HACL×N: Verified Generic SIMD Crypto
(for all your favorite platforms)

Marina Polubelova
Inria

Karthikeyan Bhargavan
Inria

Jonathan Protzenko
Microsoft Research

Benjamin Beurdouche
Inria and Mozilla

Aymeric Fromherz
Carnegie Mellon University

Natalia Kulatova
Inria

Santiago Zanella-Béguelin
Microsoft Research

ABSTRACT

We present a new methodology for building formally verified cryptographic libraries that are optimized for multiple architectures. In particular, we show how to write and verify generic crypto code in the F* programming language that exploits single-instruction multiple data (SIMD) parallelism. We show how this code can be compiled to platforms that supports vector instructions, including ARM Neon and Intel AVX, AVX2, and AVX512. We apply our methodology to obtain verified vectorized implementations on all these platforms for the Chacha20 encryption algorithm, the Poly1305 one-time MAC, and the SHA-2 and Blake2 families of hash algorithms.

A distinctive feature of our approach is that we aggressively share code and verification effort between scalar and vectorized code, between vectorized code for different platforms, and between implementations of different cryptographic primitives. By doing so, we significantly reduce the manual effort needed to add new implementations to our verified library. In this paper, we describe our methodology and verification results, evaluate the performance of our code, and describe its integration into the larger HACL* crypto library. Our vectorized code has already been incorporated into several software projects, including the Firefox web browser.

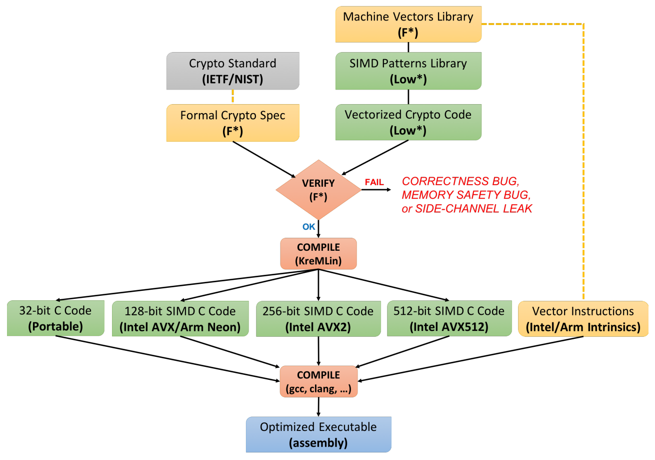
Machine Vectors Library (F*)
Crypto Standard (IETF/NIST)
SIMD Patterns Library (Low*)
Formal Crypto Spec (F*)
Vectorized Crypto Code (Low*)
VERIFY (F*)
FAIL
CORRECTNESS BUG, MEMORY SAFETY BUG, or SIDE-CHANNEL LEAK
OK
COMPILE (KreMLin)
32-bit C Code (Portable)
128-bit SIMD C Code (Intel AVX/Arm Neon)
256-bit SIMD C Code (Intel AVX2)
512-bit SIMD C Code (Intel AVX512)
Vector Instructions (Intel/Arm Intrinsics)
COMPILE (gcc, clang, ...)
Optimized Executable (assembly)

Figure 1: HACL×N programming and verification workflow. We write SIMD crypto code in Low* [29] and prove it memory-safe, secret independent, and functionally correct with respect to a high-level formal spec in F* [34], before compiling it to target-specific C code linked with compiler intrinsics. (Code components in green are verified; those in yellow are trusted and carefully tested.)

1 VERIFIED HIGH-PERFORMANCE CRYPTO

Modern cryptographic algorithms are evaluated not just for their security, but also for their performance on various platforms. Slow algorithms, even if provably secure, are rarely deployed at scale. For example, the Diffie Hellman key exchange was largely unused for decades in mainstream protocols like TLS, even though it provides strong guarantees like forward secrecy, until the advent of fast Elliptic Curve Diffie Hellman implementations. Even today, powerful cryptographic constructions like homomorphic encryption and post-quantum signatures are awaiting faster implementations before they can be considered for widespread deployment.

When a new cryptographic algorithm is standardized, the designers usually describe (and sometimes include as an appendix) a reference implementation that would work on any 32-bit computer. However, the algorithm and its parameters are often chosen carefully to admit platform-specific optimizations. For example, new authenticated encryption schemes like ChaCha20-Poly1305 and hash algorithms like Blake2 were deliberately designed to enable Single Instruction Multiple Data (SIMD) vectorization. Since most desktops and smartphones are now equipped with SIMD-enabled processors capable of computing over 4, 8, or 16 32-bit integers in parallel, the performance impact of vectorization can be dramatic.

Algorithmic Parallelism and SIMD Vectorization. Consider the ChaCha20 cipher, standardized in IETF RFC 7539. The OpenSSL library includes a reference implementation of the Chacha20 encryption algorithm (written by D.J. Bernstein) in 122 lines of portable C code. This C code takes between 4-9 cycles to encrypt a byte on a modern Intel processor. However, OpenSSL also includes at least a dozen other hand-written assembly implementations of Chacha20, for various generations of Intel and ARM processors, totaling 10056 lines of code. The payoff for this additional programming effort is improved performance: encryption takes just 1.6 cycles per byte on an Apple iPhone with the Apple A7 ARMv8 processor, and 0.73 cycles per byte on a modern laptop with an Intel Skylake processor.

An algorithm like Chacha20 can be parallelized in many different ways. First, the inner block cipher can be rearranged and executed via SIMD instructions (this is by design). Second, one may exploit the inherent parallelism in the counter-mode encryption (CTR) algorithm to process multiple blocks in parallel. Third, one may process multiple (equal sized) inputs in parallel. Each of these SIMD strategies can significantly increase performance, at the cost of adding some memory rearrangement via matrix transposition.

SIMD vectorization should be seen as a generic high-level *algorithmic transformation* that changes the shape and structure of a cryptographic computation in order to make it parallelizable. Indeed, processing blocks in parallel works for any CTR algorithm, not just Chacha20. However, since each platform may offer subtly different vector instructions, most vectorized crypto code is hand-written in unreadable low-level assembly that is specialized for the features of each algorithm and target platform.

How can we be sure that all these platform-specific implementations are correct? Testing helps, but the testing matrix gets exponentially bigger as new platforms are added. Furthermore, each new architecture inspires new implementation strategies and, therefore, increases the potential for new bugs. For example, efficient implementations of the Poly1305 algorithm interleave complex bignum arithmetic with SIMD vectorization, and so have to account for the subtleties of both. And in addition to correctness, we also need to ensure that the code is memory safe and secret independent ("constant-time"). How do we make it easier for programmers to build crypto implementations that are provably correct-by-design?

Cryptographic Software Verification. Recent works have explored several different approaches towards building verified cryptographic software [11]. Some works, like the Verified Software Toolchain [9, 14] and the Cryptol/SAW framework can prove the functional correctness of C or Java code against a high-level mathematical specification. Other tools like Vale [17, 24], Jasmin [7, 8], and CryptoLine [25] target hand-written assembly implementations of cryptographic primitives. A third approach, taken by HACL★ [36] and Fiat-Crypto [23], is to write and verify cryptographic code in a high-level programming language and compile it to C.

Each approach has its own advantages and disadvantages. VST and Cryptol can verify legacy C code, but they target unoptimized reference implementations, not vectorized high-performance crypto.

Jasmin, Vale, and CryptoLine can verify highly-optimized assembly implementations written by expert programmers who manually allocate registers (to exploit locality), manually schedule instructions (to exploit pipelining), and exploit low-level hardware instructions that are invisible to C code. Furthermore, by directly targeting assembly, these tools do not need to trust the correctness of the C compiler. The cost of this approach is that one has to write and verify custom low-level assembly code for each platform. For example, Vale and Jasmin do not currently support ARM, and adding a new verified implementation for (say) Intel AVX512 requires a significant amount of new programming and verification effort.

HACL★ and Fiat-Crypto are best suited to writing new cryptographic code and provide high-level abstractions that make programming and verification easier. However, they focus on generating portable C code, and hence do not exploit platform-specific features like vector instructions. For example, the C code generated from HACL★ is highly performant for elliptic curve algorithms, which do not usually rely on SIMD vectorization, but is significantly slower than assembly code for encryption and hashing [36].

These approaches are not mutually exclusive. EverCrypt [28] is a cryptographic provider that composes verified C code from HACL★ with verified assembly code from Vale to obtain best-in-class performance on Intel platforms for elliptic curves like Curve25519 (by relying on Intel ADX instructions) and authenticated encryption schemes like AES-GCM (by relying on Intel AES-NI and CLMUL).

Our Approach. In this paper, we present a new hybrid approach towards building a library of vectorized cryptographic algorithms. We seek to balance coding and verification effort with high performance, by following the high-level programming methodology of HACL★ but compiling it to platform-specific C code that relies on compiler intrinsics for platform-specific vector instructions.

The main insight guiding our approach is that SIMD vectorization is an algorithmic redesign which can be implemented and verified *generically*, without relying on specific details of the underlying platform. Our second observation is that modern C compilers are good enough (and constantly improving) at instruction scheduling and register allocation, so manually writing optimized assembly for each target platform is often not necessary for high performance. Finally, we note that crypto code is typically embedded into larger libraries and applications like browsers and operating systems that already place a lot of trust in the C compiler; so in many deployments, compiling crypto code using a standard C compiler like GCC or clang does not increase the Trusted Computing Base.

Figure 1 depicts our high-level methodology and workflow as a sequence of programming, verification, and compilation tasks:

High-Level Spec We write a succinct formal specification for each crypto algorithm in the F★ language [34], by carefully transcribing the corresponding IETF or NIST standard. This specification is trusted but executable; it serves as a testable, readable, reference implementation that can be audited by cryptographers.

Generic Vector Library We extend HACL★ with libraries for machine integers and vectors of integers, designed to enable generic SIMD programming. We implement the vector library as a (trusted) C header file that maps each vector operation to platform-specific vector instructions for Arm Neon, Intel AVX, AVX2, and AVX512.

SIMD Patterns for Crypto We identify, implement, and verify a series of reusable SIMD programming patterns commonly used in crypto algorithms, including generic constructions for multi-buffer parallelism, CTR encryption, and polynomial evaluation.

Verified Vectorized Implementations We build vectorized implementations of Blake2 (both versions), SHA-2 (all four versions), Chacha20, and Poly1305, in Low★ [29] (a subset of F★). These generic implementations are parameterized over a target vector size and can be instantiated with vectors of any size: 1, 2, 4, 8, 16, etc. (Vectors of size 1 correspond to scalar code.)

Target-Specific Compilation We exploit the meta-programming features of F★ to translate our generic Low★ implementations to custom C implementations for each target platform. The compiler generates both portable 32-bit C code and vectorized C code for Arm Neon and Intel AVX/AVX2/AVX512. Each C implementation can then be compiled via GCC or CLANG to target machine code.

We illustrate this workflow on four (families of) cryptographic algorithms, but our methodology is more generally applicable to other algorithms, and even for non-cryptographic SIMD code. To the best of our knowledge, ours are the first verified vectorized crypto implementations on Arm Neon and AVX512, and the first verified implementations of vectorized Blake2 and multi-buffer SHA-2. Our vectorized Chacha20-Poly1305 code is deployed in Mozilla Firefox Nightly, and our Blake2 code is used in Tezos.

Our goal is to create a usable verified crypto *library*, not just a few isolated verified algorithms. Hence, we show how to embed these vectorized algorithms into HACL⋆ and safely compose them with previously verified C and assembly code [17, 28, 36]. The resulting library provides optimized verified code for many of the most popular ciphersuites used in modern protocols like TLS 1.3, WireGuard, and Signal. By aggressively sharing code and verification effort across platforms and algorithms, we significantly reduce the cost of adding a new platform or algorithm to the library. Finally, we show how we can use our methodology to easily generate customized compact optimized verified cryptographic applications, such as the upcoming HPKE standard [13].

2 BACKGROUND: HACL⋆, F⋆, LOW⋆

The HACL⋆ cryptographic library [36] contains verified implementations of several modern cryptographic constructions such as ChaCha20-Poly1305, AES-GCM, SHA-2 (256/384/512), Curve25519, and Ed25519. HACL⋆ is written in the F⋆ programming language [6] and compiled to C for easy adoption in the existing software ecosystem. The source code is verified using the F⋆ type checker for memory safety and for a side-channel resistance guarantee called secret independence, which states that secret data cannot be used in branches or to compute memory addresses. Crucially, each implementation in HACL⋆ is verified for functional correctness against a high-level specification also written in F⋆, but closely reflecting the corresponding published crypto standard. The C code generated from our verified F⋆ library is typically as fast as hand-optimized scalar C code for each primitive. We now review these for context.

F⋆ is a state-of-the-art verification-oriented programming language [6]. It is a functional programming language with dependent types and an effect system, and it relies on SMT-based automation to prove properties about programs using a weakest-precondition calculus. Once proven correct with regards to their specification, programs written in F⋆ can be compiled to OCaml or F#. Recently [30], F⋆ gained the ability to generate C code, as long as the run-time parts of the program are written in a low-level subset called Low⋆. This allows the programmer to use the full power of F⋆ for proofs and verification and, relying on the fact that proofs are computationally irrelevant and hence erased, extract the remaining Low⋆ code to C. This approach was used by several verified software projects, such as HACL⋆, but also a cryptographic provider [28], a parsing library [31], and a QUIC implementation [20].

Specifications are expressed using the pure subset of F⋆, i.e. cannot have side-effects. Specifications use high-level concepts, such as mathematical (unbounded) integers, sequences, lists, maps and folds. As such, specifications cannot extract to C; they are, however, extracted to OCaml, and subjected to a substantial amount of testing via standard vectors, to ensure they are trustworthy.

For instance, our Poly1305 specification module Spec.Poly1305 defines a prime field $\mathbb{Z}_{2^{130}-5}$ in F⋆ as follows:

```
let pos = n:nat{n > 0}
let prime: pos = pow2 130 − 5
let felem = x:nat{x < prime}
let fadd (x:felem) (y:felem) : felem = (x + y) % prime
let fmul (x:felem) (y:felem) : felem = (x * y) % prime
```

The type pos is a *refinement* of nat, the type of natural integers. The prime modulus is a mathematical constant, and field operations are defined naturally via the use of mathematical operators for addition, multiplication and modulo. The Poly1305 MAC is then defined as the evaluation of a polynomial over this field (see §4.4).

Low⋆ is a low-level subset of F⋆ which models the C memory layout and basic data types. Using Low⋆, the programmer manipulates arrays, reasoning about their liveness, disjointness and location in memory. Low⋆ uses machine integers (instead of mathematical integers), which forces the programmer to reason about their modulo-semantics. Each stateful function is defined using one of the Low⋆ *effects*: Stack for functions that are only allowed to allocate memory on the stack (hence trivially ensuring there are no memory leaks), ST for all other Low⋆ functions.

As such, Low⋆ requires the programmer to make choices about low-level data representations for every high-level specification value. We present the low-level signature of a non-vectorized init function, which establishes the initial state for a low-level implementation of Poly1305. (§4 presents the vectorized implementation.)

```
let poly1305_ctx = lbuffer uint32 5
val poly1305_init (ctx:poly1305_ctx) (key:lbuffer uint8 32ul): Stack unit
  (requires λh → live h ctx ∧ live h key ∧ disjoint ctx key)
  (ensures λh0 _h1 → modifies (loc ctx) h0 h1 ∧ state_inv h1 ctx ∧
    (as_acc h1 ctx, as_r h1 ctx) ==
      Spec.Poly1305.poly1305_init (as_seq h0 key))
```

The state ctx is an array of type lbuffer holding 5 elements of type uint32. The pre-condition of init talks about the *liveness* and *disjointness* of the input arrays, needed for temporal and spatial memory safety. Callers can reason about the effects of this function on memory, via its modifies clause: only ctx is modified, leaving key or any other disjoint object unchanged. The post-condition guarantees that the result is a valid initial state vis-à-vis the spec.

