HACL*: A Verified Modern Cryptographic Library

Jean Karim Zinzindohoué
INRIA

Karthikeyan Bhargavan
INRIA

Jonathan Protzenko
Microsoft Research

Benjamin Beurdouche
INRIA

ABSTRACT

HACL* is a verified portable C cryptographic library that implements modern cryptographic primitives such as the ChaCha20 and Salsa20 encryption algorithms, Poly1305 and HMAC message authentication, SHA-256 and SHA-512 hash functions, the Curve25519 elliptic curve, and Ed25519 signatures.

HACL* is written in the F* programming language and then compiled to readable C code. The F* source code for each cryptographic primitive is verified for memory safety, mitigations against timing side-channels, and functional correctness with respect to a succinct high-level specification of the primitive derived from its published standard. The translation from F* to C preserves these properties and the generated C code can itself be compiled via the CompCert verified C compiler or mainstream compilers like GCC or CLANG. When compiled with GCC on 64-bit platforms, our primitives are as fast as the fastest pure C implementations in OpenSSL and Libsodium, significantly faster than the reference C code in TweetNaCl, and between 1.1x-5.7x slower than the fastest hand-optimized vectorized assembly code in SUPERCOP.

HACL* implements the NaCl cryptographic API and can be used as a drop-in replacement for NaCl libraries like Libsodium and TweetNaCl. HACL* provides the cryptographic components for a new mandatory ciphersuite in TLS 1.3 and is being developed as the main cryptographic provider for the miTLS verified implementation. Primitives from HACL* are also being integrated within Mozilla's NSS cryptographic library. Our results show that writing fast, verified, and usable C cryptographic libraries is now practical.

1 THE NEED FOR VERIFIED CRYPTO

Cryptographic libraries lie at the heart of the trusted computing base of the Internet, and consequently, they are held to a higher standard of correctness, robustness, and security than other distributed applications. Even minor bugs in cryptographic code typically result in CVEs and software updates. For instance, since 2016, OpenSSL has issued 11 CVEs[1] for bugs in its core cryptographic primitives, including 6 memory safety errors, 3 timing side-channel leaks, and 2 incorrect bignum computations. Such flaws may seem difficult to exploit at first, but as Brumley et al. [24] demonstrated, even an innocuous looking arithmetic bug hiding deep inside an elliptic curve implementation may allow an attacker to efficiently retrieve a victim's long-term private key, leading to a critical vulnerability.

Bugs in cryptographic code have historically been found by a combination of manual inspection, testing, and fuzzing, on a best-effort basis. Rather than finding and fixing bugs one-by-one, we join Brumley et al. and a number of recent works [8, 12, 25, 29, 37] in advocating the use of formal verification to mathematically prove the absence of entire classes of potential bugs. In this paper, we will show how to implement a cryptographic library and prove that it is memory safe and functionally correct with respect to its published standard specification. Our goal is to write verified code that is as fast as state-of-the-art C implementations, while implementing standard countermeasures to timing side-channel attacks.

A Library of Modern Cryptographic Primitives. To design a high-assurance cryptographic library, we must first choose which primitives to include. The more we include, the more we have to verify, and their proofs can take considerable time and effort. Mixing verified and unverified primitives in a single library would be dangerous, since trivial memory-safety bugs in unverified code often completely break the correctness guarantees of verified code. General-purpose libraries like OpenSSL implement a notoriously large number of primitives, totaling hundreds of thousands of lines of code, making it infeasible to verify the full library. In contrast, minimalist easy-to-use libraries such as NaCl [17] support a few carefully chosen primitives and hence are better verification targets. For example, TweetNaCl [19], a portable C implementation of NaCl is fully implemented in 700 lines of code.

For our library, we choose to implement modern cryptographic algorithms that are used in NaCl and popular protocols like Signal and Transport Layer Security (TLS): the ChaCha20 and Salsa20 stream ciphers [1, 15], the SHA-2 family of hash functions [36], the Poly1305 [1, 13] and HMAC [26] message authentication codes, the Curve25519 elliptic curve Diffie-Hellman group [2, 14], and the Ed25519 elliptic curve signature scheme [3, 16]. By restricting ourselves to these primitives, we obtain a compact verified library of about 7000 lines of C code that provides both the full NaCl API as well as a TLS-specific API that can be used by libraries like OpenSSL and NSS. In particular, our library is being used as the main cryptographic provider for miTLS, a verified TLS implementation [27].

Verification Goals for Cryptographic Code. Before a cryptographic library can be safely used within a larger protocol or application, the following are the most commonly desired guarantees:

Memory Safety The code never reads or writes memory at invalid locations, such as null or freed pointers, unallocated memory, or out-of-bounds of allocated memory. Also, any locally allocated memory is eventually freed (exactly once).

Functional Correctness The code for each primitive conforms to its published standard specification on all inputs.

Mitigations against Side-Channel Attacks The code does not reveal any secret inputs to the adversary, even if the adversary can observe low-level runtime behavior such as branching, memory access patterns, cache hits and misses, power consumption, etc.

[1]https://www.openssl.org/news/vulnerabilities.html

Cryptographic Security The code for each cryptographic construction implemented by the library is indistinguishable (with high probability) from some standard security definition, under well-understood cryptographic assumptions on its underlying building blocks.

For libraries written in C or in assembly, memory safety is the first and most important verification goal, since a memory error in any part of the library may compromise short- or long-term secrets held in memory (as in the infamous HeartBleed attack.) Functional correctness may be easy if the code does not diverge too far from the standard specification, but becomes interesting for highly optimized code and for elliptic curves and polynomial MACs, which need to implement error-prone bignum computations. Mitigating against all low-level side-channel attacks is an open and challenging problem. The best current practice in cryptographic libraries is to require a coding discipline that treats secret values as opaque; code cannot compare or branch on secrets or access memory at secret indices. This discipline is called *secret independence* (or *constant-time coding*), and while it does not prevent all side-channel attacks, it has been shown to prevent certain classes of timing leaks [7, 9].

In our library, we seek to verify memory safety, functional correctness, and secret independence. We do not consider cryptographic security in this paper, but our library is being used as the basis for cryptographic proofs for constructions like authenticated encryption and key exchange in miTLS [27].

Balancing Verification Effort with Performance. Making code auditable, let alone verifiable, typically comes with a significant performance cost. TweetNaCl sacrifices speed in order to be small, portable, and auditable; it is about 10 times slower than other NaCl libraries that include code optimized for specific architectures. For example, Libsodium includes three versions of Curve25519, two C implementations—tailored for 32-bit and 64-bit platforms—and a vectorized assembly implementation for SIMD architectures. All three implementations contain their own custom bignum libraries for field arithmetic. Libsodium also includes three C implementations of Poly1305, again each with its own bignum code. In order to fast verify a library like Libsodium, we would need to account for all these independent implementations, a challenging task.

Prior work on verifying cryptographic code has explored various strategies to balance verification effort with performance. Some authors verify hand-written assembly code optimized for specific architectures [25]; others verify portable C code that can be run on any platform [8, 12]; still others verify new cryptographic libraries written in high-level languages [29, 37]. The trade-off is that as we move to more generic, higher-level code, verification gets easier but at a significant cost to performance and usability. Assembly code provides the best performance, but requires considerable manual verification effort that must be repeated for each supported platform. C code is less efficient but portable; so even libraries that aggressively use assembly code often include a reference C implementation. Code in higher-level languages obtain properties like memory safety for free, but they are typically slow and difficult to protect against side-channels, due to their reliance on complex runtime components like garbage collectors.

In this paper, we attempt to strike a balance between these approaches by verifying cryptographic code written in a high-level language and then compiling it to efficient C code. For each primitive, we focus on implementing and verifying a single C implementation that is optimized for the widely used 64-bit Intel architecture, but also runs (more slowly) on other platforms. Our goal is not to replace or compete with assembly implementations; instead we seek to provide fast verified C code that can be used as default reference implementations for these primitives.

Our Approach. We take state-of-the-art optimized C implementations of cryptographic primitives and we adapt and reimplement them in F* [34] a dependently-typed programming language that supports semi-automated verification by relying on an external SMT solver (Z3). Our code is compiled to C via the KreMLin tool [20]. The resulting C code can then be compiled using the CompCert compiler [31] which results in verified machine code. Code compiled from CompCert is still not as fast as CLANG or GCC, but this gap is narrowing as more optimizations are verified and included in CompCert. In the meantime, for high-performance settings, we use GCC at optimization level -O3 to compile our C code.

To minimize the code base and the verification effort, we share as much code as possible between different primitives and different architectures. For example, we share bignum arithmetic code between Poly1305, Curve25519, and Ed25519. We also provide F* libraries that expose (and formally specify) modern hardware features such as 128-bit integer arithmetic and vector instructions, which are supported by mainstream C compilers through builtins and intrinsics. Using these libraries, we can build and verify efficient cryptographic implementations that rely on these features. On platforms that do not support these features, we provide custom implementations for these libraries, so that our compiled C code is still portable, albeit at reduced performance.

Our Contributions and Limitations. We present HACL*, a verified, self-contained, portable, reference cryptographic library that is written in F* and compiled to C. All our code is verified to be memory safe, functionally correct, and secret independent. Our library includes the first verified vectorized implementation of a cryptographic primitive (ChaCha20), the first verified implementations of SHA-512, and Ed25519, and includes new verified implementations of Salsa20, Poly1305, SHA-256, HMAC, and Curve25519. Our code is roughly as fast as state-of-the-art pure-C implementations of these primitives and is within a small factor of assembly code.

Our library is the first verified implementation of the full NaCl API and can be used as a drop-in replacement for any application that uses it via Libsodium or TweetNaCl. Our code is already being used to implement TLS ciphersuites in the miTLS project [27] and we are currently working with Mozilla to use our code within the NSS library. Our hope is that cryptographic software developers will be able to reuse our libraries and our methodology to write verified code for new primitives and new optimized implementations of existing primitives.

Throughout the paper, we will try to be precise in stating what we have proved about our code, but an early word of caution: although formal verification can significantly improve our confidence in a cryptographic library, any such guarantees rely on a large trusted computing base. The semantics of F* has been formalized [5] and our translation to C has been proven to be correct on paper [20], but we still rely on the correctness of the F* typechecker, the Z3

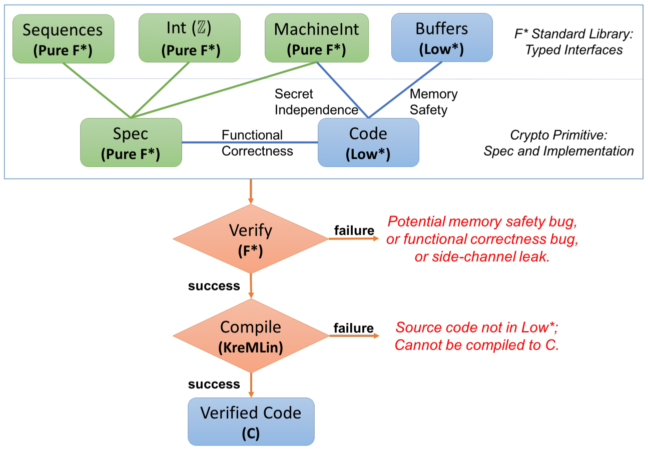

Figure 1: HACL* Verification and Compilation Toolchain

SMT solver, the KreMLin compiler, and the C compiler (that is, if we use GCC instead of CompCert.) We hope to reduce these trust assumptions over time by moving to verified F* [33] and only using CompCert. For now, we choose the pragmatic path of relying on a few carefully designed tools and ensuring that the generated C code is readable, so that it can be manually audited and tested.

Related Work. Formal verification has been successfully used on large security-critical software systems like the CompCert C compiler [31] and the sel4 operating system kernel [30]. It has been used to prove the security of cryptographic constructions like RSA-OAEP [10] and MAC-Encode-Encrypt [6]. It has even been used to verify a full implementation of the Transport Layer Security (TLS) protocol [21]. However, until recently, formal methods had not been applied to the cryptographic primitives underlying these constructions and protocols.

