# Verified Transformations and Hoare Logic: Beautiful Proofs for Ugly Assembly Language

Jay Bosamiya[1], Sydney Gibson[2], Yao Li[3], Bryan Parno[1], and Chris Hawblitzel[4]

[1] Carnegie Mellon University
[2] Massachusetts Institute of Technology
[3] University of Pennsylvania
[4] Microsoft Research

**Abstract.** Hand-optimized assembly language code is often difficult to formally verify. This paper combines Hoare logic with verified code transformations to make it easier to verify such code. This approach greatly simplifies existing proofs of highly optimized OpenSSL-based AES-GCM cryptographic code. Furthermore, applying various verified transformations to the AES-GCM code enables additional platform-specific performance improvements.

## 1 Introduction

Some of the most important code in the world is also some of the ugliest. The most commonly used implementations of cryptographic algorithms are heavily optimized, typically employing hand-crafted assembly language for maximum performance. For example, OpenSSL's implementation of AES-GCM, the cryptographic algorithm used for 91% of secure web traffic [14], contains thousands of lines of hand-optimized x86-64 assembly language code. The optimizations unroll loops, prefetch data from memory, carefully hand-schedule instructions, and interleave otherwise unrelated instructions in an effort to expose parallelism and keep the processor's functional units busy. The resulting code is extremely fast, but difficult to understand, maintain, and verify.

Recent work on EverCrypt [17] used Hoare logic to verify a variant of OpenSSL's AES-GCM x64 code. Hoare logic is a natural way to express the verification of well-structured programs. Unfortunately, the optimizations in OpenSSL's AES-GCM code obscure the natural structure of the underlying AES-GCM algorithm, making Hoare logic awkward to use directly on the optimized code. In particular, to automate the proofs, it helps to keep code units relatively small, since that keeps the proof "debug" cycle tolerable for developers. However, the interleaving of unrelated instructions makes it difficult to modularly decompose the code into smaller units with natural preconditions and postconditions. As a result, the preconditions and postconditions describe situations where natural invariants do not yet hold or have already been broken. Worse, each repeated section of code generated from loop unrolling has to be verified separately, because the instruction scheduling and interleaving cause each section to contain

slightly different code with slightly different preconditions and postconditions. The ugly code leads to ugly proofs and duplicated effort.

This creates a stark trade-off. On one hand, the performance gains from carefully optimized code are enormous and valuable: the verified code based on OpenSSL's optimized code runs 6× faster than earlier verified code written in a simpler, easier-to-verify style [7]. On the other hand, the effort involved in verifying optimized code may dissuade authors of cryptographic code from attempting any formal verification.

We argue that the trade-off is not as stark as it may seem at first glance:

- First, we demonstrate how to use verified transformers to recover the elegance of Hoare logic. In this approach, the programmer uses Hoare logic to verify a clean, modular version of the code. In addition, the programmer writes (but does not directly verify) the optimized, non-modular version of the code. Our verified transformation tool then attempts to automatically discover the relationship between the clean and optimized versions and prove their equivalence. This proves that the properties established via Hoare logic for the clean code apply to the optimized code.
- Second, we manually create a clean, modular version of EverCrypt's AES-GCM code and measure its performance. To our surprise, on some CPUs, the clean code actually runs slightly *faster* on average than the original EverCrypt code. In other words, not all of OpenSSL's optimizations are equally necessary to achieving its fast performance, and the optimization that causes the most trouble for EverCrypt's verification does not appear to pay off consistently.
- Third, inspired by the observed performance difference between the clean code and EverCrypt code, we investigate the performance of alternate interleavings of the assembly language instructions for various x86-64 processor models. We develop a tool that automatically finds interleavings that are faster than both the EverCrypt code and the clean code, and we use our verified transformation tool to verify the correctness of these new interleavings. Hence, verified transformers support automated development of hyper-targeted optimized implementations while still allowing the developer to write beautiful, Hoare-style proofs.

The rest of the paper is as follows. Section 2 presents background on the Vale language and tool [5,7], which provides the operational semantics and Hoare Logic reasoning for our assembly language code. Section 3 presents our verified transformation tool and describes how it deals with subtle equivalence issues, such as assembly language status flags. Section 4 applies the tool to an important real-world case study: OpenSSL's optimized AES-GCM. Section 5 shows that our tool can verify alternate interleavings of OpenSSL's code that are faster than the original code. Section 6 compares to related work, including related verification of cryptographic code such as Fiat-Crypto [6] and Jasmin [1,2]. Section 7 concludes with recommendations for verifying optimized code.

All of our code and proofs are available online, under an open source license.[5]

[5] `https://github.com/project-everest/hacl-star/tree/_vale_unstructured/vale`

## 2 Background: Vale and Assembly Language

In order to verify x64 code for AES-GCM, previous work [7,17] defined syntax and operational semantics for x64 instructions as F* [20] datatypes and functions. Below, we provide a representative sampling of these definitions.

```
// Instruction syntax and semantics, defined in F*
type reg = Rax | Rbx | Rcx | Rdx | ...
type operand =
  | OConst: n:int -> operand
  | OReg: r:reg -> operand
  | OMem: m:mem_addr -> operand
type ins =
  | Mov64: dst:operand -> src:operand -> ins
  | Add64: dst:operand -> src:operand -> ins
  ...
type code =
  | Ins: ins:ins -> code
  | Block: block:list code -> code
  ...
type state = {
  ok:bool;
  regs:reg -> nat64;
  flags:nat64;
  mem:map int nat8;
}
let eval_ins (ins:ins) =
  ...
  match ins with
  | Mov64 dst src -> ...
  | Add64 dst src -> ...
  ...
let rec eval_code (c:code) (s:state) ... =
  match c with
  | Ins ins -> Some (run (eval_ins ins) s)
  ...
```