KreMLin compiles Low⋆ code to *auditable, readable* C, using a series of many small, composable passes. The Low⋆ preservation theorem [30] states that once compiled to C, a Low⋆ program exhibits the same observable behavior that was specified by the Low⋆ memory model and semantics, including execution traces.

Meta-F⋆ broadly, refers to the discipline of relying on the F⋆ compiler to generate or transform existing code at F⋆-compile-time. Some meta-programming is built into F⋆, through keywords.

```
inline_for_extraction let pow4 (x: uint32 { x < 256 }) =
  [@inline_let] let pow2 = x * x in
  pow2 * pow2
```

At compile-time, F⋆, seeing that pow4 is "inline for extraction", replaces any call to pow4 with its definition; furthermore, seeing that pow4 is a pure computation that does not touch memory, the KreMLin compiler can further reduce the application of the function, eliminate the intermediary let-binding ("inline let"). Hence, if the programmer writes let x = pow4 2ul, F⋆ reduces this to let x = 16ul.

Beyond these built-in meta-programming facilities, Meta-F⋆ provides a general-purpose meta-programming framework [27], where the meta-language is F⋆ itself, an approached known as elaborator reflection and pioneered by the Lean and Idris theorem provers [18, 19]. Essentially, programs written in a special Tac effect are executed by F⋆ at compile-time and allow scripting the compiler, for proof purposes or code-generation purposes. When a meta-program generates code, F⋆ re-checks it, hence ensuring that meta-programs cannot generate ill-typed terms.

3 WRITE & VERIFY ONCE; COMPILE N TIMES

HACL×N is a new SIMD-oriented extension of HACLv1, where the motto is to *verify once*, but *extract and specialize* many times. This approach of maximizing programmer productivity is achieved through an intentional, careful scaffolding of libraries and techniques that leverage the latest advances in F⋆, including Meta-F⋆, and encourage the programmer to author generic code.

We now illustrate how this methodology plays out at all levels of granularity in HACL×N, starting with overloading in integer libraries and ending with C++-like templated meta-programming.

3.1 Universal integer and buffer libraries

By virtue of being a faithful and precise model of C, Low⋆ defines 9 different types of machine integers, including an unsigned 128-bit type. Similarly, the verification arsenal of Low⋆ features many different types of arrays, depending on whether the array is mutable or immutable, initialized with a default value or not, along with another abstraction for const pointers. HACL⋆ uses 3 of those.

Writing code directly on top of these Low⋆ base libraries is not just tedious: it is also *unproductive*. Consider a simple dereference-then-add: writing every possible instance of *x + *y in Low⋆ might require as many as 27 variants! Any C++ programmer would write template<T> T f(T *x, T *y) { return *x + *y; }, or a macro if in C.

In order to reconcile the convenience of templates or macros with precise specifications and semantics, we introduce an *agile* layer of libraries that offers universal operators for machine integers and for pointer manipulation. The module Lib.IntTypes is as follows.

```
type inttype = | U8 | U16 | U32 | U64 | U128 | S8 | S16 | S32 | S64
type secrecy_level = | SEC | PUB

inline_for_extraction val sec_int_t: inttype → Type0
inline_for_extraction let int_t (t:inttype) (l:secrecy_level) =
  match l, t with
  | PUB, U8 → LowStar.UInt8.t
  | SEC, _ → sec_int_t t | ...

inline_for_extraction val (+!): #t:inttype → #l:secrecy_level
  → a:int_t t l → b:int_t t l{range (v a + v b) t} → int_t t l
```

The type int_t is parameterized over two indices: inttype enumerates all known variants of integer types, while secrecy_level distinguishes public data from secret data. The former allows hiding the proliferation of integer models under a single type. The latter brings HACLv1's side-channel resistance modeling under the same abstraction, thus relieving the programmer from having to deal with yet another set of operators for secret data.

Hence, the operator +! for non-overflowing addition works for any width and secrecy level. The # denotes implicit arguments, inferred automatically by F⋆. Thanks to inlining, as long as +! is used at call-site with concrete values for t and l, F⋆ will normalize away the cascade of matches, resulting on a monomorphic addition on the desired machine integer, which the Low⋆ toolchain will extract either to C's native + or a library call such as LowStar_UInt128_add.

Nothing is revealed about the nature of secret integers: this means that the only operations over them are those offered by Lib.IntTypes. In particular, operations that are known to be non constant-time (e.g. division) require in their precondition that the arguments be public. Owing to the abstract type, F⋆ will treat any attempt to use a secret integer for branching or memory access as a type error, as these require bool and LowStar.UInt32.t respectively.

The library defines several abbreviations for convenience, such as let uint32 = int_t U32 SEC or uint8, which we use in this paper.

The same discipline of overloaded, universal operators is applied to array operations. Our module Lib.Buffer defines buffer #b t, where b is an index that can be one of MUT, IMMUT or CONST.

Equipped with agile operators, we define the earlier dereference-then-add snippet in a form that is both concise and generic.

```
let deref_add #b #t #s (x y: buffer #b (int_t #t #s)): Stack (int_t #t #s) ... =
  x.[0ul] + y.[0ul]
```

3.2 An abstract vector type for SIMD code

SIMD programming in C usually requires dealing with a patchwork of headers and compiler builtins. These depend on the target instruction set, as Intel and ARM offer different intrinsics.

In order to establish clear interfaces and abstraction boundaries, we first abstract away platform differences by introducing Lib.IntVector.Intrinsics, the machine vector library (Figure 1). This module axiomatizes vector operations that are general enough to be implemented using, say, either Neon or AVX. We carefully audit its semantics, and perform rigorous testing to ensure that our specifications carefully capture the intrinsics' behavior.

At compile-time, we provide hand-written definitions for the functions defined in Lib.IntVector.Intrinsics using macros. As an example, the module defines vec128_xor; we hand-write a macro that calls _mm_xor_si128 (AVX) or veorq_u32 (Neon).

Exposing a compendium of vector operations is useful, but does not enable programmer productivity. As for integer types, we define a vector library exposing a curated interface, designed to minimize verification effort by using agility as well as strong abstractions.

```
inline_for_extraction val vec_t: t:inttype → width:nat → Type0
inline_for_extraction val (+|): #t:inttype → #w:width →
  v1:vec_t t w → v2:vec_t t w → vec_t t w

val vec_v: #t:inttype → #w:width → vec t w → lseq (uint_t t SEC) w
val vec_add_mod_lemma: #t:inttype → #w:width →
  v1:vec_t t w → v2:vec_t t w → Lemma (ensures (
    vec_v (vec_add_mod v1 v2) == map2 (+.) (vec_v v1) (vec_v v2)))
```

The type vec_t is an abstract type, shielding clients from the complexity and variety of Lib.IntVector.Intrinsics. (This is crucial for SMT-based verification, where proof performance degrades as too many definitions enter the SMT context.) A vec_t is parametric over its length, or width, and the type t of its elements. The module only offers constructors for valid combinations of width and t.

Similarly to +!, we define +| which is the point-wise lifting of integer addition to vectors. Just like with integers, both the type (vec_t) and the operations (+|) are evaluated away by F* at verification-time, leaving bare calls to Lib.IntVector.Intrinsics which later become macro-expanded into the right intrinsics.

Reasoning about the semantics of vector operations is done with the vec_v function, which reflects a low-level vector as a pure data structure suitable for specification and reasoning, i.e. a sequence of the right integer type and the right length. Addition is thus specified via vec_v and +., addition-modulo restricted to unsigned integers.

3.3 Representation-agnostic algorithms

HACL×N embraces genericity not just at the level of operators, but also for entire algorithms. By making sure algorithms are *parametric* over their choice of representation, we only need to verify them *once*, but can extract and specialize them *many times*. (We leave a detailed discussion of SIMD techniques to §4 and discuss here compilation and code specialization aspects.)

For illustration, we turn to Poly1305 and revisit our earlier example (§2). This time, instead of being generic over the integer width or the flavor of buffer, we introduce an index that captures genericity at a larger scale over the choice of low-level representation:

```
type field_spec = | M32 | M128 | M256 | M512
```

We make our Poly1305 implementation parametric over field_spec, ensuring that it works for *any* choice of representation s. Doing so, we aim to maximize code sharing, i.e. never examine the value of s if avoidable: doing a case-split over field_spec requires considering four different cases, which increases the verification burden.

For the state type, we have to discriminate over s. We define poly1305_ctx to be a pointer to vectors of a suitable width:

```
let poly1305_ctx (s: field_spec) = match s with
  | M32 → buffer MUT (vec_t U64 1)
  | M128 → buffer MUT (vec_t U64 2) | ...
```

For functions, we sometimes need to perform a case analysis over s. This, however, is limited to helpers that sit at the leaves of the call-graph. For instance, one such helper might compute the block length associated to a particular representation s:

```
inline_for_extraction let blocklen (s:field_spec): int_t U32 PUB =
  match s with
  | M32 → 16ul
  | M128 → 32ul | ...
```

Helpers are by nature simple functions that pose no particular verification challenge, meaning that the cost of examining four cases is very low. The core of Poly1305 lies outside of helpers and is concerned with deep proofs and complex code. Fortunately, unlike helpers, the Poly1305 core never talks about the concrete value of s, meaning nearly all of Poly1305 is written and verified once for all four representations. (Importantly, this generic code will not need to be updated if we extend field_spec with another case.)

As an example, poly1305_update loops over the input data in block-sized chunks, then processes each block at a time. We write a generic proof stating that block-by-block processing has the desired semantics. Nowhere does the proof refer to the actual value of s, except indirectly in the 5-line definition of blocklen. We thus obtain:

```
inline_for_extraction val poly1305_update: #s:field_spec →
  ctx:poly1305_ctx s → len:size_t → text:lbuffer uint8 len → Stack unit
```

As it stands, poly1305_update cannot be extracted to C, as it manipulates types of the form poly1305_ctx s, where the value of s is not known. Such types are not valid Low*; even if they were, they would be extracted to a C union of all possible cases, which is not our goal. To generate valid Low* code, it suffices to apply the function to a concrete value for its argument s.

```
let update32 = poly1305_update #M32
```

At compile-time, F* processes the inline_for_extraction's and replaces all the functions and types of Poly1305 with their definitions:

```
let update32 =
  (λ #s (ctx: buffer MUT ((λ | M32 → vec_t U64 1 | ...) s) → ...) #M32
```

The first ... stands for the rest of the definition of poly1305_ctx, and the second for the remainder of the arguments of poly1305_update along with its body. F* then performs β-reduction, applying the outer M32 first, then reducing ((λ| M32 → vec_t U64 1 | ...) M32) to vec U64 1, and eventually to LowStar.Buffer.buffer LowStar.UInt64.t.

All that is left is a fully monomorphic implementation of Poly1305, specialized to a scalar system that uses a 64-bit integer for the internal state. Once compiled, the C function prototype is thus:

```
void Poly1305_32_update32(uint64_t *ctx, uint32_t len, uint8_t *text);
```

There is no verification cost associated to performing a partial application of poly1305_update to a concrete argument: the three other cases, for 128, 256 and 512-bit specialized variants of Poly1305, also come for free, meaning our proof-to-C-code ratio is nearly divided by 4 compared to HACLv1.

3.4 Large-Scale Program Specialization

The technique described above suffers from one caveat: the entire algorithm must be inlined into the top-level function in order to get fully applied matches to appear and be reduced away by F*. While this is fine for a mid-size algorithm such as Poly1305, for a larger piece of code such as HPKE (§5.2), this would generate prohibitively large and unreadable C code.

We now address very-large scale genericity, and use the full power of Meta-F* to solve this problem. We have written a tactic (i.e. a meta-program) that takes an algorithm written in a normal style and transforms its entire call-graph, rewriting the algorithm in a form similar to C++ templatized code. After rewriting, the programmer can generate specialized versions of their code like we did above for Poly1305, with the added benefit that the structure of the call-graph is preserved, rather than inlined away.

The tactic takes upon the burden of rewriting the code in a slightly more convoluted form, meaning there is no extra cost for its users. It is flexible, and allows the programmer to annotate their code to specify which functions should be inlined away and which ones should remain at extraction-time. As mentioned in §2, the tactic is not part of the TCB, since whatever code the tactic generates is type-checked again by F*. At the time of writing, our tactic is the second largest Meta-F* program written (> 600 LoC), and is used in almost every algorithm in HACL×N.

We now illustrate the mode of operation of our tactic on HPKE. HPKE is a high-level cryptographic functionality described extensively in §5.2 – suffices to know, for now, that HPKE calls into a

Diffie-Hellman (DH), an AEAD and a hash algorithm. HPKE is defined for *any* triplet of algorithms that implement their respective *functionalities*, meaning that HPKE is generic at a very large scale over the following index:

```
type hpke_index = dh_alg & aead_alg & hash_alg
```

HPKE differs from previous examples as each value of the index may admit multiple instances. For instance, fixing aead_alg to be Chacha20Poly1305 still allows four possible ChaCha-Poly implementations, one for each degree of vectorization. Naturally, we want to verify HPKE once, for each possible triplet of *algorithms* (*not* implementations), and enjoy specialization for free for any combination of implementations. Our tactic allows just that.

$$\text{sealBase} \overset{\text{calls}}{\rightarrow} \text{sealBase}_{\text{aux}} \overset{\text{calls}}{\rightarrow} \text{AEAD.encrypt}$$

The (simplified) call-graph of HPKE is described above. Using F⋆'s custom annotations, the programmer decorates both sealBase and AEAD.encrypt with [@@Specialize], and sealBase_aux with [@@Inline]. Doing so, they indicate to the tactic that sealBase and AEAD.encrypt should both remain in the call-graph, while sealBase_aux is to be inlined in its callers' bodies.

Upon executing, the tactic traverses the call-graph, starting from sealBase, and proceeds as follows. First, inlined functions are eliminated, leaving a call-graph only made up of specialized nodes. Then, sealBase is rewritten to take as extra parameters function pointers for every specialized function that it calls into. The signature of sealBase becomes as follows, where AEAD.encrypt_t a stands for the type of an AEAD encryption function for algorithm a, and ... stands for whichever arguments sealBase took before the rewriting.

```
let snd3 (_, x, _) = x
val sealBase: #i:hpke_index → encrypt:AEAD.encrypt_t (snd3 i) → ...
```

Instead of being applied to just a specialized index (like in Poly1305), sealBase needs to also be applied to specialized *implementations* of the *algorithms* it calls into.

```
inline_for_extraction let aead_alg = Chacha20Poly1305
let encrypt_cp32: encrypt_t aead_alg = ChachaPoly.encrypt #M32
let sealBase_cp32 = HPKE.sealBase (..., aead_alg, ...) encrypt_cp32
```

The encrypt_cp32 function above is a specialized implementation of Chacha-Poly admissible for any index (..., Chacha20Poly1305, ...). The application of the tactic-rewritten sealBase to a concrete index and encrypt_cp32 generates a valid HPKE implementation for Chacha-Poly that calls into a scalar implementation of Chacha-Poly.

The shape of the call-graph is preserved, as sealBase_cp32 calls into encrypt_cp32; furthermore, this technique allows swapping out encrypt_cp32 for any other variant, giving e.g. sealBase_cp256.

Using this technique, the proof of HPKE is exclusively concerned with *algorithmic agility*, leaving implementation choices entirely up to the module that performs concrete instantiations. This enforces strong modularity, as HPKE need not be aware of the current or future implementations for a given algorithm.