Recently, several works have taken on this challenge. Hawblitzel et al. [29] wrote and verified new implementations of SHA, HMAC, and RSA in the Dafny programming language. Appel [8] verified OpenSSL's C implementation of SHA-256 in Coq, and Behringer et al. [12] followed up with a proof of OpenSSL's HMAC code. Chen et al. [25] used a combination of SMT solving and the Coq proof assistant to verify a qhasm implementation of Curve25519. Zinzindohoue et al. [37] wrote and verified three elliptic curves P-256, Curve25519, and Curve448 in the F* programming language and compiled them to OCaml. Bond et al. [23] show how to verify assembly implementations of SHA-256, Poly1305, and AES-CBC. Cryptol and SAW [35] have been used to verify C and Java implementations of Chacha20, Salsa20, Poly1305, AES, and ECDSA. Compared to these works, we use a different methodology, by verifying code in F* and compiling it to C. Furthermore, unlike these prior works, our goal is to build a self-contained cryptographic library, so we focus on a complete set of primitives and we aggressively share code between them. Throughout the paper, we will compare our results with these works where relevant.

2 VERIFIED C CRYPTO VIA F* AND KREMLIN

Our verification approach is built on F* [34], a general purpose dependently-typed programming language that was designed to make it easier to incorporate formal verification into the software development cycle. More specifically, to obtain a verified C cryptographic library, we rely on recent work [20] that identifies a low-level subset of F* (dubbed Low*) that can be efficiently compiled to C, via a tool called KreMLin. The most up-to-date reference for the semantics of F* and the soundness of its type system appears in [5]. For a full description of Low* and its formal development, including a correctness theorem for the C compilation, we refer the reader to [20]. In this section, we focus on informally describing the parts of this framework that we use to build and verify HACL*.

The workflow for adding a new primitive in HACL* is depicted in Figure 1. We first write a high-level specification (Spec) for the primitive in a higher-order purely functional subset of F* (Pure F*). We then write a optimized implementation (Code) in Low*. The Code is then verified, using the F* typechecker, for conformance to the Spec and to ensure that it respects the logical preconditions and type abstractions required by the F* standard library. If typechecking fails, there may be potentially be a bug in the code, or it may be that the typechecker requires more annotations to prove the code correct. Finally, the Low* code for the primitive is compiled via KreMLin to C code. In the rest of this section, we describe each of these steps in more detail, and show how we use typechecking to guarantee our three target verification goals.

Writing High-Level Specifications in Pure F*. Starting from the published standard for a cryptographic primitive, our goal is to write a succinct formal specification that is as readable (if not more so) than the textual description or pseudocode included in the standard. We write all our specifications in Pure F*, a subset of F* where all code is side-effect free and guaranteed to terminate on all inputs. In particular, our specifications cannot use mutable data structures, and so must be written in a purely functional style. On the other hand, specifications can use mathematical objects like infinite precision integers (int) and natural numbers (nat) without worrying about how they would be implemented on 32-bit or 64-bit architectures. In addition, specifications can also use finite-precision integers (uint32,uint64,...) and immutable finite-length sequences (seq T). The function to_int converts a finite-precision integer to an int. We use the notation s.[i] (or equivalently index s i) to read the i'th element from a sequence s; updating a sequence s.[i] ← x returns a copy of s whose i'th element is set to x.

For example, the Poly1305 MAC algorithm, standardized in IETF RFC7539, evaluates a polynomial over the prime field $\mathbb{Z}_{2^{130}-5}$. Its field arithmetic can be fully specified in six lines of F*:

```
let prime = pow2 130 − 5
type felem = e:int{0 ≤ e ∧ e < prime}
let zero : felem = 0
let one : felem = 1
let fadd (e1:felem) (e2:felem) : felem = (e1 + e2) % prime
let fmul (e1:felem) (e2:felem) : felem = (e1 ∗ e2) % prime
```

The syntax of F* resembles functional languages like OCaml and F#. Each module is a sequence of type declarations, global constant definitions, and function definitions. The code above first defines the constant prime as $2^{130} - 5$. It then declares the type of field elements felem, as a subset or *refinement* of the unbounded integer type int. It defines the constants zero and one in the field, and finally defines the field addition (fadd) and multiplication (fmul) functions.

This high-level mathematical specification serves as the basis for verifying our optimized implementation of Poly1305, but how can we be confident that we did not make a mistake in writing the specification itself? First, by focusing on brevity and readability, we believe we are able to write a specification that can be audited by visually comparing it with the published standard. Second, our specifications are executable, so the developer can compile the F* code to OCaml and test it. Indeed, all the crypto specifications in HACL* have been run against multiple test vectors taken from the RFCs and other sources. Thirdly, we can ask F* to verify properties about the specification itself. By default, F* will already check that the F* specification obeys its declared types. For example, F* checks that all sequence accesses (s.[i]) fall within bounds ($0 \leq i <$ length s). In In the specification above, F* will also verify that zero, one, and the outputs of fadd and fmul are fall valid field elements. To prove such arithmetic properties, F* uses an external SMT solver called Z3. In addition to such sanity checks, we can also ask F* to prove more advanced properties about the specification. For example, in §4, we will prove that two variants of the ChaCha20 specification—one sequential, the other vectorized—are equivalent.

Writing C-like Code in Low*. F* supports a powerful proof style that relies on high-level invariants and a strong type system. In contrast, C programs tend to rely on low-level invariants, as the type system is not strong enough to prove properties such as memory safety. The Low* subset of F* blends the performance and control of C programming with the verification capabilities of F*. Importantly, Low* targets a carefully curated subset of C, and by eliminating the need to reason about legacy C code that may contain hard-to-prove features like pointer arithmetic, address-taking, and casts between pointers and integers, we obtain many invariants for free, leaving the programmer to only focus on essential properties and proofs.

Low* code can use finite-precision machine integers (uint8,uint32,...) but they cannot use unbounded integers (int), sequences or other heap-allocated data structures like lists, since these do not directly translate to native concepts in C. Instead, they can use immutable records (which translate to C structs) and mutable buffers (which translate to C arrays). Following OCaml notation, we use b.(i) to read the i'th element of a buffer, and b.(i) ⟵ x to overwrite it.

When implementing a cryptographic primitive in Low*, we aim to write efficient code that avoids unnecessary copying, implements algorithmic optimizations, if any, and exploits hardware features, wherever available. For example, as we will see in §5, to implement prime field arithmetic for Poly1305 on 64-bit platforms, one efficient strategy is to represent each 130-bit field element as an array of three 64-bit *limbs*, where each limb uses 42 or 44 bits and so has room to grow. When adding two such field elements, we can simply add the arrays point-wise, and ignore carries, as depicted in the fsum function below:

```
type limbs = b:buffer uint64_s{length b = 3}
let fsum (a:limbs) (b:limbs) =
    a.(0ul) ⟵ a.(0ul) + b.(0ul);
    a.(1ul) ⟵ a.(1ul) + b.(1ul);
    a.(2ul) ⟵ a.(2ul) + b.(2ul)
```

The function takes two disjoint buffers a and b, each with three limbs, adds them pointwise, and stores the result in-place within a. We will prove that this optimized code implements, under certain conditions, our high-level specification of field addition (fadd).

Verifying Memory Safety. Our first verification task is to prove that fsum is memory safe. Low* provides a Buffer library that carefully models C arrays and exposes a typed interface with pre- and post-conditions that enforces that the cryptographic code can only use them in a memory-safe manner. In particular, any code that reads or writes to a buffer must ensure that the buffer is live, which means that it points to an allocated array in the current heap, and that the index being accessed is within bounds.

To typecheck fsum against this buffer interface, we need to add a pre-condition that the buffers a and b are live in the initial heap. As a post-condition, we would like to prove that fsum only modifies the buffer a. So, we annotate fsum with the following type:

```
val fsum: a:limbs ⟶ b:limbs ⟶ Stack unit
    (requires (λ h0 ⟶ live h0 a ∧ live h0 b))
    (ensures (λ h0 _h1 ⟶ modifies_1 a h0 h1))
```

The requires clause contains pre-conditions on the inputs and initial heap h0; the ensures clause states post-conditions on the return value and any modifications between the initial heap h0 and the final heap h1. F* automatically proves that fsum meets this type, and hence that it is memory safe.

HACL* code never allocates memory on the heap; all temporary state is stored on the stack. This discipline significantly simplifies proofs of memory safety, and avoids the need for explicit memory management. More formally, Low* models the C memory layout using a Stack effect that applies to functions that do not allocate on the heap and only access heap locations that are passed to them as inputs. These functions are guaranteed to preserve the layout of the stack and can only read and write variables from their own stack frame. All HACL* code is typechecked in this effect.

Verifying Functional Correctness. To prove an implementation correct, we need to show how it maps to its specification. For example, to verify fsum, we first define a function eval that maps the contents of a limb array (limbs) to a Poly1305 field element (felem), and then extend the type of fsum with a post-condition that links it to fadd:

```
val fsum: a:limbs ⟶ b:limbs ⟶ Stack unit
    (requires (λ h0 ⟶ live h0 a ∧ live h0 b
                         ∧ disjoint a b
                         ∧ index h0.[a] 0 + index h0.[b] 0 < MAX_UINT64
                         ∧ index h0.[a] 1 + index h0.[b] 1 < MAX_UINT64
                         ∧ index h0.[a] 2 + index h0.[b] 2 < MAX_UINT64))
    (ensures (λ h0 _h1 ⟶ modifies_1 a h0 h1
                          ∧ eval h1 a = fadd (eval h0 a) (eval h0 b)))
```

To satisfy the new post-condition, we add four new pre-conditions. The first of these asks that the buffers a and b must be disjoint. The next three clauses require that the none of the limb additions will overflow (i.e. be greater than MAX_UINT64. The expression index h0.[a] i looks up the i'th element of the buffer a in the heap h0; the notation hints at how we model heaps just like arrays. F* can verify that fsum meets the above type, but it needs some additional help from the programmer, in the form of a lemma about the behavior of eval and another lemma that says how integer modulo distributes over addition. F* proves these lemmas, and then uses them to complete the full proof of fsum.

Depending on the machine model, the Low* programmer can modify how operations on machine integers are treated by the typechecker. Most of the code in this paper assumes a modular (wraparound) semantics for operations like addition and multiplication. However, in some cases, we may like to enforce a stricter requirement that integer operations do not overflow. The Low* machine integer library offers both kinds of operations and our implementations can choose between them. In particular, even if we use the strict addition operator in fsum, F* can automatically prove that none of these additions overflow.

Verifying Secret Independence. All secrets in HACL*, such as keys and intermediate cipher states, are implemented as buffers containing machine integers of some size. To protect these secrets from certain side-channel attacks, we enforce a *secret independent* coding discipline as follows. We provide two interfaces for machine integers: the regular interface (uint32,...) that is used for public values such as array indexes, lengths, and counters, and a *secure integer* interface that is used, by default, for all other (potentially secret) integers in the code.

The secure integer interface treats its integers as abstract types (uint8_s,uint32_s,...) whose internal representation is only available in specifications and proofs, via the reveal function, but cannot be accessed by cryptographic code. In code, regular integers can be cast into secure integers but not the converse, and there are no boolean comparison operators on secure integers. Instead, the interface exposes masked comparison operators, such as the following equality function:

```
val eq_mask: x:uint32_s → y:uint32_s → Tot (z:uint32_s {
    if reveal x = reveal y then reveal z = 0xfffffffful else reveal z = 0x0ul})
```

Here, both the inputs and outputs are secure integers. This function can be used to write straight-line code that depends on the result of the comparison, but prevents any branching on the result. The specification of this function says that the underlying representation of the return value (reveal z) is either 0xfffffffful or 0x0ul, and this information can be used within proofs of functions that call eq_mask. However, note that these proofs are all removed at compilation time, and the function reveal and the equality operator = over secure integers cannot be used in the cryptographic code.