Here, the big-step operational semantics defined by `eval_ins` and `eval_code` evaluate the effects of assembly language instructions on some state, producing a new state as a result. The state tracks the values in registers (`regs`), the CPU status flags (`flags`), and the memory (`mem`). An additional state field `ok` indicates whether execution has succeeded or crashed. (This description is simplified for clarity; the full F* implementation of `state` also includes multimedia (xmm) registers, a stack, and a more complex memory model.)

The previous work then used the Vale tool [5,7] to build a Hoare logic on top of the F* syntax and semantics, building the Hoare logic as verified rules on top of the operational semantics, so that the operational semantics in F* were part of the trusted computing base but the Hoare logic and the Vale tool did not need

to be trusted. The Vale language uses procedures with modifies, requires, and ensures clauses to express the Hoare logic semantics of instructions like `Add64` and to build more complex procedures on top of instructions:

```
// Two example Vale procedures
procedure Add64(inout dst:dst_opr64, in src:opr64)
    modifies efl;
    requires src + dst < pow2_64;
    ensures dst == old(dst + src);

procedure Test()
    modifies efl; rax;
    requires rax < 100;
    ensures rax == old(rax + 2);
{
    Add64(rax, 1);
    Add64(rax, 1);
}
```

In this work, we leverage the distinction between operational semantics in F$^\star$ and Hoare logic in Vale to define verified transformers (Section 3) that translate between idealized, structured code and optimized, unstructured code. Specifically, we first use Vale's Hoare logic to verify the idealized code. Since the Hoare logic rules are already built on top of the operational semantics, this gives us a proof about the idealized code in terms of the operational semantics. We then define verified transformers in terms of the x64 syntax and operational semantics, without having to modify the Hoare logic. These verified transformers prove that the operational semantics for the idealized code is equivalent to the operational semantics for the optimized code.

The transformers work by comparing the idealized code to the optimized code, where both versions of the code are expressed as an F$^\star$ datatype, as in the `ins` datatype shown above. Since the code is a datatype, the transformers can inspect the code by pattern matching on the datatype. For better modularity, though, we used a slight variant of this approach: we refactored the `ins` type to be a more general dependent type that contains an arbitrary number of input and output operands and a function that computes the values for the output operands from the values from the input operands. With this, the transformer only has to match on a small number of general instructions rather than matching separately on `Mov64`, `Add64`, etc. (For the most part, only the input and output operands matter; whether an instruction adds numbers, subtracts numbers, or just moves numbers is usually irrelevant to the transformations.)

## 3 Verified Code Transformers

It is easier to write beautiful proofs about modular code. Our goal is to enable such proofs even for high-performance ugly code. We achieve this by designing a collection of verified code transformers which allow the developer to write

proofs about the modular code and then apply one or more transformers to automatically produce the high-performance ugly code. By proving that each transformer preserves the semantics of the original code, we ensure that the results of the elegant proofs carry over to the ugly code.

Below, we describe the core workflow for a developer using these transformers (Section 3.1), details about their design, implementation, and verification (Section 3.2), as well as several transformers (Section 3.3) which have significantly improved the modularity of the proofs for AES-GCM, a case study we describe in Section 4.

### 3.1 Developer Workflow

To make use of verified code transformers, a developer first writes a clean, modular version of their code, and writes proofs about it. Next, they write, but prove nothing about, a performance-optimized version of their code. This can be based on existing code (e.g., OpenSSL's), their own intuition as to what will maximize performance for a particular architecture, or even an automated empirical search (see Section 5). Following this, the developer adds simple, high-level annotations to indicate which transformations (e.g., register re-allocation or instruction shuffling) they believe will convert their modular code into the high-performance version. At this point, the transformers take over: An untrusted tool first deduces a collection of hints necessary to apply a given transformation (e.g., which permutation should be applied to reorder the instructions). Next, a verified transformer uses the hints to validate the proposed transformation, and if successful, performs it. If unsuccessful, then the transformer indicates to the developer why it was unable to automatically perform a safe transformation.

As an example of using the workflow to verify the high-performance but ugly code `foo_ugly`, a developer first annotates `foo_ugly` with the attribute `{:codeOnly}` which indicates that no proofs have been written (yet) about it. Next, they mark the cleaner code `foo` (which they have proven against its Hoare logic spec) with the attribute `{:transform T, foo_ugly}`. This indicates that they wish to apply the `T` transformation to map `foo` to `foo_ugly`. These top-level annotations are the only ones the developer needs to supply, and the transformers automatically recursively apply themselves to internally called Vale procedures. Vale then replaces the code and proofs of `foo` with the result of applying the transformer (i.e., the code of `foo_ugly`), as well as automatically generated proofs derived from the original proof for `foo` and the generic proof for the transformer `T`, that show that the transformed code satisfies the same pre- and post-conditions as `foo`. Thus, any callers of `foo` obtain the useful Hoare conditions for `foo`, but they also transparently obtain the higher performance of `foo_ugly`.

