A Verified, Efficient Embedding of a Verifiable Assembly Language

AYMERIC FROMHERZ, Carnegie Mellon University, USA
NICK GIANNARAKIS, Princeton University, USA
CHRIS HAWBLITZEL, Microsoft Research, USA
BRYAN PARNO, Carnegie Mellon University, USA
ASEEM RASTOGI, Microsoft Research, India
NIKHIL SWAMY, Microsoft Research, USA

High-performance cryptographic libraries often mix code written in a high-level language with code written in assembly. To support formally verifying the correctness and security of such hybrid programs, this paper presents an embedding of a subset of x64 assembly language in F* that allows efficient verification of both assembly and its interoperation with C code generated from F*. The key idea is to use the computational power of a dependent type system's type checker to run a verified verification-condition generator during type checking. This allows the embedding to customize the verification condition sent by the type checker to an SMT solver. By combining our proof-by-reflection style with SMT solving, we demonstrate improved automation for proving the correctness of assembly-language code. This approach has allowed us to complete the first-ever proof of correctness of an optimized implementation of AES-GCM, a cryptographic routine used by 90% of secure Internet traffic.

CCS Concepts: • **Software and its engineering → Formal software verification**;

Additional Key Words and Phrases: Assembly language, cryptography, domain-specific languages

ACM Reference Format:
Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. 2019. A Verified, Efficient Embedding of a Verifiable Assembly Language. *Proc. ACM Program. Lang.* 3, POPL, Article 63 (January 2019), 30 pages. https://doi.org/10.1145/3290376

1 INTRODUCTION

Verifying high-performance software often requires verifying code written in more than one language. For example, OpenSSL's implementations of cryptographic primitives like AES-GCM and Poly1305 contain a mixture of C code and hand-optimized assembly language. Verification frameworks like Coq [Coq Development Team 2015], F* [Swamy et al. 2016], and Dafny [Leino 2010] usually have direct support for extracting verified programs to a small number of languages, such as extracting OCaml or Haskell from Coq, OCaml or C from F*, or C# from Dafny. Considering the diversity of programming languages in use, even just among assembly languages (x86, x64, ARM, ...), it's unlikely that any single verification framework could contain built-in support for all possible languages that programmers might need. Instead, such frameworks are flexible enough

Authors' addresses: Aymeric Fromherz, Carnegie Mellon University, USA, afromher@andrew.cmu.edu; Nick Giannarakis, Princeton University, USA, nick.giannarakis@princeton.edu; Chris Hawblitzel, Microsoft Research, USA, Chris.Hawblitzel@microsoft.com; Bryan Parno, Carnegie Mellon University, USA, parno@cmu.edu; Aseem Rastogi, Microsoft Research, India, aseemr@microsoft.com; Nikhil Swamy, Microsoft Research, USA, nswamy@microsoft.com.

2475-1421/2019/1-ART63
https://doi.org/10.1145/3290376

to embed domain-specific languages (DSLs) and verify properties of programs written in those languages. For example, Coq datatypes can express the syntax and semantics of x86 assembly language [Kennedy et al. 2013], and Coq tactics can then prove properties about assembly language programs encoded using such datatypes.

Such a deep embedding of assembly language into an existing verification framework makes a good starting point for reasoning about and manipulating assembly language, but there's more to the story. First, the embedded language needs to interoperate with the outer language, so we have to provide ways to verify how assembly language instructions interact with memory allocated by the outer language, how calling conventions are enforced between the two languages, and so on.

Second, the outer verification framework typically contains mature support for proving properties of programs, and this support isn't necessarily available to the embedded language directly. In particular, recent tools for Hoare-style verification, such as Dafny and F*, supply a rich set of features to make verification more pleasant and productive, including built-in support for preconditions, postconditions, loop invariants, assertions, ghost variables, and calls to lemmas. Such tools automatically generate verification conditions (VCs) and send them to SMT solvers like Z3 [de Moura and Bjørner 2008] and CVC4 [Barrett et al. 2011]. The tools report errors from the SMT solvers in terms of the original source code, pinpointing the line number of the failing precondition, postcondition, loop invariant, or assertion, so that programmers can reason in terms of the source code they wrote, not the verification conditions that the tool produced. We want our embedded verification languages to provide the same level of support to programmers: to support preconditions, ghost variables, etc., to automatically generate VCs for SMT solvers, to report errors in terms of the embedded language, and so on. At the same time, we want to leverage as much of the existing verification framework as we can: to reuse existing logics, libraries, and support for error reporting, for example. And while we are willing to trust the soundness of existing logics and verification frameworks, we want to add as little as possible to the trusted computing base (TCB) when embedding a language into an existing framework.

