EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider

Jonathan Protzenko*, Bryan Parno‡, Aymeric Fromherz‡, Chris Hawblitzel*, Marina Polubelova†, Karthikeyan Bhargavan†
Benjamin Beurdouche†, Joonwon Choi*§, Antoine Delignat-Lavaud*, Cédric Fournet*, Natalia Kulatova†,
Tahina Ramananandro*, Aseem Rastogi*, Nikhil Swamy*, Christoph M. Wintersteiger*, Santiago Zanella-Beguelin*

*Microsoft Research ‡Carnegie Mellon University †Inria §MIT

***Abstract*—We present EverCrypt: a comprehensive collection of verified, high-performance cryptographic functionalities available via a carefully designed API. The API provably supports agility (choosing between multiple algorithms for the same functionality) and multiplexing (choosing between multiple implementations of the same algorithm). Through abstraction and zero-cost generic programming, we show how agility can simplify verification without sacrificing performance, and we demonstrate how C and assembly can be composed and verified against shared specifications. We substantiate the effectiveness of these techniques with new verified implementations (including hashes, Curve25519, and AES-GCM) whose performance matches or exceeds the best unverified implementations. We validate the API design with two high-performance verified case studies built atop EverCrypt, resulting in line-rate performance for a secure network protocol and a Merkle-tree library, used in a production blockchain, that supports 2.7 million insertions/sec. Altogether, EverCrypt consists of over 124K verified lines of specs, code, and proofs, and it produces over 29K lines of C and 14K lines of assembly code.**

I. Introduction

Application developers seldom write their own cryptographic code; instead they rely on a *cryptographic provider*, such as OpenSSL's `libcrypto` [65], Windows Cryptography API [58], or `libsodium` [7]. Developers expect their provider to be *comprehensive* in supplying all of the functionalities they need (asymmetric and symmetric encryption, signing, hashing, key derivation, ...) for all the platforms they support.

Further, a modern cryptographic provider should be *agile*; that is, it should provide multiple algorithms (e.g., ChaCha-Poly and AES-GCM) for the same functionality (e.g., authenticated encryption) and all algorithms should employ a single unified API, to make it simple to change algorithms if one is broken.

A modern cryptographic provider should also support *multiplexing*, that is, the ability to choose between multiple implementations of the same algorithm. This allows the provider to employ high-performance implementations on popular hardware platforms (OpenSSL, for example, supports dozens), while still providing a fallback implementation that will work on any platform. Ideally, these disparate implementations should be exposed to the developer via a single unified API, so that she need not change her code when a new optimized version is deployed, and so that the provider can automatically choose the optimal implementation, rather than force the developer to do so. Historically, this process has been error prone (due in part to Intel and AMD reporting CPU features inconsistently [78]), with various cryptographic providers invoking illegal instructions on specific platforms [74], leading to killed processes and even crashing kernels.

Since a cryptographic provider is the linchpin of most security-sensitive applications, its *correctness* and *security* are crucial. However, for most applications (e.g., TLS, cryptocurrencies, or disk encryption), the provider is also on the critical path of the application's *performance*. Historically, it has been notoriously difficult to produce cryptographic code that is fast, correct, and secure (e.g., free of leaks via side channels). For instance, OpenSSL's `libcrypto` reported 24 vulnerabilities between January 1, 2016 and May 1, 2019 (Figure 15).

Such critical, complex code is a natural candidate for formal verification, which can mathematically guarantee correctness and security even for complex low-level implementations. Indeed, in recent years, multiple research groups have produced exciting examples of verified cryptographic code.

However, as we discuss in detail in §II, previous work has not produced a verified cryptographic *provider* comparable to the unverified providers in use today. Instead, some papers verify a single algorithm [14, 17, 80], or even just a portion thereof [26, 67, 75]. Others focus on many variants of a single functionality, with elliptic curves being a popular choice [36, 81]. Finally, a few works include several classes of functionality (e.g., hashing and encryption) but either in small number for specific platforms [25, 39], or with speeds below those of state-of-the-art providers [10, 44, 73, 82]. All of these works contribute important techniques and insights into how to best verify cryptographic code, but they all fall short of what application developers want from a cryptographic provider.

Building on previous algorithm verification efforts [25, 39, 82], we present EverCrypt, a comprehensive, provably correct and secure, cryptographic provider that supports agility, multiplexing, and auto-configuration. EverCrypt's agility means that multiple algorithms provably provide the same API to clients, and its multiplexing demonstrates multiple disparate implementations (almost all entirely new, including several platform-specific implementations that beat state-of-the-art unverified libraries) verifying against the same cryptographic specification. The API is carefully designed to be usable by both verified and unverified (e.g., C) clients. To support the former, we show that multiple layers of abstraction, both in the implementation and (more surprisingly) in the specification of cryptographic algorithms, naturally leads to agile code while also improving verification efficiency. We also demonstrate how the judicious use of generic programming can enhance developer productivity and simplify specifications by cleanly extracting commonality from related algorithms.

To illustrate the benefits of multiplexing, and as part of our efforts to ensure state-of-the-art performance for EverCrypt, we present new verified implementations of a large variety of algorithms (Figure 4). These include, e.g., Curve25519 [18], hash functions, and authenticated encryption with additional data (AEAD) that transparently multiplex between a generic C implementation and a hand-tuned x64 assembly implementation. When run on x64, they match or exceed the performance of the best unverified code, while the C implementations provide support across all other platforms (and offer performance competitive with unverified C code as well).

To illustrate the utility of EverCrypt's API, we develop two applications atop it. First, a new Merkle-tree [56] library leverages EverCrypt's agile hashing API. The Merkle trees support high-performance secure logging, providing over 2.7M transactions/s. They also come with comprehensive safety, correctness, and cryptographic security properties, illustrating EverCrypt's value when building formally-verified applications. Second, we build a verified transport security layer for the QUIC network protocol [45] and, by connecting it to unverified QUIC implementations, show that it supports line-rate performance.

All EverCrypt code is proven safe and functionally correct.[1] Safety means the code respects memory abstractions (e.g., never reading outside the bounds of an array) and does not perform illegal operations (e.g., divide by zero). Functional correctness means the code's computational behavior is precisely described by a pure mathematical function. Further, all EverCrypt code is proven side-channel resistant; specifically, the sequence of instructions executed and the memory addresses accessed do not depend on secret inputs; more informally, EverCrypt provably implements "constant-time cryptography" [15]. Together, the guarantees would have prevented all of `libcrypto`'s vulnerabilities, except one related to prime generation (§XIII).

EverCrypt does not (yet) include cryptographic proofs of security (e.g., we do not prove that counter-mode encryption is secure if the underlying cipher is secure). As future work, we plan to apply techniques similar to those of Bhargavan et al. [22] to provide such proofs for common security assumptions atop EverCrypt's functionalities. For now, we illustrate such proofs of high-level security properties on our sample applications, e.g., we prove that collisions on our Merkle tree can be reduced to collisions of the underlying hash function.

While EverCrypt provides provably correct and secure cryptography, like any verified artifact, its guarantees rely on the correctness of both our specifications and our verification tools. All EverCrypt code is proven using the F* proof assistant, so F* and its toolchain are part of EverCrypt's trusted computing base (TCB), which we discuss at length in §II-C. EverCrypt also currently lacks support for older algorithms like RSA and DES, and it includes hand-optimized assembly only for x64 platforms. We plan to support additional platforms in the future.

In summary, this paper makes the following contributions:

1) EverCrypt, a comprehensive, verified cryptographic provider that offers agility, multiplexing, and speed;
2) the design of a principled, abstract, agile API, usable by verified (F*) and unverified (C) clients alike;
3) its multiplexing implementation, coupled with verified zero-cost generic programming techniques to streamline the verification of cryptographic algorithms without sacrificing performance;
4) a new approach based on dependently typed generic programming [12] to prove the safety and correctness of fine-grained interoperation between C and assembly;
5) new verified cryptographic implementations (including hashes, Curve25519, and AES-GCM) whose performance matches or exceeds the best unverified implementations;
6) case studies on real applications of Merkle trees and QUIC transport security to demonstrate the utility of EverCrypt's API and the performance it enables.

The specifications, proofs, and code of EverCrypt are all available as open source (§IX).

II. Background and Related Work

Multiple research projects have contributed to our understanding of how to produce verified cryptographic code, but none provide the performance, comprehensiveness, and convenience that developers expect from a cryptographic provider, and none investigate the challenges of agility or automatic multiplexing.

Some work verifies parts of individual algorithms, particularly portions of Curve25519. For example, Chen et al. [26] verify the correctness of a Montgomery ladder using Coq [72] and an SMT solver, while Tsai et al. [75] and Polyakov et al. [67] do so using an SMT solver and a computer algebra system. All three focus on implementations written in assembly-like languages. Almeida et al. [10], in contrast, use a domain-specific language (DSL) and verified compilation (via Coq, Dafny [51], and custom scripting) to verify memory safety and side-channel resistance, and, in more recent work [11], correctness for ChaCha20 and Poly1305 using EasyCrypt [16].

Other papers verify a single algorithm, e.g., SHA-256 [14], HMAC instantiated with SHA-256 [17], or HMAC-DRBG [80]. Each is written in Coq using the Verified Software Toolchain [13] and yields C code, which is compiled by CompCert [52]. Using the Foundational Cryptography Framework [66], they prove the cryptographic constructions secure.

The SAW tool [31] proves the correctness of C code (via LLVM) and Java code with respect to mathematical specifications written in Cryptol. Dawkins et al. use it to verify reference implementations and simple in-house implementations of AES, ZUC, FFS, ECDSA, and the SHA-384 inner loop [73], but they do not report performance, nor verify assembly.

Other works verify multiple algorithms for the same functionality, typically for elliptic curves. For example, Zinzindohoué et al. develop a generic library based on templates, and instantiate it for three popular curves [81], including Curve25519, but they report 100× performance overheads compared with C code. FiatCrypto [36] employs a verified, generic compilation scheme from a high-level Coq specification to C code for

[1] The specs, proofs, and code of EverCrypt are all available as open source.

bignum arithmetic, which enables them to generate verified field arithmetic code for several curves. Even with aggressive optimization by an unverified C compiler, they still lag hand-tuned assembly implementations by 0.25-2×. They also report issues with compilers segfaulting when compiling their code, or even generating binaries that produce the wrong output, illustrating an advantage of EverCrypt's ability to produce verified assembly code when needed.

Finally, Hawblitzel et al. [44] verify a variety of algorithms (RSA signing and encryption, SHA-1, SHA-256, and HMAC) in Dafny, but they are compiled and verified as a part of a larger application, and they report performance overheads ranging from 30% to 100× compared with unverified implementations.

In the unverified space, Tuveri and Brumley propose an API for bridging the gap between cryptographic research results and deployed cryptographic systems [76], while recent providers such as `libsodium` promote simpler APIs that hide most cryptographic details.

A. Background on EverCrypt

Rather than commit to a single target language (e.g., C or assembly) as most previous work does, EverCrypt overhauls, significantly rewrites, and unifies (via multiplexing and an agile API) two open-source projects, HACL* [82] and Vale-Crypt [25, 39]. We choose these projects as starting points, since HACL* produces high-performance C code for cross-platform support, while ValeCrypt produces assembly code for maximum performance on specific hardware platforms. Both employ F* [71] for verification, which allows EverCrypt to reason about both in a single unified framework. We provide additional background on all three projects below.

a) F* and Low*: F* [71] is an ML-like programming language that, like Coq [72] or Nuprl [28], supports dependent types. This makes it easy to specify, for example, Curve25519's field (namely integers modulo the prime $2^{255} - 19$):

```
let prime = (pow2 255) − 19
type elem = e:int{0 ≤ e ∧ e < prime}
let add (e0:elem) (e1:elem) = (e0 + e1) % prime
let mul (e0:elem) (e1:elem) = (e0 ∗ e1) % prime
```

Definitions of functions and top-level values are introduced with let. A type representing field elements (elem) is defined as a restriction (shown in curly braces) of mathematical integers (int). Field operations are defined in the obvious way as pure mathematical functions, the default "effect" for F* functions. Functions can also be annotated with other effects, e.g., to indicate stateful operations affecting the heap, or a function that handles program IO. F* relies on SMT-based automation to prove properties about programs using a weakest-precondition calculus [8], and it also includes a tactic system [55] to allow the programmer greater control over how proofs are discharged.

