Verified Low-Level Programming Embedded in F*

Karthikeyan Bhargavan[2] Antoine Delignat-Lavaud[3] Cédric Fournet[3] Cătălin Hriţcu[2]
Jonathan Protzenko[3] Tahina Ramananandro[3] Aseem Rastogi[3] Nikhil Swamy[3] Peng Wang[1]
Santiago Zanella-Beguelin[3] Jean-Karim Zinzindohoué[2]

[1]MIT CSAIL [2]INRIA Paris [3]Microsoft Research

Abstract

We present Low*, a language for efficient low-level programming and verification, and its application to high-assurance 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* has a structured memory model à la CompCert; it does not involve garbage collection or implicit heap allocations; and it provides instead the control required for writing low-level security-critical code.

By virtue of typing, any Low* program is memory safe. On top of this, 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 have implemented and verified cryptographic algorithms, constructions, and tools in Low*, for a total of about 20,000 lines of code. Our code delivers performance competitive with existing (unverified) C cryptographic libraries, suggesting our approach may be applicable to larger-scale, verified, low-level software.

1. Introduction

Cryptographic software widely deployed throughout the internet is still often subject to security-critical attacks [1–4, 10, 12, 27, 29, 32, 33, 38, 41, 56, 59, 60, 62–64, 66, 67, 70]. 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 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. [46] provide nqsbTLS, an implementation of TLS in OCaml, which by virtue of its type and memory safety is impervious to attacks like Heartbleed [2] that exploit buffer overflows. Bhargavan et al. [30] 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 [6], an implementation that mixes C and OCaml, and libcrypto, a component of the widely used OpenSSL library [7]. Linking with vulnerable C code could, in the worst case, void all the security 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. Towards this end, we introduce Low*, a domain-specific language for verified, efficient low-level programming, embedded within F* [65], 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 F*'s verification capabilities to prove the code 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 initial experiments indicate significant speedups when linking Low* libraries compiled to C with F* programs compiled to OCaml, without compromising 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 [51, 52], not the "big array of bytes" model systems programmers sometimes

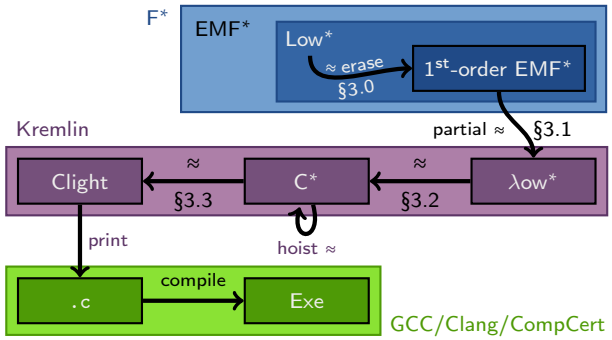

Figure 1. Low* embedded in F*, compiled to C, with soundness and security guarantees (details in §3)

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 computationally irrelevant parts of a program, e.g., proofs and specifications, using a process similar to Coq's extraction mechanism [53].

Importantly, Low* programs have a second, equivalent but more efficient semantics via compilation to C, with a predictable performance model including manual memory management—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. We instantiate the state monad of F* to use a CompCert-like structured memory model that separates the stack and the heap, supporting en masse allocation and deallocation on the stack, and allocating and freeing individual reference cells on the heap. The heap is organized into disjoint logical regions, which enables stating separation properties necessary for modular, stateful verification. On top of this, we program a library of buffers—C-style arrays passed by reference—with support for pointer arithmetic and referring to only part of an array. 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. Inasmuch as possible, we aim for the programmer to have control even over the syntactic structure of the target 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 satisfy several requirements. Specifically, the code (1) must be first order, to prevent the need to allocate closures in C; (2) must not perform any implicit allocations; (3) must not use recursive datatypes, 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. 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 give a translation from Low* to CompCert's Clight [31] and show that it preserves trace equivalence with respect to the original F* semantics of the program. In addition to ensuring that the functional behavior of a program is preserved, our trace equivalence also guarantees that the compiler does not introduce unexpected side-channels due to memory access patterns, at least until it reaches Clight—a useful sanity check for cryptographic code. Our formal results cover the translation of standalone Low* programs to C, proving that execution in C preserves the original F* semantics of the Low* program. More pragmatically, we have built several cryptographic libraries in Low*, compiled them to C, and integrated them within larger programs compiled to OCaml, relying on OCaml's foreign function interface for interoperability and trusting it for correctness.