The methodology applies, naturally, to all possible combinations of choices for DH, AEAD and hash, meaning we can obtain up to 54 specialized implementations of HPKE for free. We chose 15 relevant ones that are currently packaged within HACL×N.

```
let g (alg:blake2_alg) (st:state alg) (a b c d:idx) (x y:word alg): state alg =
    let st = st.[a] ← (st.[a] +. st.[b] +. x) in
    let st = st.[d] ← (st.[d] ^. st.[a]) >>>. (rotc alg 0) in
    let st = st.[c] ← (st.[c] +. st.[d]) in
    let st = st.[b] ← (st.[b] ^. st.[c]) >>>. (rotc alg 1) in
    let st = st.[a] ← (st.[a] +. st.[b] +. y) in
    let st = st.[d] ← (st.[d] ^. st.[a]) >>>. (rotc alg 2) in
    let st = st.[c] ← (st.[c] +. st.[d]) in
    let st = st.[b] ← (st.[b] ^. st.[c]) >>>. (rotc alg 3) in
    st
let mixing_core (alg:blake2_alg) (st:state alg) (m:state alg): state alg =
    let st = g alg st 0 4 8 12 m.[0] m.[1] in
    let st = g alg st 1 5 9 13 m.[2] m.[2] in
    let st = g alg st 2 6 10 14 m.[4] m.[5] in
    let st = g alg st 3 7 11 15 m.[6] m.[7] in
    let st = g alg st 0 5 10 15 m.[8] m.[9] in
    let st = g alg st 1 6 11 12 m.[10] m.[11] in
    let st = g alg st 2 7 8 13 m.[12] m.[13] in
    let st = g alg st 3 4 9 14 m.[14] m.[15] in
    st
```

Figure 2: F⋆ spec for the core computation in Blake2.

4 SIMD CRYPTO PROGRAMMING PATTERNS

We now explore a series of well-known SIMD programming patterns used in cryptographic libraries. We show how we can factor out these patterns into verified libraries and apply them to build and verify generic vectorized implementations for several algorithms.

4.1 Exploiting Internal Parallelism (Blake2)

We first consider algorithms that are explicitly designed to allow their core operations to be parallelized. We illustrate this pattern for the Blake2 hash algorithm, but similar strategies apply to other crypto algorithms like Chacha20 and Salsa20.

Formally Specifying Blake2. The Blake2 cryptographic hash algorithm [10] is standardized in IETF RFC 7693 [33]. We formalized this RFC in F⋆ and the main types in the resulting spec are as follows:

```
type blake2_alg = | Blake2s | Blake2b
let word_t (alg:blake2_alg) = match alg with
  | Blake2s → U32
  | Blake2b → U64
let word (alg:blake2_alg) = int_t (word_t alg) SEC
type state (alg:blake2_alg) = lseq (word alg) 16
```

Blake2 has two variants, Blake2s and Blake2b; the first uses 32-bit words, whereas the latter uses 64-bit words. We specify the word type as an algorithm-dependent machine integer that is labeled as secret (SEC), which enforces that all operations on these words must be secret independent ("constant-time"). The Blake2 state, also called a working vector, is a 4×4 matrix of words, represented in the RFC as a sequence (lseq) of 16 words, laid out row-by-row.

To hash an input message, Blake2 first splits it into state-sized blocks (64 bytes for Blake2s, 128 bytes for Blake2b), and processes each block in sequence by calling a compression function. The core computation of the compression function is a loop that repeatedly loads a message block, permutes it according to a table, and then calls the mixing_core function depicted in Figure 2.

```
let g_vec (alg:blake2_alg) (st:vec_state alg) (x y: vec_row alg) =
  let (a,b,c,d) = (0,1,2,3) in
  let st = st.[a] ← (st.[a] +| st.[b] +| x) in
  let st = st.[d] ← (st.[d] ^| st.[a]) >>>| (rotc alg 0) in
  let st = st.[c] ← (st.[c] +| st.[d]) in
  let st = st.[b] ← (st.[b] ^| st.[c]) >>>| (rotc alg 1) in
  let st = st.[a] ← (st.[a] +| st.[b] +| y) in
  let st = st.[d] ← (st.[d] ^| st.[a]) >>>| (rotc alg 2) in
  let st = st.[c] ← (st.[c] +| st.[d]) in
  let st = st.[b] ← (st.[b] ^| st.[c]) >>>| (rotc alg 3) in
  st
let diagonalize (alg:blake2_alg) (st:vec_state alg) : vec_state alg =
  let st = st.[1] ← vec_rotate_right_lanes st.[1] 1ul in
  let st = st.[2] ← vec_rotate_right_lanes st.[2] 2ul in
  let st = st.[3] ← vec_rotate_right_lanes st.[3] 3ul in
  st
let mixing_core_vec (alg:blake2_alg) (st:vec_state alg)
                  (m:vec_state alg) : vec_state alg =
  let st = g_vec alg st m.[0] m.[1] in
  let st = diagonalize alg st in
  let st = g_vec alg st m.[2] m.[3] in
  let st = undiagonalize alg st in
  st
```

Figure 3: 4-Way Vectorized Specification for Blake2

The mixing_core function in turn calls a shuffling function g 8 times. Each call takes 2 words from the message (x,y), and reads, shuffles, and writes four words in the Blake2 state (at indexes a,b,c,d), using a combination of modular addition (+.), xor (^.), and right-rotate (>>>.). We use overloaded operators that work for both uint32 and uint64, and this allows us to write a single generic, yet strongly-typed, formal specification for both Blake2s and Blake2b.

The resulting F* specification is executable and can be seen as a reference implementation of the RFC. We tested it against test vectors from the RFC and more comprehensive tests we added ourselves. Interestingly, our tests revealed a bug in a corner case of our specification when processing the last block. This bug was not exercised by the RFC test vectors, and this serves to reemphasize the need for comprehensive testing and formal verification.

Rearranging Code for 4-way Vectorization. Looking at the first four calls to g in the mixing_core function (Figure 2), we can see that each call reads and modifies a different column of the state matrix $((0, 4, 8, 12), (1, 3, 5, 7), \dots)$. Hence, these calls can be executed in parallel [10]. The next four calls process different diagonals of the state and can also be executed in parallel. To exploit this 4-way parallelism inherent in Blake2, we rearrange the state to use vectors:

```
type vec_row (alg:blake2_alg) = vec_t (word_t alg) 4
type vec_state (alg:blake2_alg) = lseq (vec_row alg) 4
val to_vec (alg:blake2_alg) (st:state alg) : vec_state alg
val from_vec (alg:blake2_alg) (st:vec_state alg) : state alg
```

The state is now explicitly a matrix with four rows, and each row is a vector with four words. Based on this vectorized state, we can define a *vectorized spec* for Blake2 in F*. We will relate the two specs using functions (to_vec, from_vec) that inter-convert between the the original scalar state and its vectorized form.

The core Blake2 computations are rewritten as shown in Figure 3. The function g_vec applies the function g to each column in parallel. Using our vector library, the code for this function is remarkably similar to that of g; we simply replace the integer operations (+.,^.,>>>.) with their vector counterparts (+|,^|,>>>|) and we set the indexes a, b, c, d to column numbers 0, 1, 2, 3.

The benefit of vectorization becomes clear in the mixing_core_vec function; it now calls g_vec only twice, since each call processes four columns at a time. If each vector operation has the same cost as a scalar operation, this transformation can provide up to a 4x performance improvement. However, one must account for the cost of loading, storing, and transforming vectors. For example, before the second call to g_vec we need to *diagonalize* the state, by rotating three of the row vectors, and undiagonalize it after.

The vectorized F* spec acts as an intermediate step between the original F* specification and the vectorized Low* implementation. We prove that the two specs are equivalent via a series of lemmas. For example, we prove that mixing_core_vec computes the same function as mixing_core, but on the vectorized state:

```
∀(alg:blake2_alg) (st:state alg) (m:state alg).
  mixing_core alg st m ==
  from_vec (mixing_core_vec alg (to_vec alg st) (to_vec alg m))
```

Implementing and Verifying Vectorized Blake2. Our Low* implementation of Blake2 closely follows the vectorized specification, but generalizes it further. On machines that support sufficiently wide vector instructions (128-bit for Blake2s, 256-bit for Blake2b), the implementation uses 4-way vectorization. On all other platforms, it defaults to scalar 32-bit code. By carefully structuring our code, we are able to define a single generic implementation for all four variants: scalar and vector, Blake2b and Blake2s.

The only other difference between the vectorized spec and our Low* code is that the code modifies the state in-place, instead of copying the state at each modification. We prove that this code is memory-safe (it does not read or write arrays outside their bounds) and we prove that it is functionally correct with respect to the vectorized F* specification, and by composing this with spec equivalence, we prove that it conforms to the original Blake2 spec.

Compiling to C with Vector Intrinsics. We compile the Low* code using KreMLin to obtain 4 C files: Blake2s_32.c, Blake2b_32.c, Blake2s_128.c, and Blake2b_256.c, each offering the same interface. The first two contain portable code that runs on any 32-bit platform.

The C code in Blake2s_128.c is essentially a sequence of calls to 128-bit vector operations. This code can be linked with our library of vector intrinsics, and executed on any machine that supports Arm Neon or Intel AVX/AVX2/AVX512. For example, on Intel AVX, the C code for the first two shuffling operations of g_vec looks like:

```
st[0U] = _mm_add_epi32(st[0U], st[1U]);
st[0U] = _mm_add_epi32(st[0U], x);
st[3U] = _mm_xor_si128(st[0U], st[3U]);
st[3U] = _mm_xor_si128(_mm_slli_epi32(st[3],32−rotc0),
                       _mm_srli_epi32(st[3],rotc0))
```

Similarly, Blake2b_256 relies on AVX2 intrinsics and can be executed on any Intel AVX2/AVX512 machine. Performance numbers for all these implementations are given in §6. On Intel processors, vectorization speeds up Blake2 by about 30%.

```
let sha2 (a:sha2_alg) (in_len:size_nat) (input:lseq uint8 in_len)
            : lbytes (hash_len a) =
  let st0 = init a in
  let blocks = in_len / blocksize a in
  let st = repeati blocks (λ i st →
                  let b = sub input (i * blocksize a) (blocksize a) in
                  compress_block a b st)
            st0 in
  let last_len = in_len % blocksize a in
  let last = sub input (in_len − last_len) last_len in
  let st = pad_compress_last a in_len last_len last st in
  emit a st
```

Figure 4: F⋆ Specification for the main SHA-2 hash function

Further Platform-Specific Optimizations. We have focused on writing *generic* code to avoid duplication of coding and verification effort. Hence, we wrote a single implementation for 4 different variants of Blake2, obtaining good performance on all platforms. However, one can sometimes get even better performance by writing platform-specific code for some operations.

Blake2 requires each input message block to be permuted according to a known permutation schedule. In our code, we implement these permutations naively, using vector loads. AVX512 offers more powerful gather instructions that allow such permutations to be performed when loading vectors. We implemented and used these intrinsics but they provide no performance benefit on the machine we tested; it is likely that they will get faster in future processors. An alternative, used by several other Blake2 implementations, is to write custom AVX2 and NEON permutation code for each round. This code is tedious and non-generic (about 300 lines of C), but can result in a significant speedup. We did not implement this optimization in our code, leaving it for future work, if needed.

4.2 Multiple Input Parallelism (SHA-2)

The next pattern generally applies to any cryptographic algorithm when it is applied to a number of independent inputs (of the same size). In particular, this pattern is used in all our primitives that process multiple block-sized inputs in parallel. Here, we illustrate this pattern in our implementation of multi-buffer SHA-2.

Specifying the SHA-2 Family. The SHA-2 family of hash functions [35] is perhaps the most widely used cryptographic construction today. It is used as a core component within method authentication codes (HMAC), key derivation (PBKDF2, HKDF), signature schemes (Ed25519, ECDSA), and Merkle trees.

SHA-2 has four variants: SHA2−224, SHA2−256, SHA2−384, and SHA2−512. The first two use 32-bit words, whereas the last two use 64-bit words. Like in Blake2, we define a generic F⋆ specification for all four variants using our integer library. The SHA2 state consists of 8 words and each block consists of 16 words (i.e. 64 or 128 bytes).

Our F⋆ spec for the main sha2 hash function is depicted in Figure 4. It calls init to initialize the state (with some known constants); it then goes into a loop (repeati) that calls compress to mix each block of the input into the state; finally, it processes the last (partial) block by calling pad_compress_last and emits the output hash.

Multi-Buffer SHA-2. The sha2 function is not obviously parallelizable, since the output of each block is fed into the input of the next. But if we were willing to hash 4, 8, or 16 independent equal-sized inputs in parallel, performance could significantly improve. This strategy is called multi-buffer SHA-2 [26] and has been applied to other serial primitives like AES-CBC.

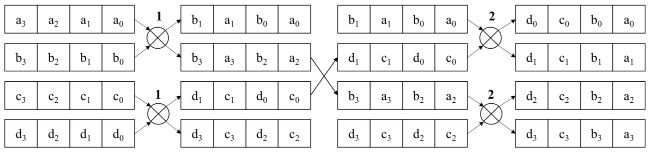

Figure 5: Transposing a 4×4 vectorized state. Each pair of vectors is interleaved element by element, then each alternate pair is interleaved 2 at a time. Transposing a $n \times n$ vectorized matrix needs $n \log(n)$ interleavings.

We write a generic vectorized specification for multi-buffer SHA-2, defining the vectorized state as an array of w-word vectors:

```
type vec_state (w:width) (alg:sha2_alg) = lseq (vec_t (word_t alg) w) 8
type multi_block (w:width) = lseq (vec_t (word_t alg) w) 16
```

Seen as a $w \times 8$ matrix, each column of this state corresponds to one input message, and hence the state represents the intermediate SHA-2 state for w inputs. We process all w inputs block-by-block by calling a vectorized version of the compress function, which takes the vectorized state and a multi_block as input. Each multi_block corresponds to the ith blocks of each of the w inputs; hence it is an array of 16 vectors and can be seen as a $w \times 16$ matrix.

Writing and verifying the vectorized compress function follows a standard pattern. Like in the Blake2 g_vec function, we replace each integer operation with the corresponding vector operation. We then prove that this transformation results in a *mapped* version of compress: it independently compresses each column in parallel.

The main remaining task for multi-buffer SHA-2 is functions for loading the message blocks and then emitting the result. Both of these operations require matrix transpositions.

A Library for Transposing Vectors. When we load an input block using vector instructions, we naturally get these blocks loaded in row-wise form. When implementing multi-buffer SHA2_256 with 128-bit vectors, for example, we process 4 inputs in parallel. We can efficiently load the 64-byte block from each input into 4 128-bit vectors, hence obtaining 16 vectors where vectors 0..3 contain data from input 0, 4..7 contain data from input 1 etc. To put this into the column-wise multi_block format needed by vectorized compress, we need to transpose vectors (0, 4, 8, 12) to obtain the first 4 vectors, then transpose (1, 5, 9, 13) to obtain the next 4 vectors, and so on.

These kinds of transpositions are routinely needed in vectorized cryptographic code (see Chacha20 below) and so we implemented and verified a generic library of vectorized transpositions called Lib.IntVector.Transpose. For each transposition, we prove that the result, seen as a matrix, is the transposition of the input.

