Zero-cost meta-programmed stateful functors in F★

Jonathan Protzenko
Microsoft Research

Son Ho
Inria

Abstract

We present zero-cost, high-level F★ functors and their compilation to low-level, efficient C code. Thanks to a combination of partial evaluation, fine-grained control of reduction, and tactic-driven C++ template-like metaprogramming, we provide the programmer with a toolkit that dramatically reduces the proof-to-code ratio, brings out the essence of algorithmic and implementation agility, and allows substantial code reuse while remaining at a very high-level of abstraction. None of our techniques require modifying the F★ compiler.

We describe a systematic process to develop functors, and illustrate it with the streaming functor, which wraps an error-prone, cryptographic block API by hiding internal buffering and state machine management to prevent C programmer mistakes. We apply this functor to 10 implementations from the HACL★ [31] cryptographic library. We then write a tactic to automate the functor encoding, allowing the programmer to author multi-argument functors with a deeply nested call graph without any syntactic overhead. We apply this general tactic on 5 algorithms from HACL★, yielding over 30 specialized functor applications. We use as an example Curve25519, a complex algorithm whose final, specialized version we express as nested functor applications.

1 Scaling cryptographic verification

In recent years [9], projects such as FiatCrypto [20], Jasmin [4, 5], CryptoLine [22], Vale-Crypto [14, 21] or HACL★ [31, 41] have demonstrated that is it now feasible to verify cryptographic algorithms whose performance matches or exceeds state-of-the-art, unverified code from projects like OpenSSL [30] or libsodium [1]. Building upon such efforts, projects like EverCrypt [32] offer entire verified cryptographic providers, which expose a wide range of modern algorithms, offering an agile and multiplexing API via CPU auto-detection.

There remain, however, several obstacles on the road to wider adoption and distribution of verified cryptography. Oftentimes, verified libraries are integrated into unverified software projects, written in C, C++ or any other popular language. Thus, when authoring verified code, it does not suffice to merely implement an algorithm as specified by the RFC; the author of verified code must strive for elegant, intuitive, easy-to-use APIs that minimize the risk of programmer error. Designing such a safe API requires more layers of abstraction, which increases the verification burden.

As an example, consider hash algorithms like SHA-2 [2] or Blake2 [8]. They are block-based, which requires clients to obey a precise state machine, feeding the data to be hashed block by block, and invalidating the state when processing the final data. This is error-prone and almost certain to result in programmer mistakes. There exists a safer API with better usability, which various libraries aim to provide. The safe API maintains an internal buffer, and eliminates almost all possible mistakes when using it from C. Yet, even Blake2’s own safe, reference API implementation [34] contained a buffer management bug that went undetected for seven years [29]!

The natural conclusion is that verified cryptographic libraries need to expose only safe, and verified APIs. Unfortunately, none of them currently do. One reason is that the amount of effort required is tremendous: verifying a non-trivial, safe API for SHA2-256 is certainly feasible [7], but doing it over and over for a dozen similar algorithms is out of the question. Verifying safe APIs is tedious, repetitive, and until now did not lend itself well to automation.

A reason for the lack of automation is that the commonality between algorithms has not yet been formally captured. Consider HACL★, which at this time offers 17 algorithms, spread across about 40 implementations. We observe that at least ten of those algorithms obey almost identical state machines and behave in a fashion almost identical to Blake2. Yet, a formal argument was never made to assert that these algorithms really do behave in the same way.

Another reason for the lack of automation is that so far, no systematic techniques have been established to “verify in the large” [18]. If those 10 algorithms behave similarly, it should be feasible to write and verify the safe API once and for all, and get 10 copies of it “for free”. In practice, this requires, in addition to capturing the commonality, devising techniques, tools and strategies to write and verify code at this very high level of abstraction. To add to the challenge, slow cryptographic code is dismissed by practitioners, which makes it impossible to use standard functional programming paradigms such as type classes with run-time dictionaries.

All of these shortcomings show up in existing work. For instance, EverCrypt offers only a single safe API for hashing, briefly mentioned but not explained in [32, XIII]. Lacking any better ways of automating those proofs, Protzenko et.al were not able to verify more than a single safe API.
This paper, using F* as our lingua franca, presents a series of language-based techniques. We explain how to conceptualize commonality between algorithms, using block APIs as our running example. We show how the programmer can think, write and verify at the highest level of abstraction, writing functors that transform an unsafe block API into a safe API once and for all, without incurring any performance penalty at run-time. We show all the techniques that enable this style of programming without needing to modify the F* compiler. Specifically, we show how to use a combination of partial evaluation and meta-programming to write code in a style akin to C++ templates, and specialize it for free for multiple instances. Our evaluation shows that programmer effort is reduced, at the expense of a modest increase in verification time for meta-generated code.

All throughout the paper, we use cryptography as the main driving force. There are two reasons. First, cryptographic APIs offer the highest level of complexity, combining mutable state, state machines, abstract representations and abstract predicates. While our techniques have been successfully applied to simpler use-cases, e.g. data structures and associative lists, we wish to showcase the full power and generality of our approach. Second, we benefit from the large existing codebase of cryptographic code in F*, meaning we can perform a real-world quantitative evaluation of how our techniques make verification more scalable.

All of our ideas have been implemented, verified and demonstrated on the HACL* and EverCrypt libraries. Our code has been integrated in the HACL* repository and is now used by Firefox, Linux, the Tezos blockchain, and others.

Our paper is structured as follows:

- we provide some detailed background (§2) on the F*-to-C toolchain that HACL* uses;
- we explore the commonality problem (§3), also known as “agility” in cryptographic lingo, using type classes as a key technical device;
- equipped with a precise definition of a block algorithm, we show how to manually write a functor that generates a safe API for any block algorithm (§4);
- getting such higher-order code to compile to C without runtime overhead is a challenge; we list a constellation of techniques (§5) that, put together, completely eliminate the cost of high-level programming thanks to partial evaluation;
- manually writing functors can prove quite tedious; we use the full power of Meta-F* (§6) to automate the functor encoding, allowing programmers to write generic code without any syntactic penalty;
- we evaluate the benefits of our techniques (§7), quantifying the verification effort for the programmer, as well as the performance impact on verification times.

While demonstrated on F*, we believe both the language techniques and the design of the cryptographic functors are reusable in other verification settings, languages and toolchains. For instance, identifying and capturing the block API is a generic contribution that can be reproduced, say, in Coq just as well. Similarly, the idea of authoring the streaming functor and relying on partial evaluation to eliminate runtime overhead could applied to, say, LMS-Scala.