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.

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

We implement in around 14,000 lines of code the Authenticated Encryption with Associated Data (AEAD) construction at the heart of the TLS-1.3 record layer—we prove memory safety, functional correctness, and cryptographic security. We also implement a robust API and side-channel protections through type abstraction. We compile our Low* AEAD code to both OCaml and C, linking it with the F* implementation of miTLS compiled to OCaml. As expected by the design of Low*, relative to the full OCaml version, miTLS linked with the C version of our libraries shows an improvement in throughput of data transfer by at least two orders of magnitude. There is still room for improvement, however—one measure shows our verified code to still be around 5× slower than OpenSSL's libcrypto. (§5.2)

We also provide HACL*, a "high-assurance crypto library" in Low*, implementing and proving several cryptographic

algorithms, including the Poly1305 MAC [22], the ChaCha20 cipher [58], and Curve25519 [23]. We package these algorithms to provide the popular `box/box_open` NaCl API [25], yielding the first performant implementation of NaCl that is verified for memory safety and side-channel resistance, including the first verified bigint libraries customized for safe, fast cryptographic use. Finally, on top of this API, we build *PneuTube*, a secure, asynchronous, file transfer application whose performance compares favorably with widely used, existing utilities like scp. (§5.1)

Supplementary materials Full formalization and hand proofs of all the theorems in this paper are available in the long version included with this submission. We also provide the KreMLin compiler and our verified Low* code for AEAD, HACL*, and PneuTube as supplementary materials.

2. A Low* tutorial

At the core of Low* is a library for programming with buffers manually allocated on the stack or heap (§2.2). Memory safety demands reasoning about the extents and liveness of these buffers, 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 two examples from our codebase: the ChaCha20 stream cipher [58], which we prove memory safe (§2.1), and the Poly1305 MAC [22], which we prove functionally correct (§2.3). We finally explain how we prove a combination of ChaCha20 and Poly1305 cryptographically secure (§2.4).

2.1 A first example: the ChaCha20 stream cipher

Figure 2 shows code snippets for the main function of ChaCha20 [58], a modern stream cipher widely used for fast symmetric encryption. 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 state on the stack; then calls a function to initialize the cipher block from the inputs (passing an inner pointer to the state); and finally 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 4–5, the C code is in one-to-one correspondence with its Low* counterpart. These specifications capture the safe memory usage of chacha20: for each argument passed by reference, and for the auxiliary state, they keep track of their liveness and size.

Line 2 uses two 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. At the beginning of line 4, Stack unit 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 same line, the pre-condition requires that all arguments passed by reference be live. On line 5, the post-condition 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.

As usual for symmetric ciphers, RFC 7539 specifies chacha20 as imperative pseudocode, and does not further discuss its mathematical properties. Our Low* code and its C extraction closely follow the RFC, and yield the same results on its test vectors. Although we could write a pure F* specification (further away from the RFC) and verify that our code implements it, it would not add much here.

2.2 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 inner pointers for arrays. 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 $\mathsf{M}\ \mathsf{t}_1 \ldots \mathsf{t}_n$ where M is an effect label and $\mathsf{t}_1 \ldots \mathsf{t}_n$ are *value types*. For example, Stack unit (...) (...) on lines 4–5 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 di-

```
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)) =
    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);
}
```

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

verges.[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 [69] covering both stack and heap. Hyper-stacks are a generalization of hyper-heaps, a memory model proposed previously for F* [65], 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 [55] 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]))
```

Since we support freeing individual references within a region, our model of regions could seem similar to Berger et al.'s [20] *reaps*. However, at present, we do no 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 [65], 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

[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.

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, by making use of 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 push), and which produce a final memory that has an equal domain to the initial memory. This ensures that Low* code with Stack effect has explicitly deallocated all heap allocated references before returning, effectively ruling out memory leaks. As such, we expect all Low* functions callable from F* through the OCaml FFI to have Stack effect. The F* code can safely pass pointers to objects allocated in the garbage-collected OCaml heap into Low* code with Stack effect since the definition of the Stack effect disallows the Low* code from freeing these references.

Modeling buffers 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. Leaving a proper low-level modeling of structs, including pointers inside structures, as future work, we focus on arrays and implement an abstract type of buffers in Low* using just the references provided by hyper-stacks. Relying on its abstraction, KreMLin compiles buffers to native C arrays. We sketch the library next.

The type buffer a below is a single-constructor inductive type whose arguments depend on each other. Its main content field holds a reference to a seq a, a purely functional sequence of a's, whose length is determined by the first field, max_length. A 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 other two fields 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 8 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' only covers 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' records the length of b' and, using the includes relation, shows that b and b' are aliased.

