correctness, by describing the final state of the output buffer using a pure function.

Lines 2–3 use type refinements to require that the len argument equals the length of the output buffer and it does not exceed the block size. (Violation of these conditions would lead to a buffer overrun in the call to chacha20_update.) Similarly, types keyBytes and nonceBytes specify pointers to fixed-sized buffers of bytes. The return type Stack unit on line 6 says that chacha20 returns nothing and may allocate on the stack, but not on the heap (in particular, it has no memory leak). On the next line, the pre-condition requires that all arguments passed by reference be live. On lines 8–10, the post-condition first ensures that the function modifies at most the contents of output (and, implicitly, that all buffers remain live). We further explain this specification in the next subsection. The rest of the post-condition specifies functional correctness: the output buffer must contain a sequence of bytes equal to the first len bytes of the cipher specified by function Spec.chacha20 for the input values of key, nonce, and counter.

As usual for symmetric ciphers, RFC 7539 specifies chacha20 as imperative pseudocode, and does not further discuss its mathematical properties. We implement this pseudocode as a series of pure functions in F*, which can be extracted to OCaml and tested for conformance with the RFC test vectors. Functions such as Spec.chacha20 then serve as logical specifications for verifying our stateful implementation. In particular, the last postcondition of chacha20 ensures that its result is determined by its inputs. We describe more sophisticated functional correctness proofs for Poly1305 in §2.3.

2.2 Low*: An Embedded DSL for Low-Level Code

As in ML, by default F* does not provide an explicit means to reclaim memory or to allocate memory on the stack, nor does it provide support for pointing to the interior of arrays. Next, we sketch the design of a new F* library that provides a structured memory model suitable for program verification, while supporting low-level features like explicit freeing, stack allocation, and interior pointers. In subsequent sections, we describe how programs type-checked against this library can be compiled safely to C. First, however, we begin with some background on F*.

Background: F* is a dependently typed language with support for user-defined monadic effects. Its types separate computations from values, giving the former *computation types* of the form M $t_1 \ldots t_n$ where M is an effect label and $t_1 \ldots t_n$ are *value types*. For example, Stack unit (...) (...) on lines 7–8 of Figure 2 is an instance of a computation type, while types like unit are value types. There are two distinguished computation types: Tot t is the type of a total computation returning a t-typed value; Ghost t, a computationally irrelevant computation returning a t-typed value. Ghost computations are useful for specifications and proofs but are erased when extracting to OCaml or C.

To add state to F*, one defines a state monad represented (as usual) as a function from some initial memory m_0:s to a pair of a result r:a and a final memory m_1:s, for some type of memory s. Stateful computations are specified using the computation type:

```
ST (a:Type) (pre: s → Type) (post: s → a → s → Type)
```

Here, ST is a computation type constructor applied to three arguments: a result type a; a pre-condition predicate on the initial memory, pre; and a post-condition predicate relating the initial memory, result and final memory. We generally annotate the pre-condition with the keyword requires and the post-condition with ensures for better readability. A computation e of type ST a (requires pre) (ensures post), when run in an initial memory m_0:s satisfying pre m, produces a result r:a and final memory m_1:s satisfying post m_0 r m_1, unless it diverges.[1] F* uses an SMT solver to discharge the verification conditions it computes when type-checking a program.

Hyper-stacks: A region-based memory model for Low.* For Low*, we instantiate the type s in the state monad to HyperStack.mem (which we refer to as just "hyper-stack"), a new region-based memory model [Tofte and Talpin 1997] covering both stack and heap. Hyper-stacks are a generalization of hyper-heaps, a memory model proposed previously for F* [Swamy et al. 2016], designed to provide lightweight support for separation and framing for stateful verification. Hyper-stacks augment hyper-heaps with a shape invariant to indicate that the lifetime of a certain set of regions follow a specific stack-like discipline. We sketch the F* signature of hyper-stacks next.

A logical specification of memory. Hyper-stacks partition memory into a set of regions. Each region is identified by an rid and regions are classified as either stack or heap regions, according to the predicate is_stack_region—we use the type abbreviation sid for stack regions and hid for heap regions. A distinguished stack region, root, outlives all other stack regions. The snippet below is the corresponding F* code.

```
type rid
val is_stack_region: rid → Tot bool
type sid = r:rid{is_stack_region r}
type hid = r:rid{¬ (is_stack_region r)}
val root: sid
```

Next, we show the (partial) signature of mem, our model of the entire memory, which is equipped with a select/update theory [McCarthy 1962] for typed references ref a. Additionally, we have a function to refer to the region_of a reference, and a relation r ∈ m to indicate that a reference is live in a given memory.

```
type mem
type ref : Type → Type
val region_of: ref a → Ghost rid
val _ ∈ _ : ref a → mem → Tot Type (* a ref is contained in a mem *)
val _ [_] : mem → ref a → Ghost a (* selecting a ref *)
val _ [_] ← _ : mem → ref a → a → Ghost mem (* updating a ref *)
val rref r a = x:ref a {region_of x = r} (* abbrev. for a ref in region r *)
```

Heap regions. By defining the ST monad over the mem type, we can program stateful primitives for creating new heap regions, and allocating, reading, writing and freeing references in those regions—we show some of their signatures below. Assuming an infinite amount of memory, alloc's pre-condition is trivial while its post-condition indicates that it returns a fresh reference in region r initialized appropriately. Freeing and dereferencing (!) require their argument to be present in the current memory, eliminating double-free and use-after-free bugs.

```
val alloc: r:hid → init:a → ST (rref r a) (ensures (λ m0 x m1 → x ∉ m0 ∧ x ∈ m1 ∧ m1 = (m0[x]← init)))
val free: r:hid → x:rref r a → ST unit (requires (λ m → x ∈ m)) (ensures (λ m0 _ m1 → x ∉ m1 ∧ ∀y≠x. m0[y] = m1[y]))
val (!): x:ref a → ST a (requires (λ m → x ∈ m)) (ensures (λ m0 y m1 → m0 = m1 ∧ y = m1[x]))
```

[1] F* recently gained support for proving stateful computations terminating. We have begun making use of this feature to prove our code terminating, wherever appropriate, but make no further mention of this.

Since we support freeing individual references within a region, our model of regions could seem similar to Berger et al. [2002]'s *reaps*. However, at present, we do not support freeing heap objects *en masse* by deleting heap regions; indeed, this would require using a special memory allocator. Instead, for us heap regions serve only to *logically* partition the heap in support of separation and modular verification, as is already the case for hyper-heaps [Swamy et al. 2016], and heap region creation is currently compiled to a no-op by KreMLin.

Stack regions, which we will henceforth call *stack frames*, serve not just as a reasoning device, but provide the efficient C stack-based memory management mechanism. KreMLin maps stack frame creation and destruction directly to the C calling convention and lexical scope. To model this, we extend the signature of mem to include a tip region representing the currently active stack frame, ghost operations to push and pop frames on the stack of an explicitly threaded memory, and their effectful analogs, push_frame and pop_frame that modify the current memory. In chacha20 in Fig. 2, the push_frame and pop_frame correspond precisely to the braces in the C program that enclose a function body's scope. We also provide a derived combinator, with_frame, which combines push_frame and pop_frame into a single, well-scoped operation. Programmers are encouraged to use the with_frame combinator, but, when more convenient for verification, may also use push_frame and pop_frame directly. KreMLin ensures that all uses of push_frame and pop_frame are well-scoped. Finally, we show the signature of salloc which allocates a reference in the current tip stack frame.

```
val tip: mem → Ghost sid
val push: mem → Ghost mem
val pop: m:mem{tip m ≠ root} → Ghost mem
val push_frame: unit → ST unit (ensures (λ m0 () m1 → m1 = push m0))
val pop_frame: unit → ST unit (requires (λ m → tip m ≠ root)) (ensures (λ m0 () m1 → m1 = pop m0))
val salloc: init:a → ST (ref a) (ensures (λ m0 x m1 → x ∉ m0 ∧ x ∈ m1 ∧ region_of x = tip m1 ∧
                                                    tip m0 = tip m1 ∧ m1 = (m0[x] ← init)))
```

The Stack effect. The specification of chacha20 claims that it uses only stack allocation and has no memory leaks, using the Stack computation type. This is straightforward to define in terms of ST, as shown below.

```
effect Stack a pre post = ST a (requires pre)
                               (ensures (λ m0 x m1 → post m0 x m1 ∧ tip m0 = tip m1 ∧ (∀ r. r ∈ m1 ⟺ r ∈ m0)))
```

Stack computations are ST computations that leave the stack tip unchanged (i.e., they pop all frames they may have pushed) and yield a final memory with the same domain as the initial memory. This ensures that Low* code with Stack effect has explicitly deallocated all heap allocated references before returning, ruling out memory leaks. As such, we expect all externally callable Low* functions to have Stack effect. Other code can safely pass pointers to objects allocated in their heaps into Low* functions with Stack effect since the definition of Stack forbids the Low* code from freeing these references.

Modeling arrays. Hyper-stacks separate heap and stack memory, but each region of memory still only supports abstract, ML-style references. A crucial element of low-level programming is control over the specific layout of objects, especially for arrays and structs. We describe first our modeling of arrays by implementing an abstract type for buffers in Low*, using just the references provided by hyper-stacks. Relying on its abstraction, KreMLin compiles our buffers to native C arrays.

The type 'buffer a' below is a single-constructor inductive type with 4 arguments. Its main content argument holds a reference to a seq a, a purely functional sequence of a's whose length is determined by the first argument max_length. The refinement type b:buffer uint32{length b = n} is translated to a C declaration uint32_t b[n] by KreMLin and, relying on C pointer decay, further referred to via uint32_t *.

```
abstract type buffer a =
  | MkBuffer: max_length:uint32
    → content:ref (s:seq a{Seq.length s = max_length})
    → idx:uint32
    → length:uint32 {idx + length ≤ max_length} → buffer a
```

The last two arguments of a buffer are there to support creating smaller sub-buffers from a larger buffer, via the Buffer.sub operation below. A call to 'Buffer.sub b i l' returning b′ is compiled to C pointer arithmetic b + i (as seen in Figure 2 line 13 in chacha20). To accurately model this, the content field is shared between b and b′, but idx and length differ, to indicate that the sub-buffer b′ covers only a sub-range of the original buffer b. The sub operation has computation type Tot, meaning that it does not read or modify the state. The refinement on the result b′ indicates its length and, using the includes relation, records that b and b′ are aliased.

```
val sub: b:buffer a → i:uint32 → len:uint32{i + len ≤ b.length} → Tot (b':buffer a{b'.length = len ∧ b includes b'})
```

We also provide statically bounds-checked operations for indexing and updating buffers. The signature of the index function below requires the buffer to be live and the index location to be within bounds. Its postcondition ensures that the memory is unchanged and describes what is returned in terms of the logical model of a buffer as a sequence.

```
let get (m:mem) (b:buffer a) (i:uint32{i < b.length}) : Ghost a = Seq.index (m[b.content]) (b.idx + i)
val index: b:buffer a → i:uint32{i < b.length} → Stack a
  (requires (λ m → b.content ∈ m))
  (ensures (λ m0 z m1 → m1 = m0 ∧ z = get m1 b i))
```

All lengths and indices are 32-bit machine integers, and refer to the number of elements in the buffer, not the number of bytes the buffer occupies. This currently prevents addressing very large buffers on 64-bit platforms. (To this end, we may parameterize our development over a C data model, wherein indices for buffers would reflect the underlying (proper) ptrdiff_t type.)