Programs written in F* can be compiled to OCaml or F#. They can also be compiled to C, as long as their run-time code is written in a low-level subset of F* called Low* [68]. The programmer can still use high-level F* code in specifications and proofs, since they are erased before extraction to C. Low* programs must obey syntactic restrictions, e.g., avoiding closures and partial applications, and, after erasure, they cannot use high-level libraries, e.g. for lists and mathematical integers. Instead, the programmer must use explicitly heap- or stack-allocated arrays and manage their lifetimes manually, and similarly rely on Low* libraries for machine integers, endianness, etc.

```
let fmul (output a b: felem): Stack unit
  (requires λh0 → {output,a,b} ∈ h0 ∧ (a == b ∨ disjoint a b))
  (ensures λh0 h1 → modifies_only output h0 h1
     ∧ elem h1.[output] == mul (elem h0.[a]) (elem h0.[b]))
```

Figure 1. Type signature for HACL*'s implementation (in Low*) of a field multiplication for Curve25519. An felem is an array of five 64-bit integers needed to represent a field element. The input arrays a,b and the output array output are required to be live (i.e., still allocated). After fmul completes, the only change from the old heap h0 to the new heap h1 is to the output array, which matches mul, the mathematical spec for field multiplication. The Stack effect annotation in its type guarantees that fmul is free of memory leaks, since its allocations are only on the call stack, and hence they are reclaimed as the function returns.

b) HACL*: HACL* [82] is a collection of cryptographic primitives: Chacha20, Poly1305, their AEAD combination, Curve25519, Ed25519, and the SHA2 family of hashes, entirely written in Low* and compiling to a collection of C files.

Like all code written in Low*, HACL* is, by construction, devoid of memory errors such as use-after-free, or out-of-bounds accesses. In addition to basic safety, HACL* proves *functional correctness* for its algorithms; e.g., it shows that optimized field operations in $2^{255} - 19$ are free of mathematical errors (see Figure 1). Finally, using a restricted model of constant-time integers HACL* ensures that one cannot branch on a secret or use it for array accesses, and thus guarantees that the resulting C code is free of the most egregious kinds of side-channels.

While HACL* achieves cross-platform compatibility, the performance of the resulting C code lags that of hand-tuned assembly used in state-of-the-art cryptographic libraries. As a cryptographic provider, it does not support multiplexing (e.g., to switch between a version using compiler intrinsics and a version in vanilla C) and it lacks agility, making further extensions to its set of algorithms difficult, and hindering verified clients from integrating HACL* in their developments. We address these shortcomings by rewriting HACL* algorithms in a style that supports agility, generic programming, and multiplexing with deep integration between Vale and Low*.

c) ValeCrypt: ValeCrypt is a collection of verified assembly programs written in Vale [25, 39], which allows developers to annotate assembly code with pre- and postconditions, loop invariants, lemmas, etc. to assist with verification (Fig. 2).

Vale models a well-structured subset of assembly language, with control flow restricted to blocks of instructions, if/then/else branching, and while loops. Although limited, this subset is well-suited to implementations of cryptographic primitives similar to those found in OpenSSL.

The latest version of Vale [39] can be viewed as a DSL that relies on deeply embedded hardware semantics formalized within F*, which also discharges proof obligations demonstrating the correctness and security of Vale assembly programs.

Vale implementations are verified for safety (programs do not crash), functional correctness (the implementations match

```
procedure mul1(ghost dst:arr64, ghost src:arr64)
  lets dst_ptr @= rdi; src_ptr @= rsi; b @= rdx;
      a := pow2_four(src[0], src[1], src[2], src[3]);
  reads dst_ptr; src_ptr; b;
  modifies rax; r8; r9; r10; r11; r12; r13; mem; efl;
  requires adx_enabled && bmi2_enabled;
    arrays_disjoint(dst, src) || dst == src;
    validDstAddrs64(mem, dst_ptr, dst, 4);
    validSrcAddrs64(mem, src_ptr, src, 4);
  ensures
    let d := pow2_five(dst[0], dst[1], dst[2], dst[3], rax);
    d == old(a * b);
```

Figure 2. Type signature for ValeCrypt's implementation of multiplying a 256-bit number (in the `src` array) by a 64-bit number in `b`. The arrays themselves are "ghost" variables, i.e., used only for proof purposes. The signature first declares some local aliases using `lets` (e.g., the pointer to the `dst` array must be in the `rdi` register). The procedure then specifies its framing (the portions of state it reads/modifies). The preconditions show that it expects the CPU to support the ADX and BMI2 extensions; the input and output arrays cannot partially overlap; and the pointers provided are valid. It returns the result of the multiplication in a combination of the destination array and `rax`.

their specification on all inputs), and robustness to cache-based and timing-based side-channel attacks.

ValeCrypt includes several implementations of cryptographic primitives. Some of these achieve good performance, such as 750 MB/s for AES-CBC [25] and 990 MB/s for AES-GCM [39], but this still falls short of the fastest OpenSSL assembly language code, which reaches up to 6400 MB/s for AES-GCM [39]. ValeCrypt's implementations are, by design, platform specific and do not offer fallbacks or even automatic detection of CPU capabilities.

In addition to improving on verified cryptographic implementations provided by ValeCrypt, we also improve on the Vale framework itself, notably the manner in which Vale and Low⋆ interoperate. Previous work on Vale [39] presented an external trusted tool called CCWrap, which would textually print out Low⋆ wrappers around ValeCrypt routines, relying on the user to edit the file and finish the proof that the Low⋆ signature provably summarizes the semantics of the ValeCrypt code. This approach is vulnerable to accidental modifications in the generated code that would affect the semantics; and to mistakes in the tool itself. More importantly, CCWrap generates a fresh model of assembly calls at each invocation, which F⋆ then needs to verify. This does not scale up to an entire provider. In contrast, we provide a new model and verified library in support of generic interoperability between Vale and Low⋆ that specifically addresses all these drawbacks (§V).

Other projects (e.g, CertiKOS [40]) also focused on verified interoperation by reasoning about deep embeddings of both C and assembly. We do not have access to a deep embedding of C semantics in F⋆ as C code is extracted from Low⋆, and then compiled using generic compilers such as CompCert or GCC. Hence, our interoperation is lighter-weight, at the price of a slightly bigger TCB that models calls into assembly, and of not supporting features such as callbacks.

B. Threat Model

EverCrypt proves that all of its implementation code is functionally correct, meaning that it matches a high-level mathematical description of its intended effect. As intermediate lemmas, it also proves memory and type safety. With respect to security, EverCrypt relies on the techniques developed by HACL* [82] and ValeCrypt to prove the absence of common side channels. Specifically, using the techniques discussed in Section II-A, it proves that secrets never influence branches, memory accesses, or timing-dependent instructions (e.g., division), hence preventing leaks based on timing or cache behavior. This does not necessarily rule out more advanced side channel attacks based, e.g., on electromagnetic radiation or speculative execution.

C. Trusted Computing Base (TCB)

As with any verification project, we must trust the correctness of our specifications and of our verification tools.

Specifications must be trusted, since they mathematically encode the specific properties we wish to verify. For EverCrypt, these properties include functional specifications to define EverCrypt's cryptographic algorithms, like SHA, and security properties that define the absence of basic digital side channels via non-interference. We inherit the latter specs from prior work [25, 39, 82] and provide a formal description in §V. Specifications also encode our assumptions about the world external to our code. Specifically, for our assembly code, we inherit Vale's specification of x64 assembly semantics [39], which define how each assembly instruction affects a model of the machine state. Because Low⋆ emits C code, we trust our spec of interoperation between C code and assembly (§V-B).

Since our specifications are trusted, we take multiple precautions to make them *trustworthy*. First and foremost, we endeavor to keep them small and simple. We have a total of roughly 8 KLOC (§VIII-D), counting whitespace and comments, compared with over 40 KLOC lines of C and assembly code. Keeping specs small and simple facilitates our second precaution: manual spec review. One team member translates specs into F⋆ while others independently review the translation against the original informal spec (e.g., from an RFC). Finally, we ensure our specs are executable, so that we can extract them via F⋆'s OCaml backend; the generated code is quite inefficient but suffices to sanity check the specs on standard test vectors. This enables early detection of basic errors, such as typos or endianness issues.

On the tool side, our proofs rely on the soundness of our verification toolchain (F⋆ and Z3). To produce executable code, we rely on F⋆'s backend for extracting Low⋆ to C code [68], which can be compiled by using a verified compiler (e.g., CompCert [52]) or by trusting a faster, unverified C compiler. We also rely on an untrusted assembler and linker to produce our final executable. These trusted tools are comparable to those found in other verification efforts; e.g., implementations verified in Coq [72] trust Coq, the Coq extraction to OCaml, and the OCaml compiler and runtime. To provide higher assurance, multiple research results show that each element of these toolchains can themselves be verified [24, 48, 52, 60, 71]. Furthermore, several studies have confirmed that, despite the use of such complex verification toolchains, the empirical result is qualitatively better code compared with traditional software development practices [37, 38, 79]. These studies

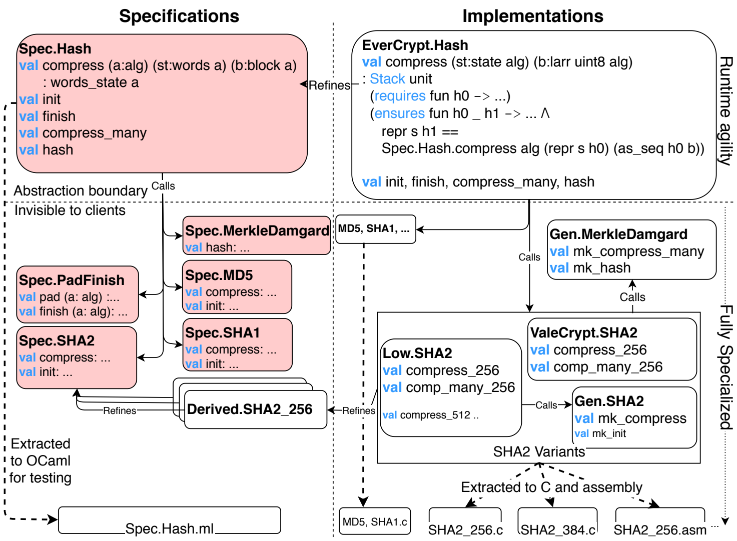

Figure 3. The modular structure of EverCrypt (illustrated on hashing algorithms)

found numerous defects and vulnerabilities in traditional software, but failed to find any in the verified software itself; the only defects found were in the specifications.

Finally, like any (verified) library, EverCrypt is subject to bugs in unverified host applications. For instance, an unverified application may misuse the API and violate a pre-condition by passing a key with an incorrect length for the chosen algorithm. For such unverified clients, we plan to add a defensive EverCrypt API, where each array pointer comes with an extra length argument. Some other classes of bugs in unverified clients cannot be trivially mitigated: for instance, if EverCrypt is used in a web browser that contains a buffer-overflow vulnerability, then an attacker may be able to read EverCrypt's secret keys.

III. Crafting an Agile, Abstract API for EverCrypt

A key contribution of EverCrypt is the design of its API, which provides abstract specifications suitable for use both in verified cryptographic applications (§VIII) and in unverified code. While agility matters for security and functionality, we also find that it is an important principle to apply throughout our code: beneath the EverCrypt API, we use agility extensively to build generic implementations with a high degree of code and proof sharing between variants of related algorithms.

Figure 3 outlines the overall structure of our API and implementations for hashing algorithms—similar structures are used for other classes of algorithms. At the top left (in red), we have trusted, pure specifications of hashing algorithms. Our specifications are optimized for clarity, not efficiency. Nevertheless, we compile them to OCaml and test them using standard test vectors. We discuss specifications further in §III-A. To the right of the figure, we have verified optimized implementations. The top-level interface is EverCrypt.Hash, which multiplexes new, efficient imperative implementations written in Low⋆ and Vale. Each of these implementations is typically proven correct against a low-level specification (e.g. Derived.SHA2_256) better suited to proofs of implementation correctness than the top-level Spec.Hash—these derived specifications are then separately proven to refine the top-level specifications. We discuss our top-level API in §III-B.

For reuse within our verified code, we identify several generic idioms. For instance, we share a generic Merkle-Damgård construction [29, 57] between all supported hash algorithms. Similarly, we obtain all the SHA2 variants from a generic template. The genericity saves verification effort at zero runtime cost—using F⋆'s support for partial evaluation, we extract fully specialized C and assembly implementations of our code (§IV).

A. Writing Algorithm Specifications for EverCrypt

Trusted specifications define what a given algorithm (e.g., SHA-256) should do; in cryptography, these definitions typically appear in a mixture of English, pseudocode, and math in an RFC or a peer-reviewed paper. Within EverCrypt, we port these specifications to pure mathematical functions within F⋆. This process is trusted, and hence we take steps to enhance its trustworthiness. We strive to keep the trusted specifications concise and declarative to facilitate human-level audits.

1) Taming Specification Explosion via Agility: We factor common structure shared by multiple specifications into "generic" functions parameterized by an algorithm parameter, and helper functions that branch on it to provide algorithm-specific details. This reduces the potential for errors, makes the underlying cryptographic constructions more evident, and provides a blueprint for efficient generic implementations (§IV)

For example, the type below enumerates the hashing algorithms EverCrypt currently supports:

```
type alg = MD5 | SHA1
         | SHA2_224 | SHA2_256 | SHA2_384 | SHA2_512
```

Although MD5 and SHA1 are known to be insecure, a practical provider must supply them for compatibility reasons. At the application level, cryptographic security theorems can be conditioned on the security of the algorithms used and, as such, would exclude MD5 or SHA1. Pragmatically, EverCrypt can also be configured to disable them, or even exclude their code at compile time.