2 Background: F*, Low*, Meta-F*

HACL* [31, 41] is a cryptographic library written in F*, which we use as a baseline to author, evaluate and integrate our proof techniques. HACL* compiles to C, and offers vectorized versions of many algorithms via C compiler intrinsics, e.g. for targets that support AVX, AVX2 or ARM Neon. EverCrypt is a high-level API that multiplexes between HACL* and Vale-Crypto [21] and supports dynamic selection of algorithms and implementations based on the target CPU’s feature set. Combined with EverCrypt, HACL* features 130k lines of F* code for 62k lines of generated C code (all excluding comments and whitespace).

F* is a state-of-the-art verification-oriented programming language. Hailing from the tradition of ML [28], F* features dependent types and a user-extensible effect system, which allows reasoning about IO, concurrency, divergence, various flavors of mutability, or any combination thereof. For verification, F* uses a weakest precondition calculus based on Dijkstra Monads [3, 37], which synthesizes verification conditions that are then discharged to the Z3 [16] SMT solver. Proofs in F* typically are a mixture of manual reasoning (calls to lemmas), semi-automated reasoning (via tactics) and fully automated reasoning (via SMT).

Low* is a subset of F* that exposes a carefully curated set of features from the C language. Using F*’s effect system, Low* models the C stack and heap, and allocations in those regions of the memory. Low* also models data-oriented features of C, such as arrays, pointer arithmetic, machine integers with modulo semantics, const pointers, and many others via a set of distinguished libraries. Programming in Low* guarantees spatial safety (no out-of-bounds accesses), temporal safety (no double frees, no use-after-free) and a form of side-channel resistance [33, 41]. All of these guarantees are enforced statically and incur no run-time checks.

To provide a flavor of programming in Low*, we present the mk_update_blake2 function below, taken from HACL*.

```
val mk_update_blake2 (a:blake_alg) (v:vectorizer) (s:state a v)
(totlen : U64.t) (data:block a): Stack U64.t
requires \h \rightarrow live h s \land live h data \land disjoint s data
ensures \h0 totlen' h1 \rightarrow modifies1 s h0 h1 \land
  (as_seq h1 s, U64.v totlen') ==
    Blak2.Spec.update a (as_seq h0 s) (U64.v totlen) (as_seq h0 data))
```

Functions in Low* are annotated with their return effect, in this case Stack, which indicates the function only performs stack allocations, and therefore is guaranteed to have no memory leaks. For all other cases, programmers may use the
ST effect. The return type of the function is U64.t, the type of 64-bit unsigned machine integers with modulo semantics. Functions are specified using pre- and two-state post-conditions. Low* relies on a modifies-clause theory [24]: the pre-condition demands that arrays data and state be disjoint, while the post-condition ensures that the only memory location affected by a call to mk_updateBlake2 is s. If a client holds an array a, then the combination of disjoint a and modifies s h0 h1 allows them to derive automatically that a is unchanged in h1. The liveness clauses ensure accesses to s and data are valid in the body of mk_updateBlake2.

Importantly, the functional behavior of mk_updateBlake2 is specified via Blake2Spec.update, a pure function that forms our specification. This style is referred to as intrinsic reasoning, or implementation refinement: we prove that the low-level behavior of mk_updateBlake2 is characterised by a pure, trusted and carefully audited specification. Various functions allow reflecting Low* objects as their pure counterpart: in this case, as_seq reflects the contents of s in h1 as a pure sequence, and U64.v reflects a machine integer as a pure unbounded mathematical number. Functions such as as_seq are in effect Ghost, meaning that they may only be used in proofs and refinements, and cannot appear at run-time.

Partial evaluation is a powerful, trusted mechanism in F*. With it, the programmer can trigger various steps of reduction using F*'s trusted normalizer without having to perform any proof. F* exposes keywords, attributes and functions to control this mechanism. We look back at the example of mk_updateBlake2: as it stands, this function is actually not valid Low*: the type state is indexed over two arguments, capturing the choice of algorithm a and implementation v. Here, lbuffer t l is an array of type t and length l.

```
inline_for_extraction let blake2_state a v =
  lbuffer (element_t a v) (4u4 x row_len a v)
```

Such a type definition cannot be safely extracted to C and is not valid Low*. The special “inline for extraction” attribute, however, indicates to F* that right before extraction, occurrences of blake2_state should be replaced with their definition. This in turns triggers more reductions steps, where matches are reduced, unreachable branches eliminated, beta-reduces evaluated away, meaning that if we apply mk_updateBlake2 to constant values for its first two arguments, we actually get Low* code after partial evaluation:

```
let updateBlake2s_32 = mk_updateBlake2s Scalar // regular C
let updateBlake2s_128 = mk_updateBlake2s AVX // vectorized
```

The “inline for extraction” mechanism applies to functions, including stateful ones: F* introduces an A-normal form [35], meaning a stateful call f e becomes let x = e in f x, which then allows β-reduction since x is a value.

We make extensive use of this keyword all throughout this paper, as it provides a way to drastically slash code duplication. For instance, HACLL* has a single implementation of Blake2, which partially evaluates to four different implementations depending on the variant (Blake2s vs. Blake2b) and the degree of vectorization (C, AVX, AVX2).

Beyond this keyword, other mechanisms exist for partial evaluation. The [@inline_let] attribute allows reducing pure let-bindings inside function definitions, and the normalize call allows very fine-grained control of the reduction flags for a sub-term. The definition of mk_updateBlake2 uses both.

Meta-F* is a recent extension of F* [26] that allows the programmer to script the F* compiler using user-written F* programs, an approach known as elaborator reflection and pioneered by Lean [17] and Idris [15]. Meta-F* offers, by design, a safe API for term manipulation, meaning it re-checks the results of meta-program execution: if a meta-program attempts to synthesise an ill-typed term, F* aborts. Therefore, tactics do not need to be statically proven correct and enjoy a great deal of flexibility. Tactics are written in the Tac effect, and in this paper we use the terms “tactic” and “meta-program” interchangeably.

Erasure and extraction in F* follows Letouzey’s extraction principles for Coq [25]. After type-checking and performing partial evaluation, F* erases computationally-irrelevant code and performs extraction to an intermediary representation dubbed the “ML AST”.

For erasure, F* eliminates refinements, pre- and post-conditions, and generally replaces computationally irrelevant terms with units, i.e. any subexpression of type Ghost becomes(). F* also removes calls to unit-returning functions, which means that calls to lemmas are also eliminated.

For extraction, F* ensures that the “ML AST” features only prefix polymorphism (i.e. type schemes), and that it is annotated with classic ML types. Naturally, not all F* programs are type-able per the ML rules: using a bidirectional approach, extraction inserts necessary casts, and replaces types that are invalid in ML with $\top$, the uninformative type.