This paper presents a technique for generating efficient VCs for embedded languages and verifying the VC generator so that it is not part of the TCB. The key idea is to use the computational power of a dependent type system's type checker to run the verified VC generator during type checking, so that the type checker (unwittingly) generates the VC and sends it to an SMT solver as part of discharging its normal type checking obligations. We apply this technique to embed x64 assembly language into F*, thereby reusing F*'s logic, libraries, and error reporting, while supporting interoperation between F* programs and assembly-language programs. We then apply the resulting system to verify efficient implementations of two widely used cryptographic algorithms, Poly1305 [Bernstein 2005] and AES-GCM [NIST 2007], that provide secrecy and integrity for over 90% of secure Internet connections [Mozilla 2018]. Each implementation relies on hand-tuned assembly and dedicated hardware instructions to achieve high performance, while still provably matching a complex specification expressed as computations over large mathematical fields.

To express preconditions, postconditions, and so on, we adopt Vale [Bond et al. 2017], a language designed for verifying assembly language. Prior work on Vale [Bond et al. 2017] simply translated Vale code into Dafny code and let Dafny generate VCs, which led to large, inefficient VCs that were slow to verify. By contrast, we write (and verify) a custom VC generator optimized for assembly language. We leverage F*'s dependent type checker to compute VCs as part of F*'s type checking process, in a style similar to proof-by-reflection. Our approach is not specific to F*; we believe it could work with other dependent type systems, like Coq. We choose F* because it directly supports interaction with SMT solvers, contains standard libraries optimized for SMT reasoning, and contains a C extraction tool that complements Vale's assembly-language extraction, particularly for cryptographic code written in a mixture of C and assembly language.

The rest of the paper presents our embedding, from definition to verification to application:

- Section 2 defines a straightforward embedding of a structured subset of x64 assembly-language syntax and semantics into F*; this embedding, along with the F* tool and SMT solver, constitutes the TCB for x64 assembly language. This TCB is no larger than for prior work on Vale [Bond et al. 2017], where the TCB consisted of an embedding of assembly language, a verification framework (Dafny), and an SMT solver.
- Section 3 presents our embedding of Vale, which we use to reason about x64 assembly language, in F*. It first presents in detail a complete, verified, and efficient VC generator as an example, showing the integration into F*'s type checking. It then briefly describes other verified extensions in our full implementation, including error reporting, control constructs, and a monadic shallow embedding of Vale's ghost variable bindings that avoids explicit reasoning about de Bruijn indices or variable substitution. Vale, and our efficient embedding of Vale in F*, are not part of the TCB (bugs in Vale and its embedding could at worst lead to a failed proof, not a successful proof of invalid assembly code).
- Section 4 describes the interoperation between x64 assembly language and C, including calling conventions and memory modelling; this constitutes the TCB for x64-C interoperation.
- Section 5 uses F* and Vale to verify real-world cryptographic code, including OpenSSL's implementation of Poly1305 (previously verified by [Bond et al. 2017]) and a new high-performance implementation of the widely used AES-GCM algorithm. We believe the latter to be the first verified high-performance AES-GCM implementation.
- Section 6 presents measurements showing significant verification speedups due to efficient VC generation, as well as Poly1305 and AES-GCM performance in the range of a gigabyte per second. For AES-GCM, this represents four orders of magnitude of improvement over the performance of previously verified implementations.

Files corresponding to our simplified examples from Sections 2 and 3, as well as our fully developed tools, specifications, implementations, and proofs (including the case studies from Section 5) are available at https://github.com/project-everest/vale/tree/popl_artifact_submit.

2 EXPRESSING ASSEMBLY LANGUAGE

Fig. 1 shows simplified examples from our deep embedding of a subset of Intel x64 assembly language into F*. This subset is similar to the deep embedding from Bond et al. [2017], albeit written in F* rather than Dafny. The embedding uses F* datatypes to represent x64 registers (reg); constant, register, and memory operands (operand); instructions (ins); and abstract syntax trees for structured assembly language code (code). As in Bond et al. [2017], we restrict control flow to structured if/else blocks and while loops, which are well-suited to our target applications (implementations of cryptographic primitives, like those in OpenSSL). A small, trusted F* program (not shown) prints code values as standard GNU assembly and Microsoft MASM assembly language formats for later assembly, linking, and execution.

We represent the machine state as a record (state) containing different machine components: general-purpose and XMM register files are functions mapping register names to values, status flags are a single word, and the memory is a partial map from integer addresses to bytes.