Similarly, memory allocation remains platform-specific. It may cause a (fatal) error as it runs out of memory. More technically, the type of create may not suffice to prevent pointer-arithmetic overflow; if the element size is greater than a byte, and if the number of elements is 2^{32}, then the argument passed to malloc will overflow on a platform where sizeof size_t == 4. To prevent such cases, KreMLin inserts defensive dynamic checks (which typically end up eliminated by the C compiler since our stack-allocated buffer lengths are compile-time constants). In the future, we may statically prevent it by mirroring the C sizeof operator at the F* level, and requiring that for each Buffer.create operation, the resulting allocation size, in bytes, is no greater than what size_t can hold.

Modeling structs. Generalizing 'buffer t' (abstractly, a reference to a finite map from natural numbers to t), we model C-style structs as an abstract reference to a 'struct key value', that is, a map from keys k:key to values whose type 'value k' depends on the key. For example, we represent the type of a colored point as follows, using a struct with two fields X and Y for coordinates and one field Color, itself a nested struct of RGB values.

```
type color_fields = R | G | B
type color = struct color_fields (λ R | G | B → uint32)
type colored_point_fields = X | Y | Color
type colored_point = struct colored_point_fields (λ X | Y → int32 | Color → color)
```

C structs are flatly allocated; the declaration above models a contiguous memory block that holds 20 bytes or more, depending on alignment constraints. As such, we cannot directly perform pointer arithmetic within that block; rather, we navigate it by referring to fields. To this end, our library of structs provides an interface to manipulate pointers to these C-like structs, including pointers

that follow a path of fields throughout nested structs. The main type provided by our library is the indexed type ptr shown below, encapsulating a base reference content: ref from and a path p of fields leading to a value of type to.

```
abstract type ptr: Type → Type = Ptr: #from:Type → content: ref from → #to: Type → p: path from to → ptr to
```

When allocating a struct on the stack, the caller provides a 'struct k v' literal and obtains a 'ptr (struct k v)', a pointer to a struct literal in the current stack frame (a Ptr with an empty path).

The extend operator below supports extending the access path associated with a 'ptr (struct k v)' to obtain a pointer to one of its fields.

```
val extend: #key: eqtype → #value: (key → Tot Type) → p: ptr (struct key value) → fd: key → ST (ptr (value fd))
  (requires (λ h → live h p))
  (ensures (λ h0 p′ h1 → h0 == h1 ∧ p′ == field p fd))
```

Finally, the read and write operations allows accessing and mutating the field referred to by a ptr.

```
val read: #a:Type → p: ptr a → ST value
  (requires (λ h → live h p))
  (ensures (λ h0 v h1 → live h0 p ∧ h0 == h1 ∧ v == as_value h0 p))

val write: #a:Type → b:ptr a → z:a → ST unit
  (requires (λ h → live h b))
  (ensures (λ h0 _h1 → live h0 b ∧ live h1 b ∧ modifies_1 b h0 h1 ∧ as_value h1 b == z))
```

2.3 Using Low* for Proofs of Functional Correctness and Side-Channel Resistance

This section and the next illustrate our "high-level verification for low-level code" methodology. Although programming at a low-level, we rely on features like type abstraction and dependently typed meta-programming, to prove our code functionally correct, cryptographically secure, and free of a class of side-channels.

We start with Poly1305 [Bernstein 2005], a Message Authentication Code (MAC) algorithm.[2] Unlike chacha20, for which the main property of interest is implementation safety, Poly1305 has a mathematical definition in terms of a polynomial in the prime field $GF(2^{130} - 5)$, against which we prove our code functionally correct. Relying on correctness, we then prove injectivity lemmas on encodings of messages into field polynomials, and we finally prove cryptographic security of a one-time MAC construction for Poly1305, specifically showing unforgeability against chosen message attacks (UF1CMA). This game-based proof involves an idealization step, justified by a probabilistic proof on paper, following the methodology we explain in §2.4.

For side-channel resistance, we use type abstraction to ensure that our code's branching and memory access patterns are secret independent. This style of F* coding is advocated by Zinzindohoué et al. [Zinzindohoué et al. 2016]; we place it on formal ground by showing that it is a sound way of enforcing secret independence at the source level (§3.1) and that our compilation carries such properties to the level of Clight (§3.3). To carry our results further down, one may validate the output of the C compiler by relying on recent tools proving side-channel resistance at the assembly level [Almeida et al. 2016a,b]. We sketch our methodology on a small snippet from our specialized arithmetic (bigint) library upon which we built Poly1305.

[2]Implementation bugs in Poly1305 are still a practical concern: in 2016 alone, the Poly1305 OpenSSL implementation experienced two correctness bugs [Benjamin 2016; Böck 2016] and a buffer overflow [CVE 2016].

```
let normalize (b:bigint) : Stack unit
  (requires (λ m0 → compact m0 b))
  (ensures (λ m0 () m1 → compact m1 b ∧ modifies1 b m0 m1 ∧
                eval m1 b = eval m0 b % (pow2 130 − 5)))
= let h0 = ST.get() in (* a logical snapshot of the initial state *)
  let ones = 67108863ul in (* 2^26 − 1 *)
  let m5 = 67108859ul in (* 2^26 − 5 *)
  let m = (eq_mask b.(4ul) ones) & (eq_mask b.(3ul) ones)
        & (eq_mask b.(2ul) ones) & (eq_mask b.(1ul) ones)
        & (gte_mask b.(0ul) m5) in
  b.(0ul) ← b.(0ul) − m5 & m;
  b.(1ul) ← b.(1ul) − b.(1ul) & m; b.(2ul) ← b.(2ul) − b.(2ul) & m;
  b.(3ul) ← b.(3ul) − b.(3ul) & m; b.(4ul) ← b.(4ul) − b.(4ul) & m;
  lemma_norm h0 (ST.get()) b m (* relates mask to eval modulo *)
```

```
val poly1305_mac:
  tag:nbytes 16ul →
  len:u32 →
  msg:nbytes len{disjoint tag msg} →
  key:nbytes 32ul{disjoint msg key ∧
                  disjoint tag key} →
  Stack unit
(requires (λ m → msg ∈ m ∧ key ∈ m ∧ tag ∈ m))
(ensures (λ m0 _m1 →
  let r = Spec.clamp m0[sub key 0ul 16ul] in
  let s = m0[sub key 16ul 16ul] in
  modifies {tag} m0 m1 ∧
  m1[tag] ==
  Spec.mac_1305 (encode_bytes m0[msg]) r s))
```

Fig. 3. Unique representation of a Poly1305 bigint (left) and the top-level spec of Poly1305 (right)

Representing field elements using bigints. We represent elements of the field underlying Poly1305 as 130-bit integers stored in Low* buffers of machine integers called *limbs*. Spreading out bits evenly across 32-bit words yields five limbs ℓ_i, each holding 26 bits of significant data. A ghost function eval = $\sum_{i=0}^{4} \ell_i \times 2^{26\times i}$ maps each buffer to the mathematical integer it represents. Efficient bigint arithmetic departs significantly from elementary school algorithms. Additions, for instance, can be made more efficient by leveraging the extra 6 bits of data in each limb to delay carry propagation. For Poly1305, a bigint `b` is in compact form in state `m` (i.e., `compact m b`) when all its limbs fit in 26 bits. Compactness does not guarantee uniqueness of representation as $2^{130} - 5$ and 0 are the same number in the field but they have two different compact representations that both fit in 130 bits—this is true for similar reasons for the range $[0, 5)$.

Abstracting integers as a side-channel mitigation. Modern cryptographic implementations are expected to be protected against side-channel attacks [Kocher 1996]. As such, we aim to show that the branching behavior and memory accesses of our crypto code are independent of secrets. To enforce this, we use an abstract type `limb` to represent limbs, all of whose operations reveal no information about the contents of the `limb`, either through its result or through its branching behavior and memory accesses. For example, rather than providing a comparison operator, `eq_leak: limb → limb → Tot bool`, whose boolean result reveals information about the input limbs, we use a masking operation (`eq_mask`) to compute equality securely. Unlike OCaml, F*'s equality is not fully polymorphic, being restricted to only those types that support decidable equality, `limb` not being among them.

```
val v : limb → Ghost nat (* limbs only ghostly revealed as numbers *)
val eq_mask: x:limb → y:limb → Tot (z:limb{if v x ≠ v y then v z = 0 else v z = pow2 26 − 1})
```

In the signature above, `v` is a function that reveals an abstract `limb` as a natural number, but only in ghost code—a style referred to as translucent abstraction [Swamy et al. 2016]. The signature of `eq_mask` claims that it returns a zero limb if the two arguments differ, although computationally relevant code cannot observe this fact. Note, the number of limbs in a Poly1305 bigint is a public constant, i.e., `bigint = b:(buffer limb){b.length = 5}`.

Proving `normalize` correct and side-channel resistant. The `normalize` function of Figure 3 modifies a compact bigint in-place to reduce it to its canonical representation. The code is rather opaque, since it operates by strategically masking each limb in a secret independent manner. However,

its specification clearly shows its intent: the new contents of the input bigint is the same as the original one, *modulo* $2^{130} - 5$. At line 14, we see a call to a F* lemma, which relates the masking operations to the modular arithmetic in the specification—the lemma is erased during extraction.

A top-level functional correctness spec. Using our bigint library, we implement poly1305_mac and prove it functionally correct. Its specification (Figure 3, right) states that the final value of the 16 byte tag (m_1[tag]) is the value of Spec.mac_1305, a polynomial of the message and the key encoded as field elements. We use this mathematical specification as a basis for the game-based cryptographic proofs of constructions built on top of Poly1305, such as the AEAD construction, described next.