In addition to masked comparisons, the secure integer interface offers a selection of primitive operators that are known to be implemented by the underlying hardware in constant time. Hence, for example, this interface excludes integer division and modulo (/,%) which are not constant-time on most architectures. On some ARM and i386 platforms, even integer multiplication may be variable-time. We do not currently account for such platforms, and leave the implementation and verification of secret independent multiplication for future work.

Theorem 1 in [20] shows that enforcing the secure integer interface for secret values guarantees secret independence. Informally, this theorem guarantees that, if a well-typed cryptographic implementation has an input containing a secure integer, then even an attacker who can observe low-level *event traces* that contain the results of all branches (left or right) and the addresses of all memory accesses cannot distinguish between two runs of the program that use different values for the secure integer input.

Secret independence is a necessary but not complete mitigation for side-channel attacks. It provably prevents certain classes of timing side-channel leaks that rely on branching and memory accesses, but does not account for other side-channels like power analysis. Furthermore, our current model only protects secrets at the level of machine integers; we treat the lengths and indexes of buffers as public values, which means that we cannot verify implementations that rely on constant-time table access (e.g. sbox-based AES) or length-hiding constructions (e.g. MAC-then-Encrypt [6]).

Extracting Verified Low* Code to C. If a program verifies against the low-level memory model and libraries, then it is passed to the KreMLin tool for translation to C [20]. KreMLin takes an F* program, erases all the proofs, and rewrites the program from an expression language to a statement language, performing numerous optimizations and rewritings in passing. If the resulting code only contains low-level code (i.e. no closures, recursive data types, or implicit allocations); then it fits in the Low* subset and KreMLin proceeds with a translation to C.

KreMLin puts a strong emphasis on readability by preserving names and generating idiomatic, pretty-printed code, meaning that the end result is a readable C library that can be audited before being integrated into an existing codebase. KreMLin can also combine modular proofs spread out across several F* modules and functions into a single C translation unit, which enables intra-procedural analyses and generates more compact code.

Formally, KreMLin implements a translation scheme from Low* to CompCert's Clight, a subset of C. This translation preserves semantics [20], which means that if a program is proven to be memory safe and functionally correct in F*, then the resulting Clight program enjoys the same guarantees. Furthermore, the translation also preserves event traces, which means our secret independence properties carry all the way down to C.

Note that our toolchain stops with verified C code. We do not consider the problem of compiling C to verified assembly, which is an important but independent challenge. Verified compilers like CompCert can preserve functional correctness and memory safety guarantees from Clight to x86, and enhanced versions of CompCert [9] can ensure that the compiler does not introduce new side-channels. However, for maximal performance, users of HACL* are likely to rely on mainstream compilers like GCC and CLANG.

3 BUILDING AND VERIFYING REFERENCE CRYPTOGRAPHIC IMPLEMENTATIONS

To aid interoperability between different implementations, popular cryptographic algorithms are precisely documented in public standards, such as NIST publications and IETF Request for Comments (RFCs). For example, the SHA-2 family of hash algorithms was standardized by NIST in FIPS 180-4 [36], which specifies four algorithms of different digest lengths: SHA-224, SHA-256, SHA-384, and SHA-512. For each variant, the standard describes, using text and pseudocode, the shuffle computations that must be performed on each block of input, and how to chain them into the final hash.

For all the cryptographic primitives in our library, our primary goal is to build a reference implementation in C that is proved to *conform* to the computation described in the standard. This section shows how we structure these conformance proofs for a straightforward implementation of SHA-256. In later sections, we will see how we can verify aggressively optimized implementations that significantly depart from the standard specification.

An F* specification for SHA-256. Based on the 25-page textual specification in NIST FIPS 180-4, we derive a 70 line Pure F* specification for SHA-256. (The spec for SHA-512 is very similar.) The specification defines a hash function that takes a input bytearray (of type seq byte) of length $< 2^{61}$ bytes and computes its 32-byte SHA-256 hash, by breaking the input byte array into 64-byte blocks and shuffling each block before mixing it into the global hash. Our F* specification for the core shuffle function is shown in Figure 2.

Each block processed by shuffle is represented as a sequence of 16 32-bit integers (uint32x16), and the intermediate hash value is represented as a sequence of 8 32-bit integers (uint32x8). The functions _Ch, _Maj, _Sigma0, _Sigma1, _sigma0, and _sigma1 represent specific operations on 32-bit integers taken directly from the FIPS spec. The constants k and h_0 are sequences of 32-bit integers. The function ws is the message scheduler, it takes a block and an index and returns the next 32-bit integer to be scheduled. The shuffle_core function performs one iteration of the SHA-256 block shuffle: it takes a block, an intermediate hash, and loop counter, and returns the next intermediate hash. Finally, the shuffle function takes an input hash value and iterates shuffle_core 64 times over a block to produce a new hash value. This function is chained over a sequence of blocks to produce the full SHA-256 hash.

Our F* specification for SHA-256 is precise, concise, and executable. We typechecked it against its declared types and tested it against all the RFC test vectors. Testing is not a complete solution; for example, we noticed that the usual test vectors do not cover certain interesting input sizes (e.g. 55bytes) that would help in catching certain padding mistakes. Consequently, it is still important for such specifications to be carefully audited.

A Low* reference implementation of SHA-256. We write a stateful reference implementation of SHA-256 in Low*, by adapting the F* specification function-by-function, and providing memory safety proofs wherever needed. In the implementation, blocks are treated as read-only buffers of 16 32-bit unsigned secure integers (uint32_s), whereas the intermediate hash value is a mutable buffer of secure integers that is modified in-place by shuffle. Other than this straightforward transformation from a functional state-passing specification to a stateful imperative programming style, the implementation incorporates two new features.

First, we precompute the scheduling function ws for each block and store its results in a block-sized buffer. This yields a far more efficient implementation than the naive recursive function in the high-level specification. Second, in addition to the one-shot hash function hash, which is suitable for scenarios where the full input is given in a single buffer, we implement an incremental interface where the application can provide the input in several chunks. Such incremental APIs are commonly provided by cryptographic libraries

```
let uint32x8 = b:seq uint32_s{length b = 8}
let uint32x16 = b:seq uint32_s{length b = 16}
let uint32x64 = b:seq uint32_s{length b = 64}

let _Ch x y z = (x & y) ^ ((lognot x) & z)
let _Maj x y z = (x & y) ^ ((x & z) ^ (y & z))
let _Sigma0 x = (x >>> 2ul) ^ ((x >>> 13ul) ^ (x >>> 22ul))
let _Sigma1 x = (x >>> 6ul) ^ ((x >>> 11ul) ^ (x >>> 25ul))
let _sigma0 x = (x >>> 7ul) ^ ((x >>> 18ul) ^ (x >> 3ul))
let _sigma1 x = (x >>> 17ul) ^ ((x >>> 19ul) ^ (x >> 10ul))

let k : uint32x64 = createL [0x428a2f98ul; 0x71374491ul; ...] // Constants
let h_0 : uint32x8 = createL [0x6a09e667ul; 0xbb67ae85ul; ...] // Constants

let rec ws (b:uint32x16) (t:nat{t < 64}) =
  if t < 16 then b.[t]
  else
    let t16 = ws b (t − 16) in
    let t15 = ws b (t − 15) in
    let t7 = ws b (t − 7) in
    let t2 = ws b (t − 2) in
    let s1 = _sigma1 t2 in
    let s0 = _sigma0 t15 in
    (s1 + (t7 + (s0 + t16)))

let shuffle_core (block:uint32x16) (hash:uint32x8) (t:nat{t < 64}) : Tot uint32x8 =
  let a = hash.[0] in let b = hash.[1] in
  let c = hash.[2] in let d = hash.[3] in
  let e = hash.[4] in let f = hash.[5] in
  let g = hash.[6] in let h = hash.[7] in
  let t1 = h + (_Sigma1 e) + (_Ch e f g) + k.[t] + ws block t in
  let t2 = (_Sigma0 a) + (_Maj a b c) in
  create_8 (t1 + t2) a b c (d + t1) e f g

let shuffle (hash:uint32x8) (block:uint32x16) =
  repeat_range_spec 0 64 (shuffle_core block) hash
```

Figure 2: F* specification of the SHA-256 block shuffle. The following operators are over 32-bit unsigned secure integers (uint32_s): >>> is right-rotate; >> is right-shift; & is bitwise AND; ^ is bitwise XOR; lognot is bitwise NOT; + is wraparound addition. The operators − and < are over unbounded integers (int).

```
val shuffle:
  hash_w :buffer uint32_s{length hash_w = 8} ⟶
  block_w:buffer uint32_s{length block_w = 16} ⟶
  ws_w :buffer uint32_s{length ws_w = 64} ⟶
  k_w :buffer uint32_s{length k_w = 64} ⟶
  Stack unit
    (requires (λ h ⟶ live h hash_w ∧ live h ws_w ∧ live h k_w ∧ live h block_w
                ∧ h.[k_w] == Spec.k
                ∧ (∀ (i:nat). i < 64 ⟹ index h.[ws_w] i == Spec.ws h.[block_w] i)) )
    (ensures (λ h0 r h1 ⟶ modifies_1 hash_w h0 h1
                ∧ h1.[hash_w] == Spec.shuffle h0.[hash_w] h0.[block_w]))
```

Figure 3: Low* type of the SHA-256 shuffle function

like OpenSSL but are not specified in the NIST standard. Our correctness specification of this API requires the implementation to maintain *ghost* state (used only in proofs, erased at compile-time) that records the portion of the input that has already been hashed.

To verify our implementation, we provide a type for each function that shows how it relates to the corresponding function in the specification. Figure 3 displays the type for our implementation of the shuffle function. The function takes as its arguments four buffers: hash_w contains the intermediate hash, block_w contains the current block, ws_w contains the precomputed schedule, k_w contains the k-constant from the SHA-256 specification. The types

```
static void
SHA2_256_shuffle(uint32_t *hash, uint32_t *block, uint32_t *ws, uint32_t *k)
{
  for (uint32_t i = (uint32_t )0; i < (uint32_t )64; i = i + (uint32_t )1)
  {
    uint32_t a = hash_0[0]; uint32_t b = hash_0[1];
    uint32_t c = hash_0[2]; uint32_t d = hash_0[3];
    uint32_t e = hash_0[4]; uint32_t f1 = hash_0[5];
    uint32_t g = hash_0[6]; uint32_t h = hash_0[7];
    uint32_t kt = k_w[i]; uint32_t wst = ws_w[i];
    uint32_t t1 = h + ((e >> (uint32_t )6 | e << (uint32_t )32 − (uint32_t )6)
          ^ (e >> (uint32_t )11 | e << (uint32_t )32 − (uint32_t )11)
            ^ (e >> (uint32_t )25 | e << (uint32_t )32 − (uint32_t )25))
          + (e & f1 ^ ¬e & g) + kt + wst;
    uint32_t t2 = ((a >> (uint32_t )2 | a << (uint32_t )32 − (uint32_t )2)
          ^ (a >> (uint32_t )13 | a << (uint32_t )32 − (uint32_t )13)
          ^ (a >> (uint32_t )22 | a << (uint32_t )32 − (uint32_t )22))
          + (a & b ^ a & c ^ b & c);
    uint32_t x1 = t1 + t2;
    uint32_t x5 = d + t1;
    uint32_t *p1 = hash_0;
    uint32_t *p2 = hash_0 + (uint32_t )4;
    p1[0] = x1; p1[1] = a; p1[2] = b; p1[3] = c;
    p2[0] = x5; p2[1] = e; p2[2] = f1; p2[3] = g;
  }
}
```

Figure 4: Compiled C code for the SHA-256 shuffle function

of these input buffers states their expected length and that their contents are secure integers (uint32_s). The function is given the Stack effect (see §2), which means that it obeys the C stack discipline, and allocates nothing on the heap.