The ML AST can then be further compiled to three different targets. Going to OCaml, $\top$ becomes Obj.t and casts become calls to Obj.magic; owing to OCaml’s uniform value representation, any F* program can be compiled to OCaml. Going to F#, only a subset of casts are admissible, since the .NET bytecode is typed. Finally, going to C only works for programs that are free of casts, as C does not admit a $\top$ type.

KreMLin [33] compiles the “ML AST” to readable, auditable C by using a series of small, composable passes. The KreMLin preservation theorem [33] states that the safety guarantees in Low* carry over to the generated C code. We present a few transformations that are relevant to this work.

The ML AST supports parameterized data types, such as pairs, tuples, and user-defined inductives, e.g. type (‘a, ‘b) pair = Pair of ‘a * ‘b. First, KreMLin removes unused type parameters. This is important for deeply dependent inductive indices: as long as they only appear in refinements, the resulting $\top$ type parameter in the ML AST is eliminated by
We posited earlier that many algorithms exhibit large amounts of commonality, and essentially behave the same way. We provide some basic cryptographic context, then show how we can describe what a block API is in F*.

3 The essence of agility

We posited earlier that many algorithms exhibit large amounts of commonality, and essentially behave the same way. We provide some basic cryptographic context, then show how we can describe what a block API is in F*.

3.1 Crafting safe APIs for C clients

Many cryptographic algorithms offer identical or similar functionalities. For example, SHA2 [2], SHA3 [19] and Blake2 [8] (in no-key mode) all implement the hash functionality, taking an input text to compute a resulting digest. As another example, HMAC [11], Poly1305 [13], GCM [27] and Blake2 implement the message authentication code (MAC) functionality, taking an input text and a key to compute a digest.

At a high level, these functionalities are simply black boxes with one or two inputs, and a single output. Taking HACL*’s SHA2-256 implementation as an example, this results in a natural, self-explanatory C API:

```c
void sha2_256(uint8_t *input, uint32_t input_len, uint8_t *dst);
```

This “one-shot” API, however, places unrealistic expectations on clients of this library. For instance, the TLS protocol, in order to authenticate messages, computes repeated intermediary hashes of the handshake data transmitted so far. Using the one-shot API would be grossly inefficient, as it would require re-hashing the entire handshake data every single time. In other situations, such as Noise protocols, just hashing the concatenation of two non-contiguous arrays with this API requires a full copy into a contiguous array.

Cryptographic libraries thus need to provide a different API that allows clients to perform incremental hash computations. A natural candidate for this is the block API: all of the algorithms we mentioned above are block-based, meaning that under the hood, they initialize their internal state, process the data block-by-block (for an algorithm-specific block size), perform some special treatment for the leftover data, and then extract the internal state onto a user-provided destination buffer, which holds the final digest. Revealing this API (Figure 1) would allow clients to feed their data into the hash incrementally.

The issue with this block API is that it is wildly unsafe to call from unverified C code. First, it requires clients to maintain a block-sized buffer, that once full must be emptied via a call to update_block. This entails non-trivial modulo-arithmetic computations and pointer manipulations, which are error-prone [29]. Second, clients can very easily violate the state machine. For instance, when extracting an intermediary hash, clients must remember to copy the internal hash state, call the sequence update_last / finish on the copy, free that copy, and only then resume feeding more data into the original hash state. Third, algorithms exhibit subtle differences: for instance, Blake2 must not receive empty data for update_last, while SHA2 is absolutely fine. In short, the block API is error-prone, confusing, and is likely to result in programmer mistakes.

We thus wish to take all of the block-based algorithms, and devise a way to wrap their respective block APIs into a uniform, safe API that eliminates all of the pitfalls above. We dub this safe API the streaming API (Figure 2): it has a degenerate state machine with a single state; it performs buffer management under the hood; it hides the differences between algorithms; and performs necessary copies as-needed when a digest needs to be extracted.

Writing and verifying a copy of the streaming API for each one of the eligible algorithms would be tedious, not very much fun, and bad proof engineering. We thus set out to write a functor, that takes any block API and returns the corresponding streaming API. But for that, we first need to state what a block API is.

3.2 The essence of stateful data

Before we get to the block API itself, we need to capture a more basic notion, that of an abstract piece of data that lives
type stateful = | Stateful:
| \{ Low-level type \} s: Type0 \to

footprint: h:mem \to s:s \to Ghost loc \to
invariant: h:mem \to s:s \to Type
| \{ A pure representation of an s \} |

t: Type0 \to
v: h:mem \to s:s \to Ghost t \to
| \{ Adequate framing lemmas, relying on v \} |
invariant_loc_in-footprint: h:mem \to s:s \to Lemma
| \{ requires \ (invariant h s) \} |
(ensures (loc_in (footprint h s) h)) \to
| \{ Stateful operations \} |
frame_invariant: l:loc \to s:s \to h:mem \to h:mem \to Lemma
| \{ requires \ (invariant h0 s \& loc_disjoint l (footprint h0 s) \& modifies l h0 h1) \} |
(ensures (invariant h1 s \& \_ \& \_ \& \_ \& footprint h1 s == footprint h0 s)) \to

\begin{figure}[h]
\centering
\begin{tabular}{ll}
1 & type stateful = | Stateful:
2 & | \{ Low-level type \} s: Type0 \to
3 & footprint: h:mem \to s:s \to Ghost loc \to
4 & invariant: h:mem \to s:s \to Type
5 & | \{ A pure representation of an s \} |
6 & t: Type0 \to
7 & v: h:mem \to s:s \to Ghost t \to
8 & | \{ Adequate framing lemmas, relying on v \} |
9 & invariant_loc_in-footprint: h:mem \to s:s \to Lemma
10 & | \{ requires \ (invariant h s) \} |
11 & (ensures (loc_in (footprint h s) h)) \to
12 & | \{ Stateful operations \} |
13 & frame_invariant: l:loc \to s:s \to h:mem \to h:mem \to Lemma
14 & | \{ requires \ (invariant h0 s \& loc_disjoint l (footprint h0 s) \& modifies l h0 h1) \} |
15 & (ensures (invariant h1 s \& \_ \& \_ \& \_ \& footprint h1 s == footprint h0 s)) \to
16 & \end{tabular}
\caption{The stateful API}
\end{figure}

is what the type class syntax desugars to. We plan to switch to the class keyword once all the bugs are fixed in F*.

3.3 The essence of block algorithms

We now capture the essence of a block algorithm by authoring a type class that encapsulates a block algorithm’s types, representations, specifications, lemmas and stateful implementations in one go.

We need the block type class to capture four broad traits of a block algorithm, namely i) explain the runtime representation and spatial characteristics of the block algorithm, ii) specify as pure functions the transitions in the state machine, iii) reveal the block algorithm’s central lemma, i.e. processing the input data block by block is the same as processing all of the data in one go, and iv) expose the low-level run-time functions that realize the transitions in the state machine. The result appears in Figure 4 in a simplified form (the actual definition is about 150 lines of F*).