2.4 Cryptographic Provable-Security Example: AEAD

Going beyond functional correctness, we sketch how we use Low* to do security proofs in the standard model of cryptography, using "authenticated encryption with associated data" (AEAD) as a sample construction. AEAD is the main protection mechanism for the TLS record layer; it secures most Internet traffic.

AEAD has a generic security proof by reduction to two core functionalities: a stream cipher (such as ChaCha20) and a one-time-MAC (such as Poly1305). The cryptographic, game-based argument supposes that these two algorithms meet their intended *ideal functionalities*, e.g., that the cipher is indistinguishable from a random function. Idealization is not perfect, but is supposed to hold against computationally limited adversaries, except with small probabilities, say, $\varepsilon_{\text{ChaCha20}}$ and $\varepsilon_{\text{Poly1305}}$. The argument then shows that the AEAD construction also meets its own ideal functionality, except with probability, say, $\varepsilon_{\text{Chacha20}} + \varepsilon_{\text{Poly1305}}$.

To apply this security argument to our implementation of AEAD, we need to encode such assumptions. To this end, we supplement our real Low* code with ideal F* code. For example, ideal AEAD is programmed as follows:

- encryption generates a fresh random ciphertext, and it records it together with the encryption arguments in a log.
- decryption simply looks up an entry in the log that matches its arguments and returns the corresponding plaintext, or reports an error.

These functions capture both confidentiality (ciphertexts do not depend on plaintexts) and integrity (decryption only succeeds on ciphertexts output by encryption). Their behaviors are precisely captured by typing, using pre- and post-conditions about the ghost log shared between them, and abstract types to protect plaintexts and keys. We show below the abstract type of keys and the encryption function for idealizing AEAD.

```
type entry = cipher * data * nonce * plain
abstract type key = { key: keyBytes; log: if Flag.aead then ref (seq entry) else unit }
let encrypt (k:key) (n:nonce) (p:plain) (a:data) =
  if Flag.aead then let c = random_bytes ℓ_c in k.log := (c, a, n, p) :: k.log; c
  else encrypt k.key n p a
```

A module Flag declares a set of abstract booleans (*idealization flags*) that precisely capture each cryptographic assumption. For every real functionality that we wish to idealize, we branch on the corresponding flag. In the code above, for instance we idealize encryption when Flag.prf is set.

This style of programming heavily relies on the normalization capabilities of F*. At verification time, flags are kept abstract, so that we verify both the real and ideal versions of the code. At extraction time, we reveal these booleans to be false, allowing the F* normalizer to drop the then branch, and replace the log field with unit, meaning that both the high-level, list-manipulating code and corresponding type definitions are erased, leaving only low-level code from the else branch to be extracted.

Using this technique, we verify by typing that our AEAD code, when using *any* ideal cipher and one-time MAC, perfectly implements ideal AEAD. We also rely on typing to verify that our code complies with the pre-conditions of the intermediate proof steps. Finally, we also prove that our code does not reuse nonces, a common cryptographic pitfall.

Inlining and Type Abstraction. In cryptographic constructions, we often rely on type abstraction to protect private state that depends on keys and other secrets.

A typical C application, such as OpenSSL, achieves limited type abstraction as follows. The library exposes a public C header file for its clients, relying on `void *` and opaque heap allocation functions for type abstraction.

```
typedef void *KEY_HANDLE;
void KEY_init(KEY_HANDLE *key);
void KEY_release(KEY_HANDLE key);
```

Opportunities for mistakes abound, since the `void *` casts are unchecked. Furthermore, abstraction only occurs at the public header boundary, not between internal translation units. Finally, this pattern does not allow the caller to efficiently allocate the actual key on the stack.

The Low* discipline allows the programmer to achieve type abstraction and modularity, while still supporting efficient stack allocation. As an example, for computing one-time MACs incrementally, we use an accumulator that holds the current value of a polynomial computation, which depends on a secret key. For cryptographic soundness, we must ensure that no information about such intermediate values leak to the rest of the code.

To this end, all operations on accumulators are defined in a single module of the form below—our code is similar but more complex, as it supports MAC algorithms with different field representations and key formats, and also keeps track of the functional correctness of the polynomial computation.

```
module OneTimeMAC
type elem = lbytes (v accLen) (* intermediate value (representing a a field element) *)
abstract type key (i:macID) = elem
abstract type accum (i:macID) = elem
(* newAcc allocates on the caller's frame *)
let newAcc (i:macID) : StackInline (accum i) (...) = Buffer.create 0ul accLen
let extend (i:macID) (key: macKey i) (acc:accum i) (word:elem) : Stack unit (...) = add acc word; mul acc key
```

The index `i` is used to separate multiple instances of MACs; for instance, it prevents calls to extend an accumulator with the wrong key. Our type-based separation between different kinds of elements is purely static. At runtime, the accumulator, and probably the key, are just bytes on the stack (or in registers), whereas the calls to `add` and `mul` are also likely to be have been inlined in the code that uses MACs.

The `newAcc` function creates a new buffer for a given index, initialized to 0. The function returns a pointer to the buffer it allocates. The `StackInline` effect indicates that the function does need to push a frame before allocation, but instead allocates in its caller's stack frame. KreMLin textually inlines the function in its caller's body at every call-site, ensuring that the allocation performed by `newAcc` indeed happens in the caller's stack frame. From the perspective of Low*, `newAcc` is a function in a separate module, and type abstraction is preserved.

3 A FORMAL TRANSLATION FROM LOW* TO CLIGHT

Figure 1 on page 3 provides an overview of our translation from Low* to CompCert Clight, starting with EMF*, a recently proposed model of F* [Ahman et al. 2017]; then λow*, a formal core of Low* after all erasure of ghost code and specifications; then C*, an intermediate language that switches the calling convention closer to C; and finally to Clight. In the end, our theorems establish that: (a)

$$
\begin{aligned}
\tau &::= \mathsf{int} \mid \mathsf{unit} \mid \{\overrightarrow{f = \tau}\} \mid \mathsf{buf}\ \tau \mid \alpha \\
v &::= x \mid n \mid () \mid \{\overrightarrow{f = v}\} \mid (b, n, \overrightarrow{f}) \\
e &::= \mathsf{let}\ x : \tau = \mathsf{readbuf}\ e_1\ e_2\ \mathsf{in}\ e \mid \mathsf{let}\ \_ = \mathsf{writebuf}\ e_1\ e_2\ e_3\ \mathsf{in}\ e \\
&\quad \mid \mathsf{let}\ x = \mathsf{newbuf}\ n\ (e_1 : \tau)\ \mathsf{in}\ e_2 \mid \mathsf{subbuf}\ e_1\ e_2 \\
&\quad \mid \mathsf{let}\ x : \tau = \mathsf{readstruct}\ e_1\ \mathsf{in}\ e \mid \mathsf{let}\ \_ = \mathsf{writestruct}\ e_1\ e_2\ \mathsf{in}\ e \\
&\quad \mid \mathsf{let}\ x = \mathsf{newstruct}\ (e_1 : \tau)\ \mathsf{in}\ e_2 \mid e_1 \rhd f \\
&\quad \mid \mathsf{withframe}\ e \mid \mathsf{pop}\ e \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\
&\quad \mid \mathsf{let}\ x : \tau = d\ e_1\ \mathsf{in}\ e_2 \mid \mathsf{let}\ x : \tau = e_1\ \mathsf{in}\ e_2 \mid \{\overrightarrow{f = e}\} \mid e.f \mid v \\
P &::= \cdot \mid \mathsf{let}\ d = \lambda y : \tau_1.\ e : \tau_2, P
\end{aligned}
$$

Fig. 4. λow* syntax

the safety and functional correctness properties verified at the F* level carry on to the generated Clight code (via semantics preservation), and (b) Low* programs that use the secrets parametrically enjoy the trace equivalence property, at least until the Clight level, thereby providing protection against side-channels.

Prelude: Internal transformations in EMF*. We begin by briefly describing a few internal transformations on EMF*, focusing in the rest of this section on the pipeline from λow* to Clight—the formal details are in the appendix. To express computational irrelevance, we extend EMF* with a primitive Ghost effect. An erasure transformation removes ghost subterms, and we prove that this pass preserves semantics, via a logical relations argument. Next, we rely on a prior result [Ahman et al. 2017] showing that EMF* programs in the ST monad can be safely reinterpreted in $\text{EMF}^{\star}_{\text{ST}}$, a calculus with primitive state. We obtain an instance of $\text{EMF}^{\star}_{\text{ST}}$ suitable for Low* by instantiating its state type with HyperStack.mem. To facilitate the remainder of the development, we transcribe $\text{EMF}^{\star}_{\text{ST}}$ to λow*, which is a restriction of $\text{EMF}^{\star}_{\text{ST}}$ to first-order terms that only use stack memory, leaving the heap out of λow*, since it is not a particularly interesting aspect of the proof. This transcription step is essentially straightforward, but is not backed by a specific proof. We plan to fill this gap as we aim to mechanize our entire proof in the future.

3.1 λow*: A Formal Core of Low* Post-Erasure

The meat of our formalization of Low* begins with λow*, a first-order, stateful language, whose state is structured as a stack of memory regions. It has a simple calling convention using a traditional, substitutive β-reduction rule. Its small-step operational semantics is instrumented to produce traces that record branching and the accessed memory addresses. As such, our traces account for side-channel vulnerabilities in programs based on the program counter model [Molnar et al. 2006] augmented to track potential leaks through cache behavior [Barthe et al. 2014]. We define a simple type system for λow* and prove that programs well-typed with respect to some values at an abstract type produce traces independent of those values, e.g., our bigint library when translated to λow* is well-typed with respect to an abstract type of limbs and leaks no information about them via their traces.