The requires clause states a series of pre-conditions. The first line asks that all the input buffers must be *live*, for memory safety. The second line asks that the buffer ks_w must contain the integer sequence specified in Spec.k. The third line asks that the buffer ws_w buffer must contain the precomputed results of the Spec.ws function applied to the current block.

The first line of the ensures clause states as a post-condition that the function only modifies the intermediate hash value hash_w; all other buffers remain unchanged. The second line asserts that the new contents of the hash_w buffer will be the same as the result of the Spec.shuffle function applied to the old hash_w and the current block_w, hence linking the specification to the implementation.

Verifying the code of shuffle against this type guarantees all our target properties: memory safety, secret independence for all inputs, and functional correctness with respect the the standard. F* verifies shuffle with a little help in the form of annotations indicating intermediate loop invariants. The full proof of SHA-256 requires a little more work; we write a total of 622 lines of Low* code and annotations, from which we generate 313 lines of C, which gives a rough indication of the annotation overhead for verification.

Generating Verified C code for SHA-256. We run KreMLin on our verified Low* implementation to generate C code. Figure 4 depicts the compiled code for shuffle. Our Low* source code is broken into many small functions, in order to improve readability, modularity and code sharing, and to reduce the complexity of each proof. Consequently, the default translation of this code to C would result in a series of small C functions, which can be overly verbose and hurts runtime performance with some compilers like CompCert.

To allow better control over the generated code, the KreMLin compiler can be directed (via program annotations) to inline certain functions and unroll certain loops, in order to obtain C code that is idiomatic and readable. The shuffle function illustrates this mechanism: the _Ch, _Maj, _Sigma0, _Sigma1, and shuffle_core functions are inlined, yielding a compact C function that we believe is readable and auditable. Furthermore, as we show in §8, the performance of our generated C code for SHA-256 (and SHA-512) are as fast as handwritten C implementations in OpenSSL and Libsodium.

Comparison with prior work. Implementations of SHA-256 have been previously verified using a variety of tools and techniques. The approach most closely-related to ours is that of Appel [8], who verified a C implementation adapted from OpenSSL using the VST toolkit. We do not operate pre-existing C code directly but instead generate the C code from our own high-level proofs and implementations. Appel wrote a high-level specification in Coq and an executable functional specification (similar to ours) in Coq; we only needed a single specification. He then manually proved memory safety and functional correctness (but not side-channel resistance) for his code using the Coq interactive theorem prover. His proof takes about 9000 lines of Coq. Our total specs + code + proofs for SHA-256 amount to 708 lines of F* code, and our proofs are partially automated by F* and the Z3 SMT solver.

Other prior work includes SAW [35], which uses symbolic equivalence checking to verify C code for HMAC-SHA-256 against a compact spec written in Cryptol. The proof is highly-automated. Vale [23] has been used to verify X86 assembly code for SHA-256 using Dafny. The verification effort of our approach is comparable to these works, but these efforts have the advantage of being able to tackle legacy hand-optimized code, whereas we focus on synthesizing efficient C code from our own implementations.

4 VERIFYING HIGH-PERFORMANCE VECTORIZED IMPLEMENTATIONS

In the previous section, we saw how we can implement cryptographic primitives in Low* by closely following their high-level F* specification. By including a few straight-forward optimizations, we can already generate C code that is as fast as hand-written C reference implementations for these primitives. However, the record-breaking state-of-the-art assembly implementations for these primitives can be several times faster than such naive C implementations, primarily because they rely on modern hardware features that are not available on all platforms and are hence not part of standard portable C. In particular, the fastest implementations of all the primitives considered in this paper make use of vector instructions that are available on modern Intel and ARM platforms.

Intel architectures have supported 128-bit registers since 1999, and, through a series of instruction sets (SSE, SSE2, SSSE3, AVX, AVX2, AVX512), have provided more and more sophisticated instructions to perform on 128, 256, and now 512-bit registers, treated as vectors of 8, 16, 32, or 64-bit integers. ARM recently introduced the NEON instruction set in 2009 that provides 128-bit vector operations. So, on platforms that support 128-bit vectors, a single vector instruction can add 4 32-bit integers using a special vector processing unit. This does not strictly translate to a 4x speedup, since vector units have their own overheads, but can significantly boost

```
val uint32x4: Type0
val to_seq: uint32x4 ⟶ GTot (s:seq uint32){length s = 4}
val load32x4: x0:uint32_s ⟶ x1:uint32_s ⟶
              x2:uint32_s ⟶ x3:uint32_s ⟶
              Tot (r:uint32x4{to_seq r = createL [x0;x1;x2;x3]})
val ( + ) : x:uint32x4 ⟶ y:uint32x4 ⟶
           Tot (r:uint32x4{to_seq r = map2 (λ x y ⟶ x + y) (to_seq x) (to_seq y)}
val shuffle_right: s:uint32x4 ⟶ n:uint32{to_int r < 4} ⟶
                    Tot (r:uint32x4{if n = 1ul then createL [s.[3];s.[0];s.[1];s.[2]]
                                    else if n == 2ul then ...})
```

Figure 5: (Partial) F* Interface for 128-bit vectors
uint32x4 models a 128-bit vector as a sequence of four 32-bit unsigned secure integers. The ghostly function to_seq is used only within specifications and proofs; load32x4 loads four secure integers into a vector; + and <<< specifies vector addition as pointwise addition on the underlying sequence of uint32_s; shuffle_right specifies vector shuffling as a permutation over the underlying sequence.

```
typedef unsigned int uint32x4 __attribute__ ((vector_size (16)));
uint32x4 load32x4(uint32_t x1, uint32_t x2, uint32_t x3, uint32_t x4){
  return ((uint32x4) _mm_set_epi32(x4,x3,x2,x1));
}
uint32x4 uint32x4_addmod(uint32x4 x, uint32x4 y) {
  return ((uint32x4) _mm_add_epi32((__m128i)x,(__m128i)y);
}
uint32x4 shuffle_right(uint32x4 x, unsigned int n) {
  return ((uint32x4) _mm_shuffle_epi32((__m128i)x,
                            _MM_SHUFFLE((3+n)%4,(2+n)%4,(1+n)%4,n%4)));
}
```

Figure 6: (Partial) GCC library for 128-bit vectors using Intel SSE3 intrinsics: (https://software.intel.com/sites/landingpage/IntrinsicsGuide/)

the speed of programs that exhibit single-instruction multiple-data (SIMD) parallelism.

Many modern cryptographic primitives are specifically designed to take advantage of vectorization. However, making good use of vector instructions often requires restructuring the sequential implementation to expose the inherent parallelism and to avoid operations that are unavailable or expensive on specific vector architectures. Consequently, the vectorized code is no longer a straightforward adaptation of the high-level specification and needs new verification. In this section, we develop a verified vectorized implementation of ChaCha20 in Low*. Notably, we show how to verify vectorized C code by relying on vector libraries provided as compiler builtins and intrinsics. We do not need to rely on or verify assembly code. We believe this is the first verified vectorized code for any cryptographic primitive and shows the way forward for verifying other record-breaking cryptographic implementations.

4.1 Modeling Vectors in F*

In F*, the underlying machine model is represented by a set of trusted library interfaces that are given precise specifications, but which are implemented at runtime by hardware or system libraries. For example, machine integers are represented by a standard library interface that formally interprets integer types like uint32 and primitive operations on them to the corresponding operations on mathematical integers int. When compiling to C, KreMLin translates these operations to native integer operations in C. However, F* programmers are free to add new libraries or modify existing

```
type state = m:seq uint32_s{length m = 16}
type idx = n:nat{n < 16}

let line (a:idx) (b:idx) (d:idx) (s:uint32{to_int s < 32}) (m:state) =
  let m = m.[a] ⟵ (m.[a] + m.[b]) in
  let m = m.[d] ⟵ ((m.[d] ^ m.[a]) <<< s) in m

let quarter_round a b c d m =
  let m = line a b d 16ul m in
  let m = line c d b 12ul m in
  let m = line a b d 8ul m in
  let m = line c d b 7ul m in m

let column_round m =
  let m = quarter_round 0 4 8 12 m in
  let m = quarter_round 1 5 9 13 m in
  let m = quarter_round 2 6 10 14 m in
  let m = quarter_round 3 7 11 15 m in m
```

Figure 7: RFC-based ChaCha20 specification in F*.
Operators are defined over 32-bit unsigned integers. + is 32-bit wraparound addition, ^ is the bitwise XOR, <<< is rotate left.

libraries to better reflect their assumptions on the underlying hardware. For C compilation to succeed, they must then provide a Low* or C implementation that meets this interface.

We follow the same approach to model vectors in HACL* as a new kind of machine integer interface. Like integers, vectors are pure values. Their natural representation is a sequence of integers. For example, Figure 5 shows a fragment of our F* interface for 128-bit vectors, represented as an abstract type uint32x4. Proofs and specifications can access the underlying sequence representation of a vector, via the to_seq function. (More generally, such vectors can be also interpreted as eight 16-bit or sixteen 8-bit integers, and we can make these representations interconvertible.) Many classic integer operations (+,−,*,&,^,<<,>>,<<<) are lifted to uint32x4, and interpreted as the corresponding point-wise operations over sequences of integers. In addition, the interface provides vector-specific operations like load32x4 to load vectors, and shuffle_right, which allows the integers in a vector to be permuted.

We provide C implementations of this interface for Intel SSE3 and ARM NEON platforms. Figure 6 shows a fragment of the Intel library relying on GCC compiler intrinsics. This C code is not verified, it is trusted. Hence, it is important to minimize the code in such libraries, and to carefully review them to make sure that their implementation matches their assumed specification in F*. However, once we have this F* interface and its C implementation for some platform, we can build and verify vectorized cryptographic implementations in Low*.

4.2 Verified Vectorized ChaCha20

The ChaCha20 stream cipher was designed by D. Bernstein [15] and standardized as an IETF RFC [1]. It is widely recommended as an alternative to AES in Internet protocols. For example, ChaCha20 is one of the two encryption algorithms (other than AES) included in TLS 1.3 [4]. The NaCl API includes Salsa20, which differs a little from ChaCha20 [15] but for the purposes of verification, these differences are irrelevant; we implemented both in HACL*.

Figure 7 depicts a fragment of our RFC-based F* specification of ChaCha20. ChaCha20 maintains an internal state that consists of 16 32-bit integers interpreted as a 4x4 matrix. This state is initialized

```
type state = m:seq uint32x4{length m = 4}
type idx = n:nat{n < 4}

let line (a:idx) (b:idx) (d:idx) (s:uint32{to_int s < 32}) (m:state) =
  let ma = m.[a] in let mb = m.[b] in let md = m.[d] in
  let ma = ma + mb in
  let md = (md ^ ma) <<< s in
  let m = m.[a] ← ma in
  let m = m.[d] ← md in m

let column_round m =
  let m = line 0 1 3 16ul m in
  let m = line 2 3 1 12ul m in
  let m = line 0 1 3 8ul m in
  let m = line 2 3 1 7ul m in m
```

Figure 8: F* specification for 128-bit vectorized ChaCha20
Operators are defined over vector of 32-bit integers: see Figure 5.

using the encryption key, nonce, and the initial counter (typically 0). Starting from this initial state, ChaCha20 generates a sequence of states, one for each counter value. Each state is serialized as a key block and XORed with the corresponding plaintext (or ciphertext) block to obtain the ciphertext (or plaintext). To generate a key block, ChaCha20 shuffles the input state 20 times, with 10 column rounds and 10 diagonal rounds. Figure 7 shows the computation for each column round.