A typical function provided by this library implements the 4×4 transposition depicted in Figure 5. It takes an array of 4 vectors each with 4 words as input. It uses a vector interleaving operation to interleave each pair of vectors element-by-element, leaving the low-half of the interleaved result in the first vector, putting the high-half in the second vector. (Both Arm and Intel platforms offer

```
val sha256_4 (r0 r1 r2 r3: lbuffer uint8 32ul)
                 (len:size_t) (b0 b1 b2 b3: lbuffer uint8 len) : Stack unit
     (requires (λ h0 → all_live h0 [ b0; b1; b2; b3; r0; r1; r2; r3 ] ∧
          pairwise_disjoint [ r0; r1; r2; r3 ]))
     (ensures (λ h0 _h1 →
          modifies (loc r0 |+| loc r1 |+| loc r2 |+| loc r3) h0 h1 ∧
          as_seq h1 r0 == sha2 SHA2_256 (v len) (as_seq h0 b0) ∧
          as_seq h1 r1 == sha2 SHA2_256 (v len) (as_seq h0 b1) ∧
          as_seq h1 r2 == sha2 SHA2_256 (v len) (as_seq h0 b2) ∧
          as_seq h1 r3 == sha2 SHA2_256 (v len) (as_seq h0 b3)))
```

Figure 6: Low* API for 4-way multi-buffer SHA2-256

these kinds of interleaving instructions.) We then interleave each pair of alternate vectors 2-by-2 to obtain the final result.

Other functions in this library extend this pattern to 8×8 and 16×16 transpositions, and also for non-square matrices. The main complexity in writing and verifying these functions is in choosing the right sequence of vector operations (some interleaving instructions can be much more expensive than others).

Implementing and Compiling Multi-Buffer SHA-2. We build a generic implementation of SHA-2 in Low* that can be instantiated for all 4 SHA-2 algorithms and can be used with 4 or 8 inputs at a time. Hence, SHA2_256 can be run on 4 inputs at a time on ARM Neon and 8 inputs at a time on Intel AVX2, while SHA2_512 can be run on 4 inputs at a time on AVX2, and 8 at a time on AVX512.

The main complexity in writing and verifying this Low* code is that each function needs to input and manipulate a large number of buffers. For example, the Low* type for 4-buffer SHA2_256 is depicted in Figure 6. It takes four equal-length buffers ($b0, b1, b2, b3$) as input and four hash-length buffers ($r0, r1, r2, r3$) as output. We require all 8 buffers to be live in the input heap, and we require the four output buffers to be disjoint. F* can then prove that the code for this function is memory safe, that it only modifies the four output buffers, and that the final value in each output buffer is the expected hash value of the corresponding input. To make these types easier to write and verify, we use a library of multi-buffer predicates like all_live, pairwise_disjoint that are meta-evaluated into conjunctions of base predicates.

The performance results for all variants of multi-buffer SHA-2 are given in §6. On Intel platforms, 4-buffer SHA-2 is about 2.5x faster than scalar SHA-2, and 8-buffer SHA-2 is up to 7x faster.

4.3 Counter Mode Encryption (Chacha20)

We next consider a SIMD pattern that applies to all counter-mode encryption (CTR) algorithms, such as Chacha20, AES, Salsa20, etc. More generally, we present a loop combinator called map_blocks, which maps a block-to-block function on some input data, and show how to parallelize any program that uses this combinator.

Specifying Generic CTR in F*. CTR is one of several block cipher modes of operation standardized by [22]. It is notably used in the two most popular modern authenticated encryption schemes: AES-GCM and Chacha20-Poly1305. We formally specify CTR as a generic construction over any block cipher that implements the interface:

```
type block = lbytes blocksize
val init: k:key → n:nonce → ctr0:nat → state
val key_block: st:state → i:nat → block
```

```
let encrypt_block (st0:state) (i:nat) (b:block) : block =
  map2 (^.) (key_block st0 i) b
let encrypt_last (st0:state) (i:nat) (len:nat{len < blocksize})
                    (r:lbytes len) : lbytes len =
  let b = create blocksize (u8 0) in
  let b = update_sub b 0 len r in
  sub (encrypt_block st0 i b) 0 len
let ctr_encrypt (k:key) (n:nonce) (ctr0:nat) (msg:bytes)
                  : cipher:bytes{length cipher == length msg} =
  let st0 = init k n ctr0 in
  map_blocks blocksize msg (encrypt_block st0) (encrypt_last st0)
```

Figure 7: Generic F* Specification for CTR.

The block cipher must define a constant blocksize, and types for the key, nonce, and cipher state. It must define a function init to initialize the state, given a key, nonce, and initial counter ctr0. Finally, it must provide a function key_block that generates a block of key bytes given a block number i.

Given such a block cipher, we specify the CTR encryption algorithm as depicted in Figure 7. The function encrypt_block encrypts the ith message block by XORing it with the ith key block. The function encrypt_last pads the last (partial) block with zeroes and then encrypts it using encrypt_block.

Finally, the main encryption function ctr_encrypt (which is the same as the decryption function) initializes the state and calls the loop combinator map_blocks, which breaks the input msg into blocks, sequentially calls encrypt_block for each block, and calls encrypt_last for the last (partial) block.

Multi-Input Parallelism for the Block Cipher. The map_blocks combinator exposes the inherently parallelism in CTR: it processes each block independently, and so can process any number of blocks in parallel. To exploit this parallelism, we first have to write a vectorized version of the block cipher:

```
type blocksize_v (w:width) = w * blocksize
type multi_block (w:width) = lbytes (blocksize_v w)
type vec_state (w:width)
val init_v: w:width → k:key → n:nonce → ctr0:nat → vec_state w
val key_block_v: w:width → vec_state w → i:nat → multi_block w
```

Following the multi-input SIMD pattern, the vectorized block cipher processes w blocks at a time. It has an internal vectorized state vec_state that is initialized by the function init_v. The function key_block_v generates w consecutive key blocks. The main proof obligation is to show that these blocks correspond to the key blocks numbered i∗w,i∗w+1,...,(i+1)∗w−1 in the original spec.

For Chacha20, writing and verifying the vectorized block cipher code follows the same pattern as SHA-2. The Chacha20 state is an array of 16 32-bit words, and so the vectorized state is an array of 16 vectors with w words each (each column corresponds to one input block). We replace each integer operation in the block cipher code with its vector equivalent, and we need to transpose the state before generating the output key blocks. By reusing library lemmas about vector operations and transpositions, we prove the correctness of the key_block_v function for Chacha20 with relatively little effort.

Parallelizing CTR. Vectorizing the block cipher effectively yields a new block cipher with a larger blocksize. Hence, we can run the standard CTR algorithm over this vectorized block cipher, by processing w sequential blocks at a time. This results in a vectorized spec for the Chacha20.

Our loop combinator library includes general lemmas about map_blocks, which allow us to prove the generic correctness of vectorized CTR given a proof of correctness for the vectorized block cipher. This gives us a proof for vectorized Chacha20, but the pattern can also be easily applied to other block ciphers like AES.

Implementing and Compiling Vectorized Chacha20. We implement Vectorized Chacha20 in Low* in two steps. We first write a module for multi-block Chacha20 that can process w blocks at the same time, for w=1,4,8,16. We then write a generic CTR module that uses the map_blocks to process w blocks at the same time.

The implementation introduces a new optimization in the vectorized code for encrypt_block, which loads w blocks of data from an input message, XORs it with w key blocks, and stores these blocks into the output ciphertext. Using vector instructions, we can implement this load-XOR-store loop generically and more efficiently than the byte-by-byte XOR in encrypt_block. In some cases, it is also beneficial to unroll this loop a few (say 4) times to take maximum advantage of instruction pipelining.

Because of our generic code structure, adding new platforms requires modest effort. For example, to add AVX512, the main additional effort was to add the relevant vector intrinsics and to define and verify a 16×16 transpose function, which is now in the library and can be used in other algorithms.

We note that this vectorization pattern is not the only one that applies to Chacha20. The inner block cipher in Chacha20 is inherently parallelizable (similarly to Blake2) and this parallelization has been exploited in prior work [16] and even verified [36]. However, in our experiments, we found that vectorizing CTR was generally more effective on our target platforms.

4.4 Polynomial Evaluation (Poly1305)

We now describe a SIMD pattern used in cryptographic algorithms like Poly1305 and GCM, which are written in terms of polynomial evaluation over a (large) arithmetic field. We show that these algorithms can be written using a loop combinator called reduce_blocks and detail how this combinator can be parallelized if the body of the loop satisfies some algebraic conditions.

Specifying Poly1305. The Poly1305 one-time MAC function [15] is standardized in IETF RFC7539 [5]. It takes a 32-byte key as input and splits into two 128-bit integers s and r. It then splits the input message into 16-byte blocks, hence transforming it to a sequence of 128-bit integers $(m_1, m_2 \dots m_n)$; if the last block is partial, it is filled out with zeroes to obtain a full block.

The main computation in the Poly1305 MAC evaluates the following polynomial in the prime field $\mathbb{Z}_p$, where $p = 2^{130} - 5$:

$$a = (m_1 \times r^n + m_2 \times r^{n-1} + \dots + m_n \times r) \mod p$$

In practice, this polynomial is evaluated block by block, by applying Horner's method to rearrange the polynomial as follows:

$$a = ((\dots((0 + m_1) \times r + m_2) \times r + \dots + m_n) \times r) \mod p$$

We maintain an accumulator a, initially set to 0, and to process each new block m_i, we first add it to the accumulator, and then multiply the result by r (all operations in $\mathbb{Z}_p$). Once the final block is processed, we compute $s + a \mod 2^{128}$ to obtain the Poly1305 MAC.

```
let process_block (r:felem) (b:block) (acc:felem) : felem =
  fmul (fadd acc (encode b)) r
let process_last (r:felem) (len:nat{len < blocksize}) (b:lbytes len)
                   (acc:felem) : felem =
  if len = 0 then acc else process_block r (pad_last len b) acc
let poly_eval (msg:bytes) (acc0 r:felem) : felem =
  reduce_blocks blocksize msg
    (process_block r)
    (process_last r)
  acc0
```

Figure 8: F* spec for Poly1305 polynomial evaluation.

```
let process_blocks_v (w:width) (r_w:felem_v w)
                        (b:lbytes (w∗blocksize)) (acc:felem_v w) : felem_v w =
  fadd_v w (fmul_v w acc r_w) (encode_v w b)
let process_last_v (w:width) (r:felem) (len:nat{len < w ∗ blocksize})
                      (b:lbytes len) (acc:felem_v w) : felem =
  poly_eval b (normalize_v w r acc) r
let poly_eval_v (w:width) (msg:bytes) (acc0 r:felem) : felem =
  let r_w = pow_v w r in
  let acc0_v = to_acc_v w acc0 in
  reduce_blocks (w∗blocksize) msg (process_blocks_v w r_w)
    (process_last_v w r) acc0_v
```

Figure 9: Vectorized spec for Poly1305 poly_eval.

Figure 8 depicts our F* specification for the polynomial evaluation described above. It uses a loop combinator called repeat_blocks that splits the input into block-sized chunks. For each block, it calls process_block, which in turn calls the two field arithmetic operations in $\mathbb{Z}_p$: fadd to add an encoded block to the accumulator acc, and fmul to multiply the result with r. The function process_last pads and processes the last block. Our full F* specification for Poly1305 is not much larger than this; it only adds some concrete details from the RFC about encoding blocks and keys.

Parallelizing Polynomial Evaluation. Several prior works (e.g. [16]) have observed that the algebraic shape of polynomial evaluation lends itself to SIMD vectorization. For example, we can process blocks two-by-two by rewriting the polynomial as follows:

$$\begin{aligned} a_1 &= (\dots((m_1 \times r^2 + m_3) \times r^2 + m_5) \times r^2 + \dots + m_{n-1}) \mod p \\ a_2 &= (\dots((m_2 \times r^2 + m_4) \times r^2 + m_6) \times r^2 + \dots + m_n) \mod p \\ a &= (a_1 \times r^2 + a_2 \times r) \mod p \end{aligned}$$

Let's assume that n is even. We split the polynomial evaluation into two computations, one processes odd-numbered blocks, and the other processes even numbered blocks, but both computations are otherwise identical. We now have two accumulators (a_1, a_2) initialized to (m_1, m_2). We process two blocks (m_{2i-1}, m_{2i}) at a time by multiplying both (a_1, a_2) by r^2 and adding the result point-wise to (m_{2i-1}, m_{2i}). After processing n blocks, a final *normalization step* multiplies a_1 by r^2 and a_2 by r and adds them.

This refactored computation effectively computes two polynomials in parallel and it is easy to informally see why it is correct. We formalize and generalize this pattern as a vectorized specification of Poly1305 in F* that can process any number (e.g. 1/2/4/8) of blocks in parallel. Figure 9 depicts the vectorized spec.

The accumulator now has the type felem_v w, which represents a vector of w field elements. The function process_blocks_v evaluates w blocks in parallel by calling vectorized versions (fmul_v, fadd_v) of the field arithmetic functions. If less than w blocks of input are left, we call the process_last_v function that *normalizes* the vectorized accumulator to get a regular field element (felem), then calls the original (scalar) poly_eval function on the remaining input.

To set up the vectorized polynomial evaluation, poly_eval_v first precomputes r^w and stores it in a vector r_W whose elements all hold r^w. It then loads the initial accumulator acc0 into the 0th element of the vectorized accumulator acc0_v (all other elements are set to zero) and calls reduce_blocks to process the input.

We generically prove, for all choices of w, that this vectorized spec is functionally equivalent to the original Poly1305 spec:

∀w msg acc0 r. poly_eval_v w msg acc0 r == poly_eval msg acc0 r

The proof relies on general lemmas about field arithmetic and the reduce_blocks combinator. We apply this lemma here to Poly1305 but it also applies to other polynomial MACs like GCM.

Implementing Multi-Input Field Arithmetic. The main effort of implementing and verifying (scalar or vectorized) Poly1305 is in the field arithmetic modulo p. Since Poly1305 uses a 130-bit field, a typical way of implementing a field element in Low⋆ is as an array of 5 26-bit *limbs*, where each limb can grow to at most 64-bits. We then need to implement custom modular Bignum arithmetic (fadd, fmul) for this representation and prove it correct. In the original HACL⋆ release, Poly1305 was one of the largest developments with 4716 lines, most of it dedicated to field arithmetic [36].

To implement vectorized Poly1305, we need to implement and verify a multi-input field arithmetic library that can add and multiply (fadd_v, fmul_v) multiple field elements in parallel. We take the original scalar Poly1305 code of HACL⋆ and generalize it using the standard multi-input pattern. Each limb is represented by a 64-bit word, and a vectorized field element is an array of vectors, each of which has w words. All functions are parameterized by the vector width w and integer operations are replaced with vector ones. The correctness proofs are adapted for vectorized inputs and outputs.

While the multi-input algorithmic transformation is itself straightforward, applying it to thousands of lines of scalar Poly1305 was a challenge and constitutes our largest case study for the SIMD coding and verification patterns in this paper. This is, however, a one-time cost. Once we vectorized all the field arithmetic in Poly1305, adding a new platform (such as AVX512) required only a modest amount of work. Furthermore, the verified vectorized bignum library we built for Poly1305 has many reusable components that can be used in other primitives like Curve25519 in future work.

5 CRYPTOGRAPHY FOR ALL YOUR NEEDS

While cryptographic algorithms are often designed, standardized, and implemented as independent components, they are typically deployed and used as part of composite constructions. For example, the Chacha20 cipher is only safe to use in conjunction with a one-time MAC like Poly1305. The SHA-256 hash algorithm is used within HMAC, HKDF, and a number of signature schemes. So, even if the code for an individual algorithm is verified for memory safety, correctness, or side-channel resistance, these guarantees become quickly meaningless if the code is composed with a buggy algorithm, or if the API provided by the algorithm is easy for an application to misuse. Consequently, it is important for verified crypto code to be deployed as part of a comprehensive verified *library* of cryptographic constructions with safe usable APIs.