Run-time characteristics. A block algorithm revolves around its state, of type stateful. A block algorithm may need to keep a key at run-time (km = Runtime, e.g. Poly1305), or keep a ghost key for specification purposes (km = Erased, e.g. keyed Blake2), or may need no key at all, in which case the key field is a degenerate instance of stateful where key.s = unit.

Specification. Using state.t, i.e. the algorithm’s state reflected as pure value, we specify each transition of the state machine at lines 18-23. Importantly, rather than specify an “update block” function, we use an “update multi” function that can process multiple blocks at a time. We don’t impose any constraints on how update_multi is authored: we just request that it obeys the fold law via the lemma at line 28. This style has several advantages. First, this leaves the possibility for optimized algorithms that process multiple blocks at a time to provide their own update_multi function, rather than being forced to inefficiently process a single block. For unoptimized algorithms that are authored with update_block, we provide a higher-order combinator that derives an update_multi function and its correctness lemma automatically. Second, by being very abstract about how the blocks are processed, we capture a wide range of behaviors for block algorithms. For instance, Poly1305 has immutable internal state for storing precomputations, along with an accumulator that changes with each call to update_block: we simply pick state.t to be a pair, where the fold only operates on the second component.

The block lemma. The spec_is_incremental lemma captures the key correctness condition and ties all of the specification functions together. It relies on a helper, split_at_last, which was carefully crafted to subsume the different behaviors between Blake2 and other block algorithms.

\begin{verbatim}
let split_at_last (block_len: U32.t) (bseq U8.t) =
let n = length b / block_len in
let rem = length b % c.block_len in
\end{verbatim}
type block = | Block:
  km: key_management (\( \text{km = Runtime} \lor \text{km = Erased}\)) →
  (\( \text{Low-level types}\))
  state: stateful →
  key: keyful →

  (\( \text{Introducing a notion of blocks and final result}\))
  max_input_length: x:nat [0 < x ∧ x < pow2 64] →
  output_len: x:U32.t \{U32.v x > 0\} →
  block_len: x:U32.t \{U32.v x > 0\} →

  (\( \text{The one-shot specification}\))
  spec_s: key.t →
  inputs:seq U8.t \{length input ≤ max_input_length\} →
  outputs:seq U8.t \{length output == U32.v output_len\} →

  (\( \text{The block specification}\))
  init_s: key.t → state.t →
  update_multi_s: state.t → prevlen:nat →
  s:seq U8.t \{length s ≤ U32.v block_len\} → state.t →
  update_last_s: state.t → prevlen:nat →
  s:seq U8.t \{length s ≤ U32.v block_len\} → state.t →
  finish_s: key.t → state.t → s:seq U8.t \{length s = U32.v output_len\} →
  update_multi_is_a_fold: ... →

  (\( \text{Central correctness lemma of a block algorithm}\))
  spec_is_incremental: key: key.t →
  inputs:seq U8.t \{length input ≤ max_input_length\} →
  Lemma:
  let bs, l = split_at_last (U32.v block_len) input in
  let hash0 = update_multi_s (init_s key) 0 bs in
  let hash1 = finish_s key (update_last_s hash0 (length bs) l) in
  hash1 == spec_s key input) →
  update_multi:
  s:state.s → prevlen:U64.t →
  blocks:buffer U8.t \{length blocks % U32.v block_len = 0\} →
  len: U32.t \{U32.v len = length blocks ∧ ... (* omitted *)\} →
  Stack unit
  (requires ... (* omitted *))
  (ensures \(\lambda h_0. h_1 →
  \text{modifies}(\text{state.footprint } h_0 s) h_0 h_1 ∧
  \text{state.footprint } h_0 s == \text{state.footprint } h_1 s ∧ 
  \text{state.invariant } h_1 s ∧
  \text{state.v.i } h_1 s == \text{update_multi_s}
  (\text{state.v.i } h_0 s) (\text{U64.v prevlen}) (\text{as_seq } h_0 \text{blocks}) ∧ ... 
  \rightarrow ... (* \text{rest of the block typeclass, e.g. } \text{init}, \text{finish...} *) \rightarrow \text{block}

[@CAbstractStruct]
type state_s (c: block) = | State:
  block_state: c.state.s →
  buf: B.buffer U8.t \{B.len buf = c.block_len\} →
  total_len: U64.t →
  seen: C.\text{Erased} (\text{S.seq } U8.t) →
  p_key: \text{optional_key} c.km c.key →
  state_s c

let state (c: block) = pointer (state_s c)

Figure 5. The streaming algorithm’s state

\textbf{Stateful implementations.} We now zoom in on the update_multi low-level signature, which describes a block’s algorithm runtime processing of multiple blocks in one go (Figure 4). This function is characterized by the spec-update_multi_s; it preserves the invariant as well as the footprint; and only affects the block algorithm’s state when called.

The combination of spec_is_incremental along with the Low\(^{\ast}\) signatures of update_multi and others restricts the API in a way that the only valid usage is dictated by Figure 1. Designing this type class while looking at a wide range of algorithms has forced us to come up with a precise, yet general enough description of what a block algorithm is: we have been able to author instances of this type class for SHA2 (4 variants), Blake2 (2 variants), Poly1305, and legacy algorithms MD5 and SHA1. We plan to significantly extend the set of available instances, adding vectorized variants of Poly1305 and Blake2 to the mix, along with new algorithms such as SHA3.

4 A streaming functor

Equipped with an accurate and precise description of what a block algorithm is, we are now ready to write an API transformer that takes an instance of block, implementing the state machine from Figure 1, and returns the safe API from Figure 2. We call this API transformer a functor, since once applied to a block it generates type definitions for the internal state, specifications, correctness lemmas and of course the five low-level, runtime functions that implement the transitions from Figure 2.

Since F\(^{\ast}\) has no native support for functors, we describe a somewhat manual encoding; \$6\$ shows how to automate this encoding with Meta-F\(^{\ast}\).

The streaming functor’s state is naturally parameterized over a block (Figure 5), and wraps the block algorithm’s state with several other fields.

The CAbstractStruct attribute ensures that the following C code will appear in the header. This pattern is known as “\text{C abstract structs},” i.e. the client cannot allocate structs or inspect private state, since the definition of the type is not known; it can only hold pointers to that state, which forces...
them to go through the API and provides a modicum of abstraction.

```c
struct state_s;
typedef struct state_s *state;
```

First, `buf` is a block-sized internal buffer, which relieves the client of having to perform modulo computations and buffer management. Once the buffer is full, the streaming functor calls the underlying block algorithm's `update_multi` function, which effectively folds the blocks into the `block_state`. The key is optional, and `total_len` keeps track of how much data has been fed so far.