```
val sub: b:buffer a → i:uint32{i + b.idx < pow2 32}
  → 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 index 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))
```

2.3 Functional correctness and side-channel resistance

This section and the next illustrates our "high-level verification for low-level code" methodology. Although programming at a low-level, we rely on high-level features of F*, including 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 [22], 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. [72]; 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

[2] Implementation bugs in Poly1305 are a reality: in 2016 alone, the Poly1305 OpenSSL implementation experienced two correctness bugs [19, 32] and a buffer overflow [9].

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 [13, 14]. We sketch our methodology on a small snippet from our specialized arithmetic (bigint) library upon which we built Poly1305.

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 (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 [49]. 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 eq_mask operation 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 [65]. 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*

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

Figure 3. Unique representation of a Poly1305 bigint

$2^{130} - 5$. 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.

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 $\varepsilon_{\mathrm{ChaCha20}}$ and $\varepsilon_{\mathrm{Poly1305}}$. The argument then shows that the AEAD construction also meets its own ideal functionality, except with probability say $\epsilon_{\mathrm{Chacha20}} + \epsilon_{\mathrm{Poly1305}}$.

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.

```
1 type entry = cipher * data * nonce * plain
2 abstract type key =
3   { key: keyBytes; log: if Flag.aead then ref (seq entry) else unit }
4 let encrypt (k:key) (n:nonce) (p:plain) (a:data) =
5   if Flag.aead
6   then let c = random_bytes ℓ_c in k.log := (c, a, n, p) :: k.log; c
7   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.

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. The F* normalizer then e.g. drops the then branch, and replaces 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 e.g. 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.

3. A formal translation from Low* to Clight

Figure 1 on page 2 provides an overview of our translation from Low* to CompCert Clight, starting with EMF*, a recently proposed model of F* [11]; then λow*, a formal core of Low*; 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) 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 [11] 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.

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

Figure 4. λow* syntax

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 [57] augmented to track potential leaks through cache behavior [18]. 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) passed by reference, where b is the buffer address and n is the offset in the buffer. Stack allocated buffers (readbuf, writebuf, newbuf, and subbuf) 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 become inaccessible.

Type system λow* types include the base types int and unit, record types $\{\overrightarrow{f = \tau}\}$, buffer types buf 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 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, and branching behavior, as shown below.

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

Figure 5 shows selected reduction rules from λow* at the left (in contrast with the reduction rules of C*, discussed in the next section). 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 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. Rule APP is a standard, substitutive β-reduction.

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 ($\Gamma = \mathsf{secret{:}limb}$), and empty store typing, using the judgment $\Gamma_{\mathsf{limb}}, \Gamma_p; \cdot; \Gamma \vdash e : \tau$. 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 $e[v_0/\mathsf{secret}]$ and $e[v_1/\mathsf{secret}]$ 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 1 (Secret independence)**.** *Given*

1. *A program well-typed against a secret interface, Γ_s, i.e, $\Gamma_s, \Gamma_P; \Sigma; \Gamma \vdash (H, e) : \tau$, where e is not a value.*
2. *A well-typed implementation of the interface Δ, P_s (i.e., $\Gamma_p; \Sigma; \cdot \vdash_\Delta P_s$), such that P_s is equivalent modulo secrets.*
3. *A pair (ρ_1, ρ_2) of well-typed substitutions for Γ.*

There exists $\ell, \Sigma' \supseteq \Sigma, \Gamma', H', e'$ and a pair (ρ_1', ρ_2') of well-typed substitutions for Γ', such that

1. *$P_s, P \vdash (H, e)[\rho_1] \rightarrow_\ell^* (H', e')[\rho_1']$ if and only if, $P_s, P \vdash (H, e)[\rho_2] \rightarrow_\ell^* (H', e')[\rho_2']$, and*
2. *$\Gamma_s, \Gamma_P; \Sigma'; \Gamma' \vdash (H', e') : \tau$*

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.