	Portable	Arm A64	Intel x64			
Algorithm	**C code**	**Neon**	**AVX**	**AVX2**	**AVX512**	**Vale**
AEAD						
Chacha20-Poly1305	✓ [36] (+)	✓ (*)	✓ (*)	✓ (*)	✓ (*)	
AES-GCM						✓ [17]
Hashes						
SHA2-224,256	✓ [36] (+)	✓ (*)	✓ (*)	✓ (*)	✓ (*)	✓ [17]
SHA2-384,512	✓ [36] (+)	✓ (*)	✓ (*)	✓ (*)	✓ (*)	
Blake2s, Blake2b	✓ [28] (+)	✓ (*)	✓ (*)	✓ (*)		
SHA3-224,256,384,512	✓ [28]					
HMAC and HKDF						
HMAC (SHA2,Blake2)	✓ [36]	✓ (*)	✓ (*)	✓ (*)	✓ (*)	
HKDF (SHA2,Blake2)	✓ [36]	✓ (*)	✓ (*)	✓ (*)	✓ (*)	
ECC						
Curve25519	✓ [36]					✓ [28]
Ed25519	✓ [36]					
P-256	✓ [28]					
High-level APIs						
Box	✓ [36]					
SecretBox	✓ [36]					
HPKE	✓ (*)	✓ (*)	✓ (*)	✓ (*)	✓ (*)	✓ (*)

Table 1: Extending HACL⋆ with vectorized crypto. Implementations marked with **(*)** were newly developed for this paper; those marked with a **(+)** replaced prior C implementations from [36]. These C implementations are composed with platform-specific Intel assembly code from Vale [17] (verified agains the same specs) to build the EverCrypt provider [28]. (Vale assembly relies on AES-NI for AES-GCM, SHAEXT for SHA2, and ADX+BMI2 instructions for Curve25519.)

5.1 Integration and Deployment with HACL*

We contributed all the verified code developed in this paper to the HACL⋆ project, and helped to integrate it with the existing constructions and APIs in the HACL⋆ library. Tab. 1 summarizes our contributions. For Chacha20, Poly1305, Blake2, and SHA2, our scalar code replaces the previous portable C code [36] and our vectorized implementations are offered as platform-dependent alternatives. For each platform, we also build verified implementations of the Chacha20Poly1305 AEAD construction, and we integrated our hash implementations into HMAC, HKDF, Ed25519, and ECDSA.

Crucially, for each algorithm, we ensure that each of our implementations meets the same high-level specifications as the original HACL⋆ code, and retains the same API.

For existing clients of the HACL⋆ library, such as the Linux Kernel, Mozilla Firefox, or the Tezos Blockchain, this means that the newer C code is a drop-in replacement, with no new specification or API to be reviewed. Indeed, some of the new vectorized code from HACL×N has already been deployed in production: Firefox Nightly now uses our vectorized Chacha20-Poly1305 code and Tezos uses our vectorized Blake2, yielding measurable performance benefits. For verified clients, such as EverQuic [20], preserving specs means that verification is not impacted.

HACL⋆ includes a verified provider called EverCrypt [28] that offers an agile, multiplexing API on top of both HACL⋆ and Vale code. It uses CPU autodetection to dynamically dispatch API calls to the most efficient implementation for the platform the code is running on. We worked with the HACL⋆ developers to make our HACL×N code available through the agile EverCrypt API. We strongly encourage clients use this verified, future-proof API: as more efficient implementations get added to HACL★, users of EverCrypt automatically get upgraded to faster variants.

5.2 HPKE: a verified application of HACL×N

We now illustrate how HACL×N serves as a platform for authoring verified cryptographic constructions and applications. We focus on Hybrid Public Key Encryption (HPKE), a new cryptographic construction that is undergoing standardization at the IETF [13], and is already being used in several upcoming protocols [12, 32].

HPKE is a public-key encryption scheme with optional sender authentication: any sender who knows the HPKE public key of a recipient can encrypt a sequence of messages under this key and send it over the network; the recipient can use the corresponding private key to decrypt the messages. Optionally, the sender may also use a pre-shared symmetric key or a private Diffie-Hellman key to authenticate the message, but we do not support this feature.

At its core, HPKE relies on three components: (1) A key encapsulation mechanism (KEM) that generates a fresh secret shared between the sender and recipient and encapsulates (encrypts) it for the recipient's public key; (2) A key derivation function (KDF) that derives an encryption context containing a key and a nonce from the shared secret; (3) An authenticated encryption algorithm (AEAD) that uses the encryption key to encrypt and decrypt a sequence of messages. Hence, computationally expensive public-key cryptography is only needed to initialize the encryption context, which can then be used to efficiently encrypt any amount of data.

HPKE is an *agile* scheme that supports multiple ciphersuites. The RFC recommends four KEMs (P-256, P-521, Curve25519 and Curve448), two KDFs (HKDF-SHA256 and HKDF-SHA512) and three AEADs (AES-GCM-128, AES-GCM-256 and ChaCha20Poly1305). Any combination is valid: HPKE thus has 24 possible ciphersuites, and many more *implementation* combinations.

Individually verifying all these would be intractable. However, using the integrated HACL★ library, we can build a *generic* implementation of HPKE in 800 lines of code, in a way that is abstract in the choice of its KEM, KDF and AEAD implementation. To instantiate this code for a specific ciphersuite on a particular platform, we only need to provide implementations for KEM, KDF and AEAD on that platform to perform program specialization (§3.4), so long as these meet our agile specifications. We instantiate and compile our code to obtain 15 verified variants of HPKE that build upon our new implementations of Chacha-Poly and SHA2, as well as previously verified implementations of AES-GCM, Curve25519 and P-256 from Vale and HACL★. Each instantiation consists of about 10 lines of F* code, and extracts to about 380 lines of verified C code.

To use our HPKE implementation, users have numerous options. They can use the original (*not* tactic-rewritten) HPKE which supports any algorithm by calling into EverCrypt. Clients get the fastest implementation available, at the expense of extra run-time checks and a large amount of code for all variants. Another option is to use one of the 15 specialized HPKE variants distributed with HACL×N. Finally, a last option is to generate a *customized*, minimal library for a specific platform and subset of ciphersuites. For example, a user can compile just the code needed for the HPKE ciphersuite consisting of Curve25519, SHA-256, and Chacha20-Poly1305 for ARM, resulting in a self-contained vectorized HPKE implementation with 3000 lines of C (compared to 91K lines for the full library.)

6 EVALUATION

Benchmarking Performance. Appendix A presents performance tables for our code obtained using the SUPERCOP framework [2] and a user-space version of KBENCH9000 [21].

We benchmarked each cryptographic algorithm on a low-end ARM device (Raspberry Pi 3B+ with a Broadcom BCM2837B0 Cortex-A53 CPU supporting NEON), a mainstream Intel laptop (Dell XPS13 with a Core i7-7560U CPU supporting AVX2), and a high-end Intel workstation (Dell Precision with a Xeon Gold 5122 CPU supporting AVX512). All three run 64-bit Ubuntu Linux and the code was compiled with the latest GCC and clang compilers available for the operating system (version ≥ 9), at optimization level `-O3`. We also benchmarked our code using SUPERCOP on 4 Amazon EC2 instances; two with Intel Xeon CPUs: a general-usage `t3.large`, and a compute-optimized `c5.metal`; two with ARMv8 CPUs: a low-cost `a1.metal`, and a higher-end `m6g.metal`. All these instances ran 64-bit Amazon Linux, and the code was compiled with GCC and CLANG (version 7 on ARMv8, versions 7 and 9 on Xeon). By benchmarking across these 7 different machines with two popular C compilers, we tried to get a wide range of performance measurements from low-end clients to high-end servers.

With SUPERCOP, we measured performance in cycles per byte for processing 16KB of input data, which is the maximum record size for the Transport Layer Security protocol. With KBENCH9000, we evaluated each algorithm on inputs sizes ranging from 1KB to 32KB. We compared the performance of our code to popular libraries like OpenSSL and LibSodium, to optimized implementations contributed to SUPERCOP, to verified assembly code from Jasmin, and to reference implementations for each algorithm.

Table 2 summarizes the main results of our evaluation. Our goal is to answer two questions: (1) what is the performance benefit of using our vectorized HACL×N code over the portable C code previously used in HACL★; and (2) how does our code compare to state-of-the-art cryptographic code, both verified and unverified, both in C and in hand-written assembly. The short answer: with the exception of hash functions on ARM devices, vectorization provides a measurable speedup for all algorithms on all platforms. Furthermore, the C code extracted from our generic verified vectorized implementations is very close in performance to the fastest available hand-optimized assembly code for each algorithm.

Chacha20 and Poly1305. On our Intel laptop, which supports AVX2 but not AVX512, our vectorized AVX2 code for Chacha20 and Poly1305 is 4.8X and 4.3X faster than portable code. On the Xeon workstation, the speedup for our AVX512 code grows to 10.3X for Chacha20 and 5.9X for Poly1305. On the Raspberry PI, the speedups are more modest: 1.7X for Chacha20, and 1.4X fo Poly1305.

Among the other implementations we measured, Jasmin had the fastest Chacha20 and Poly1305 AVX2 implementations. For inputs of 16KB, our code was 3-5% slower than Jasmin, but the difference is significantly greater for smaller inputs, where Jasmin uses specialized code, but our implementation still uses generic vectorization. For medium-to-large inputs, the speed difference is because of the manual assembly-level instruction interleaving in

Algorithm	Intel Kaby Lake Laptop			Intel Xeon Workstation			ARM Raspberry Pi 3B+			Coding and Verification Effort				
	Our Code		Other	Our Code		Other	Our Code		Other	Scalar	Vec	Equiv	Low*	Output
	Scalar	AVX2	Fastest	Scalar	AVX512	Fastest	Scalar	Neon	Fastest	Spec	Spec	Proof	Impl.	C Code
Chacha20	3.73	0.77	0.75 (j)	5.74	0.56	0.56 (d)	8.69	5.19	4.49 (o)	151	182	819	510	4083
Poly1305	1.59	0.37	0.35 (j)	2.31	0.39	0.51 (j)	4.20	3.11	1.50 (o)	56 (arith)	122	370 +3594	2361	7136
Blake2b	2.56	2.26	2.02 (b)	3.97	3.13	2.84 (b)	6.99	–	6.02 (b)	430	441	324	1077	2824
Blake2s	4.32	3.34	3.06 (b)	6.63	4.52	4.11 (b)	11.42	15.30	9.80 (b)					
SHA256-mb	7.40	1.63 ×8	4.96 (o)	11.37	1.69 ×8	7.42 (o)	15.70	12.92 ×4	15.09 (o)	213	420	662	1360	4647
SHA512-mb	5.06	1.95 ×4	3.25 (o)	7.38	1.44 ×8	4.99 (o)	11.27	–	9.77 (o)					
Total (lines of specs, proofs, and code):										850	12242			18690

Table 2: Evaluating HACL×N Performance and Development Effort.
Performance (left): For each algorithm, we measure CPU cycles per byte when processing 16384 bytes of data. We list these numbers for our portable (scalar) C code, for our best-performing vectorized implementation on the machine, and for the fastest alternative implementation we tested: (j) refers to verified assembly code from Jasmin [8]; (o) is OpenSSL, (b) is code from the Blake2 team [10], (d) is code submitted by Romain Dolbeau to SUPERCOP. For multi-buffer SHA-2, the total cycle count is divided by the number of inputs processed in parallel (indicated by $\times N$).
Development Effort (right): All our specs and proofs are written in F*, our implementations are written in Low* and then compiled to C. We calculate the size of each file in the development using `cloc`, discarding comments. The Poly1305 implementation includes a large field arithmetic component, which is separately listed. We write a single implementation of Blake2 and SHA2 for all variants of these algorithms.

the Jasmin code. By mimicking this interleaving in our C code, we were able to get closer to Jasmin's performance, but we decided not to use this optimization because it obfuscates the structure of the code and because it is unclear whether such low-level optimizations will still be effective on future platforms.

This speed difference disappears entirely on the Xeon workstation, where our Chacha20 and Poly1305 implementations are uniformly the fastest among all the code we tested, matching the performance of the fastest AVX512 implementation in SUPERCOP. Interestingly, even our AVX2 code catches up to Jasmin's AVX2 on the AVX-512 machine, where the manual instruction interleaving appears to offer less benefit. OpenSSL also includes AVX512 code that we believe is at least as fast as ours but this code appears to be disabled on our Xeon workstation and on the Amazon Xeon instances because of frequency scaling issues with AVX-512 [3], and hence could not be measured by us.

On the Raspberry Pi, the fastest implementation we found was hand-optimized assembly from OpenSSL, which was 16% faster than our Chacha20, and 2.1X faster than our Poly1305. Our code gets closer to OpenSSL on newer ARMv8 chips, and is 46% slower than OpenSSL on the Amazon Graviton2. On inspecting the OpenSSL Poly1305 code, we found that the main difference is that it was making use of efficient multiply-with-accumulate instructions available in ARM NEON (but not on Intel), and we intend to extend our vector libraries to support these instructions.

Blake2s and Blake2b. Compared to our portable C code, our 128-bit vectorized code for Blake2s offers a modest speedup on Intel machines: 1.29X on the laptop, 1.47X on Xeon. Our 256-bit vectorized code for Blake2b offers even smaller speedups: 1.13X on the laptop, 1.27X on Xeon. These measurements match the speedups we have observed for other Blake2 implementations. If the effect of vectorization seems less pronounced than for Chacha20 and Poly1305, it is perhaps because the portable C code for Blake2 is already very fast, and easy to optimize for modern C compilers.

On all the ARM64 chips, however, we see a surprising *performance loss* for vectorized code compared to portable C. This is a known issue on ARM CPUs where the latency of vector shift instructions (used extensively in hash functions like Blake2 and SHA2) is quite high [1]. Consequently, for hash functions, vectorization on the cheap ARMv8 CPUs we measured does not appear to provide many benefits. However, on higher-end ARM devices, like the Apple A9, and on upcoming ARM servers, we expect that vectorized code will reap significant benefits.

The fastest implementations of Blake2 we found were written by Samuel Neves for the BLAKE2 team. Our vectorized code is about 10% slower than this implementation on both the laptop and workstation. This difference is because Neves' implementation uses AVX2 instructions to implement the Blake2 message permutation table in code, whereas our generic vectorized code uses load instructions that are available on all platforms.

Multi-Buffer SHA2. Our multi-buffer SHA2 implementation offers a large speedup over portable code on Intel platforms, but as with Blake2, are not effective on the ARM devices we tested. Our 8-way SHA-256 implementation is 4.5X faster (per input) than portable code on AVX2, and 6.7X faster on AVX512. Our 4-way SHA-512 implementation is 2.6X faster than portable code on AVX2, our 8-way SHA-512 is and 5.1X faster on AVX512.

On all platforms, the fastest other SHA-2 implementations are from OpenSSL, which relies on vector instructions to speed up message scheduling and uses native SHA instruction (SHA-EXT) when available. On our laptop and workstation, the OpenSSL code was 1.5X faster than our scalar code, but using multi-buffer vectorization, our code leapfrogged OpenSSL by significant margins and were the fastest implementations we tested. OpenSSL also includes multi-buffer SHA-2 but offers a non-standard API for them which is hard to test; we did not benchmark this code but we believe that it will be at least as fast as our multi-buffer SHA-2 implementations.

On ARM, our 4-way vectorized SHA-256 code is 22% faster than our scalar code and 17% faster than OpenSSL. This is far less than the speedups obtained on AVX2 and AVX512, and this is because of the poor shift/rotate performance on NEON. Some Intel and ARM processors support native SHA2 instructions, and using these instructions can provide much better performance than vectorization. On Amazon Graviton, for instance, OpenSSL assembly uses hardware SHA instructions and is by far the fastest implementation.