Syntax. Figure 4 shows the syntax of λow*. A program P is a sequence of top-level function definitions, d. We omit loops but allow recursive function definitions. Values v include constants, immutable records, and buffers $(b, n, [])$ and mutable structures $(b, n, \overrightarrow{f})$ passed by reference, where b is the address of the buffer or structure, n is the offset in the buffer, and $\overrightarrow{f}$ designates the path to the structure field to take a reference of (this path, as a list, can be longer than 1 in the case of nested mutable structures.) Stack allocated buffers (readbuf, writebuf, newbuf, and subbuf), and

$$\frac{}{P \vdash (H, \mathsf{withframe}\ e) \rightarrow_\cdot (H; \{\}, \mathsf{pop}\ e)}\ \textsc{WF} \qquad \frac{}{P \vdash (H; \_, \mathsf{pop}\ v) \rightarrow_\cdot (H, v)}\ \textsc{Pop}$$

$$\frac{}{P \vdash (H, \mathsf{if}\ 0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) \rightarrow_{\mathsf{brF}} (H, e_2)}\ \textsc{LIfF} \qquad \frac{P(f) = \lambda y : \tau_1.\ e_1 : \tau_2}{P \vdash (H, \mathsf{let}\ x : \tau = f\ v\ \mathsf{in}\ e) \rightarrow (H, \mathsf{let}\ x : \tau = e_1[v/y]\ \mathsf{in}\ e)}\ \textsc{App}$$

$$\frac{H(b, n + n_1, []) = v \qquad \ell = \mathsf{read}(b, n + n_1, [])}{P \vdash (H, \mathsf{let}\ x = \mathsf{readbuf}\ (b, n, [])\ n_1\ \mathsf{in}\ e) \rightarrow_\ell (H, e[v/x])}\ \textsc{LRd}$$

$$\frac{b \notin \mathsf{dom}(H; h) \quad h_1 = h[b \mapsto v^n] \quad e_1 = e[(b, 0)/x] \qquad \ell = \mathsf{write}(b, 0), \ldots, \mathsf{write}(b, n-1)}{P \vdash (H; h, \mathsf{let}\ x = \mathsf{newbuf}\ n\ (v : \tau)\ \mathsf{in}\ e) \rightarrow_\ell (H; h_1, e_1)}\ \textsc{New}$$

Fig. 5. Selected semantic rules from λow*

their mutable structure counterparts (readstruct, writestruct, newstruct, ▷), are the main feature of the expression language, along with withframe e, which pushes a new frame on the stack for the evaluation of e, after which it is popped (using pop e, an administrative form internal to the calculus). Once a frame is popped, all its local buffers and mutable structures become inaccessible.

Mutable structures can be nested, and stored into buffers, in both cases without extra indirection. However, the converse is not true, as λow* currently does not allow arbitrary nesting of arrays within mutable structures without explicit indirection via separately allocated buffers. We leave such generalization as future work.

Type system. λow* types include the base types int and unit, record types $\{\overrightarrow{f = \tau}\}$, buffer types buf τ, mutable structure types struct τ, and abstract types α. The typing judgment has the form, $\Gamma_P; \Sigma; \Gamma \vdash e : \tau$, where Γ_P includes the function signatures; Σ is the store typing; and Γ is the usual context of variables. We elide the rules, as it is a standard, simply-typed type system. The type system guarantees preservation, but not progress, since it does not attempt to account for bounds checks or buffer/mutable structure lifetime. However, memory safety (and progress) is a consequence of Low* typing and its semantics-preserving erasure to λow*.

Semantics. We define evaluation contexts E for standard call-by-value, left-to-right evaluation order. The memory H is a stack of frames, where each frame maps addresses b to a sequence of values $\overrightarrow{v}$. The λow* small-step semantics judgment has the form $P \vdash (H, e) \rightarrow_\ell (H', e')$, meaning that under the program P, configuration (H, e) steps to (H', e') emitting a trace ℓ, including reads and writes to buffer references or mutable structure references, and branching behavior, as shown below.

$$\ell ::= \cdot \mid \mathsf{read}(b, n, \overrightarrow{f}) \mid \mathsf{write}(b, n, \overrightarrow{f}) \mid \mathsf{brT} \mid \mathsf{brF} \mid \ell_1, \ell_2$$

Figure 5 shows selected reduction rules from λow*. Rule WF pushes an empty frame on the stack, and rule Pop pops the topmost frame once the expression has been evaluated. Rule LIfF is standard, except for the trace brF recorded on the transition. Rule App is a standard, substitutive β-reduction. Rule LRd returns the value at the $(n + n_1)$ offset in the buffer at address b, and emits a $\mathsf{read}(b, n + n_1, [])$ event. Rule New initializes the new buffer, and emits write events corresponding to each offset in the buffer.

Secret independence. A λow* program can be written against an interface providing secrets at an abstract type. For example, for the abstract type limb, one might augment the function signatures Γ_P of a program with an interface for the abstract type $\Gamma_{\mathsf{limb}} = \mathsf{eq\_mask} : \mathsf{limb}^2 \rightarrow \mathsf{limb}$, and typecheck a source program with free limb variables (Γ = secret:limb), and empty store typing, using the judgment

$\Gamma_{\text{limb}}, \Gamma_p; \cdot; \Gamma \vdash e : \tau$. Given any representation τ for limb, an implementation for eq_mask whose trace is input independent, and any pair of values $v_0 : \tau, v_1 : \tau$, we prove that running $e[v_0/\text{secret}]$ and $e[v_1/\text{secret}]$ produces identical traces, i.e., the traces reveal no information about the secret v_i. We sketch the formal development next, leaving details to the appendix.

Given a derivation $\Gamma_s, \Gamma_P; \Sigma; \Gamma \vdash e : \tau$, let Δ map type variables in the interface Γ_s to concrete types and let P_s contain the implementations of the functions (from Γ_s) that operate on secrets. To capture the secret independence of P_s, we define a notion of an *equivalence modulo secrets*, a type-indexed relation for values ($v_1 \equiv_\tau v_2$) and memories ($\Sigma \vdash H_1 \equiv H_2$). Intuitively two values (resp. memories) are equivalent modulo secrets if they only differ in subterms that have abstract types in the domain of the Δ map—we abbreviate "equivalent modulo secrets" as "related" below. We then require that each function $f \in P_s$, when applied in related stores to related values, always returns related results, while producing *identical* traces. Practically, P_s is a (small) library written carefully to ensure secret independence.

Our secret-independence theorem is then as follows:

Theorem 3.1 (Secret independence). *Given*

(1) a program well-typed against a secret interface, Γ_s, i.e, $\Gamma_s, \Gamma_P; \Sigma; \Gamma \vdash (H, e) : \tau$,
(2) a well-typed implementation of the Γ_s interface, $\Gamma_s; \Sigma; \cdot \vdash_\Delta P_s$, such that P_s is equivalent modulo secrets,
(3) a pair (ρ_1, ρ_2) of well-typed substitutions for Γ,

then either:

(1) both programs cannot reduce further, i.e. $P_s, P \vdash (H, e)[\rho_1] \not\to$ and $P_s, P \vdash (H, e)[\rho_2] \not\to$, or
(2) both programs make progress with the same trace, i.e. there exists $\Sigma' \supseteq \Sigma, \Gamma' \supseteq \Gamma, H', e'$, a pair (ρ_1', ρ_2') of well-typed substitutions for Γ', and a trace ℓ such that
  i) $P_s, P \vdash (H, e)[\rho_1] \to_\ell^+ (H', e')[\rho_1']$ and $P_s, P \vdash (H, e)[\rho_2] \to_\ell^+ (H', e')[\rho_2']$, and
  ii) $\Gamma_s, \Gamma_P; \Sigma'; \Gamma' \vdash (H', e') : \tau$

3.2 C*: An Intermediate Language

We move from λow* to Clight in two steps. The C* intermediate language retains λow*'s explicit scoping structure, but switches the calling convention to maintain an explicit call-stack of continuations (separate from the stack memory regions). C* also switches to a more C-like syntax, separates side effect-free expressions from effectful statements.

$$
\begin{array}{rcl}
\hat{P} & ::= & \overrightarrow{\text{fun } f\,(x : \tau) : \tau\,\{\ \overrightarrow{s}\ \}} \\
\hat{e} & ::= & n \mid () \mid x \mid \hat{e} + \hat{e} \mid \{\overrightarrow{f = \hat{e}}\} \mid \hat{e}.f \mid \&\hat{e} \to f \\
s & ::= & \tau\ x = \hat{e} \mid \tau\ x = f(\hat{e}) \mid \text{if } \hat{e} \text{ then } \overrightarrow{s} \text{ else } \overrightarrow{s} \mid \text{return } \hat{e} \\
 & & \mid \{\ \overrightarrow{s}\ \} \mid \tau\ x[n] \mid \tau\ x = *[\hat{e}] \mid *[\hat{e}] = \hat{e} \mid \text{memset } \hat{e}\ n\ \hat{e}
\end{array}
$$

The syntax is unsurprising, with two notable exceptions. First, despite the closeness to C syntax, contrary to C and similarly to λow*, block scopes are not required for branches of a conditional statement, so that any local variable or local array declared in a conditional branch, if not enclosed by a further block, is still live after the conditional statement. Second, non-array local variables are immutable after initialization.

Operational semantics, in contrast to λow.* A C* evaluation configuration C consists of a stack S, a variable assignment V and a statement list $\overrightarrow{s}$ to be reduced. A stack is a list of frames. A frame F includes frame memory M, local variable assignment V to be restored upon function exit, and continuation E to be restored upon function exit. Frame memory M is optional: when it is $\perp$, the frame is called a "call frame"; otherwise a "block frame", allocated whenever entering a statement

$$\frac{}{\hat{P} \vdash (S, V, \{\overrightarrow{s_1}\}; \overrightarrow{s_2}) \rightsquigarrow (S; (\{\}, V, \square; \overrightarrow{s_2}), V, \overrightarrow{s_1})}\ \textsc{Block} \qquad \frac{}{\hat{P} \vdash (S; (M, V', E), V, []) \rightsquigarrow (S, V', E\,[()])}\ \textsc{Empty}$$

$$\frac{[\![\hat{e}]\!]_{(V)} = 0}{\hat{P} \vdash (S, V, \text{if } \hat{e} \text{ then } \overrightarrow{s_1} \text{ else } \overrightarrow{s_2}; \overrightarrow{s}) \rightsquigarrow_{\text{brF}} (S, V, \overrightarrow{s_2}; \overrightarrow{s})}\ \textsc{CIfF}$$

$$\frac{\hat{P}(f) = \text{fun } (y : \tau_1) : \tau_2 \{\ \overrightarrow{s_1}\ \} \qquad [\![\hat{e}]\!]_{(V)} = v}{\hat{P} \vdash (S, V, \tau\ x = f\ \hat{e}; \overrightarrow{s}) \rightsquigarrow (S; (\bot, V, \tau\ x = \square; \overrightarrow{s}), \{\}[y \mapsto v], \overrightarrow{s_1})}\ \textsc{Call}$$

$$\frac{[\![\hat{e}]\!]_{(V)} = (b, n, \overrightarrow{f}) \quad \text{Get}(S, (b, n, \overrightarrow{f})) = v \quad \ell = \text{read } (b, n, \overrightarrow{f})}{\hat{P} \vdash (S, V, \tau\ x = *[\hat{e}]; \overrightarrow{s}) \rightsquigarrow_{\ell} (S, V[x \mapsto v], \overrightarrow{s})}\ \textsc{CRead}$$

$$\frac{S = S'; (M, V, E) \qquad b \notin S \qquad V' = V[x \mapsto (b, 0, [])]}{\hat{P} \vdash (S, V, \tau\ x[n]; \overrightarrow{s}) \rightsquigarrow (S'; (M[b \mapsto \bot^n], V, E), V', \overrightarrow{s})}\ \textsc{ArrDecl}$$

Fig. 6. Selected semantic rules from C*

block and deallocated upon exiting such block. A frame memory is just a partial map from block identifiers to value lists. Each C* statement performs at most one function call, or otherwise, at most one side effect. Thus, C* is deterministic.