The state also contains a Boolean field ok representing the validity of the state. A valid state (ok = true) indicates that the machine safely executed until now. For instance, a valid state ensures that no segmentation fault occurred. Memory accesses and updates have validity checks based on membership in the domain of the memory map. An invalid memory access or update would therefore make the state invalid (ok = false).

```
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 cond = | Lt: o1:operand → o2:operand → cond | Eq: o1:operand → o2:operand → cond | ...
type code = | Ins: ins:ins → code
            | Block: block:list code → code
            | IfElse: ifCond:cond → ifTrue:code → ifFalse:code → code
            | While: whileCond:cond → whileBody:code → code
type state = { ok:bool;
               regs:reg → nat64;
               xmms:xmm → (nat32 * nat32 * nat32 * nat32);
               flags:nat64;
               mem:map int nat8; }
```

Fig. 1. Example F* definitions for Intel x64 syntax

Fig. 2 shows excerpts from our operational semantics for Intel x64 code. Our semantics are deterministic. We provide a function eval_code that takes an initial state and structured assembly code of type code, and returns the modified state after execution of the code. We must define eval_code as a total function, since mathematical definitions used in proofs must be well-founded. To account for non-terminating code, eval_code takes an additional argument providing fuel (defined to be a natural number) that is consumed during execution. If the fuel reaches zero, eval_code returns None (which is used only for termination checking; i.e., it is distinct from reaching an invalid state). Termination means that there exists some fuel f such that eval_code c f s == Some s'.

To handle failure propagation while preserving the readability of the semantics, we express our instruction semantics (eval_ins) using a state monad that transforms states into states. For example, Fig. 2 shows our model of the instruction Add64. A first check ensures that the src and dst operands are valid. An operand is invalid if it accesses an invalid memory address. If the operand is invalid, the ok flag is set to false (once this happens, the monad ensures the flag remains false in all subsequent states). We then compute the addition, and update the dst operand. We leave the flags under-specified in our semantics. We only model specific flags, such as the carry flag, that we update accordingly.

The assembly language semantics and the GNU/MASM printer form the core of our TCB. Since we write the semantics and the printer in F*, F* is also part of the TCB, as is the Z3 SMT solver [de Moura and Bjørner 2008] that F* relies on to assist type and proof checking. On the other hand, the Vale language and tool described in the next section are not trusted; Vale generates proofs that are checked, using F*, against trusted high-level correctness specifications for the assembly language code; at worst, Vale can generate invalid proofs that will be rejected by F*.

2.1 The Vale Language

Although it is possible to hand-write code values like "Block [Ins (Add64 (OReg Rax) (OConst 10))]" and prove properties about such code values directly in terms of the eval_code semantics, it is useful to have a friendlier syntax and tool for expressing and verifying assembly language programs. The Vale language [Bond et al. 2017] provides a syntax for writing assembly language annotated with preconditions, postconditions, loop invariants, ghost variables, calls to lemmas, etc. To prepare

```
let eval_operand (o:operand) (s:state) : nat64 = match o with OReg r → s.regs r | OConst n → ...| ...
let eval_ins (ins:ins) =
  s ⟵ get;
  match ins with
  | Mov64 dst src → ...
  | Add64 dst src → check (valid_operand src);; check (valid_operand dst);;
                    let sum = eval_operand dst s + eval_operand src s in
                    let new_carry = sum ≥ pow2_64 in
                    set_operand dst ins (sum % pow2_64);; set_flags (update_cf s.flags new_carry)
let rec eval_code (c:code) (f:fuel) (s:state) : option state =
  match c with
  | Ins ins → Some (run (eval_ins ins) s)
  | ...
```

Fig. 2. Example semantics for Intel x64 assembly

```
procedure{:instruction Ins(Mov64(dst, src))} Move(out dst:dst_opr64, src:opr64)
    ensures dst == old(src);
procedure{:instruction Ins(Add64(dst, src))} Add(inout dst:dst_opr64, src:opr64)
    modifies flags;
    requires dst + src < pow2_64;
    ensures dst == old(dst + src);
procedure{:instruction Ins(Add64(dst, src))} AddWrap(inout dst:dst_opr64, src:opr64)
    modifies flags;
    ensures
      dst == old(dst + src) % pow2_64;
      cf(flags) == old(dst + src ≥ pow2_64);
procedure Triple()
    modifies rax; rbx; flags;
    requires rax < 100;
    ensures rbx == 3 * old(rax);
{
    Move(rbx, rax);
    Add(rax, rbx);
    Add(rbx, rax);
}
```

Fig. 3. Example procedure declarations in Vale

for Section 3's description of how we verify Vale code, we provide a brief overview of Vale here (see Bond et al. [2017] for more details).

Vale programs consist of a series of procedures, as shown in the examples in Fig. 3. Procedure parameters may be operands, whose types specify what argument operands are allowed (e.g., dst_opr64 for a 64-bit destination operand). A procedure representing an individual instruction, indicated with the attribute {:instruction ...}, is verified directly against the operational semantics specified by eval_code; for example, both Add and AddWrap are verified relative to the semantics

for the `Add64` instruction specified in Fig. 2. Compound procedures like `Triple` contain procedure bodies inside {. . .}; the procedure bodies are verified using Hoare logic, as described in Section 3. The `Add(rax, rbx)` instruction in `Triple`, for example, must satisfy the precondition `rax + rbx < pow2_64` specified by the `Add` procedure. As in Bond et al. [2017], calls from one procedure to another procedure are inlined (macro expanded).