Coding and Verification Effort. It is hard to estimate the development cost of a verification-oriented project like HACL×N, since even the task of verifying a few algorithms often triggers the development of new reusable libraries and tool improvements that are generally useful. Table 2 tries to quantify the effort for each step in our workflow, in terms of lines of code and proof.

For Chacha20, our code and proofs total to 1511 lines, this is more than the 691 lines for scalar Chacha20 reported in [36], but less than the 1656 lines for 128-bit vectorized Chacha20 in that same paper. This three-way comparison provides a good estimate for the cost-benefit tradeoff of our methodology: by writing about the same amount of code and proofs as one vectorized implementation, we are able to compile both scalar code and vectorized code for several different platforms, generating 4083 lines of C.

For Blake2 and SHA2, our code and proofs generically cover multiple algorithms as well as multiple vector sizes, so the table includes only one row for each family.

For Chacha20, Blake2 and SHA2 most of the proof effort is in proving spec equivalence for vectorization, and the rest is in proving correctness and memory safety in Low*. If we calculate proof overhead in terms of the number of lines of Low* and F* code we had to write in order to verify each line of C code we generate, then this number would be 0.37 lines for Chacha20, 0.65 lines for Blake2, and 0.53 lines for SHA-2.

Our largest development is Poly1305, totalling 6447 lines, which can itself be broken down into vectorized bignum code for the field arithmetic (3594 lines) and vectorized polynomial evaluation (2853 lines). This implementation is about twice as large as the original scalar code in [36] but we obtain both scalar and vector implementation from it, totalling 7136 lines of C. Hence, the proof overhead for Poly1305 is at 0.9 lines of F* per line of C. However, more than half of our code is for vectorized bignum, and we expect this code to be reusable in other cryptographic algorithm implementations.

Overall, we estimate that the cost of adding a new vectorized algorithm to HACL×N is under 1 line of proof for each line of generated C, significantly better than the 3X overhead in [36].

Furthermore, the cost of adding a new vectorized platform to one of our vectorized implemementations is very low. For example, once we had verified our generic Chacha20 and Poly1305 implementations, it took a PhD student about one week to add AVX512 to our framework and extend our proofs to support this new platform.

Comparison with Verified Assembly. Our closest competition in terms of high-performance verified crypto is verified assembly from Vale, Jasmin, and CryptoLine. On AVX2, Jasmin can produce code that is measurably faster than our code by directly optimizing and verifying assembly code. However, Jasmin does not include verified code for AVX, NEON, or AVX512. Writing and verifying new implementations for these platforms may be able use some existing libraries, but are still likely to require significant effort.

In contrast, our code is compiled from a generic source implementation, yet remains competitive on all platforms, and ours are the fastest verified implementations on AVX512. The trade-off is that our code relies on unverified compilers like GCC and CLANG.

Going forward, we believe that both methodologies have a role to play: one may begin with verified generic implementations like ours, and then opportunistically replace some of their components with verified assembly, e.g. using Vale or Jasmin, for performance. Our methodology allows us to safely compose such verified implementations and use them in composite constructions.

7 DEPLOYMENT AND FUTURE WORK

HACL×N has been integrated into the HACL* cryptographic library and all our code is publicly available at:

https://github.com/project-everest/hacl-star

Our vectorized Chacha20 and Poly1305 implementations have been deployed in the NSS cryptographic library used by Mozilla Firefox, and in the TLS stack used in Microsoft's msQuic implementation. Our vectorized Blake2 code is being deployed in the Tezos blockchain. Other deployments are ongoing.

Each deployment induces a new workflow that exercises different aspects of our verified codebase. For example, integrating our code into NSS requires spec and code review by the NSS developers. Consequently, a good amount of our engineering effort goes into generating readable C code from KreMLin, in a way that follows NSS coding guidelines. The code is then subjected to static analysis tools that check for unused variables, dead code and other issues that sometimes require fixes in the Low* source code. Finally, once our C code passes the audit, it is integrated into the NSS continuous integration (CI) infrastructure, where it is regularly tested on a large number of platforms, against both hand-written unit tests and test frameworks like Wycheproof [4]. The code is then pushed to the main NSS branch and included in Firefox Nightly (a few thousand users) to find early deployment problems. After 2-4 weeks, it is deployed to Firefox Beta (a few million users) where more platform compatibility issues may be found due to the increased coverage. A month later, if no issues are found, the code is released in the Firefox browser (about 250 million users.)

The above workflow requires close coordination between NSS and HACL* developers over an extended period of time. A similar level of engagement is needed for successful deployments in Tezos and msQuic. This additional time and effort should be seen as the cost of transferring verified code from a research project like ours to real-world software applications.

This paper has focused on a few algorithms,‘ but we are working on extending the library with many more vecrotized implementations, following the same SIMD patterns we have discussed here. We also plan to optimize our code better for low-end ARM devices and investigate new vectorization strategies for such platforms.

REFERENCES

[1] Blake2b neon suffers poor performance on armv8/aarch64 with cortex-a57. https://github.com/weidai11/cryptopp/issues/367.

[2] ebacs: Ecrypt benchmarking of cryptographic systems – supercop. https://bench.cr.yp.to/supercop.html.

[3] On the dangers of intel's frequency scaling. https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/.

[4] Project wycheproof. https://github.com/google/wycheproof.

[5] ChaCha20 and Poly1305 for IETF Protocols. IETF RFC 7539, 2015.

[6] Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2017.

[7] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and

Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. 2017.

[8] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. *CoRR*, abs/1904.04606, 2019.

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

[10] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. Blake2: Simpler, smaller, fast as md5. In *Applied Cryptography and Network Security*, pages 119–135, 2013.

[11] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. Cryptology ePrint Archive, Report 2019/1393, 2019. https://eprint.iacr.org/2019/1393.

[12] R. Barnes, B. Beurdouche, J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert. The messaging layer security (mls) protocol. IETF Internet-Draft draft-ietf-mls-protocol-09, 2020.

[13] R. Barnes and K. Bhargavan. Hybrid public key encryption. IRTF Internet-Draft draft-irtf-cfrg-hpke-02, 2019.

[14] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of openssl HMAC. In *USENIX Security Symposium*, pages 207–221, 2015.

[15] Daniel J. Bernstein. The Poly1305-AES message-authentication code. In *Proceedings of Fast Software Encryption*, March 2005.

[16] Daniel J. Bernstein and Peter Schwabe. Neon crypto. In *Cryptographic Hardware and Embedded Systems (CHES)*, pages 320–339, 2012.

[17] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *Proceedings of the USENIX Security Symposium*, August 2017.

[18] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming*, 23(5):552–593, 2013.

[19] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[20] Antoine Delignat-Lavaud, Cédric Fournet, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Jay Bosamiya, Joseph Lallemand, Itsaka Rakotonirina, and Yi Zhou. A security model and fully verified implementation for the ietf quic record layer. Cryptology ePrint Archive, Report 2020/114, 2020. https://eprint.iacr.org/2020/114.

[21] Jason A. Donenfeld. kbench9000 - simple kernel land cycle counter. https://git.zx2c4.com/kbench9000/about/, February 2018.

[22] Morris J. Dworkin. Sp 800-38a 2001 edition. recommendation for block cipher modes of operation: Methods and techniques. Technical report, 2001.

[23] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1202–1219. IEEE, 2019.

[24] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. A verified, efficient embedding of a verifiable assembly language. *Proc. ACM Program. Lang.*, 3(POPL):63:1–63:30, 2019.

[25] Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Signed cryptographic program verification with typed cryptoline. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1591–1606, 2019.

[26] Shay Gueron and Vlad Krasnov. Simultaneous hashing of multiple messages. *J. Information Security*, 3(4):319–325, 2012.

[27] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. Meta-F*: Proof automation with SMT, tactics, and metaprograms. In *28th European Symposium on Programming (ESOP)*, pages 30–59. Springer, 2019.

[28] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, et al. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 634–653, 2019.

[29] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in F*. *Proceedings of the ACM on Programming Languages (PACMPL)*, 1(ICFP):17:1–17:29, 2017.

[30] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in F*. *PACMPL*, (ICFP), September 2017.

[31] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. Everparse: verified secure zero-copy parsers for authenticated message formats. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1465–1482, 2019.

[32] E. Rescorla, K. Oku, N. Sullivan, and C.A. Wood. Encrypted server name indication for tls 1.3. IETF Internet-Draft draft-ietf-tls-esni-06, 2020.

[33] M-J. Saarinen and J-P. Aumasson. The blake2 cryptographic hash and message authentication code (mac). IETF RFC 7693, 2015.

[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 Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, 2016.

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

[36] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS*, pages 1789–1806, 2017.

A PERFORMANCE BENCHMARKS

This appendix presents our performance measurements using two benchmarking frameworks across seven machines:

Tables 3 & 6: a Dell XPS13 laptop, with an Intel Core i7-7560U (Kaby Lake, AVX2) CPU

Tables 4 & 7: a Dell Precision workstation, with an Intel Xeon Gold 5122 (AVX512) CPU,

Tables 5 & 8: a Raspberry PI 3B+ single-board computer, with a Broadcom BCM2837B0 Cortex-A53 (64-bit, NEON) CPU

Table 9: an Amazon EC2 `t3.large` instance, with an Intel Xeon Platinum 8259CL (AVX512) CPU

Table 10: an Amazon EC2 `c5.metal` instance, with an Intel Xeon Platinum 8275CL (AVX512) CPU,

Table 11: an Amazon EC2 `a1.metal` instance, with an Amazon Graviton Cortex-A72 (64-bit, Neon) CPU

Table 12: an Amazon EC2 `m6g.metal` instance, with an Amazon Graviton2 Cortex-A76 (64-bit, Neon) CPU

All machines run 64-bit Linux: the first 3 run Ubuntu 18.04, the last 4 run Amazon Linux 2.

SUPERCOP. We downloaded supercop-20200417.tar.xz[1] and installed it on all seven machines above. We configured SUPERCOP to use the default GCC and CLANG compilers installed on each machine (typically gcc-7 and clang-7) and we also isntalle the latest versions of these compilers (typically gcc-9 and clang-9). SUPERCOP evaluated each algorithm for all compilers under a variety of optimization flags, with the best performance usually achieved by the combination: `-O3 -march=native -mtune=native`.

To the existing implementations in SUPERCOP, we added: (1) Jasmin Intel assembly code[2], including verified scalar x86 code, (unverified) AVX, and verified AVX2; (2) Blake2 reference source code package[3], including scalar, NEON, and AVX code; (3) OpenSSL, compiled from the latest source in the OpenSSL repository[4] with both assembly enabled and disabled (`no-asm`)

For each algorithm, we set the input size (`TUNE_BYTES`) parameter to 16384 bytes. For Poly1305, we modified the benchmarking code to measure just a single call to the Poly1305 MAC function (the original SUPERCOP measured two calls, one for MACing and one for verification.) The rest of SUPERCOP was left unchanged.

We then ran SUPERCOP which tested and measured all the implementations it could compile on each platform. For example, on the Graviton, it ignores all the Intel assembly implementations. We removed some redundant implementations from SUPERCOP (e.g. many similar variants of Blake2 with identical perfomance). Finally, we post-processed the results with a script to obtain the tables shown below, adding implementation author names for clarity.

KBENCH9000. We downloaded the kernel benchmarking suite KBENCH9000[5]. We extensively use this benchmarking suite for our own code (which runs in the Linux kernel), but many of the other implementations we wanted to measure could not be run in the kernel. Consequently, we ported this benchmarking suite to work in user-space, along with a script that turns off Turbo-Boost and HyperThreading and then runs the benchmark on a single core. We measured each algorithm for a variety of input lengths; for each length we pick the median measurement from 100000 runs.

In addition to our own code and Jasmin, Blake2 (reference) and OpenSSL, we added calls to the LibSodium library[6]. We then ran these measurements on the three machines we owned: the Dell XPS13 laptop, the Xeon workstation, and the Raspberry Pi 3B+.

OpenSSL. We note that although OpenSSL has code for AVX512, it appears to be disabled on the machines we tested because of issues with frequency scaling [3]. Furthermore, although OpenSSL includes multi-buffer SHA-2, it does not offer an easy-to-use API for this code, and we were unable to benchmark this code. In both cases, we inspected the OpenSSL code, and we expect that it should be at least as fast as our code.

[1]https://bench.cr.yp.to/supercop.html
[2]https://github.com/tfaoliveira/libjc
[3]https://github.com/BLAKE2/BLAKE2
[4]https://github.com/openssl/openssl
[5]https://git.zx2c4.com/kbench9000/about/
[6]https://github.com/jedisct1/libsodium

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
ChaCha20	dolbeau/amd64-avx2	C	AVX2	clang-11	0.75
	jasmin/avx2	assembly	AVX2	gcc-8	0.75
	openssl	assembly	AVX2	clang-11	0.75
	hacl-star/vec256	C	AVX2	gcc-8	0.77
	dolbeau/generic-gccsimd256	C	AVX2	clang-10	0.87
	goll-gueron	C	AVX2	gcc-8	0.90
	krovetz/avx2	C	AVX2	gcc-8	1.00
	jasmin/avx	assembly	AVX	gcc-9	1.44
	hacl-star/vec128	C	AVX	gcc-8	1.50
	dolbeau/generic-gccsimd128	C	AVX	clang-11	1.57
	krovetz/vec128	C	SSSE3	gcc-9	1.71
	bernstein/e/amd64-xmm6	assembly	SSE2	clang-11	1.83
	jasmin/ref	assembly		gcc-9	3.62
	hacl-star/scalar	C		gcc-8	3.73
	openssl-portable	C		clang-11	4.10
	bernstein/e/ref	C		gcc-9	4.10
Poly1305	openssl	assembly	AVX2	clang-11	0.35
	jasmin/avx2	assembly	AVX2	clang-11	0.35
	hacl-star/vec256	C	AVX2	clang-11	0.37
	moon/avx2/64	assembly	AVX2	clang-10	0.37
	jasmin/avx	assembly	AVX	clang-10	0.56
	moon/sse2/64	assembly	SSE2	clang-11	0.58
	moon/avx/64	assembly	AVX	clang-10	0.60
	jasmin/ref3	assembly		gcc-9	0.65
	hacl-star/vec128	C	AVX	clang-10	0.72
	openssl-portable	C		clang-11	1.19
	hacl-star/scalar	C		gcc-9	1.59
	bernstein/amd64	assembly	SSE2	gcc-8	1.65
	bernstein/53	C		gcc-8	1.79
Blake2b	neves/avx2	C	AVX2	clang-11	2.02
	neves/avxicc	assembly	AVX	clang-10	2.12
	moon/avx2/64	assembly	AVX2	clang-10	2.20
	moon/avx/64	assembly	AVX	gcc-9	2.21
	hacl-star/vec256	C	AVX2	clang-11	2.26
	neves/regs	C		gcc-9	2.34
	blake2-reference/sse	C	AVX	gcc-8	2.51
	blake2-reference/ref	C		gcc-9	2.52
	hacl-star/scalar	C		gcc-8	2.56
	neves/ref	C		gcc-8	2.72
Blake2s	neves/xmm	C	AVX	clang-11	3.06
	neves/avxicc	assembly	AVX	clang-11	3.07
	blake2-reference/sse	C	AVX	clang-11	3.07
	moon/ssse3/64	assembly	SSSE3	gcc-9	3.29
	hacl-star/vec128	C	AVX	gcc-9	3.34
	moon/avx/64	assembly	AVX	clang-11	3.48
	moon/sse2/64	assembly	SSE2	gcc-8	3.81
	neves/regs	C		gcc-9	4.01
	blake2-reference/ref	C		gcc-8	4.28
	hacl-star/scalar	C		gcc-9	4.32
	neves/ref	C		gcc-9	4.33
SHA256	**hacl-star/sha256-mb8**	C	AVX2	gcc-9	1.63 (13.04 / 8)
	hacl-star/sha256-mb4	C	AVX	clang-10	3.24 (12.97 / 4)
	openssl	assembly	AVX2	clang-11	4.96
	sphlib-small	C		gcc-9	7.28
	hacl-star/scalar	C		gcc-8	7.40
	sphlib	C		gcc-9	7.73
	openssl-portable	C		clang-10	10.45
SHA512	**hacl-star/sha512-mb4**	C	AVX2	clang-10	1.95 (7.81 / 4)
	openssl	assembly	AVX2	clang-11	3.25
	sphlib	C		gcc-8	4.84
	sphlib-small	C		gcc-9	4.98
	hacl-star/scalar	C		gcc-8	5.06
	openssl-portable	C		clang-10	5.83

Table 3: SUPERCOP Benchmarks on Dell XPS13 with Intel Kaby Lake i7-7560U processor, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-8, gcc-9, clang-10, and clang-11.