### 3.2 Proving a Code Transformer Correct

To support the workflow described above, we design our code transformers with an eye towards automation and one-time verification effort. In particular, we

ensure that our transformers are provably guaranteed to preserve the semantics of the original code, and we structure each transformer as a combination of an untrusted front-end that *finds* the necessary transformation steps and describes them via a series of hints, and a verified back-end that checks the proposed transformation and then performs it.

We define two blocks of code to be semantically equivalent in the standard way; i.e., if and only if starting from valid equivalent initial states, both execute to equivalent final states, i.e., roughly:

```
let semantically_equivalent (c1 c2:code) =
  (forall (s1 s2:state).
     equiv_states s1 s2 ==>
       equiv_states (eval_code c1 s1)
       (eval_code c2 s2))
```

Two states are defined to be equivalent if and only if they are pairwise equal on all of their observable projections. That is, their registers are equal, values in memory are equal, flags are equal, etc. We can then define a verified code transformer as a total function that takes code (and possibly some auxiliary data, called "hints") as input and produces code that is semantically equivalent to the original code. By allowing for untrusted hints, we follow a de Bruijn structure that allows us to use arbitrarily complicated algorithms for finding transformations, without needing to prove anything about those algorithms. Additionally, by choosing the correct representation for the hints (which can be different for each transformer), we can allow for highly expressive control over the transformer.

We describe more transformers below, but as a simple example, it is easy to see why a no-op transformer (i.e., a transformer that simply returns its input) is a verified code transformer, albeit a trivial one. It may still be practically useful, however, when integrated with a more complex transformer that might fail, since the no-op transformer can be invoked on failure and hence produce an overall total transformer. Keeping the transformers total makes it easier to stay within the pre-existing Vale framework, without needing special handling for transformed code. We show error messages that may arise during a failure via an additional field added to the internal representation of procedures, which is checked upon extraction.

The code for the transformers is completely untrusted, since their results are proven against the pre-existing Vale semantics. In addition, since the transformers prove the semantic equivalence of their results, Vale's existing correctness lemmas follow simply and immediately.

Finally, as an important security precaution, we ensure that we perform the transformations *before* Vale's verified taint analyzer runs. (The taint analyzer runs a dataflow analysis on the instructions to ensure that the code is free of basic digital side channels [5].) As a result, the taint analyzer runs directly on the final, ugly code. Hence, the transformers do not impact the results of the side-channel analysis.

### 3.3 Example Transformers

Our framework is extensible and many transformers can be written. Here we describe three transformers we developed to support the modularization of proofs for AES-GCM: the *generic peephole transformer* (particularly its instantiation for *movbe-elimination*) searches for a small pattern of instructions and replaces them with equivalent instructions; *control-flow lowering* transforms high-level if/else/while statements into low-level control-flow, and *instruction reordering* reorders instructions to improve run-time performance.

*A Generic Peephole Transformer* A peephole transformation searches for a small pattern of instructions and replaces them with equivalent instructions. For example, a simple peephole transformation might replace all occurrences of `mov {reg},0` with `xor {reg},{reg}`. Peephole optimizations are well studied in the compiler literature [13] and have been verified for CompCert [15]. Here, we implemented a generic peephole transformer which can safely perform such search-and-replace operations in a single pass over the code when provided with an arbitrary replacement pattern that (provably) preserves semantics. As a further convenience, the transformation recursively applies itself to all callees of that procedure too. Since such a replacement is locally semantics-preserving, and the rest of the code remains untouched, we prove that it is also globally semantics preserving.

As a concrete example, we used the peephole transformer to safely refactor pre-existing code, which relies on the `movbe` instruction, to work on older architectures. The `movbe` instruction is a recent addition to the x86-64 architecture (introduced on Atom, supported since Haswell on mainstream Intel processors). It performs an endianness change (i.e., a byte swap) while performing a `mov`. On earlier processor generations, this step is typically implemented via a `mov` and then a `bswap` instruction which performs an in-place endianness change. OpenSSL's version of AES-GCM (and hence the verified EverCrypt version) regularly uses `movbe`, which prevents it from running on older processors that otherwise support the necessary AES extensions.

Hence, we instantiated our generic peephole transformer with a pattern to replace `movbe {dst},{src}` with `mov {dst},{src}; bswap {dst}`. This transformation takes no auxiliary data, and it can be used to automatically update code to work on processor generations without `movbe` support, simply by adding a `{:transform movbe_eliminate}` attribute to the top-level procedure.

Similarly, we instantiated the peephole transformer to allow the insertion of `prefetch` instructions, which act as processor hints to prefetch lines of data into the cache. Our automatic optimization technique (Section 5) uses this transformer to improve performance.