The Vale tool accepts a Vale program as input and generates both `code` values and proofs about the `code` values. These values are passed to Dafny or F⋆ for verification and, if verification succeeds, for printing in GNU or MASM assembly syntax, using a trusted programmer-defined printer for the `code` values. The printed assembly code is then assembled and linked with other code compiled from unverified C code or extracted from verified Low⋆ code (currently using standard compilers and linkers such as gcc).

We refer to the original Vale tool for Dafny as Vale/Dafny [Bond et al. 2017]. For this paper, we created a variant of the Vale tool called Vale/F⋆, which retains the original Vale generation of `code` values (using F⋆ syntax rather than Dafny syntax), but uses drastically different proof generation, as described in the next section.

3 VERIFYING ASSEMBLY LANGUAGE

Domain-specific languages (DSLs) embedded in general-purpose languages ought to enjoy domain-specific benefits not applicable to all host programs. For deeply embedded DSLs like Vale, some common benefits include the ability to easily transform and analyse the syntax of embedded programs. Vale/Dafny and Vale/F⋆ both rely on this ability, for instance, to print embedded programs in assembly language, and, as we will see in Section 4.4, to analyse them for side-channel leaks.

However, for the core task of verifying the correctness of embedded Vale programs, Vale/Dafny, and our initial attempts at Vale/F⋆, generated VCs for embedded programs using the same general-purpose VC generator as was used for programs in the host language (Dafny or F⋆). While this reuse of host-language features is simple, it is also naive: the resulting VCs were bloated, leaked too many details about the underlying encoding of the embedded language, and ultimately made proving programs very inefficient, requiring hundreds of seconds of SMT solving for larger procedures. We refer to this initial attempt at implementing Vale on F⋆ as Vale/$F^{\star}_{naive}$ to distinguish it from our more optimized implementation, which we refer to simply as Vale/F⋆.

This section presents the main contribution of our work: designing and verifying a custom, efficient VC generator for Vale embedded in F⋆. The main idea is to rely on the host language's symbolic computational capabilities to partially evaluate a VC before encoding it to the SMT solver. Although we work in F⋆, our work relies on type system features available in many modern proof assistants, including full dependent types, inductive families at higher universes, higher-order abstract syntax, and type-level computations. Hence, many of our ideas should transfer to other settings. That said, we rely on F⋆'s SMT encoding facilities to actually prove VCs (and report errors on failures) using Z3 [de Moura and Bjørner 2008], and the mechanized soundness proofs presented in this section rely on Z3 as well (no tactics are required).

3.1 Vale/F⋆ Verification Conditions

Vale/F⋆ verifies each Vale procedure by generating a logical formula called a *verification condition* (VC) for the procedure. If the VC is valid, then the procedure terminates in a final state that satisfies the procedure's postcondition, assuming the initial state satisfies the procedure's precondition. For `Triple` from Fig. 3, Vale/F⋆ generates the VC shown on the right side of the turnstile ⊢ below:

```
procedure Triple()
    modifies rax; rbx;
    requires rax < 100;
    ensures rbx == 3 * old(rax);
{
    Move(rbx, rax);
    Add(rax, rbx);
    Add(rbx, rax);
}
```

```
s0 : state,
(s0 Rax < 100)                                  (*Triple's precondition*)
⊢
  (∀ (x1:nat64). x1 == s0 Rax ⟹                 (*Move(rbx, rax) post*)
                 s0 Rax + x1 < pow2_64 ∧        (*Add(rax, rbx) pre*)
  (∀ (x2:nat64). x2 == s0 Rax + x1 ⟹            (*Add(rax, rbx) post*)
                 x1 + x2 < pow2_64 ∧            (*Add(rbx, rax) pre*)
  (∀ (x3:nat64). x3 == x1 + x2 ⟹                (*Add(rbx, rax) post*)
                 x3 == 3 * s0 Rax)))            (*Triple's postcondition*)
```

This VC is valid in the context defining an initial state s0 satisfying the precondition s0 Rax < 100. The VC expresses each precondition and postcondition of each procedure called by `Triple` in a standard, straightforward manner, so that preconditions in called functions must be proven valid, and postconditions from called functions may be assumed by subsequent procedure calls. Notice that the VC hides the underlying machine model, omitting any direct mention of eval_code, eval_operand, fuel, etc. In other words, the VC optimizes away extraneous clutter from the machine model that could slow down the SMT solver.