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
Chacha20	**hacl-star/vec512**	C	AVX512	gcc-9	0.56
	dolbeau/amd64-avx2	C	AVX512	clang-10	0.56
	openssl	assembly	AVX2	clang-10	0.77
	hacl-star/vec256	C	AVX2	gcc-7	0.84
	dolbeau/generic-gccsimd256	C	AVX2	clang-10	0.99
	jasmin/avx2	assembly	AVX2	clang-10	1.12
	krovetz/avx2	C	AVX2	gcc-9	1.37
	hacl-star/vec128	C	AVX	gcc-9	1.53
	dolbeau/generic-gccsimd128	C	AVX	clang-10	1.79
	krovetz/vec128	C	SSSE3	clang-10	1.99
	jasmin/avx	assembly	AVX	clang-10	2.21
	bernstein/e/amd64-xmm6	assembly	SSE2	gcc-9	2.81
	jasmin/ref	assembly		gcc-9	5.57
	hacl-star/scalar	C		gcc-9	5.74
	bernstein/e/ref	C		gcc-9	5.97
	openssl-portable	C		clang-10	6.00
Poly1305	**hacl-star/vec512**	C	AVX512	gcc-9	0.39
	jasmin/avx2	assembly	AVX2	clang-6	0.51
	openssl	assembly	AVX2	gcc-9	0.52
	hacl-star/vec256	C	AVX2	gcc-9	0.52
	moon/avx2/64	assembly	AVX2	gcc-7	0.57
	jasmin/avx	assembly	AVX	clang-10	0.87
	moon/avx/64	assembly	AVX	gcc-7	0.88
	moon/sse2/64	assembly	SSE2	clang-10	0.89
	jasmin/ref3	assembly		clang-10	0.97
	hacl-star/vec128	C	AVX	gcc-9	1.04
	openssl-portable	C		gcc-7	1.85
	hacl-star/scalar	C		gcc-9	2.31
	bernstein/amd64	assembly		gcc-9	2.53
	bernstein/53	C		gcc-9	2.73
Blake2b	neves/avx2	C	AVX2	clang-10	2.84
	blake2-reference/sse	C	AVX	clang-10	2.98
	hacl-star/vec256	C	AVX2	clang-10	3.13
	neves/avxicc	assembly	AVX	gcc-9	3.26
	moon/avx2/64	assembly	AVX2	clang-10	3.39
	moon/avx/64	assembly	AVX	gcc-9	3.40
	neves/regs	C		gcc-9	3.61
	blake2-reference/ref	C		gcc-9	3.88
	neves/ref	C		gcc-7	3.97
	hacl-star/scalar	C		gcc-9	3.97
Blake2s	neves/xmm	C	AVX	clang-6	4.11
	blake2-reference/sse	C	AVX	clang-6	4.12
	hacl-star/vec128	C	AVX	gcc-7	4.52
	neves/avxicc	assembly	AVX	gcc-9	4.72
	moon/ssse3/64	assembly	SSSE3	gcc-9	5.06
	moon/avx/64	assembly	AVX	gcc-9	5.21
	moon/sse2/64	assembly	SSE2	gcc-9	5.85
	neves/regs	C		gcc-9	6.17
	blake2-reference/ref	C		gcc-9	6.45
	neves/ref	C		gcc-9	6.57
	hacl-star/scalar	C		gcc-9	6.63
SHA256	**hacl-star/sha256-mb8**	C	AVX2	gcc-9	1.69 (13.52 / 8)
	hacl-star/sha256-mb4	C	AVX	gcc-9	3.22 (12.90 / 4)
	openssl	assembly	AVX2	gcc-7	7.42
	sphlib-small	C		gcc-7	11.04
	sphlib	C		gcc-7	11.25
	hacl-star/scalar	C		gcc-9	11.37
	openssl-portable	C		gcc-9	15.35
SHA512	**hacl-star/sha512-mb8**	C	AVX512	clang-10	1.44 (11.49 / 8)
	hacl-star/sha512-mb4	C	AVX2	gcc-9	2.07 (8.29 / 4)
	openssl	assembly	AVX2	gcc-9	4.99
	sphlib	C		gcc-9	6.72
	sphlib-small	C		gcc-9	6.75
	hacl-star/scalar	C		gcc-7	7.38
	openssl-portable	C		clang-10	9.12

Table 4: SUPERCOP Benchmarks on Dell Precision Workstation with Intel Xeon Gold 5122 processor, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-7, gcc-9, clang-6, and clang-10.

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
ChaCha20	openssl	assembly	NEON	clang	4.49
	hacl-star/vec128	C	NEON	gcc	5.19
	dolbeau/arm-neon	C	NEON	clang	5.50
	krovetz/vec128	C	NEON	gcc	6.22
	dolbeau/generic-gccsimd128	C	NEON	clang	7.01
	hacl-star/scalar	C		gcc	8.69
	openssl-portable	C		gcc	8.84
	bernstein/e/ref	C		gcc	9.08
Poly1305	openssl	assembly	NEON	clang	1.50
	hacl-star/vec128	C	NEON	clang	3.11
	openssl-portable	C		clang	3.57
	hacl-star/scalar	C		clang	4.20
	bernstein/53	C		gcc	4.95
Blake2b	neves/regs	C		gcc	6.02
	blake2-reference/ref	C		gcc	6.70
	hacl-star/scalar	C		clang	6.99
	neves/ref	C		gcc	7.35
	blake2-reference/neon	C	NEON	gcc	10.27
Blake2s	neves/regs	C		gcc	9.80
	blake2-reference/ref	C		gcc	10.70
	blake2-reference/neon	C	NEON	clang	11.31
	neves/ref	C		gcc	11.31
	hacl-star/scalar	C		gcc	11.42
	hacl-star/vec128	C	NEON	gcc	15.30
SHA256	**hacl-star/sha256-mb4**	C	NEON	gcc	12.92 (51.66 / 4)
	openssl	assembly	NEON	clang	15.09
	hacl-star/scalar	C		clang	15.70
	sphlib-small	C	NEON	gcc	15.97
	sphlib	C	NEON	gcc	16.40
	openssl-portable	C		gcc	19.85
SHA512	openssl	assembly	NEON	gcc	9.77
	openssl-portable	C		gcc	10.07
	hacl-star/scalar	C		gcc	11.27
	sphlib	C	NEON	gcc	12.40
	sphlib-small	C	NEON	gcc	12.40

Table 5: SUPERCOP Benchmarks on Raspberry Pi 3B+, with a Broadcom BCM2837B0 quad-core Cortex-A53 (ARMv8) @ 1.4GHz. Implementations are compiled with gcc-9 and clang-9.

Algorithm	Implementation	Compiler	1024	2048	4096	8192	16384	32768
ChaCha20	jasmin/avx2	gcc-9	1.21	1.18	1.17	1.16	1.16	1.16
	openssl-assembly	gcc-9	1.24	1.19	1.17	1.16	1.17	1.17
	libsodium	gcc-9	1.34	1.28	1.25	1.24	1.24	1.23
	hacl-star/vec256	gcc-9	1.38	1.29	1.25	1.23	1.28	1.27
	hacl-star/vec128	gcc-9	2.38	2.34	2.32	2.31	2.37	2.36
	hacl-star/scalar	gcc-9	6.23	6.18	6.16	6.15	6.15	6.15
	openssl-portable	clang-9	6.23	6.20	6.18	6.17	6.17	6.16
Poly1305	openssl-assembly	clang-9	0.75	0.63	0.57	0.54	0.52	0.51
	jasmin/avx2	clang-9	0.67	0.59	0.55	0.53	0.52	0.52
	hacl-star/vec256	clang-9	0.85	0.72	0.63	0.61	0.57	0.57
	libsodium	clang-9	1.23	1.10	1.04	1.00	0.99	0.99
	hacl-star/vec128	clang-9	1.29	1.20	1.17	1.16	1.14	1.13
	openssl-portable	gcc-9	1.99	1.94	1.91	1.90	1.89	1.89
	hacl-star/scalar	gcc-9	2.50	2.44	2.41	2.39	2.38	2.39
Blake2b	libsodium	clang-9	3.34	3.22	3.15	3.12	3.11	3.10
	hacl-star/vec256	clang-9	3.71	3.63	3.60	3.58	3.57	3.58
	reference-avx	gcc-9	4.12	4.09	4.02	3.99	3.98	3.97
	hacl-star/scalar	gcc-9	4.23	4.22	4.16	4.13	4.11	4.11
	openssl-portable	clang-9	6.42	5.31	4.75	4.49	4.36	4.29
Blake2s	reference-avx	clang-9	4.97	4.96	4.90	4.87	4.86	4.85
	hacl-star/vec128	gcc-9	5.42	5.36	5.34	5.32	5.32	5.35
	hacl-star/scalar	gcc-9	7.03	6.93	6.89	6.86	6.85	6.86
	openssl-portable	gcc-9	8.96	7.87	7.33	7.07	6.94	6.95
SHA256	**hacl-star/mb8**	clang-9	2.74	2.64	2.60	2.57	2.56	2.56
	hacl-star/mb4	gcc-9	5.31	5.14	5.05	5.00	4.98	4.98
	openssl-assembly	gcc-9	8.38	8.04	7.86	7.78	7.74	7.73
	libsodium	gcc-9	12.60	12.14	11.89	11.76	11.70	11.66
	hacl-star/scalar	gcc-9	12.62	12.15	11.93	11.80	11.74	11.72
	openssl-portable	clang-9	17.30	16.73	16.43	16.27	16.19	16.15
SHA512	**hacl-star/mb4**	clang-9	3.50	3.29	3.18	3.13	3.11	3.10
	openssl-assembly	gcc-9	6.03	5.59	5.36	5.25	5.20	5.18
	libsodium	clang-9	8.62	8.01	7.72	7.56	7.49	7.45
	hacl-star/scalar	gcc-9	8.66	8.08	7.80	7.66	7.59	7.56
	openssl-portable	clang-9	10.48	9.82	9.50	9.31	9.22	9.18

Table 6: KBENCH9000 Benchmarks on Dell XPS13 with Intel Kaby Lake i7-7560U processor running 64-bit Ubuntu Linux. All implementations are compiled with gcc-9 and clang-9

Algorithm	Implementation	Compiler	1024	2048	4096	8192	16384	32768
ChaCha20	**hacl-star/vec512**	gcc-9	0.68	0.61	0.58	0.56	0.56	0.56
	openssl-assembly	gcc-9	0.89	0.83	0.81	0.80	0.79	0.79
	hacl-star/vec256	gcc-9	0.98	0.93	0.90	0.89	0.88	0.88
	jasmin/avx2	gcc-9	1.20	1.17	1.16	1.15	1.15	1.15
	libsodium	gcc-9	1.26	1.21	1.18	1.17	1.16	1.16
	hacl-star/vec128	gcc-9	1.63	1.60	1.58	1.58	1.58	1.57
	hacl-star/scalar	gcc-9	6.19	6.15	6.12	6.12	6.11	6.11
	openssl-portable	gcc-9	6.23	6.19	6.17	6.17	6.16	6.16
Poly1305	**hacl-star/vec512**	gcc-9	0.94	0.65	0.51	0.43	0.40	0.38
	jasmin/avx2	gcc-9	0.67	0.59	0.55	0.53	0.52	0.51
	openssl-assembly	clang-9	0.75	0.63	0.57	0.54	0.52	0.51
	hacl-star/vec256	gcc-9	0.82	0.66	0.58	0.54	0.52	0.51
	libsodium	clang-9	1.14	1.01	0.95	0.92	0.91	0.90
	hacl-star/vec128	gcc-9	1.27	1.16	1.11	1.09	1.07	1.06
	openssl-portable	gcc-9	1.97	1.93	1.92	1.89	1.88	1.88
	hacl-star/scalar	gcc-9	2.49	2.45	2.41	2.39	2.38	2.39
Blake2b	reference-avx	clang-9	3.23	3.14	3.09	3.07	3.06	3.05
	libsodium	gcc-9	3.34	3.22	3.18	3.15	3.14	3.14
	hacl-star/vec256	clang-9	3.40	3.36	3.33	3.32	3.31	3.31
	hacl-star/scalar	gcc-9	4.21	4.14	4.11	4.10	4.09	4.09
	openssl-portable	gcc-9	5.88	5.06	4.62	4.40	4.30	4.29
Blake2s	reference-avx	clang-9	4.60	4.53	4.50	4.49	4.48	4.48
	hacl-star/vec128	gcc-9	4.77	4.71	4.69	4.67	4.67	4.69
	openssl-portable	gcc-9	8.33	7.46	7.02	6.82	6.71	6.73
	hacl-star/scalar	gcc-9	6.90	6.86	6.85	6.83	6.82	6.84
SHA256	**hacl-star/mb8**	gcc-9	1.87	1.80	1.76	1.75	1.74	1.74
	hacl-star/mb4	gcc-9	3.55	3.43	3.36	3.33	3.32	3.31
	openssl-assembly	clang-9	8.38	8.04	7.85	7.76	7.72	7.71
	libsodium	clang-9	12.57	12.10	11.85	11.73	11.67	11.63
	hacl-star/scalar	gcc-9	12.51	12.10	11.88	11.76	11.71	11.69
	openssl-portable	clang-9	16.92	16.39	16.11	15.96	15.88	15.85
SHA512	**hacl-star/mb8**	clang-9	1.72	1.61	1.56	1.53	1.52	1.52
	hacl-star/mb4	gcc-9	2.40	2.25	2.18	2.14	2.12	2.11
	openssl-assembly	gcc-9	6.06	5.58	5.33	5.22	5.17	5.14
	libsodium	gcc-9	8.55	8.00	7.72	7.57	7.50	7.47
	hacl-star/scalar	gcc-9	8.59	8.05	7.79	7.65	7.58	7.55
	openssl-portable	gcc-9	10.63	9.95	9.63	9.45	9.37	9.32

Table 7: KBENCH9000 Benchmarks on Dell Precision workstation with Intel(R) Xeon(R) Gold 5122 CPU @ 3.60GHz processor running 64-bit Ubuntu Linux. All implementations are compiled with gcc-9 and clang-9