*Control-Flow Lowering* The Vale language supports only structured control flow statements (if/else and while) rather than unstructured control flow. Previously, this structured control flow was built directly into the operational semantics, and Vale's assembly language printer had to be trusted to correctly translate

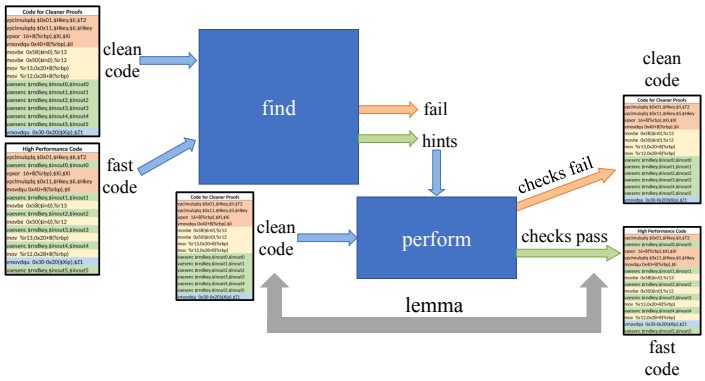


**Fig. 1.** Instruction-Reordering Transformation

these statements into the right labels and conditional branches. To add flexibility and reduce the trusted computing base, we extended the operational semantics to support unstructured control flow. We then wrote a verified transformer to translate if/else and while statements into labels and branches, in the style of certified compilation [11].

This transformer is slightly different from other transformers, in that it is applied to all Vale code, rather than only code that is explicitly user-annotated with a `{:transform ...}` declaration. Additionally, beyond the standard guarantee of semantics-preservation provided by other transformers, this transformer also guarantees preservation of digital traces, and thus strongly ensures maintenance of digital side-channel freedom, independent of whether Vale's verified taint-analyzer is run before or after the transformer.

*Instruction Reordering* Our most powerful workhorse transformer is the instruction reordering transform. As Figure 1 illustrates, this transformer takes as input two code objects (as Vale procedures), and tries to transform one into the other, as long as it is able to do so in a safe way. These code objects correspond to the verification-friendly clean modular code, and either hand-optimized code (for example, from OpenSSL), or automatically optimized candidate code (as described in Section 5). To do this transformation safely, the transformer feeds both code objects into an unverified `find` function, which produces hints that a verified `perform` then validates. If validation succeeds, it applies the transformation to the first code object in order to produce the second, along with a proof of semantic equivalence. If validation fails, it returns the first code object, and provides an informative error message to the developer.

The hints that are sent from `find` to `perform` are of the form "move the $12^{th}$ instruction to the start", "move the $5^{th}$ group of instructions next", etc., which taken as a whole specify a permutation of the original instructions. This supports permutation past Vale procedure boundaries, which allows the clean code to remain modular, despite the lack of clean modularity in the EverCrypt

code. These hints are then validated by `perform` which checks that these moves preserve the semantics of the code. It does so by decomposing moves into a series of swaps, and checking that each swap is allowed, performing it if so. If at any point a swap is disallowed, the transformer immediately stops any further processing, and sends a description of the failure to the user.

In building the transformer, we prove that a swap of two groups of instructions, say $A$ and $B$, is semantics-preserving when the locations written to by $A$'s instructions is strictly disjoint from the locations that are read by or written to by $B$, and vice-versa. That is, there are no read-write or write-write conflicts:

$$\begin{array}{l}(\forall (X, Y) \in \{(A, B), (B, A)\}. \\ \quad (\forall l \in \text{writes}(X). \\ \quad\quad (l \notin \text{reads}(Y) \wedge \\ \quad\quad\ l \notin \text{writes}(Y))))\end{array}$$

Given this proof, `perform` checks for both types of conflicts, and if the checks pass, performs the proposed swap. By proceeding through a series of such swaps, we prove that if `perform` succeeds, then it produces semantically equivalent code. Hence, in combination with `find`, we have a verified transformer that can safely reorder code from a clean, modular form into a user-chosen, performance-optimized ordering.

Unfortunately, allowing swaps only when there are no read-write or write-write conflicts disallows many reorderings that are actually safe. In particular, many instructions on x86-64 affect the flags. Changing the flags is technically a write to a location in the state, and hence any two instructions that modify the flags would be prevented from moving past one another even if the value of the flags they set was never used after their execution. This is a frequent use case in Vale, since the Vale semantics conservatively model most instructions as "havocing" the flags, i.e., setting them to an unrestricted and underspecified value. To overcome this issue, we observe that two underspecified writes are safe to exchange with each other, since their result, while different, is equivalent with respect to any value that can be observed. Thus, if we consider the value of each flag to be a ternary value (e.g., true, false, or unspecified), then we can safely and provably refine our condition for safe swaps as follows: a swap that proposes to move instruction group $B$ in front of instruction group $A$ is semantics-preserving if all of the locations written to by $A$ both do not belong to the locations read by $B$ and either do not belong to the locations that are written to by $B$ or are a constant-write for both $A$ and $B$ and the same constant is written; and vice-versa. That is, write-write conflicts are allowed as long as they write the same constant value:

$$\begin{array}{l}(\forall (X, Y) \in \{(A, B), (B, A)\}. \\ \quad (\forall l \in \text{writes}(X).\ (l \notin \text{reads}(Y) \wedge \\ \quad\quad (l \notin \text{writes}(Y) \vee \\ \quad\quad\quad (l \in \text{constwrites}(X)\ \wedge\ l \in \text{constwrites}(Y) \wedge \\ \quad\quad\quad\ \text{constwrites}(X)[l] = \text{constwrites}(Y)[l])))))) \\ \implies \text{safeswap}(A, B)\end{array}$$