As we did for SHA-256, we wrote a reference stateful implementation for ChaCha20 and proved that it conforms to the RFC-based specification. The generated code takes 6.26 cycles/byte to encrypt data on 64-bit Intel platforms; this is as fast as the unvectorized C implementations in popular libraries like OpenSSL and Libsodium, but is far slower than vectorized implementations. Indeed, previous work (see [18, 28]) has identified two inherent forms of parallelism in ChaCha20 that lend themselves to efficient vector implementations:

Line-level Parallelism: The computations in each column and diagonal round can be reorganized to perform 4 line shufflings in parallel.

Block-level Parallelism: Since each block is independent, multiple blocks can be computed in parallel.

We are inspired by a 128-bit vector implementation in SUPERCOP due to Ted Krovetz, which is written in C using compiler intrinsics for ARM and Intel platforms, and reimplement it in HACL*. Krovetz exploits line-level parallelism by storing the state in 4 vectors, resulting in 4 vector operations per column-round, compared to 16 integer operations in unvectorized code. Diagonal rounds are a little more expensive (9 vector operations), since the state vectors have to be reorganized before and after the 3 line operations. Next, Krovetz exploits block-level parallelism and the fact that modern processors have multiple vector units (typically 3 on Intel platforms and 2 on ARM) to process multiple interleaving block computations at the same time. Finally, Krovetz vectorizes the XOR step for encryption/decryption by loading and processing 128 bits of plaintext/ciphertext at once. All these strategies requires significant refactoring of the source code, so it becomes important to verify that the code is still correct with respect to the ChaCha20 RFC.

We write a second F* specification for vectorized ChaCha20 that incorporates these changes to the core algorithm. The portion of this spec up to the column round is shown in Figure 8. We modify the state to store four vectors, and rearrange the line and column_round using vector operations. We then prove that the new column_round function has the same functional behavior as the RFC-based column_round function from Figure 7. Building up from this proof, we show that the vectorized specification for full ChaCha20 computes the same function as the original spec.

Finally, we implement a stateful implementation of vectorized ChaCha20 in Low* and prove that it conforms to our vectorized specification. (As usual, we also prove that our code is memory safe and secret independent.) This completes the proof for our vectorized ChaCha20, which we believe is the first verified vectorized implementation for any cryptographic primitive.

When compiled to C and linked with our C library for uint32x4, our vectorized ChaCha20 implementation has the same performance as Krovetz's implementation on both Intel and ARM platforms. This makes our implementation the 8th fastest in the SUPERCOP benchmark on Intel processors, and the 2nd fastest on ARM. As we did with Krovetz, we believe we can adapt and verify the implementation techniques of faster C implementations and match their performance.

5 VERIFYING SECRET INDEPENDENT MODULAR BIGNUM ARITHMETIC

Asymmetric cryptographic algorithms commonly rely on prime-field arithmetic, that is, addition and multiplication modulo a prime p in $\mathbb{Z}_p$. In HACL*, the Poly1305, Curve25519, and Ed25519 algorithms all compute on various prime fields. The mathematical specification for these field operations is very simple; §2 shows the 6-line F* spec for the Poly1305 field.

For security, the primes used by cryptographic algorithms need to be quite large, which means that elements of the field cannot be represented by machine integers, and instead need to be encoded as bignums, that is, arrays of integers. Consequently, bignum arithmetic becomes a performance bottleneck for these algorithms. Furthermore, well known bignum implementation tricks that work well for numerical computations are not really suitable for cryptographic code since they may leak secrets. For example, when multiplying two bignums, a generic bignum library may shortcut the computation and return zero if one of the arguments is zero. In a crypto algorithm, however, the time taken by such optimizations may leak the value of a key. Implementing an efficient and secure generic modulus function is particularly hard. Consequently, cryptographic implementations are often faced with a trade-off between efficient field arithmetic and side-channel resistance.

5.1 Efficient Bignum Libraries for Poly1305, Curve25519, and Ed25519

For algorithms like RSA that use large and unpredictable primes, implementations often choose to forego any side-channel resistance. However, for modern fixed-prime primitives like Poly1305 and Curve25519, it is possible to choose the shape of the prime carefully so that field arithmetic can be both efficient and secret independent. For instance, given a fixed Mersenne prime of the form $2^n - 1$, the modulo operation is easy to implement: all the bits beyond n-th bit can be repeatedly lopped off and added to the low n bits, until the result is an n bit value. Computing the modulo for the Poly1305 prime $2^{130} - 5$ or Curve25519 $2^{255} - 19$ in constant time is similar.

Once a suitable prime is picked, the main implementation choice is whether to represent the field elements as *packed* bignums, where each array element (called a *limb*) is completely filled, or to use an *unpacked* representation, where the limbs are only partially filled. For example, in the Poly1305 field, elements are 130-bit values and can be stored in 3 64-bit integers. The little-endian packed layout of these elements would be $64bits|64bits|2bits$, whereas a more evenly distributed unpacked layout is $44bits\ |44bits\ |42bits$. The main advantage of the unpacked layout is that when performing several additions in a sequence, we can delay the carry propagation, since the limbs will not overflow. In the packed representation, we must propagate carries after each addition. Optimizing carry propagation by making it conditional on overflow would not be safe, since it would expose a timing side-channel. Indeed, most efficient 64-bit implementations of Poly1305 and Curve25519 use unpacked representations; Poly1305 uses the 44-44-42 layout on 64-bit platforms and 5 26-bit limbs on 32-bit platforms; Curve25519 and Ed25519 use 5 limbs of 51-bits each or 10 limbs of 25.5 bits each.

In summary, efficient implementations of Poly1305, Curve25519, and Ed25519 use prime-specific computations and different unpacked bignum representations for different platforms. Consequently, each of their implementations contains its own bignum library which must be independently verified. In particular, previous proofs of bignum arithmetic in Poly1305 [23] and Curve25519 [25] are implementation-specific and cannot be reused for other platforms or other implementations. In contrast, Zinzindohoue et al. [37] develop a generic verified bignum library in OCaml that can be used in multiple cryptographic algorithms. The cost of this genericity is significantly reduced performance. In the rest of this section, we present a novel approach that allows us to share verified bignum code across primitives and platforms, at no cost to performance.

5.2 Verifying a Generic Bignum Library

In HACL*, we uniformly adopt unpacked representations for our bignums. We define an evaluation function eval that maps a bignum to the mathematical integer it represents. This function is parametric over the base of the unpacked layout: for example, our Poly1305 elements are in base 2^{44}, which means that a bignum b represents the integer $eval(b) = b[0] + 2^{44} * b[1] + 2^{88} * b[2]$.

We observe that, except for modulo, all the bignum operations needed by our primitives are independent of the prime. Furthermore, generic bignum operations, such as addition, do not themselves depend on the specific unpacked representation; they only rely on having enough remaining space so that limbs do not overflow. Using these observations, we implement and verify a generic bignum library that includes modular addition, subtraction, multiplication, and inverse, and whose proofs do not depend on the prime or the unpacked representation. Each generic operation is parametric over the number of limbs in the bignum and requires as a pre-condition that each limb has enough spare room to avoid overflow. To satisfy these preconditions in a cryptographic primitive like Poly1305, the implementation must carefully interleave carry propagation steps and modular reduction with generic operations.

The only part of the bignum library that depends on the prime is the modular reduction, and this must be implemented and verified anew for each new prime. All other functions in the bignum library are written and verified just once. When compiling the code to C, the prime-specific code and the representation constants (e.g. the number of limbs, the evaluation base etc.) are inlined into the generic bignum code, yielding an automatically specialized bignum library in C for each primitive. As a result, our generated field arithmetic code is as efficient as the custom bignum libraries for each primitive. Hence, we are able to find a balance between generic code for verification and specialized code for efficiency. We are able to reuse more than half of the field arithmetic code between Poly1305, Curve25519, and Ed25519. We could share even more of the code if we specialized our bignum library for pseudo-Mersenne primes. For primes which shapes do not enable optimized modulo computations, we also implement and verify a generic modulo function based on Barrett reduction, which we use in the Ed25519 signature algorithm.

5.3 Preventing Bugs, Enabling Optimizations

When programming with unpacked bignums, carry propagation and modular reduction are the most expensive operations. Consequently, this style encourages programmers to find clever ways of delaying these expensive operation until they become necessary. Some implementations break long carry chains into shorter sequences that can be executed in parallel and then merged. These low-level optimizations are error-prone and require careful analysis. In particular, carry propagation bugs are the leading functional correctness flaws in OpenSSL crypto, with two recent bugs in Poly1305 [11, 22], and two others in Montgomery multiplication (CVE-2017-3732, CVE-2016-7055). A carry propagation bug was also found in TweetNaCl [19].

Our Curve25519 implementation is closely inspired by Adam Langley's `donna_c64` 64-bit implementation, which is widely used and considered the state-of-the-art C implementation. In 2014, Langley reported a bug in this implementation [2]: the implementation incorrectly skipped a necessary modular reduction step. In response, Langley explored the use of formal methods to prove the absence of such bugs, but gave up after failing to prove even modular addition using existing tools. This paper presents the first complete proof of a C implementation of Curve25519, including all its field arithmetic. In particular, our proofs guarantee the absence of carry propagation bugs in Poly1305, Curve25519, and Ed25519.

A surprising benefit of formal verification is that it sometimes identifies potential optimizations. When verifying Curve25519, we observed that `donna_c64` was too conservative in certain cases. Each multiplication and squaring operation had an unnecessary extra carry step, which over the whole Curve25519 scalar multiplication totaled to about 3400 extra cycles on 64-bit Intel processors. We removed these redundant carries in our code and proved that it was still correct. Consequently, the Curve25519 C code generated from HACL* is slightly (about 2.2%) faster than `donna_c64` making it the fastest C implementation that we know of.

[2]https://www.imperialviolet.org/2014/09/07/provers.html

```
let prime = pow2 255 − 19
type felem = e:int{0 ≤ e ∧ e < prime}
type serialized_point = b:seq uint8{length b = 32}
type proj_point = | Proj: x:felem → z:felem → proj_point

let decodePoint (u:serialized_point) =
  (little_endian u % pow2 255) % prime

let encodePoint (p:proj_point) =
  let x = fmul p.x (p.z ** (prime − 2)) in
  little_bytes 32ul x
```

Figure 9: F* specification of Curve25519 point format. Operators are over integers (field elements); fmul is the field multiplicative law, ** is the exponentiation operator based on this multiplication law (p ** x is p^x). The operators % and − and < are standard over natural integers. The little_bytes len v function returns the little endian encoding of the integer value v on len bytes.

6 VERIFYING ELLIPTIC CURVE OPERATIONS

6.1 Curve25519

Curve25519 [2, 14] a Montgomery elliptic curve designed for use in a Diffie-Hellman (ECDH) key exchange. The key operation over this curve is the multiplication nP of a public curve point P by a secret scalar n. A distinctive property of this family of curves is only the x-coordinate of P is needed to compute the x-coordinate of nP. This leads to both efficient computations and small keys.

The simplicity of the algorithm and its adoption in protocols like TLS and Signal have made it a popular candidate for formal verification. Several other works have been tackling Curve25519. However, our implementation is, to the best of our knowledge, the first implementation to verify the full byte-level scalar multiplication operation. Chen et al. [25] verified one step of the Montgomery ladder for a qhasm implementation, but did not verify the ladder algorithm or point encodings; Zinzindohoue et al. [37] implemented and verified the Montgomery ladder for Curve25519 and two other curves, but they did not verify the point encodings. Our Curve25519 implementation is verified to be fully RFC-compliant.

Figure 9 shows the F* specification for the point encoding and decoding functions that translate between curve points and byte arrays. Implementing and verifying these functions is not just a proof detail. Compliance with encodePoint avoids the missing reduction bug that Adam Langley described in `donna_c64`. The first line of encodePoint computes x as a result of the modular multiplication operation fmul (see §2). Hence, the result of encodePoint is a little-endian encoding of a number strictly less than $2^{255} - 19$. Consequently, a Low* implementation of Curve25519 that forgets to perform a modular reduction before the little-endian encoding does not meet this specification and so will fail F* verification.