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

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 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 5, 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

$$\frac{}{P \vdash (H, \mathsf{withframe}\ e) \to_{\cdot} (H; \{\}, \mathsf{pop}; e)}\ \textsc{WF}$$

$$\frac{}{P \vdash (H; \_, \mathsf{pop}\ v) \to_{\cdot} (H, v)}\ \textsc{Pop}$$

$$\frac{}{P \vdash (H, \mathsf{if}\ 0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) \to_{\mathsf{brF}} (H, e_2)}\ \textsc{LIfF}$$

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

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

$$\frac{P(f) = \lambda y : \tau_1.\ e_1 : \tau_2}{P \vdash (H, \mathsf{let}\ x : \tau = f\ v\ \mathsf{in}\ e) \to (H, \mathsf{let}\ x : \tau = e_1[v/y]\ \mathsf{in}\ e)}\ \textsc{App}$$

$$\frac{}{\hat{P} \vdash (S, V, \{\overrightarrow{s_1}\}; \overrightarrow{s_2}) \rightsquigarrow (S; (\{\}, V, \square; \overrightarrow{s_2}), V, \overrightarrow{s_1})}\ \textsc{Block}$$

$$\frac{}{\hat{P} \vdash (S; (M, V', E), V, []) \rightsquigarrow (S, V', E\ [()])}\ \textsc{Empty}$$

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

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

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

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

Figure 5. Selected semantic rules from λow* (left) and C* (right)

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 1 (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., $\hat{P} \vdash ([], V, \overrightarrow{s}; \mathsf{return}\ \hat{e}) \overset{\ell,*}{\to} ([], V', \mathsf{return}\ v)$ if, and only if, so does the λow^* program: $P \vdash (\{\}, e[V]) \overset{\ell,*}{\to} (H', v)$; 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 [31, 50] 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]

We prepend each memory access and conditional statement in the Clight generated code with a *built-in call*, a no-op annotation whose only purpose is to produce an event in the trace. The main two differences between C* and Clight, which our translation deals with as described below, are local structures, and scope management for local variables.

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

If we naively translate 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 a 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

[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.

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' [39, §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 memory-changing pass, we first reinterpret the source C* program with different event traces, before carrying out the actual transformation in a trace-preserving way. Our detailed functional correctness and security preservation proofs from λow* to Clight can be found in the appendix.

Towards assembly code We claim 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 using instrumented CompCert and LLVM [14, 18] turned 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. By contrast, applying our proof-preservation techniques to CompCert aims to avoid this further compile-time check, eliminating the risk of compilation failures.

4. KreMLin: A compiler from Low* to C

The erasure and extraction transformations described in the preceding section were already implemented as part of F*'s pre-existing OCaml extraction facility. We wrote a new tool called KreMLin that takes the extracted output from F*, then further rewrites it until it falls within the λow* subset formalized above; after which the tool performs the Low* to C* transformation; the C* to C transformation; the pretty-printing to a set of C files that can be compiled. KreMLin generates C99 code that may be compiled by GCC; Clang; Microsoft's C compiler or CompCert.

KreMLin accepts a larger input language than λow*; for the sake of programmer productivity, KreMLin handles non-recursive, parameterized data types and type abbreviations; pattern matches; tuples; nested let-bindings, assignments and conditionals. A combination of monomorphization, translation of data types to tagged unions and compilation of pattern matches reduce these constructs to λow*. Of particular interest are *stratification* and *hoisting*. Stratification goes from an expression language to a statement language of the form expected by C*, meaning that buffer allocations, assignments and conditionals are placed 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.

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; data types with only constant constructors generate an `enum` and are destructured by a `switch`; 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.

KreMLin represents about 6,800 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 accept multiple frontends and support multiple backends. We intend to announce one new backend and one new frontend soon.

5. Building verified Low* applications

Application	LoC	C LoC	%annot	Verif. time
HACL*	6,050	11,220	28%	0h 52m
AEAD	13,743	14,292	56.5%	1h 10m

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

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 implement 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. Second, 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*: Fast and safe cryptographic library

In the wake of numerous security vulnerabilities, Bernstein et al. [25] 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 [23] for key exchange, the Salsa family of symmetric encryption

algorithms [24], which includes XSalsa20 and ChaCha20, and Poly1305 for message authentication [22]. 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 [5], 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 [26] 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.

We implement the NACL API, including all its component algorithms, in a Low* library called HACL*, mechanically verifying that our code is memory safe 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.