All these algorithms use the Merkle-Damgård construction for hashing a bytestring by (1) slicing it into input blocks, with an encoding of its length in the final block; (2) calling a core, stateful compression function on each block; (3) extracting the hash from the final state. Further, the four members of the SHA2 family differ only on the lengths of their input blocks and resulting tags, and on the type and number of words in their intermediate state. Rather than write different specifications, we define a generic state type parameterized by the algorithm:

```
let word alg = match alg with
  | MD5 | SHA1 | SHA2_224 | SHA2_256 → UInt32.t
  | SHA2_384 | SHA2_512 → UInt64.t
let words alg = m:seq (word alg){length m = words_length alg}
let block alg = b:bytes {length b = block_length alg}
```

Depending on the algorithm, the type word alg selects 32-bit or 64-bit unsigned integer words; words alg defines sequences of such words of the appropriate length; and block alg defines sequences of bytes of the appropriate block length. With these types, we write a generic SHA2 compression function that updates a hash state (st) by hashing an input block (b). Note, this definition illustrates the benefits of programming within a dependently typed framework—we define a single function that operates on either 32-bit or 64-bit words, promoting code and proof reuse and reducing the volume of trusted specifications.

```
module Spec.SHA2
let compress (alg:sha2_alg) (st:words alg) (b:block alg)
  : words_state alg
  = let block_words = words_of_bytes alg 16 b in
    let st' = shuffle alg st block_words in
    seq_map2 (word_add_mod alg) st st'
```

This function first converts its input from bytes to words, forcing us to deal with endianness—being mathematical, rather than platform dependent, our specifications fix words to be little endian. The words are then shuffled with the old state to produce a new state, which is then combined with the old state via modular addition, all in an algorithm-parameterized manner (e.g., word_add_mod computes modulo 2^{32} for SHA2-224 and SHA2-256, and modulo 2^{64} for SHA2-384 and SHA2-512).

Beyond the SHA2 family of algorithms, we compose multiple levels of specification sharing. For instance, we write a single agile padding function (in Spec.PadFinish) for MD5, SHA1 and SHA2; a small helper function branches on the algorithm alg to encode the input length in little-endian or big-endian, depending on whether the algorithm is MD5.

2) Untrusted Specification Refinements: EverCrypt's trusted algorithm specifications are designed to be concise and easily auditable, but they are rarely actionable, in the sense of lending themselves well to an efficient implementation. Hence we often find it useful to develop untrusted specification refinements (e.g., Derived.SHA2_256) that provide more concrete details and are *proven* equivalent to the trusted specifications. These refinements introduce representation choices, precomputations, chunking of the operations in blocks to enable vectorized implementations, etc. In Curve25519, for example, an algorithm refinement may introduce a Montgomery ladder for efficiently computing scalar multiplications. In the case of hashes, instead of waiting for the entire message to be available and holding all the data in memory, a refined specification processes its input incrementally and is verified against the base algorithm. This process of relating specifications to implementations through a process of iterative refinement leads to well-structured, modular, and easier to maintain proofs .

B. EverCrypt's Top-level API

Verified programming is a balancing act: programs must be specified precisely, but revealing too many details of an implementation breaks modularity and makes it difficult to revise or extend the code without also breaking clients. A guiding principle for EverCrypt's API is to hide, through abstraction, as many specifics of the implementation as possible. Our choice of abstractions has been successful inasmuch as having established our verified API, we have extended its implementation with new algorithms and optimized implementations of existing algorithms without any change to the API.

We use abstraction in two flavors. *Specification* abstraction hides details of an algorithm's specification from its client, e.g., although EverCrypt.Hash.compress is proven to refine Spec.Hash.compress, only the type signature of the latter, not its definition, is revealed to clients. In addition, *representation* abstraction hides details of an implementation's data structures, e.g., the type used to store the hash's internal state is hidden.

Abstract specifications have a number of benefits. They ensure that clients do not rely on the details of a particular algorithm, and that their code will work for any present or future hash function that is based on a compression function. Abstract specifications also lend themselves to clean, agile specifications for cryptographic constructions (such as the Merkle-Damgård construction discussed above). Abstraction also allows us to provide a defensive API to unverified C code, helping to minimize basic API usage flaws. Finally, abstraction also simplifies reasoning, both formally and informally, to establish the correctness of client code. In practice, abstract specifications prune the proof context presented to the theorem prover and can significantly speed up client verification.

Our main, low-level, imperative API is designed around an algorithm-indexed, abstract type state alg. EverCrypt clients are generally expected to observe a usage protocol. For example, the hash API expects clients to allocate state, initialize it, then make repeated calls to compress, followed eventually by finalize. EverCrypt also provides a single-shot hash function in the API for convenience.

The interface of our low-level compress function is shown below, with some details elided.

```
module EverCrypt.Hash
val compress (s:state alg) (b:larr uint8 alg)
  : Stack unit
 (requires λh0 → inv s h0 ∧ b ∈ h0 ∧ fp s h0 `disjoint` loc b)
 (ensures λh0 _ h1 → inv s h1 ∧ modifies (fp s h0) h0 h1 ∧
     repr s h1 ==
     Spec.Hash.compress alg (repr s h0) (as_seq h0 b))
```

Clients of compress must pass in an abstract state handle s (indexed by an implicit algorithm descriptor alg), and a mutable array b:larr uint8 alg holding a block of bytes to be added to the hash state. As a precondition, they must prove inv s h0, the abstract invariant of the state. This invariant is established initially by the state allocation routine, and standard framing lemmas ensure the invariant still holds for subsequent API calls as long as any intervening heap updates are to disjoint regions of the heap. Separating allocation from initialization is a common low-level design decision, as it allows clients to reuse memory, an important optimization, e.g., for our Merkle tree library (§VII-B). In addition to the invariant, clients must prove that the block b is live; and that b does not overlap with the abstract footprint of s (the memory locations of the underlying hash state). The Stack unit annotation, as in Figure 1, states that compress is free of memory leaks and returns the uninformative unit value (). As a postcondition, compress preserves the abstract invariant; it guarantees that only the internal hash state is modified; and, most importantly, that the final value held in the hash state corresponds *exactly* to the words computed by the pure specification Spec.Hash.compress. It is this last part of the specification that captures functional correctness, and justifies the safety of multiplexing several implementations of the same algorithm behind the API, inasmuch as they are verified to return the same results, byte for byte.

State abstraction is reflected to C clients as well, by compiling the state type as a pointer to an incomplete struct [70]. Hence, after erasing all pre- and post-conditions, our sample low-level interface yields an idiomatic C function declaration in the extracted evercrypt_hash.h listed below.

```
struct state_s;
typedef struct state_s state;
void compress (state *s, uint8 *b)
```

Given an abstract, agile, functionally correct implementation of our 6 hash algorithms, we develop and verify the rest of our API for hashes in a generic manner. We first build support for incremental hashing (similar to compress, but taking variable-sized bytestrings as inputs), then an agile standard-based implementation of keyed-hash message authentication codes (HMAC) and finally, on top of HMAC, a verified implementation of key derivation functions (HKDF) [47].

The appendix lists the resulting C API and sample extracted code. We expect this API to be stable throughout future versions of EverCrypt. Thanks to agility, adding a new algorithm (e.g., a hash) boils down to extending an enumeration (e.g., the hash algorithm type) with a new case. This is a backward-compatible change that leaves function prototypes identical. Thanks to multiplexing, adding a new optimized implementation is purely an implementation matter that is dealt with automatically within the library, meaning once again that such a change is invisible to the client. Finally, thanks to abstract state and framing lemmas, EverCrypt can freely optimize its representation of state, leaving verified and unverified clients equally unscathed.

IV. Agile Implementations of the API with Zero-cost Generic Programming

While agility yields clean specifications and APIs, we now show how to program *implementations* in a generic manner, and still extract them to fully specialized code with zero runtime overhead. To ground the discussion, we continue with our running example of EverCrypt's hashing API, instantiating the representation of the abstract state handle (state a) and sketching an implementation of EverCrypt.Hash.compress, which supports runtime agility and multiplexing, by dispatching to implementations of specific algorithms.

A. *Implementing* EverCrypt.Hash

The abstract type state alg is defined in F* as a pointer to a a datatype holding algorithm-specific state representations, as shown below:

```
type state_s (a: alg) = match a with
| SHA2_256_s: p:hash_state SHA2_256 → state_s SHA2_256
| SHA2_384_s: p:hash_state SHA2_384 → state_s SHA2_384
| . . .
let state alg = pointer (state_s alg)
```

The state_s type is extracted to C as a tagged union, whose tag indicates the algorithm alg and whose value contains a pointer to the internal state of the corresponding algorithm. The union incurs no space penalty compared to, say, a single void*, and avoids the need for dangerous casts from void* to one of uint32_t* or uint64_t*. The tag allows an agile hash implementation to dynamically dispatch based on the algorithm, as shown below for compress:

```
let compress s blocks = match !s with
  | SHA2_256_s p → compress_multiplex_sha2_256 p blocks
  | SHA2_384_s p → compress_sha2_384 p blocks
  | . . .
```

In this code, since we only have one implementation of SHA2-384, we directly call into Low*. For SHA2-256, however, since we have multiple implementations in Low* and Vale, we dispatch to a local multiplexer that selects the right one (e.g., based on other runtime configurations in scope, including CPU identity, as described in §V).

B. *Partial Evaluation for Zero-cost Genericity*

Abstract specifications and implementations, while good for encapsulation, modularity, and code reuse, can compromise the efficiency of executable code. We want to ensure that past the agile EverCrypt.Hash nothing impedes the run-time performance of our code. To that end, we now show how to efficiently derive specialized Low* code, suitable for calling by EverCrypt.Hash, by partially evaluating our verified source *code*, reducing away several layers of abstraction before further compilation. The C code thus emitted is fully specialized and abstraction-free, and branching on algorithm descriptors only happens above the specialized code, where the API demands support for runtime configurability (e.g., only at the top-level of EverCrypt.Hash). As such, we retain the full generality of the agile, multiplexed API, while switching efficiently and only at a coarse granularity between fast, abstraction-free implementations (Figure 3).

Consider our running example: compress, the compression function for SHA-2. We managed to succinctly specify all variants of this function at once, using case-generic types like word alg to cover algorithms based on both 32- and 64-bit words. Indeed, operations on word alg like word_logand below, dispatch to operations on 32-bit or 64-bit integers depending on the specific variant of SHA-2 being specified.

```
let word_logand (alg:sha2_alg) (x y: word alg): word alg =
  | SHA2_224 | SHA2_256 → UInt32.logand x y
  | SHA2_384 | SHA2_512 → UInt64.logand x y
```

We wish to retain this concise style and, just like with specifications, write a *stateful* shared compress_sha2 *once*. This cannot, however, be done naively, and implementing bitwise-and within compress_sha2 using word_logand would be a performance disaster: every bitwise-and would also trigger a case analysis! Further, word alg would have to be compiled to a C union, also wasting space.

However, a bit of inlining and partial evaluation goes a long way. We program most of our *stateful* code in a case-generic manner. Just like in specifications, the stateful compression function is written once in a generic manner (in Gen.SHA2); we trigger code specialization at the top-level by defining all the concrete instances of our agile implementation, as follows:

```
module Low.SHA2
let compress_224 = Gen.SHA2.compress SHA2_224
let compress_256 = Gen.SHA2.compress SHA2_256
let compress_384 = Gen.SHA2.compress SHA2_384
let compress_512 = Gen.SHA2.compress SHA2_512
```

When extracting, say compress_256, F* will partially evaluate Gen.SHA2.compress on SHA2_256, eventually encountering word_logand SHA2_256 and reducing it to UInt32.logand. By the time all reduction steps have been performed, no agile code remains, and all functions and types that were parameterized over alg have disappeared, leaving specialized implementations for the types, operators and constants that are specific to SHA2-256 and can be compiled to efficient, idiomatic C code.

We take this style of partial evaluation one step further, and parameterize stateful code over algorithms *and* stateful functions. For instance, we program a generic, *higher-order* Merkle-Damgård hash construction [29, 57] instantiating it with specific compression functions, including multiple implementations of the same algorithm, e.g., implementations in Low* or Vale. Specifically, the compress_many function is parameterized by a compression function f, which it applies repeatedly to the input.

```
val mk_compress_many (a:hash_alg) (f:compress_st a)
  : compress_many_st a
```

We obtain several instances of it, for the same algorithm, by applying it to different implementations of the same compression function, letting F* specialize it as needed.

```
let compress_many_256_vale: compress_many_st SHA2_256 =
  mk_compress_many SHA2_256 compress_sha2_256_vale
let compress_many_256: compress_many_st SHA2_256 =
  mk_compress_many SHA2_256 compress_sha2_256
```

This higher-order pattern allows for a separation of concerns: the many-block compression function need not be aware of *how* to multiplex between Low* and Vale, or even of *how many* choices there might be. We extend this higher-order style to our entire hash API: for instance, mk_compress_last generates a compression function for the remainder of the input data, given an implementation of compress and pad; or, mk_hash generates a one-shot hash function. We then instantiate the entire set of functions, yielding two specialized APIs with no branching: one for Low* code, one for Vale code.