Ed25519. The Ed25519 signature scheme [3, 16] is an EdDSA algorithm based on the twisted Edwards curve birationally equivalent to Curve25519. Despite their close relation, the implementation of Ed25519 involves many more components than Curve25519. It uses a different coordinate system and different point addition and doubling formulas. The signature input is first hashed using the SHA-512 hash function, which we verify separately. The signature operation itself involves prime-field arithmetic over two primes: the Curve25519 prime $2^{255} - 19$ and a second non-Mersenne prime $2^{252} + 27742317777372353535851937790883648493$. This second prime does not enjoy an efficient modulo operation, so we implement and verify a slower but generic modulo function using the Barrett reduction. We thus obtain the first verified implementation of Ed25519 in any language. In terms of size and proof complexity, Ed25519 was the most challenging primitive in HACL*; implementing and verifying the full construct took about 3 person-weeks, despite our reuse of the Curve25519 and SHA-512 proofs.

Our implementation is conservative and closely follows the RFC specification. It is faster than the naive Ed25519 reference implementation (`ref`) in TweetNaCl, but about 2.5x slower than the optimized `ref10` implementation, which relies on a precomputed table containing multiples of the curve base point. Our code does not currently use precomputation. Using precomputed tables in a provably secret independent way is non-trivial; for example, [32] demonstrate side-channel attacks on Ed25519 precomputations on certain platforms. We leave the implementation and verification of secure precomputation for Ed25519 as future work.

7 MEETING HIGH-LEVEL CRYPTO APIS

HACL* offers all the essential building blocks for real-world cryptographic application: authenticated encryption, (EC)DH key exchange, hash functions, and signatures. The C code for each of our primitives is self-contained and easy to include in C applications. For example, we are currently engaged in transferring multiple verified primitives into Mozilla's and RedHat's NSS cryptographic library.

In the rest of this section, we describe three more advanced ways of integrating our verified library in larger software developments.

NaCl. The APIs provided by mainstream cryptographic libraries like OpenSSL are too complex and error-prone for use by non-experts. The NaCl cryptographic API [17] seeks to address this concern by including a carefully curated set of primitives and only allowing them to be used through simple secure-by-default constructions, like box/box_open (for public-key authenticated encryption/decryption). By restricting the usage of cryptography to well-understood safe patterns, users of the library are less likely to fall into common crypto mistakes.

The NaCl API has several implementations including TweetNaCl, a minimal, compact, portable library, and Libsodium, an up-to-date optimized implementation. HACL* implements the full NaCl API and hence can be used as a drop-in replacement for any application that relies on just the NaCl functions in TweetNaCl or Libsodium. Our code is as fast as libsodium's unvectorized C code on 64-bit Intel platforms, and is many times faster than TweetNaCl on all platforms. Hence, we offer the first high-performance verified C implementation of NaCl.

TLS 1.3. TLS 1.3 [4] will soon become the new standard for secure communications over the internet. HACL* implements all the primitives needed for one TLS 1.3 ciphersuite: IETF Chacha20Poly1305 authenticated encryption with associated data (AEAD), SHA-256 and HMAC-SHA-256, Curve25519 key exchange, and Ed25519 signatures. We do not yet implement RSA or ECDSA signatures which are needed for X.509 certificates.

OpenSSL implements the current TLS 1.3 draft and hence uses many of these primitives; OpenSSL does not yet implement Ed25519.

Algorithm	Spec (F^* loc)	Code+Proofs (Low^* loc)	C Code (C loc)	Verification (s)
Salsa20	70	651	372	280
Chacha20	70	691	243	336
Chacha20-Vec	100	1656	355	614
SHA-256	96	622	313	798
SHA-512	120	737	357	1565
HMAC	38	215	28	512
Bignum-lib	-	1508	-	264
Poly1305	45	3208	451	915
X25519-lib	-	3849	-	768
Curve25519	73	1901	798	246
Ed25519	148	7219	2479	2118
AEAD	41	309	100	606
SecretBox	-	171	132	62
Box	-	188	270	43
Total	801	22,926	7,225	9127

Table 1: HACL* code size and verification times

OpenSSL allows other libraries to provide cryptographic implementations via an *engine* interface. We package HACL* as an OpenSSL engine so that our primitives can be used within OpenSSL and by any applications built on top of OpenSSL. We use this engine to compare the speed of our code with the native implementations in OpenSSL. Our Curve25519 implementation is significantly faster than OpenSSL, and our other implementations are as fast as OpenSSL's C code, but slower than its assembly implementations.

miTLS. A key advantage of developing HACL* in F* is that it can be integrated into larger verification projects in F*. For example, the miTLS project is developing a cryptographically secure implementation of the TLS 1.3 protocol in F*. Previous versions of miTLS relied on an unverified (OpenSSL-based) cryptographic library, but the new version now uses HACL* as its primary cryptographic provider. The functional correctness proofs of HACL* form a key component in the cryptographic proofs of miTLS. For example, our proofs of ChaCha20 and Poly1305 are composed with cryptographic assumptions about these primitives to build a proof of the TLS record layer protocol [27]. In the future, we plan to build simpler verified F* applications, that rely on HACL*'s NaCl API.

8 EVALUATION AND DISCUSSION

In this section, we assess the coding and verification effort that went into the HACL* library, and evaluate its performance relative to state-of-the-art cryptographic libraries.

Coding and Verification Effort. Taking an RFC and writing a specification for it in F* is straightforward; similarly, taking inspiration from existing C algorithms and injecting them into the Low* subset is a mundane task. Proving that the Low* code is memory safe, secret independent, and that it implements the RFC specification is the bulk of the work. Table 1 lists, for each algorithm, the size of the RFC-like specification and the size of the Low* implementation, in lines of code. Specifications are intended to be read by experts and are the source of "truth" for our library: the smaller, the better. The size of the Low* implementation captures both the cost of going into a low-level subset (meaning code is more imperative and verbose) and the cost of verification (these include lines of proof). We also list the size of the resulting C program, in lines of code. Since the (erased) Low* code and the C code are in close correspondence, the ratio of C code to Low* code provides a good estimate of code-to-proof ratio.

One should note that a large chunk of the bignum verified code is shared across Poly1305, Curve25519 and Ed25519, meaning that this code is verified once but used in three different ways. The sharing has no impact on the quality of the generated code, as we rely on KreMLin to inline the generic code and specialize it for one particular set of bignum parameters. The net result is that Poly1305 and Curve25519 contain separate, specialized versions of the original Low* bignum library. Chacha20 and Salsa20, just like SHA-256 and SHA-512, are very similar to each other, but the common code has not yet been factored out. We intend to leverage recent improvements in F* to implement more aggressive code sharing, allowing us to write, say, a generic SHA-2 algorithm that can be specialized and compiled twice, for SHA-256 and SHA-512.

Our estimates for the human effort are as follows. Symmetric algorithms like Chacha20 and SHA2 do not involve sophisticated math, and were in comparison relatively easy to prove. The proof-to-code ratio hovers around 2, and each primitive took around one person-week. Code that involves bignums requires more advanced reasoning. While the cost of proving the shared bignum code is constant, each new primitive requires a fresh verification effort. The proof-to-code ratio is up to 6, and verifying Poly1305, X25519 and Ed25519 took several person-months. High-level APIs like AEAD and SecretBox have comparably little proof substance, and took on the order of a few person-days.

Finally, we provide timings, in seconds, of the time it takes to verify a given algorithm. These are measured on an Intel Xeon workstation without relying on parallelism. The total cost of one-time HACL* verification is a few hours; when extending the library, the programmer writes and proves code interactively, and may wait for up to a minute to verify a fragment depending on its complexity.

The HACL* library is open source and is being actively developed on GitHub. Expert users can download and verify the F* code, and generate the C library themselves. Casual users can directly downloaded the generated C code. The full C library is about 7Kloc and compresses to a 42KB zip file. Restricting the library to just the NaCl API yields 5Kloc, which compresses to a 25KB file. For comparison, the TweetNaCl library is 700 lines of C code and compresses to 6Kb, whereas Libsodium is 95Kloc (including 24K lines of pure C code) and compresses to a 1.8MB distributable. We believe our library is quite compact, auditable, and easy to use.

Measuring Performance. We focus our performance measurements on the popular 64-bit Intel platforms found on modern laptops and desktops. These machines support 128-bit integers as well as vector instructions with up to 256-bit registers. We also measured the performance of our library on a 64-bit ARM device (Raspberry Pi 3) running both a 64-bit and a 32-bit operating system.

On each platform, we measured the performance of the HACL* library in several ways. First, for each primitive, we uses the CPU performance counter to measure the average number of cycles needed to perform a typical operation. (Using the median instead of

Algorithm	HACL*	OpenSSL	libsodium	TweetNaCl	OpenSSL (asm)
SHA-256	13.43	16.11	12.00	-	7.77
SHA-512	8.09	10.34	8.06	12.46	5.28
Salsa20	6.26	-	8.41	15.28	-
ChaCha20	6.37 (ref)	7.84	6.96	-	1.24
	2.87 (vec)				
Poly1305	2.19	2.16	2.48	32.65	0.67
Curve25519	154,580	358,764	162,184	2,108,716	-
Ed25519 sign	63.80	-	24.88	286.25	-
Ed25519 verify	57.42	-	32.27	536.27	-
AEAD	8.56 (ref)	8.55	9.60	-	2.00
	5.05 (vec)				
SecretBox	8.23	-	11.03	47.75	-
Box	21.24	-	21.04	148.79	-

Table 2: Intel64-GCC: Performance Comparison in cycles/byte on an Intel(R) Xeon(R) CPU E5-1630 v4 @ 3.70GHz running 64-bit Debian Linux 4.8.15. All measurements (except Curve25519) are for processing a 16KB message; for Curve25519 we report the number of cycles for a single ECDH shared-secret computation. All code was compiled with GCC 6.3. OpenSSL version is 1.1.1-dev, compiled with no-asm, which disables assembly code; Libsodium version is 1.0.12-stable, compiled with –disable-asm, which disables assembly code and platform-specific optimizations; TweetNaCl version is 20140427.

the average yielded similar results.) Second, we used the SUPERCOP benchmarking suite to compare HACL* with state-of-the-art assembly and C implementations. Third, we used the OpenSSL speed benchmarking tool to compare the speed of the HACL* OpenSSL engine with the builtin OpenSSL engine. In the rest of this section, we describe and interpret these measurements.

Performance on 64-bit Platforms. Table 2 shows our cycle measurements on a Xeon workstation; we also measured performance on other Intel processors, and the results were quite similar. We compare the results from HACL*, OpenSSL, and two implementations of the NaCl API: Libsodium and TweetNaCl. OpenSSL and Libsodium include multiple C and assembly implementations for each primitive. We are primarily interested in comparing like-for-like C implementations, but for reference, we also show the speed of the fastest assembly code in OpenSSL. In the Appendix, Table 4 ranks the top performing SUPERCOP implementations on our test machine, and Table 8 displays the OpenSSL speed measurements.

For most primitives, our HACL* implementations are as fast as (and sometimes faster than) the fastest unvectorized C implementations in OpenSSL, Libsodium, and SUPERCOP. Notably, all our code is significantly faster than the naive reference implementations included in TweetNaCl and SUPERCOP. However, some assembly implementations and vectorized C implementations are faster than HACL*. Our vectorized Chacha20 implementation was inspired by Krovetz's 128-bit vectorized implementation, and hence is as fast as that implementation, but slower than implementations that use 256-bit vectors. Our Poly1305 and Curve25519 implementations rely on 64x64 bit multiplication; they are faster than all other C implementations, but slower than vectorized assembly code. Our Ed25519 code is not optimized (it does not precompute fixed-base scalar multiplication) and hence is significantly slower than the fast C implementation in Libsodium, but still is much faster than the reference implementation in TweetNaCl.