Algorithm	HACL*	Sodium	TweetNaCl
Poly1305	2.2 cycle/B	2.15 cycle/B	27.5 cycle/B
XSalsa20	8.6 cycle/B	7.2 cycle/B	10.1 cycle/B
Box	10.8 cycle/B	9.5 cycle/B	37.5 cycle/B
Curve25519	281 µs/mul	59 µs/mul	522 µs/mul

Table 2. Performance Comparison: 64-bit HACL*, 64-bit Sodium (pure C, no assembly), and TweetNaCl, all compiled using `gcc -O3` and run on an Intel Core i7-4600U CPU

Performance Table 2 compares the performance of HACL* to TweetNaCl and Sodium by running each primitive on a 1MB input. For Curve25519, we measure the time taken for one call to scalar multiplication. Box uses Curve25519 to compute a symmetric key, which it then uses to encrypt a 1MB input. (In this case, the cost of symmetric encryption dominates over Curve25519.) We report averages over 1000 iterations expressed in cycles/byte. We observe that for Poly1305 and XSalsa20, HACL* achieves comparable performance to Sodium's optimized C code and significantly better performance than TweetNaCl's concise C implementation. Sodium also offers an assembly implementation of XSalsa20 which is 3 times faster than the C version. Our Curve25519 implementation is about 5 times slower than Sodium, and we pay this performance penalty in exchange for functional correctness, as explained below.

The cost of verification Poly1305 and Curve25519 implementations rely on bigint libraries that implement modular arithmetic over prime fields in terms of operations over bitstrings. For efficiency, and to mitigate certain side-channels, they cannot use generic bigint libraries; instead they implement custom libraries optimized for 8-bit, 32-bit, and 64-bit architectures. For Curve25519, HACL* includes a side-channel resistant, 64-bit bigint proven correct against its mathematical specification. Proving functional correctness required code restructuring and the introduction of auxiliary values in order to express and prove lemmas, and these changes account for our performance loss. Even so, our Curve25519 code is two orders of magnitude faster than the 20ms reported for a verified OCaml implementation of Curve25519 [72]. The 64-bit Poly1305 code in Table 2 is only verified for memory safety and side-channels, and consequently pays no penalty. We additionally verified for correctness a 32-bit bigint library for Poly1305 and it experiences a similar performance degradation in comparison to 32-bit Sodium, taking 16.0 cycles/byte versus 5.2 cycles/byte for the unverified code. We plan to continue to optimize our code and proofs. All the above results were obtained with GCC; to use verified assembly code via CompCert, we pay an additional cost. For example, our 64-bit XSalsa20, when compiled via CompCert incurs a 6.1x slowdown over GCC.

PneuTube: Fast encrypted file transfer Efficient security applications are typically written in C, not just for cryptography, but also for access to low-level libraries for disk and network I/O. We use Low* to provide high-assurance C libraries like HACL* that can be included within unverified C projects, but we can also write standalone 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. Hence, PneuTube provides better traffic analysis protection than SSH and TLS, which leak file length.

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. We link PneuTube to standard TCP sockets, and to a memory-mapped file I/O library written in C, and we carefully choose the block-size to balance the load across our encryption/decryption pipelines. 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 a file between calling open and close.)

Since PneuTube is asynchronous, a sender does not have to wait for a key exchange to be complete before it starts sending data. This 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 Fast cryptographically secure AEAD for miTLS

We use our cryptographically secure AEAD library (§2.4) within miTLS [28], an existing implementation of TLS in F*. In a previous verification effort, AEAD encryption was 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. 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 approach is perfectly secure, in as much as the security of TLS is now reduced to the PRF and MAC idealizations in AEAD when extracted to OCaml. For performance, we also 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. The OCaml-extracted version only achieves a 74 KB/s throughput on a 1GB file transfer. Thus, while it is useful as a reference for verification, it is not usable in practice. In C, we achieve a 69 MB/s throughput, which is about 20% of the 354 MB/s obtained with OpenSSL. There is a noticeable cost stemming from the AEAD proof infrastructure, such as indirections needed for algorithmic agility. Better specialization support in KreMLin may help reduce this overhead.

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 [36, 44, 47], 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 [17, 21, 34] and Isabelle [61, 71] 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 [48], Greenawayet al. [42, 43] 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 [17, 21, 40]. 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 [37, 45, 68]. 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 [15, 54]. 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 [8] 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 [16] 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 that the verification happens very far away from the original source code.