The most subtle point is the use of a ghost sequence of bytes, which keeps track of the past, i.e. the bytes we have fed so far into the hash. This is reflected in the functor's invariant, which states that if we split the input data into blocks, then the current block algorithm state is the result of accumulating all of the blocks into the block state; the rest of the data that doesn't form a full block is stored in `buf`.

```c
let state_invariant (c: block) (h:mem) (s:state c) =  
let s = deref h s in  
let State block_state buffer total_len seen key = s in  
let blocks, rest = split_at_last (U32v c.block_len) seen in  

(* omitted *) ... ∧ c.state.v h block_state ==  
c.update_multi_s c.init_s (optional_reveal h key) 0 blocks ∧  
slice (as_seq h buffer) 0 (length rest) == rest
```

The `mk_finish` function takes a block algorithm `c` and returns a suitable finish function usable with a state `c`. Under-the-hood, it calls `c.copy` to avoid invalidating the `block_state`; then `c.update_last` followed by `c.finish`, the last two transitions of Figure 1. Thanks to the correctness lemma in `c` along with the invariant, `mk_finish` states that the digest written in `dst` is the result of applying the full block algorithm to the data the was fed into the streaming state so far.

```c
val mk_finish :  
c:block → s:state c →  
dst:B.buffer U8.t { B.len dst == c.output_len } →  
Stack unit (requires λh0 → ... (* omitted *))  
(ensures λh0 s’ h1 → ... ∧ (* some omitted *))  
as_seq h1 dst == c.spec_s (get_key c h0 s) (get_seen c h0 s))
```

This particular usage of a ghost variable is actually the third iteration of the streaming API, and the one that we have found easiest to use and be productive with. It allows authoring a `function get_seen` that in any heap returns the bytes seen so far; previously, the user was required to materialize the previously-seen bytes as a ghost argument to `mk_finish`, which incurred a substantial syntactic burden.

This streaming API has two limitations. First, we cannot prove the absence of memory leaks. This is a fundamental limitation of Low*, which cannot show that a malloc followed by a `free` is morally equivalent to being in the Stack effect. However, this can easily be addressed with manual code review or off-the-shelf tools, such as clang's `--sanitize=memory`. The second is that there is still a source of unsafety for C clients: they may exceed the maximum amount of data that can be fed into the block algorithm. This is purely a design decision: since the limit is never less than 261 bytes (that’s two million terabytes), we chose to not penalize the vast majority of C clients who will never exceed that limit, and leave it up to clients who may encounter such extreme cases to perform length-checking themselves.

### 4.1 A note on meta-arguments

We now focus on the usage of meta-level arguments, which act as tweaking knobs to control the shape of the streaming API. Clients can of course choose a suitable block size and suitable types for the block state and key representation, which influences the result of the functor application. But the key management is of particular interest.

```c
noextract type key_management = | Runtime | Erased  
inline_for_extraction  
let optional_key (km: key_management) (key: stateful) : Type =  
match km with  
| Runtime → key.s  
| Erased → Ghost.eraser.key.t
```

The `km` parameter of the block type class is purely meta, and will never be examined at run-time. It allows the block algorithm to indicate whether it needs a key. In the streaming code, every reference to key goes through a wrapper like the one above. After partial application, the `optional_*` wrappers reduce to either a proper key type, or to a ghost value, which then gets erased to unit, which means the key field of the state entirely disappears thanks to KreMLin's unit field elimination. The streaming API's `init` function unconditionally takes a key at run-time; but for algorithms like hashes, it suffices to pick `c.key.s = unit` and the superfluous argument to `init` gets eliminated too.

### 4.2 Runtime agility

In reality, the entire type class is parameterized over an index, omitted here for conciseness. The index is used ghostly, except for the `init` function. This allows doing run-time agility, e.g. by having a streaming API for any hash algorithm; the state then becomes a state where a is the chosen hash algorithms, and every single definition we have seen becomes agile over the choice of a. Using this, we trivially re-implement EverCrypt's old incremental hashing module, making it a mere application of the streaming functor, where the index allows choosing a particular hash algorithm at `init-time`.

### 5 From high-level functors to Low* code

We now turn our attention to extraction, and explain how to carefully tweak the streaming functor so that, once applied
to a given block algorithm, it yields first-order, specialized code that fits in the Low\(^*\) subset that can compile to \(\text{C}\).

In this section, we use \(\text{F}^*\) flags and attributes without resorting to Meta-\(\text{F}^*\) tactics. While we use the streaming functor as our running example, the techniques are systematic and can be applied to any hand-written functor in \(\text{F}^*\).

5.1 Specialization via partial evaluation
We now show how to use both the \texttt{inline_for_extraction} keyword and the \texttt{[@inlinelet]} attribute (§2) to ensure that, after \(\text{F}^*\) has performed its extraction-specific normalization run, no traces are left of the functor argument, and the resulting code only contains first-order Low\(^*\) code. In other words, we completely eliminate accesses to the type class dictionary via partial evaluation, meaning no run-time overhead.

We focus on \texttt{mk\_finish}, the streaming API’s finish function (Figure 6). The function contains numerous patterns that need to be inlined away, which makes it representative of the streaming functor as a whole.

The let-binding at line 2 serves only to bring the associativity lemma from the type class into the scope of the SMT context, so that its associated pattern can trigger and save the programmer from having to call the lemma manually. The usage of \texttt{[@inlinelet]} eliminates this partial application. We use \texttt{[@inlinelet]} in numerous other places in the streaming functor, to generate cosmetically more pleasant \(\text{C}\) code.

Built-in constructs of Low\(^*\) such as lines 4 and 7 receive special treatment in the toolchain and are eliminated. Calls to lemma and assertions at line 9 and 13 have type \texttt{Ghost unit}. The KreMLin compiler eliminates superfluous units, so these disappear too.

At lines 15-16, we need to copy the block state into a temporary, in order to obtain the state machine mentioned earlier (Figure 2). The syntax hides nested calls to projectors of the block and stateful type classes respectively. To make sure these reduce, we mark both type class definitions as "inline for extraction", which in turns makes their projectors reduce. At extraction-time, provided \texttt{mk\_finish} is applied to an instance, lines 15-16 reduce into direct calls to the original alloca and copy functions found in the type class.

At line 11, we call rest, a helper shared across multiple functions that returns the amount of data currently in the internal buffer. As such, rest needs access to the type class, if only to know the block algorithm’s block size. To avoid generating a run-time access to \(\text{c}\) in the call to rest, we mark the definition of rest as “inline for extraction”; provided rest undergoes the same treatment we described above, all references to \(\text{c}\) are now eliminated at extraction-time.

5.2 Recovering ML polymorphism
Playing with normalization attributes and keywords guarantees that any reference to the type class argument disappears; but this is not enough to make the code valid Low\(^*\). The issue lies with our earlier type definition (Figure 5).