This technique is akin to C++ templates, in that both achieve zero-cost abstractions. F*, however, allows us to verify by construction that any specialization satisfies an instantiation of the generic specification, unlike C++ templates which perform textual expansion and repeated checks at each instantiation for typability in C++'s comparatively weak type system.

V. Safely Traversing Abstraction Boundaries between C and Assembly

Implementing cryptography in assembly enables the use of hardware-specific features, such as the SHA-EXT [43] and AES-NI [41] instruction sets. It also enables manual optimizations that general-purpose compilers might not detect. To achieve high-performance, EverCrypt therefore needs the ability to verifiably call Vale assembly routines from Low*. To support fine-grained interactions between C and assembly, we provide a verified interoperation framework to reconcile differences in their execution models (e.g., different memory models) while ensuring that specifications of verified Low* and Vale code match precisely.

A. Overview

In contrast to prior work that considers C programs interoperating with potentially malicious assembly language contexts [9], the setting in EverCrypt is simpler—C code verified in Low* interacts with assembly code verified in Vale. Given Vale's focus on verified cryptography, it has sufficed, so far, for Low* programs to call Vale while sharing mutable arrays of fixed-width types, for Vale to read and write those arrays, and to return word-sized results back to Low*. In the future, we plan to extend our support for interoperability to other architectures (beyond x64), while sharing richer, structured types between Low* and Vale, should the need arise. We summarize the main features of our interoperation model below.

- Verified Low* programs call into verified Vale procedures or inline assembly fragments.
- Control transfers from Vale back to Low* only via returns; there are no callbacks from Vale to Low*.
- The only observable effects of Vale in Low* are:
  - updates to memory, observed atomically as the Vale program returns (since there are no callbacks, intermediate memory states are not observable);
  - the value in the `rax` register in the final machine state of the Vale program as it returns to Low*; and,
  - digital side-channels due to the trace of instructions or memory addresses accessed by the Vale program.

As such, Vale procedures extend the semantics of Low* with an atomic computational step with effects on memory and a single word-sized result (but with variable execution time as a potential side channel). The goal of our interoperation framework is to safely lift the fine-grained Vale semantics to a Low* specification, so that Low* programs containing atomic Vale steps can be verified within Low*'s program logic.

B. Modeling Interoperation

Abstractly, a verified Vale procedure satisfies a Hoare triple $\{P\}\ c\ \{Q\}$, meaning that for all Vale machine states s_0 that satisfies P, it is safe to evaluate the Vale instructions c, producing a final state that satisfies Q; i.e., the following property is provable in F*: $\forall s_0.\ P\ s_0 \implies Q\ (\mathsf{eval}\ c\ s_0)$, where eval is a definitional interpreter for the semantics of Vale in F*. We make use of this definitional interpreter to lift a Vale Hoare triple to Low*, as shown in the sketch below:

```
let call_assembly c arg1 ... argn
  : Stack uint64 (requires lift_pre P) (ensures lift_post Q) =
  let h0 = get () in
  let s0 = initial_vale_state h0 arg1 ...argn in
  let s1 = eval c s0 in
  let rax, h1 = final_lowstar_state h0 s1 in
  put h1; rax
```

The Low* function call_assembly models calling the Vale code c with arguments $\mathsf{arg}_1 \ldots \mathsf{arg}_n$. Operationally, the call is modeled as follows: at line 3, we retrieve the initial Low* heap h_0; at line 4, we construct the initial Vale state s_0 from h_0 and all the arguments; at line 5, we run the Vale definitional interpreter to obtain the final Vale state s_1; at line 6, we translate this s_1 back to a Low* heap h_1 and return value rax; finally, we update the Low* state atomically with h_1 and return rax. Most importantly, at line 2, we prove that whenever the Vale code satisfies $\{P\}\ c\ \{Q\}$, our operational model of Low*/Vale interoperation is soundly specified by the Low* computation type: Stack uint64 (requires lift_pre P) (ensures lift_post Q). Concretely, each call to assembly from Low* is extracted by our compiler as a C extern signature, whereas the assembly code itself is printed by Vale's simple assembly printer.

Trust assumptions: The definition of the call_assembly wrapper is at the core of our trusted assumptions: it defines the semantics of a call from Low* into Vale. Some salient parts of this trusted definition include the translation of the Low* memory into a Vale memory and back, and the parameter-passing convention. Of course, the Vale evaluator is part of the trusted assumptions about Vale itself, regardless of interoperation with Low*. Notably, the specification of call_assembly (line 2) is not trusted itself, since it is merely an abstraction that is proven to be satisfied by the operational definition. Additionally, we rely on the C compiler, assembler and linker to concretely implement the call as modeled by our formal development.

Defining the call_assembly wrapper posed several challenges, described in the next sub-section. For a sense of the scale, our full call_assembly wrapper is 17 lines long, and is supported by a further 1,595 lines of F* specification and modeling code; 2,194 lines of untrusted F* proofs establishing various other lemmas for convenient use of call_assembly wrapper from Low* (including functions analogous to lift_pre and lift_post); and 11,558 lines of untrusted proofs for 31 specific calls from Low* into Vale.

Our work is based closely on prior work relating Low* and Vale by Fromherz et al. [39]. However, unlike Fromherz et al., who rely on an ad hoc external tool to relate Low* and Vale, our new approach is generic and modular, and done entirely within F* without the need for external tools. This has made it easy for us to extend it with new features, e.g., to add support for inline assembly. Further, by abstracting interoperation into a generically programmed and proven layer, we significantly reduce the verification burden for each Vale procedure that uses the interoperation layer—in effect, we relate the abstractions of Vale and Low* once and for all, instead of repeatedly proving a relation between the abstractions each time we call from Low* into Vale code, allowing our work to scale to the 31 (and counting) procedures mentioned previously, including the largest single Vale procedure that we have lifted to Low*, which takes 17 arguments and is specified in approximately 200 lines of F* code.

C. Relating Low* and Vale

There are four main elements of our model, which we describe in more detail below:

- Relating memory models: Constructing an initial Vale state from the Low* heap and interpreting the final Vale state as a Low* heap requires relating the Low* structured memory model to Vale's "flat array of bytes" memory model.
- Modeling the calling convention: Constructing the initial Vale state also requires modeling the calling convention to place arguments in specific registers.
- Lifting specifications: The lift_pre and lift_post functions interpret Vale pre- and post-conditions as Low* pre- and post-conditions.
- Side-channel analysis: We use a taint analysis to show that execution traces of Vale programs are secret independent, and relate this to Low*'s notion of secret independence.

Relating memory models: The memory models used by Low* and Vale differ significantly. The Low* memory model stores values of structured types: the types include machine integers of various widths (8–128 bits) and signedness; and arrays of structured values (as in C, pointers are just singleton arrays). In contrast, Vale treats memory as just a flat array of bytes. At each call, we may pass several pointers from Low* to Vale; to do so, we assume the existence of a physical address map that assigns a Vale address for each shared pointer. Given this address map, we can build an explicit correspondence from the fragment of Low* memory containing the shared pointers to Vale's flat memory—this involves making explicit the layout of each structured type (e.g., endianness of machine integers), and the layout in contiguous memory of the elements of an array. For EverCrypt's specific goal of accelerating the inner loops of crypto routines, our current focus on just these types for interoperation has sufficed.

Modeling the calling convention: From Vale's perspective, arguments are received in specific registers and spilled on the stack if needed; in contrast, in Low* as in C, arguments are just named. As we construct the initial vale state, we translate between these (platform-specific) conventions, e.g., on an x64 machine running Linux, the first argument of a function must be passed in the rdi register, and the second in rsi. Further, the wrapper requires that *callee-saved registers* (e.g., r15 for Windows on x64) have the same value when entering and exiting the Vale procedure. Aside from x64 standard calls on Windows and Linux, we also support custom calling conventions in support of inline assembly (subject to restrictions, e.g., the stack register rsp cannot be used to pass arguments, and distinct arguments must be passed in distinct registers). One of the subtleties of modeling the calling convention is to define it once, while accounting for all arities (notice the $\mathsf{arg}_1 \ldots \mathsf{arg}_n$, in the sketch of call_assembly of the previous section)—F*'s support for dependently typed arity-generic programming [12] makes this possible.

Lifting specifications generically from Vale to Low*: To preserve the verification guarantees of a Vale program when called from Low*, the Vale preconditions must be provable in the initial Low* state and arguments in scope. Dually, the Vale postconditions must suffice to continue the proof on the Low* side. A key feature of our interoperation layer is to lift Vale specifications along the mapping between Vale and

Low⋆ states, e.g., lift_pre and lift_post reinterpret soundly and generically (i.e., once and for all) pre- and postconditions on Vale's flat memory model and register contents in terms of Low⋆'s structured memory and named arguments in scope. As explained in §V-B, relating specifications between Vale and Low⋆ is untrusted—thankfully so, since this is also perhaps the most complex part of our interoperation framework, for two reasons. First, Vale and Low⋆ use subtly different core concepts (each optimized for their particular setting) including different types for integers, and different predicates for memory footprints, disjointness, and liveness of memory locations. Hence, relating their specifications involves working deep within the core of the semantic models of the two languages and proving compatibility properties among these different notions. This is only possible because both languages are emdedded within the same underlying logic, i.e., F⋆. Second, because we model the calling convention generically for all arities, the relations among Vale and Low⋆ core concepts must also be generic, since these state properties of the variable number and types of arguments in scope. However, the payoff for these technical proofs is that they are done once and for all, and their development cost is easily amortized by the relative convenience of instantiating the framework at a specific arity for each call from Low⋆ to Vale.

Side-channel analysis: Whereas the relation between Hoare triples between Low⋆ and Vale is fully mechanized within F⋆'s logic (as illustrated by the call_assembly sketch), we rely on a meta-theorem to connect the side-channel guarantees provided by Low⋆ and Vale. Specifically, whereas Low⋆'s guarantees rely on enforcing constant-time properties using a syntactic analysis based on abstract types, Vale's constant-time guarantees arise from a certified taint analysis that is executed on the Vale instructions [25, 39]. We extended and improved Vale's existing taint analysis, allowing it to more generically process Vale instructions, but we reuse without change the *Secret Independence for Hybrid Low⋆/Vale Programs* theorem, proven by Fromherz et al. [39, Theorem 4.2]—we reproduce the theorem and explain it in Appendix X. This metatheorem is based on a notion of combined execution traces that include an abstraction of both Low⋆ and Vale instruction sequences, and it proves that when a well-typed Low⋆ program interoperates with a Vale program proven constant-time by the Vale taint analysis, the combined traces produced by these programs are independent of their secret inputs.

D. CPU-ID Detection

Aside from generic support for relating memories, calling conventions, correctness and security specifications, our interoperation layer also provides specialized support for CPU-ID detection—a crucial feature for a cryptographic provider with platform-specific optimizations. For instance, the Vale implementation of mul1 expects the CPU to support the ADX and BMI2 extensions. Otherwise, executing this code would result in errors due to illegal (unsupported) instructions. To safely call this code, we need to propagate this precondition to Low⋆. Functions calling this procedure will then need to prove that these CPU extensions are supported. Conditions such as the presence of CPU extensions are stated identically in Low⋆ and Vale, without any reinterpretation—this makes their lifting trivial, but (rightfully) forces Low⋆ code calling Vale procedures to establish preconditions about which CPU extensions are supported. To discharge these proof obligations, EverCrypt calls into other Vale routines we developed that make a series of CPUID calls to provably determine the features supported by the current CPU. EverCrypt caches the results to avoid costly processor-pipeline flushes.

Guarding CPU-ID detection: A further complication arises because the CPUID instruction is itself only supported on x86 and x64 platforms, but not, say, on ARM. Hence, we add another layer of flags representing static compiler-level platform information (e.g., TargetConfig.x64). Checks for these flags are compiled as a C #ifdef. Hence, the emitted code for auto-detecting CPU features looks as follows:

```
#if TARGETCONFIG_X64
  if (check_aesni () != 0U) cpu_has_aesni[0U] = true;
#endif
```

This ensures that no link-time error occurs when compiling EverCrypt for platforms which do not support the CPUID instruction. The connection of the TARGETCONFIG_ macros to standard compiler definitions, e.g.,

```
#if defined(__x86_64__) || defined(_M_X64)
#define TARGETCONFIG_X64 1
#elif ...
```

are hand-written, and as such, must be carefully audited.

VI. Achieving Best-In-Class Run-Time Performance

Cryptographic performance is often a bottleneck for security-sensitive applications (e.g., for TLS or disk encryption). Given a choice between a complex high-performance crypto library, and a simple, potentially more secure one, historically much of the world has opted for better performance.

With EverCrypt, we aim to obviate the need for such a choice; thanks to verification, we can offer developers best-in-class performance *and* provable security guarantees. Below, we highlight two examples of how we achieve this.

A. AES-GCM

AES-GCM [62] may be the world's most widely used cryptographic algorithm; a recent study from Mozilla shows it in use for 91% of secure web traffic [59]. AES-GCM provides authenticated encryption with additional data (AEAD), meaning that it protects the secrecy and integrity of plaintext messages, and it can provide integrity for additional data as well (e.g., a packet header). This makes it a popular choice for protecting bulk data, e.g., in protocols like TLS and QUIC, which also places it on the critical path for many applications.