Note that we refer to "group of instructions" above, when performing the moves and swaps. This is because it is convenient to move certain instructions as a coherent block of code to avoid having a proposed swap be rejected by the conservative checks in `perform`. As a toy example, consider two groups of instructions `add rax, 1; adc rbx, 1` and `add rcx, 1; adc rdx, 1`. In both cases, the `add` instruction sets the carry-flag (based on its arguments), and then the add-with-carry instruction (`adc`) reads that flag as part of its addition calculation. It is easy to see that both orderings of the two groups are semantically equivalent (assuming the flags can be ignored after these instructions). However, if one were to naively try to change one into the other using a series of *instruction-only* swaps, then one runs into trouble. The `add`s set up the carry flag specifically for their immediately succeeding `adc`, and simple instruction-only swapping would lead to an ordering that consisted of two `add`s followed by two `adc`s, along the way to actually reordering them. However, if we consider them as groups, then they satisfy the constraints for the swap. A large portion of `find` is thus dedicated to automatically finding groups of instructions to move together, rather than individually. Note that this is where the separation between an unverified `find` and the verified `perform` shines: we can have arbitrarily complicated heuristics for finding good groups of instructions, without needing to write any proofs about the heuristics, since the transformation and hints they produce are validated.

## 4 Verifying AES-GCM

As a case study on the utility of verified code transformations, we apply them to a version of OpenSSL's implementation that was verified in prior work [17].

### 4.1 Background on AES-GCM

AES-GCM is a cryptographic scheme for Authenticated Encryption with Authenticated Data (AEAD). In other words, it protects the secrecy and integrity of a message (e.g., the payload of a network packet) and (optionally) protects the integrity of some additional public information (e.g., a network packet's header). AES-GCM is one of the world's main tools for protecting bulk data, particularly on the Internet, where it is used for 91% of secure traffic [14]. In many of these settings, AES-GCM is on the critical path, since it dictates how quickly data can be read/written.

Given this ubiquity, the world has devoted considerable effort to optimizing the performance of AES-GCM, both in hardware and software. On the hardware side, in 2008, Intel introduced AES-NI [9], a collection of new instructions devoted to accelerating portions of the AES-GCM computation. Even with these hardware instructions, optimal software implementations are non-trivial. For example, OpenSSL's most optimized implementation on x64 involves over 950

```
lea         ($in0,%r12),$in0
vpclmulqdq  \$0x01,$Hkey,$Ii,$T2
vpclmulqdq  \$0x11,$Hkey,$Ii,$Hkey
vpxor       16+8(%rsp),$Xi,$Xi
vmovdqu     0x40+8(%rsp),$Ii
vmovdqu     0x30-0x20($Xip),$Z1
movbe       0x58($in0),%r13
movbe       0x50($in0),%r12
mov         %r13,0x20+8(%rsp)
mov         %r12,0x28+8(%rsp)
vaesenc     $rndkey,$inout0,$inout0
vaesenc     $rndkey,$inout1,$inout1
vaesenc     $rndkey,$inout2,$inout2
vaesenc     $rndkey,$inout3,$inout3
vaesenc     $rndkey,$inout4,$inout4
vaesenc     $rndkey,$inout5,$inout5
```

(a) **A modular version of the code**

```
vpclmulqdq  \$0x01,$Hkey,$Ii,$T2
lea         ($in0,%r12),$in0
vaesenc     $rndkey,$inout0,$inout0
vpxor       16+8(%rsp),$Xi,$Xi
vpclmulqdq  \$0x11,$Hkey,$Ii,$Hkey
vmovdqu     0x40+8(%rsp),$Ii
vaesenc     $rndkey,$inout1,$inout1
movbe       0x58($in0),%r13
vaesenc     $rndkey,$inout2,$inout2
movbe       0x50($in0),%r12
vaesenc     $rndkey,$inout3,$inout3
mov         %r13,0x20+8(%rsp)
vaesenc     $rndkey,$inout4,$inout4
mov         %r12,0x28+8(%rsp)
vmovdqu     0x30-0x20($Xip),$Z1
vaesenc     $rndkey,$inout5,$inout5
```

(b) **Representative snippet of OpenSSL**

**Fig. 2.** *We write proofs about the cleaner, more modular version of the AES-GCM code (a) and then use verified code transformers to connect these proofs to the original OpenSSL code (b). AES operations are highlighted in blue, GCM in green, prefetching and processing of input data in red, and loop control checks in yellow.*

SLOC of Perl scripts [21], which generate 724 SLOC of assembly. Using Perl allows the code to, for example, generate assembly for unrolled loops and customize the registers used in each unrolling. The complexity of these optimizations can lead to concerete security vulnerabilities: In 2013, a performance improvement was added to OpenSSL's codebase, passed all tests, and was on its way into the mainline code when two cryptographers noticed that the optimization would allow an attacker to forge arbitrary messages [8].

Conceptually, computing AES-GCM involves splitting the input into 128-bit blocks, computing AES counter-mode encryption on each block to produce a ciphertext, and finally computing the GCM message authentication algorithm on the resulting ciphertext and any additional authenticated data. A naive implementation written along these conceptual lines is relatively straightforward to verify, but results in performance that is 6× slower than OpenSSL's [17].

Indeed, in its drive for better performance, OpenSSL's implementation merges these conceptual operations so that it need only perform a single pass over the data. It also, when able, processes the input six blocks at a time, so as to make maximal use of available registers. Even at the block level, the individual instructions for performing AES steps are intermixed with those for performing the GCM steps as well as with memory manipulation steps (e.g., loading input data, transforming it into a suitable form to feed to AES, and storing results back to memory), presumably in an effort to keep the processor's functional units fully saturated. The GCM instructions for carryless multiplies and polynomial reductions are carefully ordered to improve parallelism, and various modular reduction steps are delayed to amortize their cost. Individual instructions themselves rely heavily on SIMD operations over 128-bit XMM registers.