Bedrock [35] 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.

A concurrent submission to Oakland, "Implementing and Proving the TLS 1.3 Record Layer", is included as an anonymous supplementary material. 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 Oakland submission 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*.

References

[1] Common weakness enumeration (CWE-415: Double free). URL `http://cwe.mitre.org/data/definitions/415.html`.

[2] The Heartbleed bug. `http://heartbleed.com/`.

[3] Common weakness enumeration (CWE-190: Integer overflow or wraparound). URL `https://cwe.mitre.org/data/definitions/190.html`.

[4] Common weakness enumeration (CWE-416: Use after free). URL `http://cwe.mitre.org/data/definitions/416.html`.

[5] The sodium crypto library (libsodium). URL `https://www.gitbook.com/book/jedisct1/libsodium/details`.

[6] nocrypto: OCaml cryptographic library. URL `https://github.com/mirleft/ocaml-nocrypto`.

[7] OpenSSL: Cryptography and SSL/TLS toolkit. URL `https://www.openssl.org/`.

[8] The Rust programming language. URL `https://www.rust-lang.org`.

[9] CVE-2016-7054: ChaCha20/Poly1305 heap-buffer-overflow, Nov. 2016. URL `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-7054`.

[10] J. Afek and A. Sharabani. Dangling pointer – smashing the pointer for fun and profit. BlackHat USA, July 2007.

[11] D. Ahman, C. Hriţcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy. Dijkstra monads for free. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Jan. 2017. URL `https://www.fstar-lang.org/papers/dm4free/`. To appear.

[12] N. J. AlFardan and K. G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540, 2013.

[13] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir. Verifiable side-channel security of cryptographic implementations: Constant-time MEE-CBC. In *Fast Software Encryption - 23rd International Conference, FSE 2016*, pages 163–184, 2016.

[14] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium, USENIX Security 16*, pages 53–70, 2016.

[15] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, et al. COGENT: Verifying high-assurance file system implementations. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–188. ACM, 2016.

[16] N. Amin and T. Rompf. 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. URL `https://www.cs.purdue.edu/homes/rompf/papers/amin-draft2016b.pdf`.

[17] A. W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Trans. Program. Lang. Syst.*, 37(2):7, 2015.

[18] G. Barthe, G. Betarte, J. D. Campo, C. D. Luna, and D. Pichardie. System-level non-interference for constant-time cryptography. In *2014 ACM SIGSAC Conference on Computer and Communications Security, CCS 2014*, pages 1267–1279, 2014.

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

[20] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA 2002*, pages 1–12. ACM, 2002.

[21] L. Beringer, A. Petcher, Q. Y. Katherine, and A. W. Appel. Verified correctness and security of OpenSSL HMAC. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 207–221, 2015.

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

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

[24] D. J. Bernstein. The Salsa20 family of stream ciphers. In *New stream cipher designs*, pages 84–97. Springer, 2008.

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

[26] D. J. Bernstein, B. Van Gastel, W. Janssen, T. Lange, P. Schwabe, and S. Smetsers. TweetNaCl: A crypto library in 100 tweets. In *International Conference on Cryptology and Information Security in Latin America, LATINCRYPT 2014*, pages 64–83, 2014.

[27] K. Bhargavan and G. Leurent. 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`.

[28] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with verified cryptographic security. In *IEEE Symposium on Security and Privacy*, pages 445–459, 2013.

[29] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, , A. Pironti, and P.-Y. Strub. Triple handshakes and cookie cutters: Breaking

and fixing authentication over TLS. In *2014 IEEE Symposium on Security and Privacy*, pages 98–113, 2014.

[30] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Zanella-Béguelin. Proving the TLS handshake secure (as it is). In *Advances in Cryptology–CRYPTO 2014*, pages 235–255. Springer, 2014.

[31] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43 (3):263–288, 2009.

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

[33] H. Böck, A. Zauner, S. Devlin, J. Somorovsky, and P. Jovanovic. Nonce-disrespecting adversaries: Practical forgery attacks on GCM in TLS. Cryptology ePrint Archive, Report 2016/475, 2016. `http://eprint.iacr.org/2016/475`.

[34] H. Chen, X. N. Wu, Z. Shao, J. Lockerman, and R. Gu. Toward compositional verification of interruptible OS kernels and device drivers. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016*, pages 431–447, 2016.