How to produce such a compact VC from F⋆ and relate it to the semantics of the embedded assembly language is not obvious, since we don't have direct control over F⋆'s internal VC generation. In the rest of this section we develop, in several stages, our own verified generator for Vale that computes VCs (including the VC shown above), and show how to convince F⋆ to use our generated VC. For simplicity and clarity, we first present a version of the generator for a subset of assembly language whose state consists solely of a register file (no flags, memory, etc.):

```
type reg = Rax | Rbx | Rcx | Rdx
type operand = | OReg: r:reg → operand | OConst: n:nat64 → operand
type state = reg → nat64
```

We present a verification of this VC generator's soundness with respect to the underlying eval_code semantics of the underlying machine model. We then describe, less formally, how we extended and verified the VC generator for additional assembly language and Vale features, including all the features from Fig. 2, error reporting, control constructs, and Vale ghost variables. However, to set the stage, we start by describing a naive proof strategy implemented first by Vale/Dafny and also in an early version of Vale/F⋆.

3.2 A Naive Proof Strategy

Attempting to prove Vale programs by directly reasoning about the embedded eval_code semantics is intractable. We prefer instead to reason about programs through the indirection of a Hoare logic proven sound with respect to eval_code.

Towards this end, for each Vale procedure, Vale/F⋆ constructs an F⋆ lemma that proves that the procedure terminates and that procedure's postconditions are satisfied, assuming the procedure's preconditions. A lemma in F⋆ is a computationally irrelevant, total (a.k.a. Ghost) function whose type is of the form $x_1{:}t_1 \rightarrow \ldots \rightarrow x_n{:}t_n \rightarrow$ Ghost t (requires pre) (ensures λ(x:t) → post). The formal parameters $x_i{:}t_i$ are in scope to the right of the arrows that bind them, i.e., the arrows are dependent. When applied to arguments that validate the precondition pre, the function returns a value v:t that validates the postcondition post, where x is bound to v. For example, here is the statement of the lemma generated for the `Add` procedure (slightly hand-edited for clarity):

```
val lemma_Add (s0:state) (dst:operand) (src:operand) : Ghost (state * fuel)
  (requires OReg? dst ∧ eval_operand dst s0 + eval_operand src s0 < pow2_64)
  (ensures λ(sM, fM) → eval_code (Ins (Add64 dst src)) fM s0 == Some sM ∧
                       eval_operand dst sM == eval_operand dst s0 + eval_operand src s0 ∧
                       sM == update_state dst.r sM s0)
```

The statement of lemma_Add guarantees that when evaluating the code of the Add instruction (Ins (Add64 dst src)) with operands and initial state that satisfy Add's preconditions, it can compute a final state sM and fuel amount fM such that eval_code run in the initial state with fM produces the final state sM satisfying the postconditions.

Using lemma_Add, one can try to prove a corresponding lemma_Triple for `Triple`. The statement of val lemma_Triple is shown in Fig. 4 and is analogous to lemma_Add. The last conjunct in its postcondition expresses `Triple`'s "`modifies rax; rbx;`" clause by ensuring that the final state is equal to the initial state except for updates to `rax` and `rbx`.

```
let update_reg (s:state) (r:reg) (v:nat64) : state = λr' → if r = r' then v else s r'
let update_state (r:reg) (s' s:state) : state = update_reg s r (s' r)
val lemma_merge (c:code) (cs:list code) (s0:state) (f0:fuel) (sM:state) (fM:fuel) (sN:state) : Ghost fuel
  (requires eval_code c f0 s0 == Some sM ∧ eval_code (Block cs) fM sM == Some sN)
  (ensures λfN → eval_code (Block (c::cs)) fN s0 == Some sN)
let codes_Triple : list code = [Ins (Mov64 (OReg Rbx) (OReg Rax));
  Ins (Add64 (OReg Rax) (OReg Rbx)); Ins (Add64 (OReg Rbx) (OReg Rax))]
val lemma_Triple (s0:state) : Ghost (state * fuel)
  (requires s0 Rax < 100)
  (ensures λ(sM, f0) → eval_code (Block codes_Triple) f0 s0 == Some sM ∧
                       sM Rbx == 3 * s0 Rax ∧ sM == update_state Rax sM (update_state Rbx sM s0))
let lemma_Triple s0 =
  let b1 = codes_Triple in
  let (s2, fc2) = lemma_Move s0 (OReg Rbx) (OReg Rax) in let b2 = b1.tl in
  let (s3, fc3) = lemma_Add s2 (OReg Rax) (OReg Rbx) in let b3 = b2.tl in
  let (s4, fc4) = lemma_Add s3 (OReg Rbx) (OReg Rax) in let b4 = b3.tl in
  let (sM, f4) = (s4, 0) in
  let f3 = lemma_merge b3.hd b4 s3 fc4 s4 f4 sM in
  let f2 = lemma_merge b2.hd b3 s2 fc3 s3 f3 sM in
  let fM = lemma_merge b1.hd b2 s0 fc2 s2 f2 sM in
  assert (FStar.FunctionalExtensionality.feq sM (update_state Rax sM (update_state Rbx sM s0)));
  (sM, fM)
```

Fig. 4. Declaration and naive proof of the correctness lemma for the `Triple` procedure