The semantics of C* is shown to the right in Figure 6, also illustrating the translation from λow* to C*. There are three main differences. First, C*'s calling convention (rule Call) shows an explicit call frame being pushed on the stack, unlike λow*'s β reduction. Additionally, C* expressions do not have side effects and do not access memory; thus, their evaluation order does not matter and their evaluation can be formalized as a big-step semantics; by themselves, expressions do not produce events. This is apparent in rules like CIfF and CRead, where the expressions are evaluated atomically in the premises. Finally, newbuf in λow* is translated to an array declaration followed by a separate initialization. In C*, declaring an array allocates a fresh memory block in the current memory frame, and makes its memory locations available but uninitialized. Memory write (resp. read) produces a write (resp. read) event. memset $\hat{e}_1\ m\ \hat{e}_2$ produces m write events, and can be used only for arrays.

Correctness of the λow-to-C* transformation.* We proved that execution traces are exactly preserved from λow* to C*:

Lemma 3.2 (λow* to C*). *Let P be a λow* program and e be a λow* entry point expression, and assume that they compile: $\Downarrow (P) = \hat{P}$ for some C* program $\hat{P}$ and $\downarrow (e) = \overrightarrow{s}; \hat{e}$ for some C* list of statements $\overrightarrow{s}$ and expression $\hat{e}$.*

Let V be a mapping of local variables containing the initial values of secrets. Then, the C program $\hat{P}$ terminates with trace ℓ and return value v, i.e., $\hat{P} \vdash ([], V, \overrightarrow{s}; \text{return } \hat{e}) \overset{\ell,*}{\rightarrow} ([], V', \text{return } v)$ if, and only if, so does the λow* program: $P \vdash (\{\}, e[V]) \overset{\ell,*}{\rightarrow} (H', v)$; and similarly for divergence.*

In particular, if the source λow* program is safe, then so is the target C* program. It also follows that the trace equality security property is preserved from λow* to C*. We prove this theorem by bisimulation. In fact, it is enough to prove that any λow* behavior is a C* behavior, and flip the diagram since C* is deterministic. That C* semantics use big-step semantics for C* expressions complicates the bisimulation proof a bit because λow* and C* steps may go out-of-sync at times.

Within the proof we used a relaxed notion of simulation ("quasi-refinement") that allows this temporary discrepancy by some stuttering, but still implies bisimulation.

3.3 From C* to CompCert Clight and Beyond

CompCert Clight is a deterministic (up to system I/O) subset of C with no side effects in expressions, and actual byte-level representation of values. Clight has a realistic formal semantics [Blazy and Leroy 2009; Leroy 2016] and tractable enough to carry out the correctness proofs of our transformations from λow* to C. More importantly, Clight is the source language of the CompCert compiler backend, which we can thus leverage to preserve at least safety and functional correctness properties of Low* programs down to assembly.[3]

Recall that we need to produce an event in the trace whenever a memory location is read or written, and whenever a conditional branch is taken, to account for memory accesses and statements in the semantics of the generated Clight code for the purpose of our noninterference security guarantees. However, the semantics of CompCert Clight *per se* produces no events on memory accesses; instead, CompCert provides a syntactic program annotation mechanism using no-op *built-in calls*, whose only purpose is to add extra events in the trace. Thus, we leverage this mechanism by prepending each memory access and conditional statement in the Clight generated code with one such built-in call producing the corresponding events.

The main two differences between C* and Clight, which our translation deals with as described below, are immutable local structures, and scope management for local variables.

Immutable local structures. C* handles immutable local structures as first-class values, whereas Clight only supports non-compound data (integers, floating-points or pointers) as values.

If we naively translate immutable local C* structures to C structures in Clight, then CompCert will allocate them in memory. This increases the number of memory accesses, which not only introduces discrepancies in the security preservation proof from C* to Clight, but also introduces significant performance overhead compared to GCC, which optimizes away structures whose addresses are never taken.

Instead, we split an immutable structure into the sequence of all its non-compound fields, each of which is to be taken as a potentially non-stack-allocated local variable,[4] except for functions that return structures, where, as usual, we add, as an extra argument to the callee, a pointer to the memory location written to by the callee and read by the caller.

Local variable hoisting. Scoping rules for C* local arrays are not exactly the same as in C, in particular for branches of conditional statements. So, it is necessary to hoist all local variables to function-scope. CompCert 2.7.1 does support such hoisting but as an unproven elaboration step. While existing formal proofs (e.g., Dockins' [Dockins 2012, §9.3]) only prove functional correctness, we also prove preservation of security guarantees, as shown below.

Proof techniques. Contrary to the λow*-to-C* transformation, our subsequent passes modify the memory layout leading to differences in traces between C* to Clight, due to pointer values. Thus, we need to address security preservation separately from functional correctness.

For each pass changing the memory layout, we split it into three passes. First, we *reinterpret* the program by replacing each pointer value in event traces with the function name and recursion depth of its function call, the name of the corresponding local variable, and the array index and

[3] As a subset of C, Clight can be compiled by any C compiler, but only CompCert provides formal guarantees.

[4] Our benchmark without this structure erasure runs 20% slower than with structure erasure, both with CompCert 2.7. Without structure erasure, code generated with CompCert is 60% slower than with `gcc -O1`. CompCert-generated code without structure erasure may even segfault, due to stack overflow, which structure erasure successfully overcomes.

structure field name within this local variable. Then, we perform the actual transformation and prove that it exactly preserves traces in this new "abstract" trace model. Finally, we reinterpret the generated code back to concrete pointer values. We successfully used this technique to prove functional correctness and security preservation for variable hoisting.

For each pass that adds new memory accesses, we split it into two passes. First, a reinterpretation pass produces new events corresponding to the provisional memory accesses (without actually performing those memory accesses). Then, this pass is followed by the actual trace-preserving transformation that goes back to the non-reinterpreted language but adds the actual memory accesses into the program. We successfully used this technique to prove functional correctness and security preservation for structure return, where we add new events and memory accesses whenever a C* function returns a structure value.

In both cases, we mean *reinterpretation* as defining a new language with the same syntax and small-step semantic rules except that the produced traces are different, and relating executions of the same program in the two languages. There, it is easy to prove functional correctness, but for security preservation, we need to prove an invariant on two small-step executions of the same program with different secrets, to show that two equal pointer values in event traces coming from those two different executions will actually turn into two equal abstract pointer values in the reinterpreted language.

Our detailed functional correctness and security preservation proofs from λow* to Clight can be found in the appendix.

Towards verified assembly code. We conjecture that our reinterpretation techniques can be generalized to most passes of CompCert down to assembly. While we leave such generalization as future work, some guarantees from C to assembly can be derived by instrumenting CompCert [Barthe et al. 2014] and LLVM [Almeida et al. 2016b; Zhao et al. 2012, 2013] and turning them into *certifying* (rather than certified) compilers where security guarantees are statically rechecked on the compiled code through translation validation, thus re-establishing them independently of source-level security proofs. In this case, rather than being fully preserved down to the compiled code, Low*-level proofs are still useful to *practically* reduce the risk of failures in translation validation.

4 KREMLIN: A COMPILER FROM LOW* TO C

4.1 From Low* to Efficient, Elegant C

As explained previously, λow* is the core of Low*, post erasure. Transforming Low* into λow* proceeds in several stages. First, we rely on F*'s existing normalizer and erasure and extraction facility (similar to features in Coq [Letouzey 2008]), to obtain an ML-like AST for Low* terms. Then, we use our new tool KreMLin that transforms this AST further until it falls within the λow* subset formalized above. KreMLin then performs the λow* to C* transformation, followed by the C* to C transformation and pretty-printing to a set of C files. KreMLin generates C11 code that may be compiled by GCC; Clang; Microsoft's C compiler or CompCert. We describe the main transformations performed by KreMLin, beyond those formalized in §3, next.

Structures by value. We described earlier (§2.2) our Low* struct library that grants the programmer fine-grained control over the memory layout, as well as mutability of interior fields. As an alternative, KreMLin supports immutable, by-value structs. Such structures, being pure, come with no liveness proof obligations. The performance of the generated C code, however, is less predictable: in many cases, the C compiler will optimize and pass such structs by reference, but on some ABIs (x86), the worst-case scenario may be costly.

Concretely, the F* programmer uses tuples and inductive types. Tuples are monomorphized into specialized inductive types. Then, inductive types are translated into idiomatic C code: single-branch inductive types (e.g., records) become actual C structs, inductives with only constant constructors become C enums, and other inductives becomes C tagged unions, leveraging C11 anonymous unions for syntactic elegance. Pattern matches become, respectively, switches, let-bindings, or a series of cascading if-then-elses.

Whole-program transformations. KreMLin perform a series of whole-program transformations. First, the programmer is free to use parameterized type abbreviations. KreMLin substitutes an application of a type abbreviation with its definition, since C's `typedef` does not support parameters. (C++11 alias templates would support this use-case.) Second, KreMLin recursively inlines all `StackInline` functions, as required for soundness (cf. §2.4). Third, KreMLin performs a reachability analysis. Any function that is not reachable from the `main` function or, in the case of a library, from a distinguished API module, is dropped. This is essential for generating palatable C code that does not contain unused helper functions used only for verification. Fourth, KreMLin supports a concept of "bundle", meaning that several F* modules may be grouped together into a single C translation unit, marking all of the functions as `static`, except for those reachable via the distinguished API module. This not only makes the code much more idiomatic, but also triggers a cascade of optimizations that the C compiler is unable to perform across translation units.

Going to an expression language. F* is, just like ML, an expression language. Two transformations are required to go to a statement language: *stratification* and *hoisting*. Stratification places buffer allocations, assignments and conditionals in statement position before going to C*. Hoisting, as discussed in §3.3, deals with the discrepancy between C99 block scope and Low* `with_frame`; a buffer allocated under a `then` branch must be hoisted to the nearest enclosing `push_frame`, otherwise its lifetime would be shortened by the resulting C99 block after translation.