Recognizing the importance of AES-GCM, Intel introduced seven dedicated instructions to improve its performance and side-channel resistance [41]. Even given these instructions, considerable work remains to construct a complete AES-GCM implementation. For example, the GCM-support instruction (PCLMULQDQ) performs a carryless multiply of its arguments, but the GCM algorithm operates over the Galois field $GF(2^{128})$, and hence multiplication in the field requires both a carryless multiply *and* a further polynomial reduction. As further evidence of the complexity of using these instructions, in 2013, a performance enhancement was committed to OpenSSL to speed-up their implementation of AES-GCM. However, despite passing the standard test suite, the enhancement introduced a subtle bug in the GCM calculation that would allow an attacker to produce arbitrary message forgeries [42]. Fortunately, the bug was discovered before it trickled into an official release, but with EverCrypt, we aim to prove that no such bugs exist. While previous work verified their own implementation of AES-GCM [39], the result was 6× slower than OpenSSL's.

Hence, rather than invent our own optimizations, we port OpenSSL's implementation (written in over 1100 lines of Perl and C preprocessor scripts) to Vale. This created numerous challenges beyond tricky reasoning about $GF(2^{128})$ and the sheer scale of the code (the core loop body has over 250 assembly instructions). For example, the implementation makes heavy use of SIMD instructions operating over 128-bit XMM registers, which require reasoning about all of the parallel operations happening within the registers. At an algorithmic level, the simplest way to implement AES-GCM would be to do a first pass to encrypt the plaintext (in AES-CTR mode) and then make a second pass to compute the MAC over the ciphertext and authenticated data. To achieve better performance, AES-GCM is designed such that both operations can be computed in a single pass, and of course OpenSSL does so. Further, it deeply interleaves the instructions for encryption and authentication, and it processes six 128-bit blocks in parallel to make maximum use of the SIMD registers. During encryption, it carefully runs the AES-CTR process 12 blocks ahead of the authentication calculations to ensure the processor pipeline remains fully saturated. The OpenSSL GCM computation also rearranges the carryless multiplies and reductions to improve parallelization, precomputing six carryless multiply powers during initialization, delaying reduction steps, and cleverly integrating bitwise operations into the reduction calculations.

To cope with this complexity, we specified the effects of many of the XMM instructions in terms of "opaque" functions, which hide the details of the effects from F*, and hence simplify reasoning about the basic operations of the code itself. We then reveal the effect details only within separately written lemmas that reason about, say, the resulting computations over $GF(2^{128})$. To simplify the task of deciphering OpenSSL's code and reconstructing the invariants the original programmer had in mind, we initially ported the encryption and the authentication operations separately, proving that each accomplished its goals individually. We then manually merged the implementations and proofs into single implementation and proof. In the future, we hope to develop techniques to automate such merges.

B. *Curve25519*

The Curve25519 elliptic curve [18], standardized as IETF RFC7748 [50], is quickly emerging as the default curve for modern cryptographic applications. It is the only elliptic curve supported by protocols like Signal [6] and Wireguard [32], and is one of two curves commonly used with Transport Layer Security (TLS) and Secure Shell (SSH). Curve25519 was designed to be fast, and many high-performance implementations optimized for different platforms have been published [18, 27, 35, 63] and added to mainstream crypto libraries.

a) Implementing Curve25519: Curve25519 can be implemented in about 500 lines of C. About half of this code consists of a customized bignum library that implements modular arithmetic over the field of integers modulo the prime $p_{25519} = 2^{255} - 19$. The most performance-critical functions implement multiplication and squaring over this field. Since each field element has up to 255 bits, it can be stored in 4 64-bit machine words, encoding a polynomial of the form:

$$e3 * 2^{192} + e2 * 2^{128} + e1 * 2^{64} + e0$$

where each coefficient is less than 2^{64}. Multiplying (or squaring) field elements amounts to textbook multiplication with a 64-bit radix: whenever a coefficient in an intermediate polynomial goes beyond 64-bits, we need to carry over the extra bits to the next-higher coefficient. To avoid a timing side-channel, we must assume that every 64-bit addition may lead to a carry and propagate the (potentially zero) carry bit regardless.

Propagating these carries can be quite expensive, so a standard optimization is to delay carries by using an *unpacked* representation, with a field element stored in 5 64-bit machine words, each holding 51 bits, yielding a radix-51 polynomial:

$$e4 * 2^{204} + e3 * 2^{153} + e2 * 2^{102} + e1 * 2^{51} + e0$$

While this representation leads to 9 more 64x64 multiplications, since each product now has only 102 bits, it has lots of room to hold extra carry bits without propagating them until the final modular reduction.

b) Correctness Bugs and Formal Verification: Even with these delayed carries, carry propagation continues to be a performance bottleneck for modular multiplication, and high-performance implementations of Curve25519 implement many low-level optimizations, such as interleaving carry chains, and skipping some carry steps if the developer believes that a given coefficient is below a threshold. Such delicate optimizations have often lead to functional correctness bugs [20, 49], both in popular C implementations like Donna-64 and in high-performance assembly like amd64-64-24k. These bugs are particularly hard to find by testing or auditing, since they only occur in low-probability corner cases deep inside arithmetic. Nevertheless, such bugs may be exploitable by a malicious adversary once they are discovered. Hence elliptic curves have been a popular target for verification efforts (§II).

c) Faster Curve25519 with Intel ADX: In 2017, Oliveira et al. [63] demonstrated a significantly faster implementation of Curve25519 on Intel platforms that support Multi-Precision Add-Carry Instruction Extensions, also called Intel ADX. Unlike other fast Curve25519 implementations, Oliveira et al. use a radix-64 representation and instead optimize the carry propagation code by carefully managing Intel ADX's second carry flag. The resulting performance improvement is substantial—at least 20% faster than prior implementations.

Oliveira et al. wrote their implementation mostly in assembly. A year after its publication, when testing and comparing this code against formally verified implementations taken from HACL* [82] and Fiat-Crypto [36], Donenfeld and others found several critical correctness bugs [34]. These bugs were fixed [30], with a minor loss of performance, but they raised concerns as to whether this Curve25519 implementation, with the best published performance, is trustworthy enough for deployment in mainstream applications.

d) Fast Verified Curve25519 in EverCrypt: In EverCrypt, we include two new versions of Curve25519. The first is written in Low* and generates portable C code that uses a radix-51 representation. The second relies on verified Vale assembly for low-level field arithmetic that uses a radix-64 representation, inspired by Oliveira et al.'s work.

Notably, we carefully factor out the *generic* platform-independent Curve25519 code, including field inversion, curve operations, key encodings, etc., so that this code can be shared between our two implementations. In other words, we split our Curve25519 implementation into two logical parts: **(1)** the low-level field arithmetic, implemented both in Vale and in Low*, but verified against the same mathematical specification in F*, and **(2)** a high-level curve implementation in Low* that can use either of the two low-level field-arithmetic implementations.

We implement and verify optimizations at both levels. In Vale, we implement modular addition, multiplication, and squaring using the double carry flags of Intel ADX. We also implement functions that can multiply and square two elements at the same time. In Low*, we optimize our ladder implementation to reduce the number of conditional swaps, and to skip point addition in cases where the bits of the secret key are fixed. With the aid of these optimizations, we are able to produce a mixed assembly-C implementation of Curve25519 that slightly outperforms that of Oliveira et al. but with formal proofs of memory safety, functional correctness, and secret independence for the assembly code, C code, and the glue code in between.

Our implementation is the fastest verified implementation of Curve25519 to date, and it is comparable to the fastest published benchmarks for any implementation (§VIII-B).

VII. Secure programming above EverCrypt

As discussed in Section III, one of EverCrypt's contributions is an agile API designed to support both verified and unverified clients, while still providing best-in-class performance. In particular, verified clients should be able to code and reason abstractly against agile functionalities and their multiple implementations. To support this claim, we present two representative examples of verified applications. First, we present a memory-safe, functionally correct, constant-time implementation of QUIC's transport key derivation and packet encryption, parameterized by the choice of a TLS ciphersuite. Second, we present the design and implementation of a Merkle-tree library, parameterized by a hash algorithm, and we prove its security by reduction to collision-resistance on the hash function: given any forgery (that is, any evidence of membership on an element that has not been inserted in the tree), we efficiently extract two different bytestrings that yield the same hash.

A. QUIC Transport Cryptography

QUIC [45] is a new secure transport protocol originally designed by Google and now undergoing standardization at the IETF. One can think of QUIC as an alternative to the TLS record layer (which fragments and encrypts application data into TCP packets) built on top of UDP. Therefore, it is similar to DTLS (the UDP variant of TLS), but offers many new features (such as explicit authenticated acknowledgements, multiple application data streams, advanced flow control, etc.).

One such feature is packet header protection, which aims to enhance user privacy by encrypting the sequence numbers of QUIC packets, thereby restricting the ability of an observer to correlate the use of the same QUIC session on multiple networks. QUIC header protection uses an *ad hoc* cryptographic construction that combines an authenticated encryption scheme with a raw block or stream cipher to encrypt some packet headers. As pointed out by the specification [54, Section 9.4], if the two schemes are not composed correctly and in constant time, the sequence number can be inferred from timing side-channels. Our implementation offers the following functions:

- *Packet encryption and decryption*: We write a complete formal specification of packet encryption and decryption, which includes the formatting and parsing of short and long headers, and the encryption and decryption of protected fields (some flags, and the packet number). We prove that the functional specification of packet encryption is correct, that is, that a packet produced by the encryption function correctly decrypts to the same input arguments (with a caveat in the proof: for short headers, the recipient must know in advance the size of the connection identifier). We implement this specification on top of EverCrypt and prove that the implementation is memory safe and correct. Additionally, we prove that decryption fails in constant time if the packet has been tampered with in any way, by ensuring that the bytes of the encrypted packet number are treated abstractly (following the approach of HACL*).
- *Key derivation*: we write a formal specification of the key derivation operations (for payload AEAD encryption, static IV, header encryption, and re-keying), using the EverCrypt interface for HKDF. We write an implementation of QUIC key derivation that is proved memory safe and correct with respect to this specification.

We validate our specification by checking interoperability of our code integrated into two open-source QUIC libraries. We measure performance in Section VIII-C.

B. Merkle Trees

A Merkle tree [56] is a cryptographic construction for authenticating large data structures by compressing them into a single hash tag: the tree root. Our library supports binary Merkle trees with fast incremental updates. The leaves of the tree are a sequence of (hashes of) transactions. The root is computed by recursively hashing pairs of nodes. Compared with a plain hash of the sequence of transactions, this recursive construction enables clients to verify the presence of any transaction in the tree by recomputing the root along its path, given a logarithmic number of intermediate hashes as auxiliary evidence. Figure 11 shows the structure of our incremental Merkle tree.

a) Using Merkle Trees to Authenticate a Log of Transactions: Our library is used to authenticate the secure ledger of transactions for an industrial permissioned blockchain [69]. Its API provides three main functions: i) *extend* the tree with a transaction at the next index; ii) compute the current root of the tree; iii) given an index of a transaction in the tree, return a sequence of intermediate hashes (the Merkle *path*) enabling the recomputation of its root.

Our application provides high throughput for transactions and their resulting Merkle paths. To this end, it cryptographically signs the root only from time to time. Signing is significantly slower than hashing, so each signature computation is amortized over the evidence for thousands of transactions.

We designed, implemented, and verified our library on top of the agile EverCrypt API for hash algorithms. We also developed and verified a generic library of resizable vectors for caching its intermediate hashes. The Merkle Tree library is parameterized by the choice of a hash function. In all our experiments, we instantiate it to the core compression function of SHA2-256 (compressing 64 bytes into 32 bytes).

b) An Optimized, Incremental Implementation: Our implementation computes all intermediate hashes on-demand and caches them so that we can return paths for any recent transactions via table lookups. An auxiliary function flushes older intermediate hashes to reclaim memory, while retaining the ability to look up paths for all recent indices.

Because all hashes are cached and there are (roughly) as many intermediate nodes as there are leaves, each transaction requires on average just one call to the compression function. Still, as evidence of EverCrypt's clean API design (and our efforts to optimize the Merkle tree library), performance is dominated by these calls, which account for 95% of the total runtime for our benchmarks (Section VIII-C).

c) Functional Correctness: In addition to runtime safety, we verified the correctness of our incremental implementation against a simple high-level functional specification. The proof includes auxiliary lemmas on index computations, and a refinement step to account for the implicit padding of sequences of transactions up to the next power of 2.

We separately proved high-level properties of our specification, showing for instance that any root recomputation from (our specification of) Merkle paths yields back the root computed from the whole tree. We also verified its security, as follows.

d) Cryptographic Security: Merkle trees are meant to provide strong integrity protection: at any tree size, it should be computationally hard to exhibit two different sequences of transactions that yield the same root. This property clearly depends on the concrete strength of the hash function. Even with a strong hash function, this property should not be taken for granted for trees; for instance, it took three years to uncover a design flaw in the Merkle tree used by Bitcoin [61, 77].