While the projection \texttt{c.block\_len} is innocuous, and disappears (redefinitions are erased), the type of \texttt{block\_state} is an application of a type-level function (the projector) to the type argument \(\text{c}\). Upon seeing such a type, \(\text{F}^*\)’s extraction simply inserts \(\top\). This means that a partial application of \texttt{state\_s} to a class \(\text{c}\) will generate:

```plaintext
type c state\_s = { block\_state: Obj\_t; ... }  
type c\_state = unit state\_s
```

Indeed, the partial application of an inductive does not trigger partial evaluation. \(\text{F}^*\) does not generate a fresh, specialized copy of an inductive when it encounters a partial application. All seems lost, for a field \texttt{block\_state: t} cannot be compiled to \(\text{C}\). We can, however, regain ML-like polymorphism with this "one simple trick":

```plaintext
noeq type state\_s (c: block) (t: Type0 { t == c.state\_s }) = | State:  
  block\_state: t \rightarrow  
  buf: B.buffer U8.t { B.len buf = c.block\_len } \rightarrow  
  ... \rightarrow state\_s c
```

This second version is curiously convoluted; from the point of view of \(\text{F}^*\), however, this is a perfectly valid ML type whose definition in the ML AST becomes:

```plaintext
type c 't state\_s = { block\_state: 't; buf: ... }
```

We apply a similar trick to \texttt{functions}, where for instance the prototype of \texttt{mk\_finish} becomes:

```plaintext
val mk\_finish: c.block \rightarrow t:Type0 { t == c.state\_s } \rightarrow s:state c t \rightarrow ...
```

This extracts to an ML AST that is free of casts, since all types within the body of \texttt{mk\_finish} are now of type \(\text{t}\) (a type parameter of the function) instead of \texttt{c.state\_s} (a non-extractable function call at the \texttt{Type0} level). We obtain an ML-polymorphic definition, along with monomorphic uses:
6 Tactic-based metaprogramming
The process of hand-writing a functor (§4) gives the programmer fine-grained control over reduction and type monomorphization. However, this manual encoding requires the programmer to mark the entire call-graph as "inline for extraction", in order to properly eliminate occurrences of the meta-argument `c` in the generated code. In many situations, this is not acceptable: a huge blob of code would not pass muster with software engineers who want to use verified libraries, and as such we need to retain the structure of the call-graph in the generated C code.

6.1 General-purpose meta-level indices
We now abstract over, and generalize the setting of §4. We consider call-graphs of arbitrary depth, where the only restriction is the absence of recursion. This is a safe assumption: most `Low*` code uses loop combinators, as recursion results in unpredictable performance, owing to the uneven support for tail-call optimizations in C compilers.

We assume every function in the call-graph is parametric over a meta-parameter, which we call from here on an index. In order for the functions in our call-graph to be valid `Low*`, they must be applied to a concrete index in order to trigger enough partial evaluation.

In §4, the meta-parameter was the type class `c`, which contained type definitions followed by specifications, lemmas, low-level implementations, helper definitions, etc. This style is burdensome, as a large algorithm will typically incur a type class with dozens of fields, which makes authoring instances tedious and non-modular. We now present a different style, which we have found minimizes syntactic overhead, and is easier to work with in day-to-day proof engineering.

For the rest of this paper, we choose for the index a finite enumeration, accompanied with a set of helper definitions over that index. To illustrate that second style, we use HPKE [10] (Hybrid Public-Key Encryption), a cryptographic construction that combines AEAD (Authenticated Encryption with Additional Data), DH (Diffie-Hellman) and hashing.

We wish to generate specialized instances of HPKE for a given triplet of implementations. Using C++ as an analogy, we wish to author the equivalent of template `HPKE<typename AEAD, typename DH, typename Hash>);

```c
typedef struct {
  uint32_t block_state;
  uint8_t *buf;
  uint64_t total_len;
} Hacl_Streaming_Functor_state_sha2_256;
```

The index `Spec.HPKE.alg` captures all possible algorithm choices prescribed by the HPKE RFC. We thus write specifications, lemmas, helpers and types parametrically over the index as standalone definitions. The key `aead.type`, for example, is parametric over triplets of algorithms, and defines a low-level key to be an array of bytes whose length is the key length for the chosen AEAD. The same systematic parametrization over `alg` can be carried to functions and their types, shown with `sign` as an example. An important point is that for a given `sign alg`, there may be multiple implementations of the given algorithm. So, the finite combinations for `Spec.HPKE.alg` admit an infinite number of implementations.

6.2 Call-graph rewriting
Figure 7 describes the call-graph of HPKE: nodes are `F*` top-level functions and arrows indicate function calls. The helper node makes verification modular but would pollute the generated C code, and therefore must be evaluated away. All other circled nodes must appear in the generated C code.

This call-graph is representative of a typical large-scale verification effort: in order to make verification robust in the presence of an SMT solver; to ensure modularity of the proofs; and to increase the likelihood that a future programer can understand and maintain a given proof, we encourage a proliferation of small helpers with crisp pre- and post-conditions. However, these helpers would typically amount, if they were extracted, to a mere line or two of C code.

We now wish to obtain a copy of this call-graph where helper has been inlined away, and where all functions in the generated call-graph are specialized variants for a given triplet of algorithms. We proceed in two steps: first, we rewrite the call-graph to look as follows:

```c
let mk_finish (type c t) (s: (c, t) state) ... = ...
let finish_sha2_256 = mk_finish <unit, U32 t buffer>
```
Figure 7. Simplified call-graph of HPKE

```
inline_for_extraction let helper (alg: Spec.HPKE.alg)
  (sign: sign_t alg): helper_t alg = λ... → ...

inline_for_extraction let hpke (alg: Spec.HPKE.alg)
  (sign: sign_t alg) (enc: enc_t alg): hpke_t alg = λ... → ...
  helper alg sign ...
  ... enc ...
```

This form is convoluted: we have recursively parameterized the hpke to accept *specialized* versions of all the functions that we wish to retain in the call-graph after partial evaluation. Then, we let the user perform instantiations to obtain a specialized HPKE algorithm:

```
// ChachaPoly.AVX2.fstl:
val enc: enc_t Spec.AEAD.ChachaPoly

// HPKE.ChachaPolyl.fstl:
let hpke_chachapoly: hpke_t (..., Spec.AEAD.ChachaPoly, ...) =
hpke (..., Spec.AEAD.ChachaPoly, ...) ChachaPoly.AVX2.enc
```

This results in a *specialized* version of hpke, which calls a *specialized* version of encrypt. The index is gone and there is no run-time overhead; we have in effect specialized the entire call-graph for a specific value of the index. Note that we can provide many more specializations of HPKE for the same value of the index, e.g., with the AVX512 version of ChachaPoly. Back to our earlier C++ template analogy, we have in effect written: `HPKE<..., ChachaPolyAvx, ...> Hpke_ChachaPoly`

6.3 Equational rewriting by tactic

We now formalize the call-graph specialization logic as a set of rewriting rules (Figure 8), which are to be understood as follows. The user annotates with an attribute `af` every function `f` in the call-graph, where `af` is either `[@ Eliminate]` or `[@ Specialize]`. Un-annotated functions are understood to be outside the call-graph and are ignored.