Readability. KreMLin puts a strong emphasis on generating readable C, in the hope that security experts not familiar with F* can review the generated C code. Names are preserved; we use `enum` and `switch` whenever possible; functions that take `unit` are compiled into functions with no parameters; functions that return `unit` are compiled into `void`-returning functions. The internal architecture relies on an abstract C AST and what we believe is a correct C pretty-printer.

Implementation. KreMLin represents about 10,000 lines of OCaml, along with a minimal set of primitives implemented in a few hundred lines of C. After F* has extracted and erased the AEAD development, KreMLin takes less than a second to generate the entire set of C files. The implementation of KreMLin is optimized for readability and modularity; there was no specific performance concern in this first prototype version. KreMLin was designed to support multiple backends; we are currently implementing a WebAssembly backend to provide verified, efficient cryptographic libraries for the web.

4.2 Integrating KreMLin's Output

KreMLin generates a set of C files that have no dependencies, beyond a single `.h` file and C11 standard headers, meaning KreMLin's output can be readily integrated into an existing source tree.

To allow code sharing and re-use, programmers may want to generate a shared library, that is, a `.dll` or `.so` file that can be distributed along with a public header (`.h`) file. The programmer can achieve this by writing a distinguished API module in F*, exposing only carefully-crafted function signatures. As exemplified earlier (Figure 2), the translation is predictable, meaning the programmer can precisely control, in F*, what becomes, in C, the library's public header. The bundle feature of

KreMLin then generates a single C file for the library; upon compiling it into a shared object, the only visible symbols are those exposed by the programmer in the header file.

We used this approach for our HACL* library. Our public header file exposes functions that have the exact same signature as their counterpart in the NaCL library. If an existing binary was compiled against NaCL's public header file, then one can configure the dynamic linker to use our HACL* library instead, without recompiling the original program (using the infamous "LD preload trick").

The functions exposed by the library comply with the C ABI for the chosen toolchain. This means that one may use the library from a variety of programming languages, relying on foreign-function interfaces to interoperate. One popular approach is to generate bindings for the C library *at run-time* using the ctypes and the libffi [Green 2014] libraries. This is an approach leveraged by languages such as JavaScript, Python or OCaml, and requires no recompilation.

An alternative is to write bindings by hand, which allows for better performance and control over how data is transformed at the boundary, but requires writing and recompiling potentially error-prone C code. This is the historical way of writing bindings for many languages, including OCaml. We plan to have KreMLin generate these bindings automatically. We used this approach in miTLS, effectively making it a mixed C/OCaml project. We intend to eventually lower all of miTLS into Low*.

5 BUILDING VERIFIED LOW* LIBRARIES AND APPLICATIONS

Table 1. Evaluation of verified Low* libraries and applications (time reported on an Intel Core E5 1620v3 CPU)

Codebase	LoC	C LoC	%annot	Verif. time
Low* standard library	8,936	N/A	N/A	8m
HACL*	6,050	11,220	28%	12h
miTLS AEAD	13,743	14,292	56.5%	1h 10m

In this section, we describe two examples (summarized in Table 1) that show how Low* can be used to build applications that balance complex verification goals with high performance. First, we describe HACL*, an efficient library of cryptographic primitives that are verified to be memory safe, side-channel resistant, and, where there exists a simple mathematical specification, functionally correct. Then, we show how to use Low* for type-based cryptographic security verification by implementing and verifying the AEAD construction in the Transport Layer Security (TLS) protocol. We show how this Low* library can be integrated within miTLS, an F* implementation of TLS that is compiled to OCaml.

5.1 HACL*: A Fast and Safe Cryptographic Library

In the wake of numerous security vulnerabilities, Bernstein et al. [2012] argue that libraries like OpenSSL are inherently vulnerable to attacks because they are too large, offer too many obsolete options, and expose a complex API that programmers find hard to use securely. Instead they propose a new cryptographic API called NaCl that uses a small set of modern cryptographic primitives, such as Curve25519 [Bernstein 2006] for key exchange, the Salsa family of symmetric encryption algorithms [Bernstein 2008], which includes Salsa20 and ChaCha20, and Poly1305 for message authentication [Bernstein 2005]. These primitives were all designed to be fast and easy to implement in a side-channel resistant coding style. Furthermore, the NaCl API does not directly expose these low-level primitives to the programmer. Instead it recommends the use of simple

composite functions for symmetric key authenticated encryption (`secretbox`/`secretbox_open`) and for public key authenticated encryption (`box`/`box_open`).

The simplicity, speed, and robustness of the NaCl API has proved popular among developers. Its most popular implementation is Sodium [lib 2017], which has bindings for dozens of programming languages and is written mostly in C, with a few components in assembly. An alternative implementation called TweetNaCl [Bernstein et al. 2014] seeks to provide a concise implementation that is both readable and *auditable* for memory safety bugs, a useful point of comparison for our work. With Low*, we show how we can take this approach even further by placing it on formal, machine-checked ground, without compromising performance.

A Verified NaCl Library. We implement the NaCl API, including all its component algorithms, in a Low* library called HACL*, mechanically verifying that all our code is memory safe, functionally correct, and side-channel resistant. The C code generated from HACL* is ABI-compatible and can be used as a drop-in replacement for Sodium or TweetNaCl in any application, in C or any other language, that relies on these libraries. Our code is written and optimized for 64-bit platforms; on 32-bit machines, we rely on a stub library for performing 64x64-bit multiplications and other 128-bit operations.

We implement and verify four cryptographic primitives: ChaCha20, Salsa20, Poly1305, and Curve25519, and then use them to build three cryptographic constructions: AEAD, `secretbox` and `box`. For all our primitives, we prove that our stateful optimized code matches a high-level functional specification written in F*. These are new verified implementations. Previously, Tomb [2016] used SAW and Cryptol to verify C and Java implementations of Chacha20, Salsa20, Poly1305, AES, and ECDSA. Using a different methodology, Bond et al. [2017] verifies an assembly version of Poly1305. Curve25519 has been verified before: Chen et al. [2014] verified an optimized low-level assembly implementation using an SMT solver; Zinzindohoué et al. [2016] wrote and verified a high-level library of three curves, including Curve25519, in F* and generated an OCaml implementation from it. Our verified Curve25519 code explores a third direction by targeting reference C code that is both fast and readable.

A companion paper currently under review [Zinzindohoué et al. 2017] is entirely devoted to the HACL* library, and contains an in-depth evaluation of the proof methodology, several new algorithms that were verified since the present paper was written, along with a more comprehensive performance analysis.

Table 2. Performance in CPU cycles: 64-bit HACL*, 64-bit Sodium (pure C, no assembly), 32-bit TweetNaCl, 64-bit OpenSSL (pure C, no assembly), and the fastest assembly implementation included in eBACS SUPERCOP. All code was compiled using `gcc -O3` optimized and run on a 64-bit Intel Xeon CPU E5-1630. Results are averaged over 1000 measurements, each processing a random block of 2^{14} bytes; Curve25519 was averaged over 1000 random key-pairs.

Algorithm	HACL*	Sodium	TweetNaCl	OpenSSL	eBACS Fastest
ChaCha20	6.17 cy/B	6.97 cy/B	-	8.04 cy/B	1.23 cy/B
Salsa20	6.34 cy/B	8.44 cy/B	15.14 cy/B	-	1.39 cy/B
Poly1305	2.07 cy/B	2.48 cy/B	32.32 cy/B	2.16 cy/B	0.68 cy/B
Curve25519	157k cy/mul	162k cy/mul	1663k cy/mul	359k cy/mul	145k cy/mul
AEAD-ChaCha20-Poly1305	8.37 cy/B	9.60 cy/B	-	8.53 cy/B	
SecretBox	8.43 cy/B	11.03 cy/B	50.56 cy/B	-	
Box	18.10 cy/B	20.97 cy/B	149.22 cy/B	-	

Performance. Table 2 compares the performance of HACL* to Sodium, TweetNaCl, and OpenSSL by running each primitive on a 16KB input; we chose this size since it corresponds to the maximum

Table 3. Performance in operations per second: 64-bit HACL*, 64-bit OpenSSL (assembly disabled) and Microsoft's "Crypto New Generation" (CNG) library on a 64-bit Windows 10 machine. These results were obtained by writing an OpenSSL engine that calls back to either HACL*, CNG, or OpenSSL itself (so as to include the overhead of going through a pluggable engine). The `speed ecdhx25519` command runs multiplications for 10s, then counts the number of multiplications performed. We show the average over 10 runs of this command. The machine is a desktop machine with a 64-bit Intel Xeon CPU E5-1620 v2 nominally clocked at 3.70Ghz.

Algorithm	HACL*	OpenSSL	CNG
Curve25519	17700 mul/s (σ = 246)	8033 mul/s (σ = 120)	7490 mul/s (σ = 114)

record size in TLS and represents a good balance between small network messages and large files. We report averages over 1000 iterations expressed in cycles/byte. For Curve25519, we measure the time taken for one call to scalar multiplication. For comparison with state-of-the-art assembly implementations, for each primitive, we also include the best performance for any implementation (assembly or C) included in the eBACS SUPERCOP benchmarking framework.[5] These fastest implementation are typically in architecture-specific assembly.

We performed these tests on a variety of 64-bit Intel CPUs (the most popular desktop configuration) and these performance numbers were similar across machines. To confirm these measurements, we also ran the full eBACS SUPERCOP benchmarks on our code, as well as the OpenSSL speed benchmarks, and the results closely mirrored Table 2. However, we warn the performance numbers could be quite different on (say) 32-bit ARM platforms.

We observe that for ChaCha20, Salsa20, and Poly1305, HACL* achieves comparable performance to the optimized C code in OpenSSL and Sodium and significantly better performance than TweetNaCl's concise C implementation. Assembly implementations of these primitives are about 3-4 times faster; they typically rely on CPU-specific vector instructions and careful hand-optimizations.

Our Curve25519 implementation is about the same speed as Sodium's 64-bit C implementation (`donna_c64`) and an order of magnitude faster than TweetNaCl's 32-bit code. It is also significantly faster than OpenSSL because even 64-bit OpenSSL uses a Curve25519 implementation that was optimized for 32-bit integers, whereas the implementations in Sodium and HACL* take advantage of the 64x64-bit multiplier available on Intel's 64-bit platforms. The previous F* implementation of Curve25519 [Zinzindohoué et al. 2016] running in OCaml was not optimized for performance; it is more than 100x slower than HACL*. The fastest assembly code for Curve25519 on eBACS is the one verified in [Chen et al. 2014]. This implementation is only 1.08x faster than our C code, at least on the platform on which we tested, which supported vector instructions up to 256 bits. We anticipate that the assembly code may be significantly faster on platforms that support larger 512-bit vector instructions.