A proof of a lemma in F* must ensure the lemma's postcondition assuming the lemma's precondition; typical F* proofs consist of calls to other lemmas, whose postconditions help prove the outer lemma's postcondition. As shown in Fig. 4, Vale/$F^{\star}_{naive}$ generates a proof of lemma_Triple in an obvious way, calling lemma_Move and lemma_Add for each Move and Add instruction in the `Triple` procedure, peeling back the list of code instructions in codes_Triple along the way. The generated code keeps track of the fuel required to execute each call to `Add`, and then applies a library lemma lemma_merge to add the fuels together, rolling the list of instructions back together, to compute the total fuel fM needed by `Triple`. The proof relies on F*'s built-in VC generator to compute a VC

that chains together the pre- and postconditions for each invoked lemma to show that they imply the statement of the lemma.

Unfortunately, this naive style of proof leads to large, inefficient VCs, both for Dafny and F*. Below is a fragment of the VC that F* computes for lemma_Triple, specifically the fragment corresponding to checking the precondition of *just the second* Add, assuming the postcondition for the first Add. For brevity, it omits expressions for one of the calls to lemma_merge which it uses to compute, via Z3, the correct amount of fuel to prove termination. This is a far cry from the compact VC we produced in Section. 3.1.

```
(∀ (ghost_result0:(state * fuel)).
  (let (s3, fc3) = ghost_result0 in
    eval_code (Ins (Add64 (OReg (Rax)) (OReg (Rbx)))) fc3 s2 == Some s3 ∧
    eval_operand (OReg Rax) s3 == eval_operand (OReg Rax) s2 + eval_operand (OReg Rbx) s2 ∧
    s3 == update_state (OReg Rax).r s3 s2) ⟹
  lemma_Add s2 (OReg Rax) (OReg Rbx) == ghost_result0 ⟹
  (∀ (s3:state) (fc3:fuel). lemma_Add s2 (OReg Rax) (OReg Rbx) == Mktuple2 s3 fc3 ⟹
    Cons? codes_Triple.tl ∧
    (∀ (any_result0:list code). codes_Triple.tl == any_result0 ⟹
      (∀ (any_result1:list code). codes_Triple.tl.tl == any_result1 ⟹
        OReg? (OReg Rbx) ∧ eval_operand (OReg Rbx) s3 + eval_operand (OReg Rax) s3 < pow2_64
```

The problem is that in reusing the host language's VC generator as is, we have imposed all the requirements of the full generality of a VC generator for a dependently typed, higher order host language on our small, compact DSL for assembly. We ought to be able to better exploit the narrow feature set of the DSL to compute an optimized VC expressed in concepts as close as possible to the DSL constructs, not the host language.

After computing the naive VC for `Triple`, F* passes it to Z3, which proves the formula valid in less than a second. For a small procedure like `Triple`, this process is fast enough. For larger procedures, though, Z3 can take hundreds of seconds, and this slow response time makes it painful for users to debug their preconditions and postconditions. Partly, Z3 runs slowly because the formula is large (in the worst case, SAT solvers can take exponential time relative to the size of the formula). But a deeper issue is that the naive VC forces Z3 to perform computations that our ideal VC does not. Consider the expression eval_operand (OReg Rbx) s3 that appears near the end of the naive VC shown above. This looks up register rbx in the state s3, which triggers reasoning about map select/update operations on the state s2 to find the value of rbx produced by the first Move instruction. Long procedures often require repeated instantiations of select/update lemmas to discover values computed in far-away instructions. Worse, such select/update reasoning often gets discarded and recomputed as the SMT solver backtracks during its search.

Rather than let Z3 search through possible instantiations of select/update lemmas, it's more efficient to first use an algorithm to compute selects and updates, then pass the result to Z3. We can make an analogy with proof-by-reflection. To use an example from CPDT [Chlipala 2017], given a unary representation of a number, it's possible to prove the number even or odd by repeatedly applying tactics: apply a tactic that shows a number is even if its predecessor is odd, then apply a tactic showing that a number is odd if its predecessor is even, and so on. Repeatedly applying tactics is inefficient, though; it's more efficient to write an executable function that computes whether a number is even or odd, prove the correctness of the function, and then simply run the function on the particular number we're interested in to prove that it's even or odd. This technique might not sound impressive at first (any novice programmer can write a function that computes whether a

number is even or odd), but the beauty of proof by reflection is that it works within existing proof systems like F* or Coq, exploiting the proof system's ability to compute during type checking.

Our custom VC generator isn't exactly the same as computing even-or-odd via reflection; when we compute a VC, we're only partially computing the correctness of a Vale procedure, and then leaving the rest of the work for an SMT solver. But we follow the same high-level strategy: replace a slow proof process (the naive approach's overuse of SMT solving) with a faster function (a custom VC generation function), prove the correctness of the function, and then run the function as part of type checking (using F*'s normalization features).