Based on our specification, we provide a constructive proof in F* of the following security property: we program and verify a pure, total, linear function that takes a collision between two trees and returns a collision between two inputs of the hash function. We also show that, unsurprisingly, a collision between the core compression function immediately yields a collision on the SHA2-256 standards (by completing the hash computation with the length block). Together, this shows that the security of our construction perfectly reduces to the collision-resistance assumption on 64-byte inputs of the SHA2-256 standard.

Algorithm	**C version**	**Targeted ASM version**
AEAD		
AES-GCM		AES-NI + PCLMULQDQ + AVX
Chacha-Poly	yes	
High-level APIs		
Box	yes	
SecretBox	yes	
Hashes		
MD5	yes	
SHA1	yes	
SHA2	yes	SHA-EXT (for SHA2-224+SHA2-256)
SHA3	yes	
Blake2	yes	
MACS		
HMAC	yes	agile over hash
Poly1305	yes	X64
Key Derivation		
HKDF	yes	agile over hash
ECC		
Curve25519	yes	BMI2 + ADX
Ed25519	yes	
P-256	yes	
Ciphers		
ChaCha20	yes	
AES128, 256		AES NI + AVX
AES-CTR		AES NI + AVX

Figure 4. ***Supported algorithms and systems.*** *All are newly developed for EverCrypt, with the exception of Box, SecretBox, Ed25519, and Poly1305.*

VIII. Evaluation

In this section, we evaluate EverCrypt's success in achieving the criteria we set out in Section I for a cryptographic provider. Specifically, we evaluate EverCrypt's comprehensiveness, support for agility & multiplexing, performance, and usability by client applications. We also report on the effort involved in developing EverCrypt itself. Unless specified otherwise, for all performance graphs, lower is better.

A. *EverCrypt Features*

With EverCrypt, we aim to provide both cross-platform support and optimized implementations for specific platforms. Figure 4 summarizes our progress in this direction. All implementations are new to EverCrypt, except the targeted Poly1305 implementation, which we obtained from ValeCrypt [39], and the implementations of Ed25519, Box, and SecretBox from HACL* [82]; the latter two are simple, secure-by-default APIs for performing public-key and secret key encryption, respectively. As the table highlights, EverCrypt provides a variety of functionalities, including hashing, key derivation, cipher modes, message authentication, authenticated encryption with additional data (AEAD), signatures (Ed25519 [19, 46]), and elliptic curve operations. In most cases, EverCrypt provides both a generic C implementation for cross-platform purposes, as well as an optimized implementation for an x64-based target. EverCrypt automatically detects whether to employ the latter,

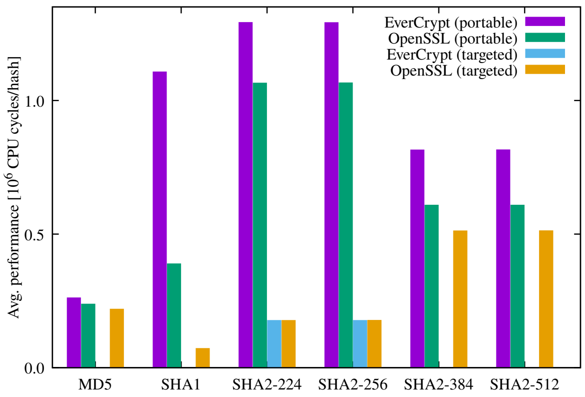

Figure 5. ***Avg. CPU cycles to compute a hash of 64 KB of random data.***

and it offers agile interfaces for AEAD, hashing, HMAC, and HKDF. HMAC and HKDF build on the agile hash interface, and hence inherit targeted implementations on supported platforms. EverCrypt does not yet support agility over elliptic curves, nor does it yet support older asymmetric algorithms like RSA. We have also, thus far, focused on optimized implementations for x64, but prior work with Vale [25] demonstrates that it can just as easily target other platforms, and hence we plan to target ARM in the future.

With regards to the comprehensiveness of EverCrypt's API, the most natural (unverified) comparison is with `libsodium` [7], which also aims to offer a clean API for modern cryptographic algorithms. The functionality exposed by each is quite comparable, with a few exceptions. EverCrypt does not yet offer an API for obtaining random data (which would require axiomatizing calls to the relevant OS-dependent APIs or hardware instructions like `RdRand`) or for securely deleting data. It also currently lacks a dedicated password-hashing API that uses a memory-hard hash function like Argon2 [23].

B. EverCrypt Run-Time Performance

EverCrypt aims to demonstrate that verification need not mean sacrificing performance. Below, we compare EverCrypt's performance against OpenSSL's implementations, which prior work measured as meeting or exceeding that of other popular open-source crypto providers [25]. We also compare with representative verified implementations [11, 14, 36, 39].

In our results, each data point represents an average of 1000 trials; error bars are omitted as the tiny variance makes them indistinguishable. All measurements are collected with hyper-threading and dynamic-processor scaling (e.g., Turbo Boost) disabled. We collect measurements on different platforms, since no single CPU supports all of our various targeted CPU features.

In Figure 5, we report on the performance of our targeted hash implementations when available (i.e., for SHA2-224 and SHA2-256), and our portable implementation otherwise, comparing with OpenSSL's corresponding implementations. Appel verified [14] an older version of the portable OpenSSL implementation, which runs about 10% slower than the latest version reported here. Figure 6 provides further detail on SHA2-256. We collect the measurements on a 1.5 GHz Intel Celeron J3455 (which supports SHA-EXT [43]) with 4 GB of RAM.

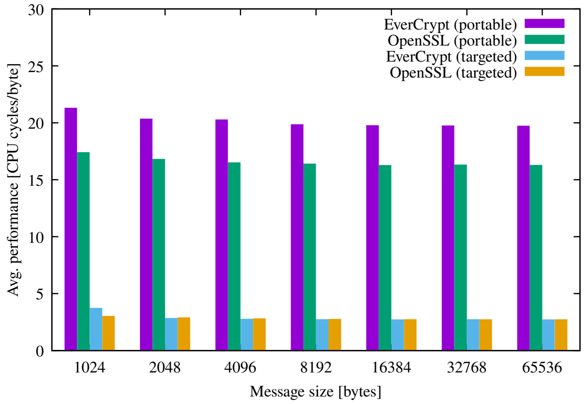

Figure 6. ***Avg. CPU cycles/byte to hash (SHA2-256) variable random data.***

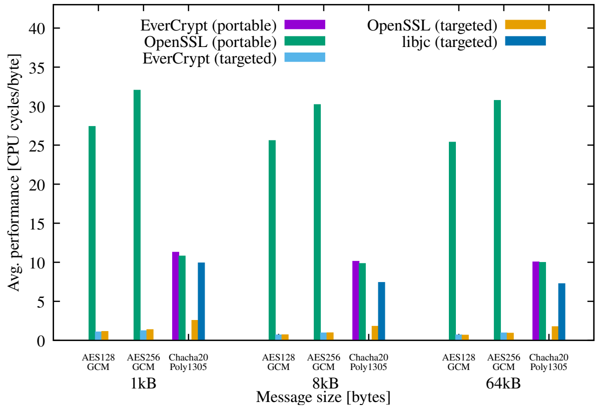

Figure 7. ***Cycles/byte to encrypt blocks of random data using AEAD.***

The results demonstrate the value of optimizing for particular platforms, as hardware support increases our performance for SHA2-224 and SHA2-256 by 7×, matching that of OpenSSL's best implementation. EverCrypt's portable performance generally tracks OpenSSL's, indicating a respectable fallback position for algorithms and platforms we have not yet targeted.

Similarly, Figures 7 and 8 report on the performance of our AEAD algorithms, with the latter focusing on targeted implementations. We also compare against `libjc` [11], a recently released library. It includes verified, targeted implementations of Poly1305 and ChaCha20. It does not yet include a verified ChaCha-Poly implementation [53], but we combined the two primitives in an unverified implementation. We measure on a 3.6GHz Intel Core i9-9900K with 64 GB of RAM.

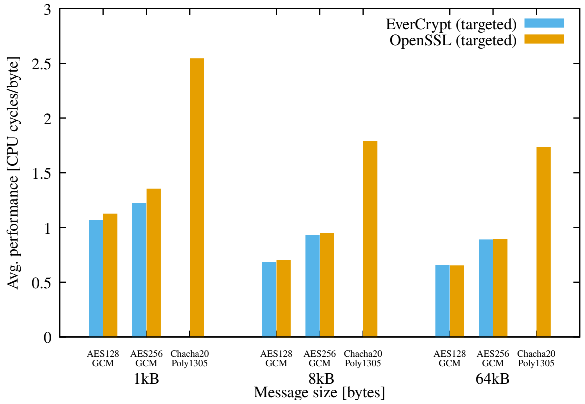

Figure 8. ***Cycles/byte to encrypt blocks of random data with targeted AEAD.***

Implementation	Radix	Language	CPU cy.
`donna64` [2]	51	64-bit C	159634
`fiat-crypto` [36]	51	64-bit C	145248
`amd64-64` [26]	51	Intel x86_64 asm	143302
`sandy2x` [27]	25.5	Intel AVX asm	135660
EverCrypt portable (*this paper*)	51	64-bit C	135636
`openssl*` [5]	64	Intel ADX asm	118604
Oliveira et al. [63]	64	Intel ADX asm	115122
EverCrypt targeted (*this paper*)	64	64-bit C + Intel ADX asm	113614

Figure 9. ***Performance comparison between Curve25519 Implementations.***

For cross-platform performance, we see that EverCrypt with ChaCha-Poly matches OpenSSL's equivalent, and both surpass OpenSSL's portable AES-GCM implementation. Targeting, however, boosts both EverCrypt and OpenSSL's AES-GCM implementations beyond that of even the targeted version of ChaCha-Poly. Note that EverCrypt's targeted performance meets or exceeds that of OpenSSL and achieves speeds of less than one cycle/byte for larger messages. This puts EverCrypt more than 6× ahead of the performance of the previous best verified implementation of AES-GCM [39].

Meanwhile, the performance of `libjc`'s targeted ChaCha-Poly slightly beats that of OpenSSL and EverCrypt's portable implementations, but it is $\sim$ 4× slower than OpenSSL's targeted ChaCha-Poly and $\sim$9-11× slower than EverCrypt's targeted AES128-GCM. We attribute this to the fact that the latter two each jointly optimize encryption and authentication together, whereas `libjc` optimizes the two primitives separately.

Finally, Figure 9 measures the performance of our implementations of Curve25519 against that of other implementations, including OpenSSL, Erbsen et al. [36] (one of the fastest verified implementations), and Oliveira et al. [63] (one of the fastest unverified implementations). All measurements are collected on an Intel Kaby Lake i7-7560 using a Linux kernel crypto benchmarking suite [33]. OpenSSL cannot be called from the kernel and was benchmarked using the same script, but in user space. All code was compiled with GCC 7.3 with flags `-O3 -march=native -mtune=native`.

Our results show that, with optimizations from Section VI-B, EverCrypt's combined Low* +Vale implementation narrowly exceeds that of Oliveira et al. by about 1%, which in turn exceeds that of OpenSSL by 3%. We also exceed the previous best verified implementation from Erbsen et al. by 22%. To the best of our knowledge, this makes EverCrypt's implementation of Curve25519 the fastest verified or unverified implementation on record. Notably, our optimizations also improve our portable C implementation, which is now the second fastest verified implementation, beating Erbsen et al.'s implementation by 6.6%.

C. EverCrypt Applications

Components	**LOC**
Cryptographic algorithm specs	3782
Vale interop specs	1595
Vale hardware specs	3269
Low* algorithms	25097
Low* support libraries	9943
Vale algorithms (written in Vale)	24574
Vale interop wrappers	13836
Vale proof libraries	23819
EverCrypt	5472
EverCrypt tests	4131
Merkle Tree	6510
QUIC transport cryptography	2282
Vale algorithms (F* code generated from Vale files)	72039
Total (hand-written F* and Vale)	124310
Compiled code (.c files)	25052
Compiled code (.h files)	4082
Compiled code (ASM files)	14740

Figure 10. ***System Line Counts.***

a) QUIC: To benchmark our verified implementation of QUIC packet encryption, we integrate our library with two open-source, OpenSSL-based implementations of QUIC: picoquic [3] (which uses picotls for the TLS 1.3 handshake), and ngtcp2 [4]. We measure the throughput of each library by downloading a 1GB random file over a local network interface, using each library's respective HTTP3-over-QUIC test application, acting both as a server and as a client. Our implementation exercises our Vale x64 implementation of AES-GCM (and AES-CTR, for header protection) and our Low* implementation of ChaCha20-Poly1305 (and ChaCha20), on a Xeon W-2155 CPU with 128GB of RAM running Windows 10.