Algorithm	Implementation	Compiler	1024	2048	4096	8192	16384	32768
ChaCha20	openssl-assembly	clang	4.59	4.53	4.50	4.49	4.50	4.55
	hacl-star/vec128	gcc	5.42	5.32	5.27	5.25	5.27	5.32
	openssl-portable	clang	8.88	8.84	8.82	8.82	8.84	8.94
	hacl-star/scalar	gcc	8.91	8.86	8.84	8.83	8.87	8.99
	libsodium	clang	9.33	9.25	9.21	9.21	9.25	9.37
Poly1305	openssl-assembly	gcc	1.97	1.72	1.59	1.53	1.50	1.51
	hacl-star/vec128	clang	3.48	3.30	3.21	3.17	3.15	3.16
	openssl-portable	gcc	3.74	3.65	3.61	3.58	3.57	3.59
	hacl-star/scalar	gcc	4.61	4.52	4.48	4.46	4.45	4.47
	libsodium	gcc	5.35	5.27	5.23	5.21	5.20	5.22
Blake2b	openssl-portable	gcc	11.33	8.71	7.39	6.74	6.43	6.30
	hacl-star/scalar	gcc	7.12	7.01	6.95	6.93	6.92	6.96
	libsodium	gcc	7.60	7.42	7.34	7.29	7.28	7.33
	reference-neon	gcc	11.13	10.96	10.87	10.82	10.81	10.91
Blake2s	openssl-portable	gcc	14.92	12.60	11.44	10.89	10.63	10.56
	hacl-star/scalar	gcc	11.59	11.51	11.48	11.46	11.47	11.57
	reference-neon	gcc	11.83	11.68	11.61	11.57	11.58	11.66
	hacl-star/vec128	gcc	16.58	16.52	16.49	16.49	16.61	16.64
SHA256	**hacl-star/mb4**	gcc	13.68	13.23	13.01	12.99	13.08	13.00
	openssl-assembly	gcc	16.22	15.58	15.26	15.10	15.15	15.12
	hacl-star/scalar	gcc	17.49	16.92	16.64	16.51	16.58	16.54
	libsodium	gcc	19.43	18.68	18.31	18.13	18.22	18.19
	openssl-portable	clang	21.01	20.25	19.88	19.70	19.79	19.73
SHA512	openssl-assembly	gcc	11.10	10.37	10.01	9.83	9.76	9.82
	openssl-portable	gcc	11.45	10.70	10.33	10.14	10.08	10.12
	hacl-star/scalar	gcc	12.70	11.94	11.55	11.36	11.28	11.34
	libsodium	gcc	13.73	12.76	12.27	12.03	11.94	11.98

Table 8: KBENCH9000 Benchmarks on Raspberry Pi 3B+, with a Broadcom BCM2837B0 quad-core Cortex-A53 (ARMv8) @ 1.4GHz running 64-bit Ubuntu Linux. All implementations are compiled with gcc-9 and clang-9

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
ChaCha20	**hacl-star/vec512**	C	AVX512	gcc-9	0.52
	dolbeau/amd64-avx2	C	AVX512	clang	0.52
	openssl	assembly	AVX2	gcc-9	0.64
	hacl-star/vec256	C	AVX2	gcc-9	0.71
	jasmin/avx2	assembly	AVX2	gcc-9	0.93
	dolbeau/generic-gccsimd256	C	AVX2	gcc-9	0.94
	krovetz/avx2	C	AVX2	gcc-9	1.14
	hacl-star/vec128	C	AVX	gcc-9	1.27
	dolbeau/generic-gccsimd128	C	AVX	gcc-9	1.51
	jasmin/avx	assembly	AVX	gcc-9	1.83
	krovetz/vec128	C	SSSE3	clang	1.88
	bernstein/e/amd64-xmm6	assembly	SSE2	gcc-9	2.33
	jasmin/ref	assembly		clang-9	4.62
	hacl-star/scalar	C		gcc-9	4.76
	bernstein/e/ref	C		gcc-9	4.95
	openssl-portable	C		gcc-9	4.98
Poly1305	**hacl-star/vec512**	C	AVX512	gcc-9	0.40
	jasmin/avx2	assembly	AVX2	gcc-9	0.49
	openssl	assembly	AVX2	gcc-9	0.49
	hacl-star/vec256	C	AVX2	gcc-9	0.49
	moon/avx2/64	assembly	AVX2	clang-9	0.53
	jasmin/avx	assembly	AVX	gcc-9	0.72
	moon/avx/64	assembly	AVX	gcc-9	0.77
	jasmin/ref3	assembly		gcc-9	0.80
	moon/sse2/64	assembly	SSE2	gcc-9	0.86
	hacl-star/vec128	C	AVX	gcc-9	0.88
	openssl-portable	C		gcc-9	1.53
	hacl-star/scalar	C		gcc-9	1.92
	bernstein/amd64	assembly		gcc-9	2.20
	bernstein/53	C		gcc-9	2.51
Blake2b	neves/avx2	C	AVX2	clang-9	2.60
	neves/avxicc	assembly	AVX	gcc-9	2.70
	moon/avx/64	assembly	AVX	gcc-9	2.82
	blake2-reference/sse	C	AVX	clang-9	2.83
	neves/regs	C		gcc-9	2.99
	hacl-star/vec256	C	AVX2	gcc-9	2.99
	blake2-reference/ref	C		gcc-9	3.21
	moon/avx2/64	assembly	AVX2	clang-9	3.23
	hacl-star/scalar	C		gcc-9	3.29
	neves/ref	C		gcc-9	3.34
Blake2s	blake2-reference/sse	C	AVX	clang	3.33
	neves/xmm	C	AVX	clang	3.37
	hacl-star/vec128	C	AVX	gcc-9	3.76
	neves/avxicc	assembly	AVX	gcc-9	3.91
	moon/ssse3/64	assembly	SSSE3	gcc-9	4.20
	moon/avx/64	assembly	AVX	gcc-9	4.32
	moon/sse2/64	assembly	SSE2	gcc-9	4.85
	neves/regs	C		gcc-9	5.11
	blake2-reference/ref	C		gcc-9	5.35
	neves/ref	C		gcc-9	5.45
	hacl-star/scalar	C		gcc-9	5.49
SHA256	**hacl-star/sha256-mb8**	C	AVX2	gcc-9	1.40 (11.21 / 8)
	hacl-star/sha256-mb4	C	AVX	gcc-9	2.68 (10.70 / 4)
	openssl	assembly	AVX2	clang-9	6.23
	sphlib-small	C		gcc	9.15
	sphlib	C		gcc-9	9.34
	hacl-star/scalar	C		gcc-9	9.43
	openssl-portable	C		gcc-9	12.73
SHA512	**hacl-star/sha512-mb8**	C	AVX512	clang	1.39 (11.11 / 8)
	hacl-star/sha512-mb4	C	AVX2	gcc-9	1.72 (6.89 / 4)
	openssl	assembly	AVX2	gcc-9	4.19
	sphlib	C		gcc-9	5.63
	sphlib-small	C		gcc-9	5.64
	hacl-star/scalar	C		gcc-9	6.19
	openssl-portable	C		gcc-9	7.56

Table 9: SUPERCOP Benchmarks on Amazon EC2 `t3.large` instance with Intel(R) Xeon(R) Platinum 8259CL CPU @ 2.50GHz processor, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-7, clang-7, gcc-9, and clang-9.

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
ChaCha20	**hacl-star/vec512**	C	AVX512	gcc-9	0.44
	dolbeau/amd64-avx2	C	AVX512	clang-9	0.44
	openssl	assembly	AVX2	gcc-9	0.61
	hacl-star/vec256	C	AVX2	gcc-9	0.67
	dolbeau/generic-gccsimd256	C	AVX2	clang-9	0.81
	jasmin/avx2	assembly	AVX2	gcc-9	0.89
	krovetz/avx2	C	AVX2	gcc-9	1.08
	hacl-star/vec128	C	AVX	gcc-9	1.21
	dolbeau/generic-gccsimd128	C	AVX	clang-9	1.44
	krovetz/vec128	C	SSSE3	clang-9	1.61
	jasmin/avx	assembly	AVX	gcc-9	1.74
	bernstein/e/amd64-xmm6	assembly	SSE2	gcc-9	2.22
	jasmin/ref	assembly		gcc-9	4.40
	hacl-star/scalar	C		gcc-9	4.54
	bernstein/e/ref	C		gcc-9	4.72
	openssl-portable	C		gcc-9	4.75
Poly1305	**hacl-star/vec512**	C	AVX512	gcc-9	0.31
	jasmin/avx2	assembly	AVX2	gcc-9	0.41
	openssl	assembly	AVX2	clang	0.41
	hacl-star/vec256	C	AVX2	gcc-9	0.41
	moon/avx2/64	assembly	AVX2	gcc	0.46
	jasmin/avx	assembly	AVX	gcc-9	0.69
	moon/avx/64	assembly	AVX	gcc-9	0.71
	moon/sse2/64	assembly	SSE2	gcc-9	0.72
	jasmin/ref3	assembly		gcc-9	0.76
	hacl-star/vec128	C	AVX	gcc-9	0.82
	openssl-portable	C		gcc-9	1.46
	hacl-star/scalar	C		gcc-9	1.83
	bernstein/amd64	assembly		gcc-9	2.00
	bernstein/53	C		gcc-9	2.14
Blake2b	neves/avx2	C	AVX2	clang-9	2.74
	moon/avx2/64	assembly	AVX2	clang-9	2.75
	hacl-star/vec256	C	AVX2	clang-9	2.85
	blake2-reference/sse	C	AVX	clang-9	3.06
	moon/avx/64	assembly	AVX	clang-9	3.28
	neves/avxicc	assembly	AVX	clang-9	3.35
	hacl-star/scalar	C		clang-9	3.85
	neves/regs	C		clang-9	4.48
	blake2-reference/ref	C		clang-9	4.58
	neves/ref	C		clang-9	4.68
Blake2s	blake2-reference/sse	C	AVX	clang-9	3.53
	moon/ssse3/64	assembly	SSSE3	clang-9	4.01
	hacl-star/vec128	C	AVX	clang-9	4.05
	neves/xmm	C	AVX	clang	4.11
	neves/avxicc	assembly	AVX	clang-9	4.15
	moon/avx/64	assembly	AVX	clang	4.59
	moon/sse2/64	assembly	SSE2	clang	5.02
	hacl-star/scalar	C		gcc-9	5.68
	neves/regs	C		clang	5.73
	blake2-reference/ref	C		gcc-9	6.22
	neves/ref	C		gcc-9	6.99
SHA256	**hacl-star/sha256-mb8**	C	AVX2	gcc-9	1.33 (10.60 / 8)
	hacl-star/sha256-mb4	C	AVX	gcc-9	2.55 (10.20 / 4)
	openssl	assembly	AVX2	gcc-9	6.26
	sphlib	C		clang-9	9.78
	hacl-star/scalar	C		clang	9.81
	sphlib-small	C		clang-9	9.93
	openssl-portable	C		clang	12.14
SHA512	**hacl-star/sha512-mb8**	C	AVX512	clang	1.18 (9.43 / 8)
	hacl-star/sha512-mb4	C	AVX2	clang	1.75 (6.99 / 4)
	openssl	assembly	AVX2	clang-9	3.98
	hacl-star/scalar	C		clang	6.39
	sphlib	C		clang-9	6.67
	openssl-portable	C		clang-9	7.40
	sphlib-small	C		clang-9	7.45

Table 10: SUPERCOP Benchmarks on Amazon EC2 `c5.metal` instance with Intel(R) Xeon(R) Platinum 8275CL CPU @ 2.50GHz processor, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-7, clang-7, gcc-9, and clang-9.

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
ChaCha20	openssl	assembly	NEON	clang	4.40
	hacl-star/vec128	C	NEON	gcc	5.10
	dolbeau/arm-neon	C	NEON	gcc	5.16
	krovetz/vec128	C	NEON	clang	5.79
	dolbeau/generic-gccsimd128	C	NEON	clang	5.87
	hacl-star/scalar	C		gcc	5.95
	openssl-portable	C		gcc	8.43
	bernstein/e/ref	C		clang	8.90
Poly1305	openssl	assembly	NEON	clang	1.16
	hacl-star/vec128	C	NEON	clang	1.98
	openssl-portable	C		clang	3.08
	bernstein/53	C		clang	3.74
	hacl-star/scalar	C		gcc	5.13
Blake2b	neves/regs	C		gcc	5.46
	blake2-reference/ref	C		gcc	5.78
	hacl-star/scalar	C		gcc	5.95
	neves/ref	C		gcc	6.21
	blake2-reference/neon	C	NEON	clang	11.63
Blake2s	neves/regs	C		gcc	9.10
	blake2-reference/ref	C		gcc	9.34
	hacl-star/scalar	C		gcc	9.78
	neves/ref	C		gcc	10.06
	blake2-reference/neon	C	NEON	clang	17.15
	hacl-star/vec128	C	NEON	gcc	19.15
SHA256	openssl	assembly	SHA-EXT	clang	2.01
	hacl-star/sha256-mb4	C	NEON	clang	10.12 (40.46 / 4)
	sphlib-small	C	NEON	clang	12.08
	hacl-star/scalar	C		gcc	12.15
	sphlib	C	NEON	gcc	12.31
	openssl-portable	C		gcc	14.58
SHA512	openssl	assembly	NEON	gcc	7.28
	openssl-portable	C		gcc	7.75
	hacl-star/scalar	C		gcc	7.93
	sphlib-small	C	NEON	clang	9.81
	sphlib	C	NEON	clang	9.82

Table 11: SUPERCOP Benchmarks on Amazon EC2 `a1.metal` instance with Amazon Graviton1 Cortex-A72 @ 2.3GHz, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-7 and clang-7.

Algorithm	Implementation	Language	SIMD Features	Compiler	Cycles/Byte
ChaCha20	openssl	assembly	NEON	gcc	2.36
	hacl-star/vec128	C	NEON	gcc	2.95
	dolbeau/arm-neon	C	NEON	gcc	3.16
	krovetz/vec128	C	NEON	gcc	3.58
	hacl-star/scalar	C		gcc	3.66
	dolbeau/generic-gccsimd128	C	NEON	gcc	3.74
	openssl-portable	C		gcc	5.78
	bernstein/e/ref	C		clang	5.98
Poly1305	openssl	assembly	NEON	clang	1.05
	hacl-star/vec128	C	NEON	clang	1.54
	openssl-portable	C		gcc	2.82
	bernstein/53	C		gcc	2.93
	hacl-star/scalar	C		gcc	5.07
Blake2b	neves/regs	C		clang	3.78
	blake2-reference/ref	C		gcc	3.82
	hacl-star/scalar	C		gcc	3.98
	neves/ref	C		gcc	3.99
	blake2-reference/neon	C	NEON	clang	7.83
Blake2s	neves/regs	C		gcc	6.26
	blake2-reference/ref	C		gcc	6.45
	neves/ref	C		gcc	6.54
	hacl-star/scalar	C		gcc	6.60
	hacl-star/vec128	C	NEON	clang	10.44
	blake2-reference/neon	C	NEON	clang	11.16
SHA256	openssl	assembly	SHA-EXT	gcc	1.57
	hacl-star/sha256-mb4	C	NEON	clang	6.52 (26.09 / 4)
	sphlib-small	C	NEON	clang	9.76
	sphlib	C	NEON	gcc	10.16
	hacl-star/scalar	C		gcc	10.44
	openssl-portable	C		gcc	11.72
SHA512	openssl	assembly	NEON	gcc	6.03
	openssl-portable	C		gcc	6.31
	hacl-star/scalar	C		gcc	7.00
	sphlib-small	C	NEON	clang	7.96
	sphlib	C	NEON	clang	7.99

Table 12: SUPERCOP Benchmarks on Amazon EC2 `m6g.metal` instance with Amazon Graviton2 Cortex-A76 @ 2.3GHz, running 64-bit Ubuntu Linux. Implementations are compiled with gcc-7 and clang-7.