3.3 QuickCode: Custom, Verified, Efficient VC Generation

This section defines a custom verification condition generator for Vale and describes how we formally proved it sound in F*. At first, we were apprehensive about formally verifying a VC generator. As Jean-Christophe Filliatre said, "formally verifying a tool such as Why3 is a lot of work and it is not that obvious that we should do it right now" [Filliâtre 2011]. An immediate problem is to formalize variable binding with all its notorious administrative overhead (e.g., using a deep embedding with explicit De Bruijn indices), as earlier VC-generation formalizations have done [Herms et al. 2012]. Although our assembly language has no bound variables, our custom VCs have universal quantifiers with bound variables, and Vale has ghost variable bindings. Fortunately, we discovered that a shallow embedding of both of these kinds of variable bindings using higher-order abstract syntax [Pfenning and Elliott 1988] sufficed, so we did not have to reason explicitly about binding, substitution, etc. to prove soundness.

Although we are computing VCs for deeply embedded programs, our approach is hybrid, relying on a layer that deeply embeds just the monadic control structure of a program, while exhibiting for each instruction and control construct in the embedded language an instance of a typeclass in the host language (i.e., a shallow term) that is capable of computing VCs for just that construct. VC generation for a given program then simply involves composing the VCs for typeclass instances corresponding to instructions in that program. This hybrid technique also allowed us to freely mix and match our VC generation with F*'s own VC generation. For example, we used a mixture of F*'s own VC generation and custom VC generation for while loops. This saved considerable effort.

Our VC generator constructs weakest precondition (WP) predicate transformers of type t_wp:

```
let t_wp = (state → Type) → (state → Type)
```

Elements wp of t_wp map postconditions of type state → Type (i.e., predicates on final states), to preconditions of type state → Type (i.e., predicates on initial states). To tie a wp to the code it describes, we define the relation has_wp, shown below:

```
let has_wp (c:code) (wp:t_wp) = k:(state → Type) → s0:state → Ghost (state * fuel)
                                (requires wp k s0)
                                (ensures λ(sM, f0) → eval_code c f0 s0 == Some sM ∧ k sM)
```

A useful intuition for WP calculi is continuation passing style. The type has_wp represents a proof of soundness for a particular code c and weakest precondition wp: for any postcondition k (i.e., the logical continuation) and initial state s0, if we assume that the precondition wp k holds on s0, then the code c successfully evaluates to a state sM such that the postcondition k holds on sM.

The type quickCode packages a code value c, weakest precondition wp, and proof hasWp about c and wp into a datatype value. One way to see quickCode c is as a typeclass that associates with a code value (c) a wp computation that is proven sound for c:

```
type quickCode : code → Type = QProc: c:code → wp:t_wp → hasWp:has_wp c wp → quickCode c
```

```
let wp_Add (dst:operand) (src:operand) (k:state → Type) (s0:state) : Type =
  OReg? dst ∧ eval_operand dst s0 + eval_operand src s0 < pow2_64 ∧
  (∀ (x:nat64). let sM = update_reg s0 dst.r x in
            eval_operand dst sM == eval_operand dst s0 + eval_operand src s0 ⟹ k sM)
let hasWp_Add (dst:operand) (src:operand) (k:state → Type) (s0:state) : Ghost (state * fuel)
  (requires wp_Add dst src k s0)
  (ensures λ(sM, f0) → eval_code (Ins (Add64 dst src)) f0 s0 == Some sM ∧ k sM)
 = lemma_Add s0 dst src
let quick_Add (dst:operand) (src:operand) : quickCode (Ins (Add64 dst src)) =
  QProc (Ins (Add64 dst src)) (wp_Add dst src) (hasWp_Add dst src)
```

Fig. 5. Automatically generated values that build a quickCode value for Add

As an example of a value of type quickCode, Fig. 5 shows a sample wp and hasWp definition for the Add instruction. Given a Vale declaration for Add with user-supplied preconditions and postconditions, Vale/F* automatically generates values wp_Add, hasWp_Add, and quick_Add. Notice that the weakest precondition function wp_Add generates universal quantifiers with bound variables, using F*'s built-in universal quantifiers (a shallow embedding). This shallow embedding makes the proof of hasWp_Add simple: it just calls the existing lemma_Add lemma.

To combine quickCode instances into a structure suitable for VC generation, we define quickCodes that reveals the sequential skeleton of the instructions in a program.

```
type quickCodes : list code → Type =
  | QEmpty: quickCodes []
  | QSeq: #c:code → #cs:list code → quickCode c → quickCodes cs → quickCodes (c::cs)
  | QLemma: #cs:list code → pre:Type → post:Type →
            lem:(unit → Lemma (requires pre) (ensures post)) → quickCodes cs → quickCodes cs
```

In the QSeq constructor, the arguments marked with a # are implicit arguments: F* computes their instantiations by unification at each use. The QLemma constructor uses F*'s syntactic sugar for Lemma, a unit-returning Ghost function, to represent a proof of pre ⟹ post (it has an additional unit argument because in F*, Lemma is an effect annotation on a function type, not a type by itself).