QUIC Library	AES128-GCM	AES256-GCM	C20-P1305
ngtcp2 (OpenSSL)	41.0 MB/s	40.9 MB/s	42.2 MB/s
ngtcp2 (EverCrypt)	41.1 MB/s	41.1 MB/s	41.5 MB/s
picoquic (OpenSSL)	125.2 MB/s	121.5 MB/s	90.9 MB/s
picoquic (EverCrypt)	129.6 MB/s	129.5 MB/s	84.3 MB/s

The reported figures are averaged over 10 runs (removing the top and bottom decile) and include the respective message and header overheads of TLS, QUIC and HTTP. These results show that the performance of current open-source implementations of HTTP3-over-QUIC is not limited by the underlying cryptography. In synthetic tests, our implementation can process over 1.74M packets/s with AES128-GCM and an MTU set to 1300 bytes, which amounts to over 2.1GB/s of raw data. The performance for application traffic is lower due to the TLS, QUIC and HTTP3 overheads.

b) Merkle Tree: Our Merkle tree implementation required a non-trivial amount of proof effort (6.5K lines of F* code), but extracts to a relatively small amount of code (2.3K lines of C, not including EverCrypt and supporting libraries). We measure the time to fill a tree of various sizes by repeatedly inserting new nodes, and compute the average rate of of insertions/s. On a Core i9 9900K CPU, we are able to consistently achieve over 2.7M insertions/s on trees ranging in size from 1k to 1M nodes (see Figure 12 in appendix).

To compare with other implementations, we use the built-in `bench_bitcoin` tool provided in the Bitcoin [1] source tree. We run the MerkleRoot benchmark, which fills a Merkle tree with 9k nodes. The tool reports a median time of 947ms with the default 800 iterations and 5 measurements, which amounts to about 950k insertions/s, i.e., 40% of our verified code.

D. Verification Effort

Figure 10 summarizes line counts for EverCrypt and the applications built atop it. We include white space and comments. Line counts for Low* and Vale algorithms include some inline proofs, since F* makes it tricky to definitively separate implementation from proof. Yet, to give a sense of the ratio between actual implementation code and proof overhead, we also report the lines of C/ASM extracted from our implementations. We count assembly for one toolchain: Windows/MASM syntax.

Overall, verifying EverCrypt sequentially takes 5 hours 45 minutes. Individual functions/procedures (where developers mostly spend their time) verify much faster, typically in less than 10-20 seconds. We also exploit the parallelism of modular verification to bring the latency of a full system verification down to less than 18 minutes on a powerful desktop machine.

Altogether, designing, specifying, implementing, and verifying EverCrypt took three person-years, plus approximately one person year spent on infrastructure, testing, benchmarking, and contributing bug fixes and other improvements to F⋆.

IX. Conclusions

EverCrypt provides application developers with the functionality, ease of use, and speed they expect from existing cryptographic providers, while guaranteeing correctness and security of the code. Its API is carefully designed to support algorithm agility and multiplexing, but seals many details behind an abstraction boundary to simplify verification for clients. We program its implementation generically to reduce specification bloat and increase code and proof reuse, without harming performance. Indeed, our targeted implementations of hashes, AEAD, and ECC all meet or exceed the performance of state-of-the-art unverified implementations, and handily exceed previous verified implementations. Our QUIC transport security layer and our Merkle trees demonstrate the utility of EverCrypt's API in verifying client applications and meeting their demands for high performance.

Online Materials

EverCrypt is developed as part of Project Everest [21] and is available under an Apache 2.0 license on GitHub:

https://github.com/project-everest/hacl-star/tree/fstar-master/

This repository combines HACL∗, ValeCrypt, and EverCrypt. More information on the tools (F⋆, Vale, KreMLin) and how to get started with them, is available on our GitHub project page at https://project-everest.github.io/.

Acknowledgments

We would like to thank the entire Everest Team, with special thanks to Gustavo Varo for CI support.

Work at Carnegie Mellon University was supported in part by the Department of the Navy, Office of Naval Research under Grant No. N00014-18-1-2892, and a grant from the Alfred P. Sloan Foundation. Work at Inria was funded by ERC grant agreement number 683032 - CIRCUS. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these sponsors.

References

[1] Bitcoin Core integration/staging tree. https://github.com/bitcoin/bitcoin Git commit cbe7ef.

[2] curve25519-donna: Implementations of a fast Elliptic-curve Diffie-Hellman primitive. https://github.com/agl/curve25519-donna.

[3] Minimal implementation of the QUIC protocol. https://github.com/private-octopus/picoquic Git commit 65eee1.

[4] ngtcp2: an effort to implement IETF QUIC protocol. https://github.com/ngtcp2/ngtcp2 Git commit 79cde0.

[5] OpenSSL. https://github.com/openssl/openssl Commit e8fb288.

[6] Signal Protocol Library for Java/Android. https://github.com/signalapp/libsignal-protocol-java.

[7] The sodium crypto library (libsodium). https://github.com/jedisct1/libsodium.

[8] D. Ahman, C. Hriţcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy. Dijkstra monads for free. In *ACM Symposium on Principles of Programming Languages (POPL)*, Jan. 2017.

[9] A. Ahmed, D. Garg, C. Hritcu, and F. Piessens. Secure Compilation (Dagstuhl Seminar 18201). *Dagstuhl Reports*, 8(5):1–30, 2018.

[10] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[11] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P.-Y. Strub. The last mile: High-assurance and high-speed cryptographic implementations. arXiv:1904.04606 https://arxiv.org/abs/1904.04606.

[12] T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*, 2003.

[13] A. W. Appel. Verified software toolchain. In *Proceedings of the European Conference on Programming Languages and Systems (ESOP/ETAPS)*, 2011.

[14] A. W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Trans. Program. Lang. Syst.*, Apr. 2015.

[15] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie. System-level non-interference for constant-time cryptography. In *Proc. ACM CCS*, Nov. 2014.

[16] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella-Béguelin. Computer-aided security proofs for the working cryptographer. In *Proceedings of IACR CRYPTO*, 2011.

[17] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel. Verified correctness and security of OpenSSL HMAC. In *Proceedings of the USENIX Security Symposium*, 2015.

[18] D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *Proceedings of the IACR Conference on Practice and Theory of Public Key Cryptography (PKC)*, 2006.

[19] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2), Sept. 2012.

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

[21] K. Bhargavan, B. Bond, A. Delignat-Lavaud, C. Fournet, C. Hawblitzel, C. Hritcu, S. Ishtiaq, M. Kohlweiss, R. Leino, J. Lorch, K. Maillard, J. Pang, B. Parno, J. Protzenko, T. Ramananandro, A. Rane, A. Rastogi, N. Swamy, L. Thompson, P. Wang, S. Zanella-Beguelin, and J.-K. Zinzindohoué. Everest: Towards a verified drop-in replacement of HTTPS. In *Proceedings of the Summit on Advances in Programming Languages (SNAPL)*, 2017.

[22] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Pan, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Béguelin, and J. K. Zinzindohoué. Implementing and proving the TLS 1.3 record layer. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2017.

[23] A. Biryukov, D. Dinu, and D. Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. In *Proceedings of the IEEE European Symposium on Security and Privacy*, Mar. 2016.

[24] S. Böhme and T. Weber. Fast lcf-style proof reconstruction for

z3. In *Proceedings of Interactive Theorem Proving*, 2010.

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

[26] Y.-F. Chen, C.-H. Hsu, H.-H. Lin, P. Schwabe, M.-H. Tsai, B.-Y. Wang, B.-Y. Yang, and S.-Y. Yang. Verifying Curve25519 software. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2014.

[27] T. Chou. Sandy2x: New Curve25519 speed records. In *Selected Areas in Cryptography (SAC)*, 2016.

[28] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., 1986.

[29] I. Damgard. A design principle for hash functions. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 1989.

[30] P. Dettman. Problems with field arithmetic. https://github.com/armfazh/rfc7748_precomputed/issues/5, Feb. 2018.

[31] R. Dockins, A. Foltzer, J. Hendrix, B. Huffman, D. McNamee, and A. Tomb. Constructing semantic models of programs with the software analysis workbench. In *Conference on Verified Software - Theories, Tools, and Experiments (VSTTE)*, 2016.

[32] J. A. Donenfeld. Wireguard: Next generation kernel network tunnel. Jan. 2017.

[33] J. A. Donenfeld. kbench9000 - simple kernel land cycle counter. https://git.zx2c4.com/kbench9000/about/, Feb. 2018.

[34] J. A. Donenfeld. new 25519 measurements of formally verified implementations. http://moderncrypto.org/mail-archive/curves/2018/000972.html, Feb. 2018.

[35] M. Düll, B. Haase, G. Hinterwälder, M. Hutter, C. Paar, A. H. Sánchez, and P. Schwabe. High-speed curve25519 on 8-bit 16-bit and 32-bit microcontrollers. *Design Codes Cryptography*, 2015.

[36] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[37] K. Fisher, J. Launchbury, and R. R. . doi:10.1098/rsta.2015.0401. The HACMS program: Using formal methods to eliminate exploitable bugs. *Philosophical Transactions A, Math Phys Eng Sci.*, 375(2104), Sept. 2017.

[38] P. Fonseca, X. W. Kaiyuan Zhang, and A. Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of ACM EuroSys Conference*, Apr. 2017.

[39] A. Fromherz, N. Giannarakis, C. Hawblitzel, B. Parno, A. Rastogi, and N. Swamy. A verified efficient embedding of a verifiable assembly language. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, 2019.

[40] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, 2015.

[41] S. Gueron. Intel® Advanced Encryption Standard (AES) New Instructions Set. https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf, Sept. 2012.

[42] S. Gueron and V. Krasnov. The fragility of AES-GCM authentication algorithm. In *Proceedings of the Conference on Information Technology: New Generations*, Apr. 2014.

[43] S. Gulley, V. Gopal, K. Yap, W. Feghali, J. Guilford, and G. Wolrich. Intel® SHA Extensions. https://software.intel.com/sites/default/files/article/402097/intel-sha-extensions-white-paper.pdf, July 2013.

[44] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-end security via automated full-system verification. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[45] J. Iyengar and M. Thompson. QUIC: A UDP-Based Multiplexed and Secure Transport. draft-ietf-quic-transport-20, Apr. 2019.

[46] S. Josefsson and I. Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, Jan. 2017.

[47] H. Krawczyk and P. Eronen. HMAC-based extract-and-expand key derivation function (HKDF). RFC 5869, May 2010.

[48] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, Jan. 2014.

[49] A. Langley. A shallow survey of formal methods for C code, 2014. https://www.imperialviolet.org/2014/09/07/provers.html.

[50] A. Langley, M. Hamburg, and S. Turner. Elliptic curves for security. https://tools.ietf.org/html/rfc7748, Jan. 2016.

[51] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2010.

[52] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand. Compcert – a formally verified optimizing compiler. In *Embedded Real Time Software and Systems (ERTS)*. SEE, 2016.

[53] libjc. Combined AEAD ChaCha/Poly. https://github.com/tfaoliveira/libjc/issues/2, July 2019.

[54] M. Thompson and S. Turner. Using TLS to Secure QUIC. Internet-Draft, draft-ietf-quic-tls-20, Apr. 2019.

[55] G. Martínez, D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, C. Hritcu, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, T. Ramananandro, A. Rastogi, and N. Swamy. Meta-F*: Metaprogramming and tactics in an effectful program verifier. In *European Symposium on Programming (ESOP)*, 2019.

[56] R. C. Merkle. A digital signature based on a conventional encryption function. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 1987.

[57] R. C. Merkle. A certified digital signature. In *Proceedings of CRYPTO*, 1989.

[58] Microsoft. Cryptographic API: Next generation - Windows Applications. http://https://docs.microsoft.com/en-us/windows/desktop/seccng/cng-portal, May 2018.

[59] Mozilla. Measurement dashboard. https://mzl.la/2ug9YCH, July 2018.

[60] E. Mullen, S. Pernsteiner, J. R. Wilcox, Z. Tatlock, and D. Grossman. Œuf: Minimizing the coq extraction TCB. In *Proceedings of the ACM Conference on Certified Programs and Proofs (CPP)*, 2018.

[61] National Vulnerability Database. CVE-2012-2459. https://nvd.nist.gov/vuln/detail/CVE-2012-2459, June 2012.

[62] NIST. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D, Nov. 2007.

[63] T. Oliveira, J. López, H. Hışıl, A. Faz-Hernández, and F. Rodríguez-Henríquez. How to (pre-)compute a ladder: Improving the performance of X25519 and X448. In *Proceedings of Selected Areas in Cryptography (SAC)*, Aug. 2017.

[64] OpenSSL. Vulnerabilities. https://www.openssl.org/news/vulnerabilities.html Retrieved July, 2019.

[65] OpenSSL Team. OpenSSL. http://www.openssl.org/, May 2005.

[66] A. Petcher and G. Morrisett. The foundational cryptography framework. In R. Focardi and A. Myers, editors, *Principles of Security and Trust*, 2015.

[67] A. Polyakov, M.-H. Tsai, B.-Y. Wang, and B.-Y. Yang. Verifying Arithmetic Assembly Programs in Cryptographic Primitives. In *Conference on Concurrency Theory (CONCUR)*, 2018.

[68] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hritcu, K. Bhargavan, C. Fournet, and N. Swamy. Verified low-level programming embedded in F*. *PACMPL*, (ICFP), Sept. 2017.