Table 5 measures performance on a cheap ARM device (Raspberry Pi 3) running a 64-bit operating system. The cycle counts were estimated based on the running time, since the processor does not expose a convenient cycle counter. The performance of all implementations is worse on this low-end platform, but on the whole, our HACL* implementations remain comparable in speed with Libsodium, and remains significantly faster than TweetNaCl. OpenSSL Poly1305 and SHA-512 perform much better than HACL* on this device.

Performance on 32-bit Platforms. Our HACL* code is tailored for 64-bit platforms that support 128-bit integer arithmetic, but our code can still be run on 32-bit platforms using our custom library for 128-bit integers. However, we expect our code to be slower on such platforms than code that is optimized to use only 32-bit instructions. Table 6 shows the performance of our code on an ARM device (Raspberry Pi 3) running a 32-bit OS. In the Appendix, Table 7 ranks the top SUPERCOP implementations on this device.

For symmetric primitives, HACL* continues to be as fast as (or faster than) the fastest C implementations of these primitives. In fact, our vectorized Chacha20 implementation is the second fastest implementation in SUPERCOP. However, the algorithms that rely on Bignum operations, such as Poly1305, Curve25519, and Ed25519, suffer a serious loss in performance on 32-bit platforms. This is because we represent 128-bit integers as a pair of 64-bit integers, and we encode 128-bit operations in terms of 32-bit instructions. Using a generic 64-bit implementation in this way results in a 3x penalty. If performance on 32-bit machines is desired, we recommend writing custom 32-bit implementations for these algorithms. As an experiment, we wrote and verified a 32-bit implementation of Poly1305 and found that its performance was close to that of Libsodium. We again note that even with the performance penalty, our code is faster than TweetNaCl.

CompCert Performance. Finally, we evaluate the performance of our code when compiled with the new 64-bit CompCert compiler (version 3.0) for Intel platforms. Although CompCert supports 64-bit instructions, it still does not provide 128-bit integers. Consequently, our code again needs to encode 128-bit integers as

Algorithm	HACL*	Libsodium	TweetNaCl
SHA-256	25.71	30.87	-
SHA-512	16.15	26.08	97.80
Salsa20	13.63	43.75	99.07
ChaCha20 (ref)	10.28	17.69	-
Poly1305	13.89	10.79	111.42
Curve25519	980,692	458,561	4,866,233
Ed25519 sign	276.66	70.71	736.07
Ed25519 verify	272.39	58.37	1153.42
Chacha20Poly1305	23.28	28.21	-
NaCl SecretBox	27.51	54.31	206.36
NaCl Box	94.63	83.64	527.07

Table 3: Intel64-CompCert: Performance Comparison in cycles/byte on an Intel(R) Xeon(R) CPU E5-1630 v4 @ 3.70GHz running 64-bit Debian Linux 4.8.15. Code was compiled with CompCert 3.0.1 with a custom library for 128-bit integers.

pairs of 64-bit integers. Furthermore, CompCert only includes verified optimizations and hence does not compile code that is as fast as GCC. Table 3 depicts the performance of HACL*, Libsodium, and TweetNaCl, all compiled with CompCert. As with 32-bit platforms, HACL* performs well for symmetric algorithms, and suffers a penalty for algorithms that rely on 128-bit integers. If CompCert supports 128-bit integers in the future, we expect this penalty to disappear.

9 CONCLUSION

We presented the design, implementation, and evaluation of HACL*, an open-source verified cryptographic library that implements the full NaCl API and many of the core primitives used in TLS 1.3. All our code is verified to be memory safe, secret independent, and functionally correct with respect to high-level, concise RFC-based specifications. We deliver verified C code that can be readily integrated into existing software. Our code is already being used in larger verification projects like miTLS. In collaboration with Mozilla, parts of HACL* code are also being incorporated within the NSS cryptographic library, and should soon be used by default in the Firefox web browser.

HACL* continues to evolve as we add more primitives and faster implementations. The performance of our library is already comparable to state-of-the-art C implementations and is within a small factor of hand-optimized assembly code. Our results indicates that security researchers should expect far more than auditability from modern cryptographic libraries; with some effort, their full formal verification is now well within reach.

ONLINE MATERIALS

The HACL* library is being actively developed as an open source project at:

https://github.com/mitls/hacl-star/

All the code, specifications, and benchmarks mentioned in this paper are available at the above URL, along with instructions for installing our verification and compilation tools.

REFERENCES

[1] 2015. ChaCha20 and Poly1305 for IETF Protocols. IETF RFC 7539. (2015).

[2] 2016. Elliptic Curves for Security. IETF RFC 7748. (2016).

[3] 2017. Edwards-Curve Digital Signature Algorithm (EdDSA) . IETF RFC 8032. (2017).

[4] 2017. The Transport Layer Security (TLS) Protocol Version 1.3. IETF Internet Draft 20. (2017).

[5] 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. https://doi.org/10.1145/3009837.3009878

[6] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2015. Verifiable side-channel security of cryptographic implementations: constant-time MEE-CBC. *IACR Cryptology ePrint Archive* 2015 (2015), 1241. http://eprint.iacr.org/2015/1241

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

[8] Andrew W Appel. 2015. Verification of a cryptographic primitive: SHA-256. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 37, 2 (2015), 7.

[9] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. 2014. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1267–1279.

[10] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. 2011. Computer-aided security proofs for the working cryptographer. In *Annual Cryptology Conference*. 71–90.

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

[12] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *USENIX Security Symposium*. 207–221.

[13] Daniel J. Bernstein. 2005. The Poly1305-AES Message-Authentication Code. In *Fast Software Encryption (FSE)*. 32–49.

[14] Daniel J Bernstein. 2006. Curve25519: new Diffie-Hellman speed records. In *Public Key Cryptography-PKC 2006*. Springer, 207–228.

[15] Daniel J Bernstein. 2008. ChaCha, a variant of Salsa20.

[16] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering* (????), 1–13.

[17] 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)*. Springer, 159–176.

[18] Daniel J Bernstein and Peter Schwabe. 2012. NEON crypto. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 320–339.

[19] 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)*. Springer, 64–83.

[20] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Catalin Hritcu, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. 2017. Verified Low-Level Programming Embedded in F*. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

[21] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. 2013. Implementing TLS with Verified Cryptographic Security. In *IEEE Symposium on Security & Privacy (Oakland)*. 445–462.

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

[23] 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*.

[24] Billy B Brumley, Manuel Barbosa, Dan Page, and Frederik Vercauteren. 2012. Practical realisation and elimination of an ECC-related software bug attack. In *Topics in Cryptology–CT-RSA 2012*. Springer, 171–186.

[25] 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 *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 299–309.

[26] Quynh H Dang. 2008. The Keyed-Hash Message Authentication Code (HMAC). NIST FIPS-198-1. (2008).

[27] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Beguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue. 2017. Implementing and Proving the TLS 1.3 Record Layer. In *IEEE Symposium on Security and Privacy (Oakland)*. 463–482.

[28] Martin Goll and Shay Gueron. 2014. Vectorization on ChaCha stream cipher. In *Information Technology: New Generations (ITNG)*. 612–615.

[29] Chris Hawblitzel, Jon Howell, Jacob R Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 165–181.

[30] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 207–220.

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

[32] Erick Nascimento, Łukasz Chmielewski, David Oswald, and Peter Schwabe. 2016. Attacking embedded ECC implementations through cmov side channels. In *Selected Areas in Cryptology – SAC 2016 (Lecture Notes in Computer Science)*.

[33] Pierre-Yves Strub, Nikhil Swamy, Cedric Fournet, and Juan Chen. 2012. Self-Certification: Bootstrapping Certified Typecheckers in F* with Coq. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 571–584.

[34] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent Types and Multi-Monadic Effects in F*. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 256–270.

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

[36] National Institute of Standards US Department of Commerce and Technology (NIST). 2012. Federal Information Processing Standards Publication 180-4: Secure hash standard (SHS). (2012).

[37] Jean Karim Zinzindohoue, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. 2016. A Verified Extensible Library of Elliptic Curves. In *IEEE Computer Security Foundations Symposium (CSF)*. 296–309.

PERFORMANCE BENCHMARKS

Algorithm	Implementation	Language	Architecture	Cycles
ChaCha20	moon/avx2/64	assembly	AVX2	1908
	dolbeau/amd64-avx2	C	AVX2	2000
	goll_guerin	C	AVX2	2224
	krovetz/avx2	C	AVX2	2500
	moon/avx/64	assembly	AVX	3584
	moon/ssse3/64	assembly	SSSE3	3644
	krovetz/vec128	C	SSSE3	4340
	hacl-star/vec128	C	SSSE3	4364
	moon/sse2/64	assembly	SSE2	4528
	e/amd64-xmm6	assembly	SSE	4896
	e/x86-xmm6	assembly	SSE	5656
	hacl-star/ref	C	x86_64	9248
	e/amd64-3	assembly	x86_64	9280
	e/ref	C	x86	9596
Poly1305	moon/avx2/64	assembly	AVX2	2508
	moon/avx/64	assembly	AVX	4052
	moon/sse2/64	assembly	SSE2	4232
	hacl-star	C	x86_64	5936
	amd64	assembly	x86_64	8128
	x86	assembly	x86	8160
	53	C	x86	11356
	avx	assembly	AVX	13480
	ref	C	x86	111212
Curve25519	amd-64-64	assembly	x86_64	580132
	sandy2x	assembly	AVX	595272
	amd-64-51	assembly	x86_64	617244
	hacl-star	C	x86_64	632544
	donna_c64	C	x86_64	635620
	donna	assembly	x86	1026040
	ref10	C	x86	1453308
	athlon	assembly	x86	1645992
	ref	C	x86	17169436
SHA-512	openssl	assembly	x86	9028
	ref	C	x86	12620
	sphlib	C	x86	13396
	hacl-star	C	x86	15844
Ed25519	amd64-64-24k	assembly	x86_64	235932
	ref10	C	x86	580232
	hacl-star	C	x86_64	1353932
	ref	C	x86	5234724

Table 4: Intel64 SUPERCOP Benchmarks: ranked list of best performing implementations on an Intel(R) Xeon(R) CPU E5-1630 v4 @ 3.70GHz running 64-bit Debian Linux 4.8.15. All numbers are estimated CPU cycles. Curve25519 is measured for two variable-base and two fixed-base scalar multiplications. All other primitives are measured for an input of 1536 bytes: Chacha20 is measured for a single encryption; Poly1305 is measured for one MAC plus one verify; SHA-512 is measured for a single hash computation; Ed25519 is measured for one sign plus one verify.

Algorithm	Operation	HACL*	OpenSSL (C)	Libsodium (C)	TweetNaCl	OpenSSL (asm)
SHA-256	Hash	45.83	40.94	37.00	-	14.02
SHA-512	Hash	34.76	20.58	27.26	37.70	15.65
Salsa20	Encrypt	13.50	-	27.24	40.19	-
ChaCha20	Encrypt	17.85 (ref) 14.45 (vec)	30.73	19.60	-	9.61
Poly1305	MAC	11.09	7.05	10.47	310.84	3.00
Curve25519	ECDH	833,177	890,283	810,893	5,873,655	-
Ed25519	Sign	310.07	-	84.39	1157.73	-
Ed25519	Verify	283.86	-	105.27	2227.41	-
Chacha20Poly1305	AEAD	29.32	26.48	30.40	-	13.05
NaCl SecretBox	Encrypt	24.56	-	38.23	349.96	-
NaCl Box	Encrypt	85.62	-	97.80	779.91	-

Table 5: AARCH64-GCC: Performance Comparison in cycles/byte on an ARMv7 Cortex A53 Processor @ 1GHz running 64-bit OpenSuse Linux 4.4.62. All code was compiled with GCC 6.2.