As shown in Figure 2b, the result is Perl code that mixes, instruction-by-instruction, conceptually different operations (shown by the various colors).

Unsurprisingly, such code is far more challenging to verify. Prior work [17] relies on SMT solvers and hence is limited in how many instructions can reasonably fit into a single procedure. As a result, they divide OpenSSL's code into smaller units demarcated with Hoare-logic pre- and postconditions. Unfortunately, the intermingled nature of the code makes it difficult to decompose the code in a clean modular fashion, since at any given instruction boundary, the invariants for one conceptual step do not yet hold or have already been broken. The result is large (∼3500 LOC), inelegant proofs, despite the automation provided by the SMT solver (and a custom VC generator [7]).

### 4.2 Verifying AES-GCM via Code Transformations

With the power of verified code transformers at hand, we re-wrote the previously verified, OpenSSL-based AES-GCM code [17], in a clean, modular fashion. Figure 2a shows a representative snippet, where the instructions for each conceptual operation are now grouped, and even within a group, logically similar operations (e.g., `vpclmulqdq`) are themselves grouped together. This reordering already makes the conceptual structure of the code much simpler to see and reason about (e.g., the AES instructions are now more obviously computing six 128-bit blocks in parallel, using the same round key for each block).

Furthermore, with the ability to reorder instructions, we were able to extract the three major functional steps (AES, GCM, and input manipulation) into generic procedures that utilize Vale's *operand parameters* and *inline arguments* to customize each procedure at compile time. Hence, each procedure can be customized at its invocation point to, for example, use a particular register assignment in a given round of the AES computation, as shown in Figure 3. This eliminates the need for custom per-round procedures and dramatically reduces the total amount of code and proofs needed.

For our case study, we applied our transformers to the inner loop of EverCrypt's AES-GCM implementation, where the proof-to-code ratio is 5.0:1 (compared to 2.6:1 in the remaining 3250 lines of proof and code). Overall, the instruction-reordering transformer enabled us to write the inner loop of AES-GCM in 450 lines of code and proof, compared with EverCrypt's version, which required 1250 lines, a nearly 3× reduction. Both versions produce the same approximately 250 lines of assembly code.

Given our clean, modular version and the original ugly EverCrypt version, the instruction-reordering transformer automatically discovered the necessary instruction permutations; the only annotation needed was to specify which transformation we desired.

We subsequently employed our peephole transformer to create custom variants of the code that can run on older platforms that do not support the `movbe` instruction. Also, using our peephole transformer, we show that the `prefetch` optimizations are indeed safe.

```
procedure Loop6x_plain(
  inline alg:algorithm, // inline argument
  inline rnd:nat, // inline argument
  ...
  out rndkey:xmm) // rndkey is an operand parameter
...
{
  Load128_buffer(rndkey, rcx, // rcx is base address
      16 * (rnd + 1) - 0x80, // offset from rcx
      ...);
  VAESNI_enc(inout0, inout0, rndkey);
  VAESNI_enc(inout1, inout1, rndkey);
  VAESNI_enc(inout2, inout2, rndkey);
  VAESNI_enc(inout3, inout3, rndkey);
  VAESNI_enc(inout4, inout4, rndkey);
  VAESNI_enc(inout5, inout5, rndkey);
...
}
...
Loop6x_plain(alg, 0, ..., xmm2);
Loop6x_plain(alg, 1, ..., xmm15);
...
Loop6x_plain(alg, 8, ..., xmm15);
```

**Fig. 3.** Compile-time customization of procedures in our modular AES-GCM code

Finally, since we apply the control-flow lowering transform to all Vale code, we automatically obtain provably-correct unstructured code.

## 5 Optimizing Code for Each Processor Generation

When writing high-performance software, micro-architectural details of a given processor can influence which style of code performs best. In particular, code that is optimized for one processor generation may not perform optimally on another generation of the same processor. Nevertheless, maintaining a different version of the code for each generation of each processor is a daunting task, and hence, even OpenSSL (which supports a wide variety of CPUs with and without various extensions like SSE2, AVX, AES-NI) typically does not go to these lengths to squeeze out additional performance. With the use of verified code transformers, however, we show that we can now safely and automatically produce code that is optimized on a per-generation basis. Hence, verification enables us to reap the rewards of higher performance, in a provably safe way, with marginal extra effort.

The key observation is that having done the work to produce a verified transformer (in particular, the instruction-reordering transformer from Section 3.3),

we can supply it with our clean modular code and *any* unverified code that produces a performance improvement. As long as the transformer accepts that code, we can safely employ it.

Hence, we developed a genetic algorithm to search for faster instruction orderings on a given processor. The algorithm takes as input an initial code object and applies a series of random mutations (shuffling of instructions) to create the first "generation" of candidates. Candidates also are allowed to mutate with a small chance to have random `prefetch` instructions added. Each candidate is sanity-checked for correctness on a single input-output test pair. Candidates that pass this fast sanity-check are then automatically benchmarked on the processor. The fastest candidates then "breed" by combining portions of their mutations (along with a small chance of new random mutations appearing), to produce a new generation of candidates. This process continues looping up to a time or generation bound provided by the developer, at which time the overall fastest candidate across all the generations is returned.