[35] A. Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *ACM SIGPLAN Notices*, volume 48, pages 391–402. ACM, 2013.

[36] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *International Conference on Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

[37] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *European Symposium on Programming*, pages 520–535. Springer, 2007.

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

[39] R. W. Dockins. *Operational Refinement for Compiler Correctness*. PhD thesis, Princeton University, 2012.

[40] J. Dodds. Part one: Verifying s2n HMAC with SAW. Galois Blog, Sept. 2016. URL `https://galois.com/blog/2016/09/specifying-hmac-in-cryptol/`.

[41] T. Duong and J. Rizzo. Here come the ⊕ ninjas. Available at `http://nerdoholic.org/uploads/dergln/beast_part2/ssl_jun21.pdf`, May 2011.

[42] D. Greenaway, J. Andronick, and G. Klein. Bridging the gap: Automatic verified abstraction of C. In *3rd International Conference on Interactive Theorem Proving, ITP 2012*, volume 7406 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2012.

[43] D. Greenaway, J. Lim, J. Andronick, and G. Klein. 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*, pages 429–439. ACM, 2014.

[44] B. Jacobs, J. Smans, and F. Piessens. The VeriFast program verifier: A tutorial. iMinds-DistriNet, Department of Computer Science, KU Leuven - University of Leuven, Belgium, 2014. URL `https://people.cs.kuleuven.be/~bart.jacobs/verifast/tutorial.pdf`.

[45] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[46] D. Kaloper-Meršinjak, H. Mehnert, A. Madhavapeddy, and P. Sewell. Not-quite-so-broken TLS: Lessons in re-engineering a security protocol specification and implementation. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 223–238, 2015.

[47] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.

[48] 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. seL4: Formal verification of an OS kernel. In *Proceedings of the Symposium on Operating Systems Principles*, pages 207–220. ACM, 2009.

[49] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology – CRYPTO 1996*, pages 104–113. Springer, 1996.

[50] X. Leroy. The CompCert C verified compiler. `http://compcert.inria.fr/`, 2004–2016.

[51] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[52] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert memory model, version 2. Research report RR-7987, INRIA, June 2012. URL `http://hal.inria.fr/hal-00703441`.

[53] P. Letouzey. A new extraction for Coq. In *Types for proofs and programs*, pages 200–219. Springer, 2002.

[54] N. D. Matsakis and F. S. Klock II. The Rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[55] J. McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

[56] B. Möller, T. Duong, and K. Kotowicz. This POODLE Bites: Exploiting The SSL 3.0 Fallback. Available at `https://www.openssl.org/~bodo/ssl-poodle.pdf`, 2014.

[57] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. 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*, pages 156–168. Springer, 2006.

[58] Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF protocols. IETF RFC 7539, 2015.

[59] J. D. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security & Privacy*, 2(4):20–27, 2004.

[60] J. Rizzo and T. Duong. The CRIME Attack, September 2012.

[61] N. Schirmer. *Verification of sequential imperative programs in Isabelle-HOL*. PhD thesis, Technical University Munich, 2006.

[62] B. Smyth and A. Pironti. Truncating TLS connections to violate beliefs in web applications. Technical Report hal-01102013, Inria, Oct. 2014. URL `https://hal.inria.fr/hal-01102013`.

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

[64] M. Stevens, P. Karpman, and T. Peyrin. Freestart collision for full SHA-1. In *Advances in Cryptology – EUROCRYPT 2016*, pages 459–483. Springer, 2016.

[65] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, 2016.

[66] R. Święcki. ChaCha20/Poly1305 heap-buffer-overflow. CVE-2016-7054, 2016.

[67] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, pages 48–62. IEEE Computer Society, 2013.

[68] D. Tarditi. Extending C with bounds safety. Checked C Technical Report, Version 0.6, Nov. 2016. URL `https://github.com/Microsoft/checkedc`.

[69] M. Tofte and J.-P. Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, Feb. 1997.

[70] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *2nd USENIX Workshop on Electronic Commerce, WOEC 1996*, pages 29–40, 1996.

[71] S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock, and M. Norrish. Mind the gap. In *22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 500–515. Springer, 2009.

[72] J. K. Zinzindohoué, E.-I. Bartzia, and K. Bhargavan. A verified extensible library of elliptic curves. In *IEEE Computer Security Foundations Symposium (CSF)*, 2016.