We use `f → g` to state that `f` calls into `g`. We define `spec(f)` to be all the functions annotated with `[@ Specialize]` that are called by `f` through `[@ Eliminate]` functions. This set never contains `f`, since we ruled out recursion. One property of interest is:

\[
\text{spec}(g) ⊂ \text{spec}(f) \text{ if } f → g \quad (p)
\]

In rule (i), each function `f` is rewritten to take as extra parameters specialized versions of all the functions `f_i` it might (transitively) call into. When such a specialized function call is encountered in (iii), it is rewritten into a call to the specialized variant the function received as a parameter. Note that the index disappears: the parameter `f_i` is of type `t_f i`, i.e., it is a *specialized* instance of `f` for the current index `i`.

Calls to functions that are to be eliminated (ii) are rewritten differently: since they disappear from the call-graph, we rely on the "inline for extraction" attribute to inline their definitions away. They do take, however, extra arguments, for all the specialized functions they eventually call: we pass those as well, which are always bound thanks to (p).

6.4 Implementation in Meta-F*

We have implemented these rewriting rules as a recursive traversal of the call-graph. Our tactic, at 620 lines, (including whitespace and comments) is the second largest Meta-F* program written to date. We now briefly give an overview of the implementation. The tactic is written in Tac, the effect of meta-programs. As mentioned earlier (§2), the design of Meta-F* means any fresh term generated by a tactic must be re-checked for soundness.

The tactic is written in a state-passing style, as meta-programs do not have access to mutable state, and revolves around the following internal definitions:

```
noeq type mapping =
  | Eliminate new_name:name → mapping
  | Specialize: mapping

let state = list (name & (term & mapping & list name))
```

The state type is an associative list that to each `f` (of type `name`) associates: its type `t_f` (of type `term`, the safe view of terms exposed to meta-programs); its `af` and new name `g` (of type `mapping`); and its set `spec(f)` (of type `list name`).

The core of our tactic is `visit_f`, which returns an updated state along with a set of fresh definitions to be inserted into the current module. In order to call our tactic, the user passes the roots of the call-graph traversal, along with the type of the index. The `@specialize` directive inserts meta-generated definitions at the current point, and requires the user to pass the `names` of all the specialized nodes that they wish to call later, in order to establish a lexical scope (scope resolution happens before meta-program evaluation in F*).

```
% [Meta.Interface specialize (Spec.HPKE.alg) {
  Impl.HPKE.setupBase; Impl.HPKE.setupBaseR;
  Impl.HPKE.sealBase; Impl.HPKE.openBase;
}]
```

One possible specialization, out of more than a hundred possible options, is P256 for elliptic curve DH, AVX 128-bit ChachaPoly for AEAD and SHA256 for hashing. Note how
We now present the application of our automated tactic to a particularly gnarly example. Curve25519 is an elliptic curve algorithm [12]; suffice to say that it relies on a mathematical field, which admits two efficient implementations; furthermore, one of these two implementations relies on a core set of primitives (e.g. multiplication) which themselves admit two different implementations, one in Low* from the HACL* library, and one in Vale assembly [21].

In Figure 9, circles denote interfaces. We define the index as follows, along with some helpers for the low-level representation of field elements:

```
type field_repr = | Field51 | Field64

let felem (s: field_repr) = ...
```

We then make sure the whole Curve25519 module is written against an abstract Field.fsti, where all definitions are polymorphic over the index; Field.fsti captures the signature to be provided by field implementations. This relies on the extra feature mentioned earlier, where our tactic stops at abstraction boundaries.

As an example, Field.fsti contains signatures that are implemented by individual field implementations, such as Field51.fsti:

```
(* Field51.fsti *)
[@ Specialize ] val store_felem : #s:field_repr ->
  lbuffer U64 4ul -> f:felem s -> Stack unit ...

(* Field51.fsti *)
let store_felem (u64s:lbuffer U64 4ul) (f:Field51.felem) Stack unit... = ...
```

We apply this pattern once more for Field64.fsti, which is itself written against an abstract Core64.fsti:

```
(* Helpers.fsti *)
inline_for_extraction let fadd_t (s:field_spec) =
  out:felem s -> f1:felem s -> f2:felem s -> Stack unit ...

(* Core64.fsti *)[@ Meta.Attribute.specialize] val fadd: fadd_t Field64
(* Core64.Vale.fsti *) let fadd: fadd_t Field64 = ...
```

This relies on another extra feature mentioned earlier, namely the ability to have, at the leaves of the call-graph, functions that omit the index if only one case is possible.

We obtain three specialized applications. One would be, in OCaml, module Curve51 = Curve25519(Field51), while another would be module Curve64Vale = Curve25519(Field64(Core64)). Unlike in OCaml, though, our functors reduce via partial evaluation and incur no run-time overhead.
7 Evaluation

7.1 Streaming functor
To evaluate the applicability of the streaming functor, we compare lines of code (LoC) for the F* source code and the final C code as a proxy for programmer effort. Our point of reference is a first, non-generic streaming API that previously operated atop the EverCrypt agile hash layer.

Table 1 presents the evaluation. For the old, non-generic streaming API, the proof-to-code ratio was 1.11, meaning we had to write more than one line of code in F* for every line of generated C code.

Capturing the block API and implementing the functor uses 1505 lines of F* code. The extra verification effort is quickly amortized across the 10 applications of the streaming functor, which each requires a modest amount of proofs to implement the exact signature of the block API. Poly1305 and Blake2 were originally authored without bringing out the functional, fold-like nature of the algorithms, which led to some glue code and proofs to meet the block API. Altogether, we obtain a final proof-to-code ratio of 0.83, which we interpret to coarsely mean a 34% improvement in programmer productivity. We expect this number to further decrease, as more applications of the streaming functor follow.

For execution times, we present the verification time of the functor itself, and the verification time of each of the instances, including glue proofs. Compared to fully verifying Blake2 (7.5 minutes), or Poly1305 (~14 minutes), the verification cost is modest. Applying the streaming functor to a type class argument incurs no verification cost, so the extraction column measures the cost of partial evaluation and extraction to the ML AST, which is negligible.