[69] M. Russinovich, E. Ashton, C. Avanessians, M. Castro, A. Chamayou, S. Clebsch, M. Costa, C. Fournet, M. Kerner, S. Krishna, J. Maffre, T. Moscibroda, K. Nayak, O. Ohrimenko, F. Schuster, R. Schwartz, A. Shamis, O. Vrousgou, and C. M. Wintersteiger. CCF: A framework for building confidential verifiable replicated services. Technical Report MSR-TR-2019-16, Microsoft, April 2019.

[70] R. Seacord. Implement abstract data types using opaque types. https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=87151966, Oct. 2018.

[71] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, 2016.

[72] The Coq Development Team. The Coq Proof Assistant Reference Manual, version 8.7, Oct. 2017.

[73] A. Tomb. Automated verification of real-world cryptographic implementations. *IEEE Security Privacy Magazine*, 14(6), Nov. 2016.

[74] D. B. Tracker. libssl1.0.0: illegal instruction crash on amd64. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=793557#132.

[75] M.-H. Tsai, B.-Y. Wang, and B.-Y. Yang. Certified verification of algebraic properties on low-level mathematical constructs in cryptographic programs. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[76] N. Tuveri and B. B. Brumley. Start your ENGINEs: Dynamically loadable contemporary crypto. Technical report, 2018.

[77] F. Voight. CVE-2012-2459 (block merkle calculation exploit). https://bitcointalk.org/?topic=102395, Aug. 2012.

[78] Wikipedia. Bit manipulation instruction sets (last retrieved 2019-03-22). https://en.wikipedia.org/wiki/Bit_Manipulation_Instruction_Sets#ABM_(Advanced_Bit_Manipulation).

[79] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, June 2011.

[80] K. Q. Ye, M. Green, N. Sanguansin, L. Beringer, A. Petcher, and A. W. Appel. Verified correctness and security of mbedTLS HMAC-DRBG. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

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

[82] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

X. Appendix: Secret Independence

We restate and discuss the *Secret Independence for Hybrid Low*/Vale programs* theorem proven by Fromherz et al. [39].

Constant-time properties are proven of Low* programs using a syntactic, type-based analysis for secure information flow. In particular, Low* programmers can designate specific inputs in their programs as secret and give them abstract types to force them to be manipulated only through a given interface of constant-time primitives, e.g., only additions, multiplications, and bitwise operations, but no comparisons or divisions. A meta-theorem about Low* establishes that such well-typed programs produce execution traces (sequences of instructions, including branches taken and memory addresses accessed) that are independent of their secret inputs.

In contrast, Vale programs are analyzed for leakage using a taint analysis programmed and proven correct with F*. The taint analysis is proven to only accept programs whose execution traces (which also includes the sequences of instructions executed and the memory addresses they access) do not depend on inputs that are designated as secret.

To compose the two properties, Fromherz et al. [39] define an extended Low* syntax and semantics, which includes trace-instrumented atomic computation steps for the Vale program fragments executed. This extended language has an execution semantics that produces traces that are concatenations of Low* and Vale execution traces. The theorem (stated below) shows that the two notions of secret independence compose well, establishing that well-typed Low* programs that call into leakage-free Vale programs produce combined traces that are also secret independent.

The theorem is a bisimulation between runs of a pair of related, well-typed Vale-extended Low* runtime configurations (H_1, e_1) and (H_2, e_2), showing that they either both step to related configurations maintaining their invariants and producing identical traces, or they are both terminal configurations.

Theorem: Secret Independence: Given well-typed extended Low* configurations (H_1, e_1) and (H_2, e_2), where $\Gamma \vdash (H_1, e_1) : \tau$, $\Gamma \vdash (H_2, e_2) : \tau$, $H_1 \equiv_\Gamma H_2$ and $e_1 \equiv_\Gamma e_2$, and a secret-independent implementation of the secret interface P_s, either both the configurations cannot reduce further, or $\exists \Gamma' \supseteq \Gamma$ s.t. $P_s \vdash (H_1, e_1) \rightarrow^+_{\ell_1} (H'_1, e'_1)$, $P_s \vdash (H_2, e_2) \rightarrow^+_{\ell_2} (H'_2, e'_2)$, $\Gamma' \vdash (H'_1, e'_1) : \tau$, $\Gamma' \vdash (H'_2, e'_2) : \tau$, $\ell_1 = \ell_2$, $H'_1 \equiv_{\Gamma'} H'_2$, and $e'_1 \equiv_{\Gamma'} e'_2$,

XI. Appendix: Incremental Merkle trees

Figure 11 shows how our Merkle trees are incrementally constructed, while Figure 12 gives performance measurements for tree insertions, given various tree sizes. The results confirm that the incremental construction keeps insertion a constant-time operation, regardless of tree size.

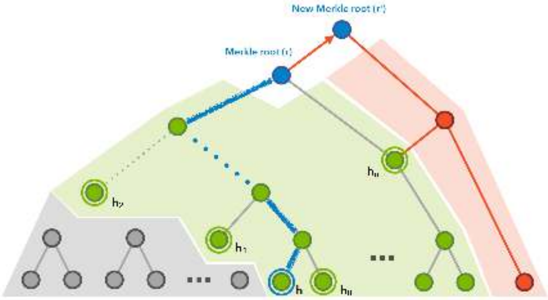

Figure 11. ***Incremental Binary Merkle Trees.*** *The leaves are individual transactions. The red fringe of the tree represents the insertion of a new transaction in the tree. (Note that we skip levels for partial binary trees.) The green part of the tree represents cached intermediate hashes, enabling immediate lookup of any paths for the corresponding leaves. For example, looking up the path from* h *to the root would return a logarithmic number of hashes* $h_0, h_1, \dots, h_2, \dots$ *enabling the recomputation of* r *from* h*, The grey part of the tree represents intermediate hashes that have been erased to reclaim memory.*

XII. Appendix: Code snippets for C and header files

Figures 13 and 14 illustrate the C code extracted from EverCrypt.

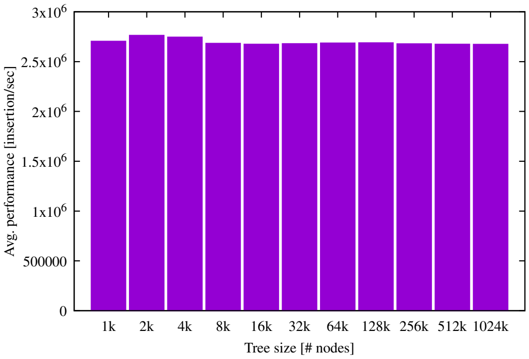

Figure 12. ***Avg. number of insertions of a node into our verified Merkle tree of various sizes, per second. Higher is better.***

```
// From the C implementation of hashes.
static void Hash_hash_256(uint8_t *input, uint32_t input_len,
  uint8_t *dst)
{
  uint32_t s[8U] = {
    0x6a09e667U, 0xbb67ae85U, 0x3c6ef372U, 0xa54ff53aU,
    0x510e527fU, 0x9b05688cU, 0x1f83d9abU, 0x5be0cd19U
  };
  uint32_t blocks_n = input_len / (uint32_t)64U;
  uint32_t blocks_len = blocks_n * (uint32_t)64U;
  uint8_t *blocks = input;
  uint32_t rest_len = input_len − blocks_len;
  uint8_t *rest = input + blocks_len;
  bool has_shaext1 = AutoConfig2_has_shaext();
  if (has_shaext1) {
    uint64_t n1 = (uint64_t)blocks_n;
    uint64_t scrut = sha256_compress(s, blocks, n1,
      Hash_Core_SHA2_Constants_k224_256);
  }
  else {
    Hash_SHA2_compress_many_256(s, blocks, blocks_n);
  }
  Hash_compress_last_256(s, (uint64_t)blocks_len, rest, rest_len);
  Hash_Core_SHA2_finish_256(s, dst);
}
```

Figure 13. ***A representative snippet of the C code generated for EverCrypt. This is a specialized instance of the generic Merkle-Damgård construction for a compression function that multiplexes between Low⋆ and Vale.***

XIII. Appendix: Analysis of Vulnerabilities in OpenSSL's Cryptographic Provider

In Figure 15, we summarize all 24 vulnerabilities reported against OpenSSL's cryptographic provider, `libcrypto`, between January 2016 and July 2019 [64]. Each row explains what the vulnerability affected and which of EverCrypt's properties (safety, functional correctness, or side-channel freedom) would have prevented it. The list covers both `libcrypto` algorithms that have been re-implemented in EverCrypt, and algorithms that, if implemented in EverCrypt, would be immune to those vulnerabilities. The only vulnerability EverCrypt's methodology might not have prevented is 2016-0701, in which OpenSSL added a feature to allow the loading of Diffie-Hellman primes from X9.42-style parameter files. The library trusted these primes to be safe, rather than checking them for safety.

```
// This type is shared with specifications.
#define Hash_SHA2_224 0
#define Hash_SHA2_256 1
#define Hash_SHA2_384 2
#define Hash_SHA2_512 3
#define Hash_SHA1 4
#define Hash_MD5 5
typedef uint8_t Hash_hash_alg;

// Interface for hashes, block-aligned input data
struct Hash_state_s_s;
typedef struct Hash_state_s_s Hash_state_s;

Hash_state_s *Hash_create_in(Hash_hash_alg a);
void Hash_init(Hash_state_s *s);
void Hash_compress(Hash_state_s *s, uint8_t *block1);
void Hash_compress_many(Hash_state_s *s,
  uint8_t *blocks, uint32_t len1);
void Hash_compress_last(Hash_state_s *s,
  uint8_t *last1, uint64_t total_len);
void Hash_finish(Hash_state_s *s, uint8_t *dst);
void Hash_free(Hash_state_s *s);
void Hash_copy(Hash_state_s *s_src, Hash_state_s *s_dst);

void Hash_hash(Hash_hash_alg a,
  uint8_t *dst, uint8_t *input, uint32_t len1);

// Incremental interface for hashes, arbitrary input data,
// relies on an internal array (not covered in the paper).
typedef struct Hash_Incremental_state_s {
  Hash_state_s *hash_state;
  uint8_t *buf;
  uint64_t total_len;
} Hash_Incremental_state;

Hash_Incremental_state Hash_Incremental_create_in(
  Hash_hash_alg a);
Hash_Incremental_state Hash_Incremental_compress(
  Hash_hash_alg a, Hash_Incremental_state s,
  uint8_t *data, uint32_t len1);
void Hash_Incremental_finish(Hash_hash_alg a,
  Hash_Incremental_state s, uint8_t *dst);

// Specialized versions of HMAC
void HMAC_compute_sha1(uint8_t *mac, uint8_t *key,
  uint32_t keylen, uint8_t *data, uint32_t datalen);
void HMAC_compute_sha2_256(uint8_t *mac, uint8_t *key,
  uint32_t keylen, uint8_t *data, uint32_t datalen);
void HMAC_compute_sha2_384(uint8_t *mac, uint8_t *key,
  uint32_t keylen, uint8_t *data, uint32_t datalen);
void HMAC_compute_sha2_512(uint8_t *mac, uint8_t *key,
  uint32_t keylen, uint8_t *data, uint32_t datalen);

// Agile HMAC
bool HMAC_is_supported_alg(Hash_hash_alg x);
void HMAC_compute(Hash_hash_alg a, uint8_t *mac,
  uint8_t *key, uint32_t keylen, uint8_t *data, uint32_t datalen);
```

Figure 14. ***A representative subset of the EverCrypt API. This file is taken as-is from the output of our toolchain (i.e. from the .h file). The file was edited only to remove some module name prefixes to make the code more compact.***

CVE	Severity	Vulnerability	Broken property	Prevented?
2019-1543	Low	improper IV handling	functional correctness	✓
2018-5407	Low	EC multiplication timing leak	side-channel resistance	✓
2018-0734	Low	bignum timing leak	side-channel resistance	✓
2018-0735	Low	bignum allocation leak	side-channel resistance	✓
2018-0737	Low	bignum timing leak	side-channel resistance	✓
2018-0733	Moderate	incorrect hand-written assembly	functional correctness	✓
2017-3738	Low	buffer overflow	functional correctness	✓
2017-3736	Moderate	carry propagation bug	functional correctness	✓
2017-3733	High	inconsistent agility parameter	functional correctness	✓
2017-3732	Moderate	carry propagation bug	functional correctness	✓
2017-3731	Moderate	out of bounds access	memory safety	✓
2016-7055	Low	carry propagation bug	functional correctness	✓
2016-7054	High	incorrect memset	memory safety	✓
2016-6303	Low	integer overflow	functional correctness	✓
2016-2178	Low	cache timing leak	side-channel resistance	✓
2016-2177	Low	undefined behavior	memory safety	✓
2016-2107	High	missing bounds check	functional correctness	✓
2016-2106	Low	integer overflow	functional correctness	✓
2016-2105	Low	integer overflow	functional correctness	✓
2016-0705	Low	double free	memory safety	✓
2016-0704	Moderate	key recovery	side-channel resistance	✓
2016-0703	High	key recovery	side-channel resistance	✓
2016-0702	Low	cache timing leak	side-channel resistance	✓
2016-0701	High	load unsafe DH primes	misconfiguration	×

Figure 15. Recent CVEs in `libcrypto` that would have been prevented by EverCrypt's methodology.