AEAD and `secretbox` essentially amount to a ChaCha20/Salsa20 cipher sequentially followed by Poly1305, and their performance reflects the sum of the two primitives. Box uses Curve25519 to compute a symmetric key, which it then uses to encrypt a 16KB input. Here, the cost of symmetric encryption dominates over Curve25519.

In summary, our measurements show that HACL* is as fast as (or faster than) state-of-the-art C crypto libraries and within a small factor of hand-optimized assembly code. This finding is not entirely unexpected, since we wrote our Low* code by effectively porting the fastest C implementations to F*, and any algorithmic optimization that is implemented in C can, in principle, be written (and verified) in Low*. What is perhaps surprising is that we get good performance even though our Low* code, and consequently the generated C, heavily relies on functional programming

[5] https://bench.cr.yp.to/supercop.html

patterns such as tail-recursion, and even though we try to write generic compact code wherever possible, rather than trying to mimic the verbose inlined style of assembly code. We find that modern compilers like GCC and CLANG are able to optimize our code quite well, and we are able to benefit from their advancements, without having to change our coding style. Where needed, KreMLin helps the C compiler by inserting attributes like `const`, `static` and `inline` that act as optimization hints.

Balancing Trust and Performance. All the above performance numbers were obtained with GCC-6 with most architecture-specific optimizations turned on (`-march=native`). Consequently, any bug in GCC or its plugins could break the correctness and security guarantees we proved in F* for our source code. For example, GCC has an auto-vectorizer that significantly improves the performance of our ChaCha20 and Salsa20 code in certain use cases, but does so by substantially changing its structure to take advantage of the parallelism provided by SIMD vector instructions. To avoid trusting this powerful but unverified mechanism, and for more consistent results across platforms, we turned off auto-vectorization (`-fno-tree-vectorize`) for the numbers in Table 2. For similar reasons, we turned off link-time optimization (`-fno-lto`) since it relies on an external linker plugin, and can change the semantics of our library every time it is linked with a new application.

Ideally, we would completely remove the burden of trust on the C compiler by moving to CompCert, but at significant performance cost. Our Salsa20 and ChaCha20 code incurs a relatively modest 3x slowdown when compiled with CompCert 3.0 (with `-O3`). However, our Poly1305 and Curve25519 code incurs a 30-60x slowdown, which makes the use of CompCert impractical for our library. We anticipate that this penalty will reduce as CompCert improves, and as we learn how to generate C code that would be easier for CompCert to optimize. For now, we continue to use GCC and CLANG and comprehensively test the generated code using third-party tools. For example, we test our code against other implementations, and run all the tests packaged with OpenSSL. We also test our compiled code for side-channel leaks using tools like DUDECT.[6]

PneuTube: Fast encrypted file transfer. Using HACL*, we can build a variety of high-assurance security applications directly in Low*. PneuTube is a Low* program that securely transfers files from a host *A* to a host *B* across an untrusted network. Unlike classic secure channel protocols like TLS and SSH, PneuTube is *asynchronous*, meaning that if *B* is offline, the file may be cached at some untrusted cloud storage provider and retrieved later.

PneuTube breaks the file into *blocks* and encrypts each block using the `box` API in HACL* (with an optimization that caches the result of Curve25519). It also protects file metadata, including the file name and modification time, and it hides the file size by padding the file before encryption to a user-defined size. We verify that our code is memory-safe, side-channel resistant, and that it uses the I/O libraries correctly (e.g., it only reads or writes a file or a socket between calling open and close).

PneuTube's performance is determined by a combination of the crypto library, disk access (to read and write the file at each end) and network I/O. Its aynchronous design is particularly rewarding on high-latency network connections, but even when transferring a 1GB file from one TCP port to another on the same machine, PneuTube takes just 6s. In comparison, SCP (using SSH with ChaCha20-Poly1305) takes 8 seconds.

5.2 Cryptographically Secure AEAD for miTLS

We use our cryptographically secure AEAD library (§2.4) within miTLS [Bhargavan et al. 2013], an existing implementation of TLS in F*. In a previous verification effort, AEAD encryption was

[6] https://github.com/oreparaz/dudect

idealized as a cryptographic assumption (concretely realized using bindings to OpenSSL) to show that miTLS implements a secure authenticated channel. However, given vulnerabilities such as CVE-2016-7054, this AEAD idealization is a leap of faith that can undermine security when the real implementation diverges from its ideal behavior.

We integrated our verified AEAD construction within miTLS at two levels [Bhargavan et al. 2017]. First, we replace the previous AEAD idealization with a module that implements a similar ideal interface but translates the state and buffers to Low* representations. This reduces the security of TLS to the PRF and MAC idealizations in AEAD. We integrate AEAD at the C level by substituting the OpenSSL bindings with bindings to the C-extracted version of AEAD. This introduces a slight security gap, as a small adapter that translates miTLS bytes to Low* buffers and calls into AEAD in C is not verified. We confirm that miTLS with our verified AEAD interoperates with mainstream implementations of TLS 1.2 and TLS 1.3 on ChaCha20-Poly1305 ciphersuites.

6 RELATED WORK

Many approaches have been proposed for verifying the functional correctness and security of efficient low-level code. A first approach is to build verification frameworks for C using verification condition generators and SMT solvers [Cohen et al. 2009; Jacobs et al. 2014; Kirchner et al. 2015]. While this approach has the advantage of being able to verify existing C code, this is very challenging: one needs to deal with the complexity of C and with any possible optimization trick in the book. Moreover, one needs an expressive specification language and escape hatches for doing manual proofs in case SMT automation fails. So others have deeply embedded C, or C-like languages, into proof assistants such as Coq [Appel 2015; Beringer et al. 2015; Chen et al. 2016] and Isabelle [Schirmer 2006; Winwood et al. 2009] and built program logics and verification infrastructure starting from that. This has the advantage of using the full expressive power of the proof assistant for specifying and verifying properties of low-level programs. This remains a very labor-intensive task though, because C programs are very low-level and working with a deep embedding is often cumbersome. Acknowledging that uninteresting low-level reasoning was a determining factor in the size of the seL4 verification effort [Klein et al. 2009], Greenaway et al. [2012, 2014] have recently proposed sophisticated tools for automatically abstracting the low-level C semantics into higher-level monadic specifications to ease reasoning. We take a different approach: we give up on verifying existing C code and embrace the idea of writing low-level code in a subset of C shallowly embedded in F*. This shallow embedding has significant advantages in terms of reducing verification effort and thus scaling up verification to larger programs. This also allows us to port to C only the parts of an F* program that are a performance bottleneck, and still be able to verify the complete program.

Verifying the correctness of low-level cryptographic code is receiving increasing attention [Appel 2015; Beringer et al. 2015; Dodds 2016]. The verified cryptographic applications we have written in Low* and use for evaluation in this paper are an order of magnitude larger than most previous work. Moreover, for AEAD we target not only functional correctness, but also cryptographic security.

In order to prevent the most devastating low-level attacks, several researchers have advocated dialects of C equipped with type systems for memory safety [Condit et al. 2007; Jim et al. 2002; Tarditi 2016]. Others have designed new languages with type systems aimed at low-level programming, including for instance linear types as a way to deal with memory management [Amani et al. 2016; Matsakis and Klock II 2014]. One drawback is the expressiveness limitations of such type systems: once memory safety relies on more complex invariants than these type systems can express, compromises need to be made, in terms of verification or efficiency. Low* can perform arbitrarily sophisticated reasoning to establish memory safety, but does not enjoy the benefits of efficient decision procedures [rus 2017] and currently cannot deal with concurrency.

We are not the first to propose writing efficient and verified C code in a high-level language. LMS-Verify [Amin and Rompf 2017] recently extended the LMS meta-programming framework for Scala with support for lightweight verification. Verification happens at the generated C level, which has the advantage of taking the code generation machinery out of the TCB, but has the disadvantage of being far away from the original source code.

Bedrock [Chlipala 2013] is a generative meta-programming tool for verified low-level programming in Coq. The idea is to start from assembly and build up structured code generators that are associated verification condition generators. The main advantage of this "macro assembly language" view of low-level verification is that no performance is sacrificed while obtaining some amount of abstraction. One disadvantage is that the verified code is not portable.

Our companion paper "Implementing and Proving the TLS 1.3 Record Layer" [Bhargavan et al. 2017] is available online. It describes a cryptographic model and proof of security for AEAD using a combination of F* verification and meta-level cryptographic idealization arguments. To make the point that verified code need not be slow, the paper mentions that the AEAD implementation can be "extracted to C using an experimental backend for F*", but makes no further claims about this backend. The current work introduces the design, formalization, implementation, and experimental evaluation of this C backend for F*.

7 CONCLUSION

This paper advocates a new methodology for carrying out high-level proofs on low-level code. By embedding a low-level language and memory model within F*, the programmer not only enjoys sophisticated proofs but also gets to write their low-level code in a more modular style, using features functional programmers take for granted, including recursion and type abstraction. Our toolchain, relying on partial evaluation and the latest advances in C compilers, shows that we can write code in a style suitable for verification *and* enjoy the same performance as hand-written C code.

We are currently making progress in three different directions. First, continuing our integration of AEAD within miTLS, we aim to port the miTLS protocol layer to Low*, in order to get an entire verified, TLS library in C. Second, parts of our toolchain are unverified. We plan to formalize and verify using F* parts of the KreMLin tool, notably the λow* to C* transformation. Third, we are working on embedding assembly instructions within Low*, allowing us to selectively optimize our code further towards closing the performance gap that still remains relative to architecture-specific, hand-written assembly routines.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their excellent reviews. We also thank Abhishek Anand and Mike Hicks, for useful feedback and discussion which helped shape the work presented here, as well as Armaël Guéneau, for his work on a mechanized proof.

REFERENCES

2008–2017. The Sodium crypto library (libsodium). (2008–2017). https://www.gitbook.com/book/jedisct1/libsodium/details
2010–2017. The Rust Programming Language. (2010–2017). https://www.rust-lang.org
2016. CVE-2016-7054: ChaCha20/Poly1305 heap-buffer-overflow. (Nov. 2016). http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-7054
2017. Common Weakness Enumeration (CWE-190: Integer Overflow or Wraparound). (2017). https://cwe.mitre.org/data/definitions/190.html
2017. Common Weakness Enumeration (CWE-415: Double Free). (2017). http://cwe.mitre.org/data/definitions/415.html
2017. Common Weakness Enumeration (CWE-416: Use After Free). (2017). http://cwe.mitre.org/data/definitions/416.html
J. Afek and A. Sharabani. 2007. Dangling Pointer – Smashing The Pointer For Fun And Profit. BlackHat USA. (July 2007).

Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. 2017. Dijkstra Monads for Free. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 515–529. DOI: http://dx.doi.org/10.1145/3009837.3009878

Nadhem J. AlFardan and Kenneth G. Paterson. 2013. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *2013 IEEE Symposium on Security and Privacy*. 526–540.

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2016a. Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC. In *Fast Software Encryption - 23rd International Conference, FSE 2016*. 163–184.

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016b. Verifying Constant-Time Implementations. In *25th USENIX Security Symposium, USENIX Security 16*. 53–70.

Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, and others. 2016. COGENT: Verifying High-Assurance File System Implementations. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 175–188.

Nada Amin and Tiark Rompf. 2017. LMS-Verify: Abstraction without Regret for Verified Systems Programming. To appear in 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'17). (2017). https://www.cs.purdue.edu/homes/rompf/papers/amin-draft2016b.pdf

Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.* 37, 2 (2015), 7.

Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. 2014. System-level Non-interference for Constant-time Cryptography. In *2014 ACM SIGSAC Conference on Computer and Communications Security, CCS 2014*. 1267–1279.

David Benjamin. 2016. poly1305-x86.pl produces incorrect output. https://mta.openssl.org/pipermail/openssl-dev/2016-March/006161. (2016).

Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. 2002. Reconsidering Custom Memory Allocation. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA 2002*. ACM, 1–12.

Lennart Beringer, Adam Petcher, Q Ye Katherine, and Andrew W Appel. 2015. Verified correctness and security of OpenSSL HMAC. In *24th USENIX Security Symposium (USENIX Security 15)*. 207–221.

Daniel J Bernstein. 2005. The Poly1305-AES message-authentication code. In *International Workshop on Fast Software Encryption*. Springer, 32–49.

Daniel J Bernstein. 2006. Curve25519: new Diffie-Hellman speed records. In *International Workshop on Public Key Cryptography*. Springer, 207–228.

Daniel J Bernstein. 2008. The Salsa20 family of stream ciphers. In *New stream cipher designs*. Springer, 84–97.

Daniel J Bernstein, Tanja Lange, and Peter Schwabe. 2012. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America, LATINCRYPT 2012*. Springer, 159–176.

Daniel J Bernstein, Bernard Van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. 2014. TweetNaCl: A crypto library in 100 tweets. In *International Conference on Cryptology and Information Security in Latin America, LATINCRYPT 2014*. 64–83.

Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, , Alfredo Pironti, and Pierre-Yves Strub. 2014. Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS. In *2014 IEEE Symposium on Security and Privacy*. 98–113.

Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jianyang Pan, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella Béguelin, and Jean Karim Zinzindohoue. 2017. Implementing and Proving the TLS 1.3 Record Layer. *IEEE Security & Privacy* (2017).

Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and P Strub. 2013. Implementing TLS with verified cryptographic security. In *IEEE Symposium on Security and Privacy*. 445–459.

Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Santiago Zanella-Béguelin. 2014. Proving the TLS handshake secure (as it is). In *Advances in Cryptology–CRYPTO 2014*. Springer, 235–255.

Karthikeyan Bhargavan and Gaëtan Leurent. 2016. On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN. Cryptology ePrint Archive, Report 2016/798. (2016). http://eprint.iacr.org/2016/798.

Sandrine Blazy and Xavier Leroy. 2009. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning* 43, 3 (2009), 263–288.

Hanno Böck. 2016. Wrong results with Poly1305 functions. https://mta.openssl.org/pipermail/openssl-dev/2016-March/006413. (2016).

Hanno Böck, Aaron Zauner, Sean Devlin, Juraj Somorovsky, and Philipp Jovanovic. 2016. Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS. Cryptology ePrint Archive, Report 2016/475. (2016). http://eprint.iacr.org/

2016/475.

Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *Proceedings of the USENIX Security Symposium*.

Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2016. Toward compositional verification of interruptible OS kernels and device drivers. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016*. 431–447.

Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. 2014. Verifying curve25519 software. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 299–309.

Adam Chlipala. 2013. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 391–402.

Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A practical system for verifying concurrent C. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 23–42.

Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C Necula. 2007. Dependent types for low-level programming. In *European Symposium on Programming*. Springer, 520–535.

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS (Lecture Notes in Computer Science)*, Vol. 4963. Springer, 337–340. DOI: http://dx.doi.org/10.1007/978-3-540-78800-3_24

I. Dobrovitski. 2003. Exploit for CVS double free() for Linux pserver. (Feb. 2003). http://archives.neohapsis.com/archives/fulldisclosure/2003-q1/0545.html

Robert W. Dockins. 2012. *Operational Refinement for Compiler Correctness*. Ph.D. Dissertation. Princeton University.

Joey Dodds. 2016. Part one: Verifying s2n HMAC with SAW. Galois Blog. (Sept. 2016). https://galois.com/blog/2016/09/specifying-hmac-in-cryptol/

Thai Duong and Juliano Rizzo. 2011. Here Come The ⊕ Ninjas. Available at http://nerdoholic.org/uploads/dergln/beast_part2/ssl_jun21.pdf. (May 2011).

Anthony Green. 2014. The libffi home page. (2014). http://sourceware.org/libffi

David Greenaway, June Andronick, and Gerwin Klein. 2012. Bridging the Gap: Automatic Verified Abstraction of C. In *3rd International Conference on Interactive Theorem Proving, ITP 2012 (Lecture Notes in Computer Science)*, Vol. 7406. Springer, 99–115.

David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. 2014. Don't sweat the small stuff: formal verification of C code without the pain. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014*. ACM, 429–439.

Heartbleed. 2014. The Heartbleed Bug. http://heartbleed.com/. (2014).

Bart Jacobs, Jan Smans, and Frank Piessens. 2014. The VeriFast Program Verifier: A Tutorial. iMinds-DistriNet, Department of Computer Science, KU Leuven - University of Leuven, Belgium. (2014). https://people.cs.kuleuven.be/~bart.jacobs/verifast/tutorial.pdf

Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C.. In *USENIX Annual Technical Conference, General Track*. 275–288.

David Kaloper-Meršinjak, Hannes Mehnert, Anil Madhavapeddy, and Peter Sewell. 2015. Not-quite-so-broken TLS: Lessons in re-engineering a security protocol specification and implementation. In *24th USENIX Security Symposium (USENIX Security 15)*. 223–238.

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Asp. Comput.* 27, 3 (2015), 573–609.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the Symposium on Operating Systems Principles*. ACM, 207–220.

Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology – CRYPTO 1996*. Springer, 104–113.

Xavier Leroy. 2004–2016. The CompCert C verified compiler. http://compcert.inria.fr/. (2004–2016).

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2*. Research report RR-7987. INRIA. http://hal.inria.fr/hal-00703441

Pierre Letouzey. 2002. A new extraction for Coq. In *Types for proofs and programs*. Springer, 200–219.

Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *4th Conference on Computability in Europe (Lecture Notes in Computer Science)*, Vol. 5028. Springer, 359–369. DOI: http://dx.doi.org/10.1007/978-3-540-69407-6_39

Nicholas D Matsakis and Felix S Klock II. 2014. The Rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.

John McCarthy. 1962. Towards a Mathematical Science of Computation. In *IFIP Congress*. 21–28.

Microsoft Research and INRIA. 2016. Everest: VERifiEd Secure Transport. https://project-everest.github.io/. (2016).

Bodo Möller, Thai Duong, and Krzysztof Kotowicz. 2014. This POODLE Bites: Exploiting The SSL 3.0 Fallback. Available at https://www.openssl.org/~bodo/ssl-poodle.pdf. (2014).

David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2006. The Program Counter Security Model: Automatic Detection and Removal of Control-flow Side Channel Attacks. In *8th International Conference on Information Security and Cryptology, ICISC 2005*. Springer, 156–168.

Yoav Nir and Adam Langley. 2015. ChaCha20 and Poly1305 for IETF Protocols. IETF RFC 7539. (2015).

nocrypto. 2014–2017. nocrypto: OCaml cryptographic library. (2014–2017). https://github.com/mirleft/ocaml-nocrypto

OpenSSL library. 1998–2017. OpenSSL: Cryptography and SSL/TLS Toolkit. (1998–2017). https://www.openssl.org/

Jonathan D. Pincus and Brandon Baker. 2004. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns. *IEEE Security & Privacy* 2, 4 (2004), 20–27.

Jonathan Protzenko. 2017. The KreMLin compiler. (2017). https://www.github.com/FStarLang/kremlin

Julian Rizzo and Thai Duong. 2012. The CRIME Attack. (September 2012).

Norbert Schirmer. 2006. *Verification of sequential imperative programs in Isabelle-HOL*. Ph.D. Dissertation. Technical University Munich.

Ben Smyth and Alfredo Pironti. 2014. *Truncating TLS Connections to Violate Beliefs in Web Applications*. Technical Report hal-01102013. Inria. https://hal.inria.fr/hal-01102013

Juraj Somorovsky. 2016. Systematic fuzzing and testing of TLS libraries. In *23rd ACM Conference on Computer and Communications Security, CCS 2016*.

Marc Stevens, Pierre Karpman, and Thomas Peyrin. 2016. Freestart Collision for Full SHA-1. In *Advances in Cryptology – EUROCRYPT 2016*. Springer, 459–483.

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. https://www.fstar-lang.org/papers/mumon/

Robert Święcki. 2016. ChaCha20/Poly1305 heap-buffer-overflow. CVE-2016-7054. (2016).

Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 48–62.

David Tarditi. 2016. Extending C with bounds safety. Checked C Technical Report, Version 0.6. (Nov. 2016). https://github.com/Microsoft/checkedc

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Inf. Comput.* 132, 2 (Feb. 1997), 109–176.

A. Tomb. 2016. Automated Verification of Real-World Cryptographic Implementations. *IEEE Security Privacy* 14, 6 (2016), 26–33.

David Wagner and Bruce Schneier. 1996. Analysis of the SSL 3.0 Protocol. In *2nd USENIX Workshop on Electronic Commerce, WOEC 1996*. 29–40.

Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. 2009. Mind the Gap. In *22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009 (Lecture Notes in Computer Science)*, Vol. 5674. Springer, 500–515.

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 427–440.

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2013. Formal verification of SSA-based optimizations for LLVM. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 175–186.

Jean Karim Zinzindohoué, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. 2016. A Verified Extensible Library of Elliptic Curves. In *IEEE Computer Security Foundations Symposium (CSF)*.

Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. Cryptology ePrint Archive, Report 2017/536. (2017). http://eprint.iacr.org/2017/536.

Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. (2017). https://www.github.com/mitls/hacl-star