7.2 Tactic
From a qualitative point of view, the usage of the call-graph rewriting tactic significantly improved programmer experience and addressed many fundamental engineering roadblocks in one go. First, programmers would tweak F*’s include path to switch between implementations (e.g. Field51 vs Field64), effectively making it impossible to build verified applications on top of HACL*. Second, the lack of modularity and call-graph specialization in old versions of HACL* made C and vectorized implementations appear in the same file; since we had to use -maxv -maxv2 for compiling instrics, the C compiler would use AVX2 instructions for our non-vectorized, regular C version, causing illegal instruction errors later on [39]. We now put one tactic instantiation (i.e. one %splice) per file, which solves the problem definitively. Finally, previous versions of HACL* did not distinguish between algorithmic agility and choice of implementation for a given algorithm. This made a modular and specializable HPKE just impossible to author.

From a quantitative point of view, we measure the overhead incurred by re-checking the tactic-generated call-graph, relative to the total verification time for a given algorithm. In most cases, the overhead is < 100%, because we don’t rewrite lemmas and proofs. Curve25519 is an outlier because we thread a precondition, resulting in additional verification burden. In practice, build time matters little in the face of improved programmer productivity.

7.3 Usability of tactics
Tactics are not part of the trusted computing base (§2); unlike, say, MTac2 [23], Meta-F* [26] does not allow the user to prove properties about tactics, trading provable correctness for ease-of-use and programmer productivity.

The debugging experience for tactics is thus pleasant. If the tactic itself fails, F* points to the faulty line in the meta-program. If the generated code is ill-typed, we examine it like any other F* program. We debugged this tactic on Curve25519; once debugged, the tactic never generated ill-typed code and was used successfully by other co-authors.

A technical detail relates to lexical scoping (§6.3): the user must somewhat materialize in the source code the names of all the g’s that are generated by the tactic, in order to establish proper lexical scope. For that, regular users can observe the standard output, where the tactic prints a summary of the functions it generated, their types, and their names. Equipped with the names of the generated g’s, users then edit their source code to fill in the first argument to %splice. Alternatively, power users just look up the mangling scheme and directly write arguments to %splice in one go.

8 Future work
Based on our experience performing very large-scale verification in F*, we have shown an array of techniques to make program proof a productive endeavor. Establishing clear, crisp abstractions that highlight commonality between related pieces of code sets a foundation for higher-level APIs. With partial evaluation and meta-programming, programmers can think at the highest levels of abstractions, while
minimizing effort and increasing code reuse. Quantitative evaluations provides evidence that our techniques improve programmer experience. In our view, scaling software verification is just as important as verification through tours de force.

We intend to continue our efforts on the HACL* codebase. An immediate goal is to extend the streaming functor to take a meta-level parameter that allows storing $n$ blocks in the internal buffer, which is essential for vectorized implementations that process multiple blocks at a time. We also intend to author a new type class to capture the commonality between Chacha20, AES, SHA3 and Blake2 in PRF mode and the matching flavors of AEAD, along with a corresponding functor to automatically generate safer APIs for those.
References

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

[2] Federal Information Processing Standards Publication 180-4: Secure hash standard (SHS), 2012. NIST.

[3] Daniel Ahman, Cátlin Hrítcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In ACM Symposium on Principles of Programming Languages (POPL), January 2017.

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. 2017.

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

[6] Nada Amin and Karthik Rompf. Lms-verify: Abstraction without regret for verified systems programming. In POPL, 2017.

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

[8] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O’Hearn, and Christian Winnerlein. BLAKE2: Simpler, smaller, fast as MD5. In Applied Cryptography and Network Security, pages 119–135, 2013.

[9] Manuel Barbosa, Gilles Barthe, Karthikeyan Bhargavan, Bruno Blanchet, Cas Cersmers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. IACR Cryptol. ePrint Arch., 2019:1393, 2019.

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

[11] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of OpenSSL HMAC. TACMAS. 2015.

[12] 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.

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

[14] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, L. de Moura and N. Bjørner. Z3: An efficient SMT solver. 2008.

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

[16] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. Technical report, 2015.

[17] Leonardo de Moura, Soo Hong Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover. In Proceedings of the Conference on Automated Deduction (CADE), 2015.

[18] Frank DeRemer and Hans Kron. Programming-in-the-large versus programming-in-the-small. In Proceedings of the International Conference on Reliable Software, page 114–121, New York, NY, USA, 1975. Association for Computing Machinery.

[19] Morris J Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions. Technical report, 2015.

[20] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. 2019.

[21] Aymeric Fromhner, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. A verified, efficient embedding of a verifiable assembly language. Proc. ACM Program. Lang., 3(POPL), January 2019.

[22] Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Signed cryptographic program verification with typed combinators. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS ’19, page 1391–1400, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Jan-Oliver Kaiser, Beta Ziliani, Robbert Kebbers, Yann Régnis-Gianus, and Derek Dreyer. Mtc2: typed tactics for backward reasoning in coq. Proceedings of the ACM on Programming Languages, 2(3):1–31, 2018.

[24] K. Rustan 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.

[25] Pierre Letouzey. A new extraction for coq. In International Workshop on Types for Proofs and Programs, pages 200–219. Springer, 2002.

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

[27] David A. McGrew and John Viega. The security and performance of the Galois/counter mode of operation. In Proceedings of the International Conference on Cryptology in India (INDOCRYPT), 2004.

[28] Robin Milner. A theory of type polymorphism in programming. Journal of computer and system sciences, 17(3):348–375, 1978.

[29] Nicky Mouha, Mohammad S Rana, D Richard Kuhn, and Raghu Kacker. Finding bugs in cryptographic hash function implementations. IEEE transactions on reliability, 67(3):870–884, 2018.

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

[31] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromhner, Natalia Kulatova, and Santiago Zanella-Béguelin. Hacl!+n: Verified generic simd crypto (for all your favorite platforms). Cryptology ePrint Archive, Report 2020/572, 2020. https://eprint.iacr.org/2020/572.

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

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

[34] M-J. Saarinen and J-P. Aumasson. Blake2 cryptographic hash and message authentication code (mac). IETF RFC 7693, 2015.

[35] Amir Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. Lisp and symbolic computation, 63(4):289–360, 1993.

[36] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, Haskell ’02, page 1–16, New York, NY, USA, 2002. Association for Computing Machinery.

[37] Nikhil Swamy, Cátlin Hrítcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and monadic effects in F*. In Proceedings of the ACM Conference on Principles of Programming Languages (POPL), 2016.

[38] The Coq Development Team. The Coq Proof Assistant Reference Manual, version 8.7, October 2017.

[39] Guido Vranken. Cryptofuzz. https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=1461, May 2019.
[40] Stephen Weeks. Whole-program compilation in milton. *ML*, 6:1–1, 2006.

[41] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. 2017.