To evaluate the effectiveness of this approach, we have run this algorithm on five x86-64 processors of varying generations, namely Intel's i5-2500, i7-3770, i7-7600U, and i9-9900K, and AMD's Ryzen 7 3700X. For each processor, we experimented with starting the genetic algorithm both from the original EverCrypt version of OpenSSL's hand-optimized assembly, and from our clean, modular version. We took the resulting optimized algorithm from each processor and automatically verified them using the transformers from our cleaned up modular version to confirm semantic equivalence. For the i5-2500 and i7-3770 processors, which do not support the `movbe` instruction that is used in EverCrypt, we applied the movbe-elimination transformer (Section 3.3) before starting optimization. We also applied it to implementations optimized for newer platforms before running them on the older non-`movbe` platforms. As additional context, we also include the (unverified) OpenSSL code that was the basis for the EverCrypt implementation; running this code on the i5-2500 and i7-3770 processors entailed manual modifications to replace `movbe` instructions. We then ran all of these implementations on all of our processors.

The results of these benchmarks are shown in Table 1. Each row of the table corresponds to a different instruction-ordering of the code, while each column corresponds to a different processor. Each cell in the table shows the minimum number of cycles it took to encrypt 4096 bytes of data, with zero bytes of additional data, across 20 million iterations. The smallest value in each column is marked in **bold**, and represents the fastest code for that processor. As the table illustrates, each processor has an optimal ordering, and this optimal ordering can give speedups on top of state-of-the-art OpenSSL or EverCrypt code by up to 27% or 13% respectively.

Overall, our results show that highly targeted code implementations can give non-trivial performance improvements. Further optimizations are, of course, still possible, either via an improved automated search algorithm, or via targeted changes from performance optimization experts. Either way, the verified code transformers conveniently and automatically connect the clean, proven code with

| Code \ Tested on | i5-2500 | i7-3770 | i7-7600U | i9-9900K | 3700X |
|---|---|---|---|---|---|
| Optimized for i5-2500 | **^12957** | ^12560 | ^2454 | **^2378** | ^3492 |
| Optimized for i7-3770 | ^12960 | **^12557** | ^2454 | ^2382 | ^3456 |
| Optimized for i7-7600U | ^14340 | ^13917 | **2450** | 2476 | 3528 |
| Optimized for i9-9900K | ^14382 | ^13941 | 2453 | **2378** | 3528 |
| Optimized for 3700X | ^14127 | ^13696 | 2452 | 2486 | **3168** |
| Clean | ^13632 | ^13222 | 2463 | 2486 | 3420 |
| EverCrypt | ^14619 | ^14198 | 2452 | 2474 | 3492 |
| OpenSSL | *14943 | *14428 | 2986 | 2980 | 4032 |

**Table 1.** Cycle counts for various reorderings on different processors. Code with ^ used the movbe-elimination transformer to run/optimize on older processors. Code with * denotes manual elimination of `movbe` from OpenSSL's interleaved variant.

the optimized versions, ensuring that such optimizations will never violate correctness or security (unlike some previous optimizations attempts [8]).

## 6 Related Work

Although few projects have explored verified translations at the level of assembly language, verified transformations are well understood at higher levels. In particular, verified transformations are the basis for certified compilers such as CompCert [11], which applies repeated translations and optimizations to compiler intermediate languages. The VST project [3] built a Hoare logic on top of CompCert, so that Hoare logic proofs about high-level code imply properties about compiler-generated low-level code too. An example application of VST was a cryptographic primitive (SHA) [4], although compiler-generated cryptographic code often runs slower than hand-optimized assembly language code [5].

To support hand-optimized code, our reordering transformation must examine two existing versions of code and discover the relationship between them. This contrasts with typical verified compilers and optimizers, which are given just one version of the code and can then decide which code to generate.

Translation validation is a pragmatic alternative to compiler verification [16]. In contrast to the latter, which aims to verify that a compiler *always* produces the correct code, this approach verifies that a *particular* compiled code correctly implements its source program. For example, Sewell et. al. [19] use this approach to extend the verification of the source code of seL4, an operating system microkernel [10], written in C, to that of the compiled binary. The validation is based on a refinement proof between the two programs. TInA [18] takes a different approach, lifting inline assembly into C, to improve the precision of existing C analyzers, by applying translation validation to confirm that the lifted-and-recompiled code is equivalent to the original. While translation validation can leverage the general-purpose reasoning of an SMT solver, it can also suffer from the unpredictability of SMT solvers. Targeted verified translations are less general, but produce more predictable results.

Fiat-Crypto [6] and Jasmin [1,2] both support verified translation from higher-level code to lower-level code. Jasmin uses Hoare logic at a high level; its lowest is slightly higher than assembly language, although low enough to make compilation straightforward (for example, the Jasmin compiler's register allocator never spills variables to the stack). Fiat-Crypto includes high-level domain-specific optimizations for elliptic curve cryptography, relieving the programmer of having to generate low-level optimized C code. Nevertheless, for widely used algorithms like AES-GCM, cryptography developers still consider it worthwhile to develop hand-optimized assembly code, and we believe that it is valuable to verify this hand-optimized code.

Superoptimizers [12] search for fast assembly language sequences and try to automatically establish equivalence with the original source code. However, typical superoptimizers can only generate short code sequences. Our genetic algorithm and verified transformer works on much longer instruction sequences (100s of instructions), albeit only for specific types of transformations.