Algorithm	HACL*	OpenSSL	Libsodium	TweetNaCl	OpenSSL (asm)
SHA-256	25.70	30.41	25.72	-	14.02
SHA-512	70.45	96.20	101.97	100.05	15.65
Salsa20	14.10	-	19.47	21.42	-
ChaCha20	15.21 (ref) 7.66 (vec)	18.81	15.59	-	5.2
Poly1305	42.7	17.41	7.41	140.26	1.65
Curve25519	5,191,847	1,812,780	1,766,122	11,181,384	-
Ed25519	1092.83	-	244.75	1393.16	-
Ed25519	1064.75	-	220.92	2493.59	-
Chacha20Poly1305	62.40	33.43	23.35	-	7.17
NaCl SecretBox	56.79	-	27.47	161.94	-
NaCl Box	371.67	-	135.80	862.58	-

Table 6: ARM32-GCC: Performance Comparison in cycles/byte on an ARMv7 Cortex A53 Processor @ 1GHz running 32-bit Raspbian Linux 4.4.50. All code was compiled with GCC 6.3 with a custom library providing 128-bit integers.

Algorithm	Implementation	Language	Architecture	Cycles
ChaCha20	moon/neon/32	assembly	NEON	9694
	hacl-star/vec128	C	NEON	12602
	dolbeau/arm-neon	C	NEON	13345
	hacl-star/ref	C	NEON	17691
	moon/armv6/32	assembly	ARM	18438
	e/ref	C	ARM	22264
Poly1305	moon/neon/32	assembly	NEON	10475
	neon2	assembly	NEON	11403
	moon/armv6/32	assembly	ARM	18676
	53	C	ARM	20346
	hacl-star	C	ARM	127134
	ref	C	ARM	395722
Curve25519	neon2	assembly	NEON	1935283
	ref10	C	ARM	4969185
	hacl-star	C	ARM	13352774
	ref	C	ARM	60874070
SHA-512	sphlib	C	ARM	82589
	ref	C	ARM	118118
	hacl-star	C	ARM	121327
Ed25519	ref10	C	ARM	2,093,238
	ref	C	ARM	18,763,464
	hacl-star	C	ARM	29,345,891

Table 7: ARM32 SUPERCOP Benchmarks: ranked list of best performing implementations on an ARMv7 Cortex A53 Processor @ 1GHz running 32-bit Raspbian Linux 4.4.50.

Algorithm	Implementation	16by	64by	256by	1024by	8192by	16384by
ChaCha20	HACL*	90381.10k	353297.74k	377317.29k	380701.70k	386591.17k	385418.53
	HACL* vec	115770.29k	486701.81k	728594.24k	860998.38k	910695.60k	924024.72
	OpenSSL C	204657.84k	318616.27k	342565.63k	346045.80k	371442.81k	370262.02
	OpenSSL ASM	285974.37k	526845.47k	1165745.92k	2382449.36k	2452002.59k	2470173.90
ChachaPoly	HACL*	39405.99k	143626.18k	238075.98k	277331.74k	292995.07k	302145.07
	OpenSSL C	169799.71k	262761.53k	285738.89k	304376.49k	300509.41k	290193.41
	OpenSSL ASM	217872.74k	399483.59k	848875.62k	1518847.66k	1632862.87k	1638246.57
SHA-256	HACL*	20331.67k	54075.54k	106500.44k	141369.19k	158401.50k	153695.16
	OpenSSL C	18121.99k	49251.87k	104402.28k	144965.29k	161028.97k	166327.74
	OpenSSL ASM	25321.67k	78481.92k	201910.03k	310514.47k	375845.67k	389046.03
SHA-512	HACL*	16513.59k	65673.72k	127720.99k	201159.46k	234087.09k	236592.63
	OpenSSL C	17280.47k	68173.85k	135549.35k	213524.48k	263108.41k	264705.37
	OpenSSL ASM	20556.52k	82447.35k	194595.05k	368933.21k	519731.71k	546442.02
Poly1305	HACL*	33945.66k	125367.98k	382090.15k	817432.47k	1204432.92k	1246641.57
	OpenSSL C	35947.80k	134963.35k	421210.62k	928101.54k	1355694.08k	1418755.77
	OpenSSL ASM	33354.96k	125854.18k	433647.19k	1383256.87k	3630256.03k	4032672.28
Curve25519	HACL*	144895					
	OpenSSL C	68107					

Table 8: OpenSSL speed comparison for our algorithms. Each algorithm is run repeatedly for three seconds on different input sizes, and we measure the number of bytes per second via the openssl speed command. The experiment is performed on an Intel Core i7 @ 2.2Ghz running OSX 10.12.4. For Curve25519, we measure the number of ECDH computations per second.

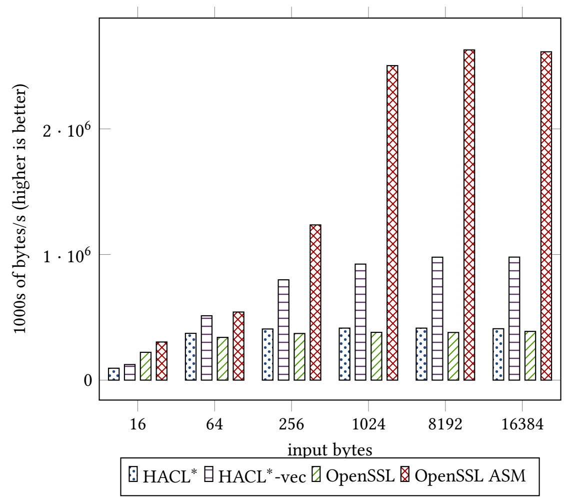

Figure 10: OpenSSL speed comparison for the Chacha20 algorithm. The algorithm is run repeatedly for three seconds on different input sizes, and we measure the number of operations via the openssl speed command. The experiment is performed on an Intel Core i7 @ 2.2Ghz running OSX 10.12.4.

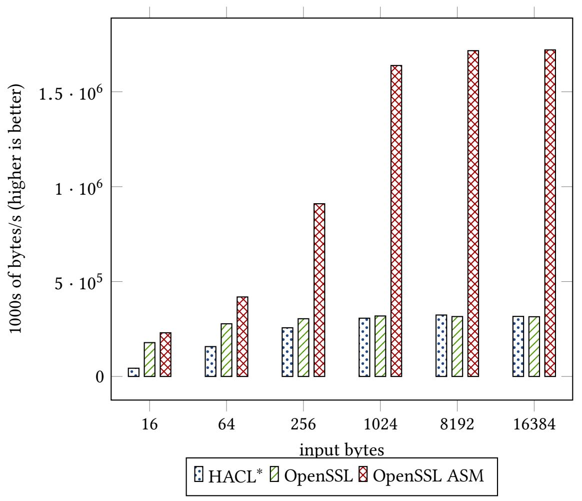

Figure 11: OpenSSL speed comparison for the AEAD algorithm. The algorithm is run repeatedly for three seconds on different input sizes, and we measure the number of operations via the openssl speed command. The experiment is performed on an Intel Core i7 @ 2.2Ghz running OSX 10.12.4.

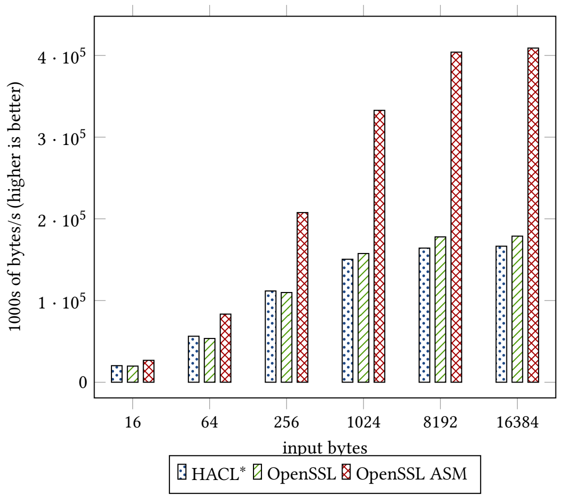

Figure 12: OpenSSL speed comparison for the SHA2-256 algorithm. The algorithm is run repeatedly for three seconds on different input sizes, and we measure the number of operations via the openssl speed command. The experiment is performed on an Intel Core i7 @ 2.2Ghz running OSX 10.12.4.

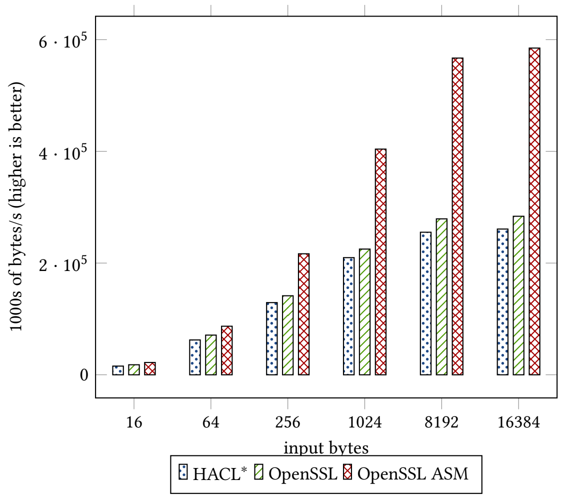

Figure 13: OpenSSL speed comparison for the SHA2-512 algorithm. The algorithm is run repeatedly for three seconds on different input sizes, and we measure the number of operations via the openssl speed command. The experiment is performed on an Intel Core i7 @ 2.2Ghz running OSX 10.12.4.

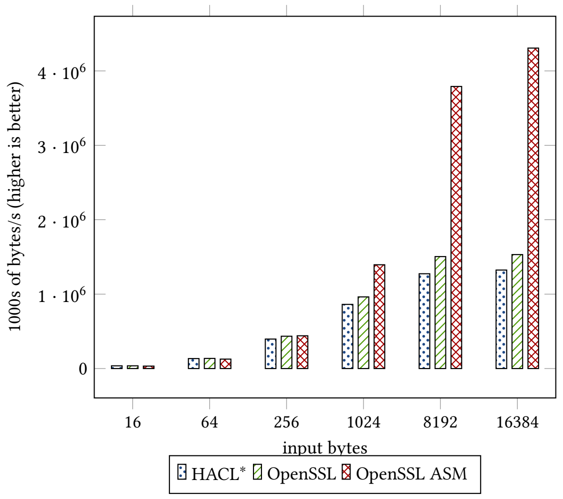

Figure 14: OpenSSL speed comparison for the Poly1305 algorithm. The algorithm is run repeatedly for three seconds on different input sizes, and we measure the number of operations via the openssl speed command. The experiment is performed on an Intel Core i7 @ 2.2Ghz running OSX 10.12.4.

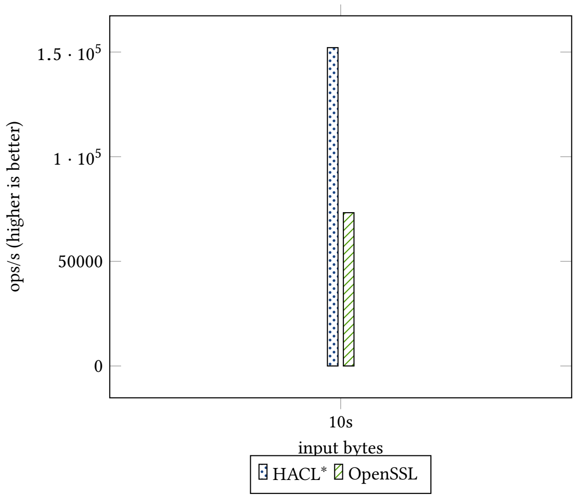

Figure 15: OpenSSL speed comparison for the X25519 algorithm. The algorithm is run repeatedly for ten seconds, and we measure the number of operations via the openssl speed command. The experiment is performed on an Intel Core i7 @ 2.2Ghz running OSX 10.12.4.