## 7 Conclusions and Future Work

Code designed to be verified is not necessarily the same as code designed to run fast. However, some simple transformations can connect the two versions of the code. We have demonstrated such transformations both to increase confidence in existing code (in particular, OpenSSL's highly interleaved AES-GCM implementation) and to point the way towards alternate implementations that can have even higher performance on some platforms. Although the performance impact of interleaving instructions is small on the most recent Intel processors, we did find significant differences in performance both on older Intel processors and on a recent AMD processor; based on this, we speculate that such differences may be even more significant on less recent and/or non-Intel processors, such as less-powerful embedded processors.

For people focused on verification, it is heartening (and surprising) that the verification-friendly version of the AES-GCM code that we developed often outperformed the original, more interleaved code on several platforms. This suggests that developers of high-performance verified code should focus on major optimizations like domain-specific algorithm optimizations, loop unrolling and careful register allocation, and worry less about the exact sequence of instructions; this sequence may be better determined by automated search algorithms, supported by verified transformation tools.

One limitation of our current verified transformations is that writes to the heap cannot be reordered relative to other heap reads and writes. We are currently exploring ways to relax this restriction. For example, we are annotating heap loads and stores with identifiers that represent disjoint regions of memory; with these annotations, a transformer can safely reorder memory operations annotated with distinct identifiers.

## Acknowledgments

We thank the anonymous reviewers for valuable feedback. 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 grants from the Intel Corporation and the Alfred P. Sloan Foundation.

## References

1. Almeida, J.B., Barbosa, M., Barthe, G., Blot, A., Grégoire, B., Laporte, V., Oliveira, T., Pacheco, H., Schmidt, B., Strub, P.Y.: Jasmin: High-assurance and high-speed cryptography. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (2017). https://doi.org/10.1145/3133956.3134078
2. Almeida, J.B., Barbosa, M., Barthe, G., Grégoire, B., Koutsos, A., Laporte, V., Oliveira, T., Strub, P.: The last mile: High-assurance and high-speed cryptographic implementations. CoRR **abs/1904.04606** (2019), `http://arxiv.org/abs/1904.04606`
3. Appel, A.W.: Verified software toolchain. In: ESOP: 20th European Symposium on Programming (2011)
4. Appel, A.W.: Verification of a cryptographic primitive: SHA-256. ACM Trans. Program. Lang. Syst. **37**(2), 7:1–7:31 (Apr 2015)
5. Bond, B., Hawblitzel, C., Kapritsos, M., Leino, K.R.M., Lorch, J.R., Parno, B., Rane, A., Setty, S., Thompson, L.: Vale: Verifying high-performance cryptographic assembly code. In: Proceedings of the USENIX Security Symposium (August 2017)
6. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In: Proceedings of the IEEE Symposium on Security and Privacy (2019)
7. Fromherz, A., Giannarakis, N., Hawblitzel, C., Parno, B., Rastogi, A., Swamy, N.: A verified, efficient embedding of a verifiable assembly language. In: Proceedings of the ACM Symposium on Principles of Programming Languages (POPL) (Jan 2019)
8. Gueron, S., Krasnov, V.: The fragility of AES-GCM authentication algorithm. In: Proceedings of the Conference on Information Technology: New Generations (Apr 2014)
9. Gueron, S.: Intel® Advanced Encryption Standard (AES) New Instructions Set. `https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf` (Sep 2012)
10. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. ACM Transactions on Computer Systems **32**(1) (2014)
11. Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: Compcert – a formally verified optimizing compiler. In: Embedded Real Time Software and Systems (ERTS). SEE (2016)
12. Massalin, H.: Superoptimizer – a look at the smallest program. In: Architectural Support for Programming Languages and Operating Systems (ASPLOS) (1987)
13. McKeeman, W.M.: Peephole optimization. Commun. ACM **8**(7) (Jul 1965)
14. Mozilla: Measurement dashboard. `https://mzl.la/2ug9YCH` (Jul 2018)

15. Mullen, E., Zuniga, D., Tatlock, Z., Grossman, D.: Verified peephole optimizations for compcert. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2016)
16. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings. pp. 151–166 (1998). https://doi.org/10.1007/BFb0054170, https://doi.org/10.1007/BFb0054170
17. Protzenko, J., Parno, B., Fromherz, A., Hawblitzel, C., Polubelova, M., Bhargavan, K., Beurdouche, B., Choi, J., Delignat-Lavaud, A., Fournet, C., Kulatova, N., Ramananandro, T., Rastogi, A., Swamy, N., Wintersteiger, C., Zanella-Beguelin, S.: EverCrypt: A fast, verified, cross-platform cryptographic provider. In: Proceedings of the IEEE Symposium on Security and Privacy (May 2020)
18. Recoules, F., Bardin, S., Bonichon, R., Mounier, L., Potet, M.L.: Get Rid of Inline Assembly through Verification-Oriented Lifting. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE) (Nov 2019)
19. Sewell, T.A.L., Myreen, M.O., Klein, G.: Translation validation for a verified os kernel. In: Proceedings of ACM PLDI (2013)
20. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoué, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in F*. In: Proceedings of the ACM Conference on Principles of Programming Languages (POPL). pp. 256–270. ACM (Jan 2016)
21. Wheeler, D.A.: SLOCCount. Software distribution, http://www.dwheeler.com/sloccount/