With this machinery in place, building the top-level VC generator is pleasantly simple.

```
let rec vc_gen (cs:list code) (qcs:quickCodes cs) (k:state → Type) : state → Type = λ(s0:state) →
  match qcs with | QEmpty → k s0
                 | QSeq qc qcs → qc.wp (vc_gen cs.tl qcs k) s0
                 | QLemma pre post _ qcs → pre ∧ (post ⟹ vc_gen cs qcs k s0)
```

Given a list of code values cs and a quickCodes data structure qcs, the VC generation function vc_gen builds a weakest precondition for the series of procedure calls and lemma invocations described by qcs. The main work of the VC generator is to sequentially compose the individual WPs. For example, the weakest precondition for a lemma invocation says that the lemma's precondition pre must hold and, assuming the lemma's postcondition post holds, the verification condition for the subsequent procedure calls and lemma calls must hold.

The vc_sound function is the complete proof of soundness of vc_gen:

```
let rec vc_sound (cs:list code) (qcs:quickCodes cs) : has_wp (Block cs) (vc_gen cs qcs) =
  λ(k:state → Type) (s0:state) → match qcs with
    | QEmpty → (s0, 0)
    | QSeq qc qcs → let (sM, fM) = qc.hasWp (vc_gen cs.tl qcs k) s0 in
                    let (sN, fN) = vc_sound cs.tl qcs k sM in
                    let fN' = lemma_merge cs.hd cs.tl s0 fM sM fN sN in
                    (sN, fN')
    | QLemma pre post lem qcs' → lem (); vc_sound cs qcs' k s0
```

The statement of the lemma is in the type of vc_sound on the first line, establishing that the weakest precondition of Block cs is really vc_gen cs qcs. The proof is by induction on the structure of qcs, checked automatically by F⋆ and Z3 for termination; it is quite short with just a few lines for each case. For example, for the case of a lemma invocation, the proof simply invokes the lemma lem and then recurses on the subsequent procedure calls and lemma invocations in qcs'.

Vale/F⋆ automatically generates quickCodes values for Vale procedures. The quickCodes value for `Triple` uses QSeq three times, once for each procedure call that `Triple` makes to `Move` or `Add`:

```
let qcodes_Triple : quickCodes codes_Triple = QSeq (quick_Move (OReg Rbx) (OReg Rax)) (
                                              QSeq (quick_Add (OReg Rax) (OReg Rbx)) (
                                              QSeq (quick_Add (OReg Rbx) (OReg Rax)) (
                                              QEmpty)))
```

Given this quickCodes value, the earlier correctness lemma from Section 3.2 can now be replaced with a much shorter proof, consisting simply of a call to vc_sound, passing in the final postcondition for `Triple` as the argument k to vc_sound:

```
let state_eq (s0 s1:state) : Ghost Type (requires True) (ensures λb → b ⟹ s0 == s1) =
  let b = s0 Rax == s1 Rax ∧ s0 Rbx == s1 Rbx ∧ s0 Rcx == s1 Rcx ∧ s0 Rdx == s1 Rdx in
  assert (b ⟹ FStar.FunctionalExtensionality.feq s0 s1);
  b
let lemma_Triple s0 =
  let k sM = sM Rbx == 3 * s0 Rax ∧ state_eq sM (update_state Rax sM (update_state Rbx sM s0)) in
  vc_sound codes_Triple qcodes_Triple k s0
```

The state_eq function computes whether the final state sM satisfies the `Triple`'s modifies clause, i.e. that sM is the same as the initial state s0 except for updates to any modified registers (Rax and Rbx in this example). We use F⋆'s extensional equality for functions to show that state_eq s0 s1 implies s0 == s1, to satisfy the == in lemma_Triple's postcondition.

To generate an efficient VC for use by lemma_Triple, F⋆ needs to actually run the VC generator vc_gen to produce a VC for the SMT solver. F⋆ provides a function normalize that runs a user-supplied total function during F⋆'s type checking. To exploit this feature, the code below shows a variant of vc_sound called vc_sound_norm that applies normalize to the VC generator.

```
let vc_sound_norm (cs:list code) (qcs:quickCodes cs) (k:state → Type) (s0:state) : Ghost (state * fuel)
  (requires normalize (vc_gen cs qcs k s0))
  (ensures λ(sN, fN) → eval_code (Block cs) fN s0 == Some sN ∧ k sN)
 = vc_sound cs qcs k s0
```

We change lemma_Triple to call vc_sound_norm rather than vc_sound; when it calls vc_sound_norm with particular cs and qcs values, F⋆ normalizes the application of vc_gen to these values to produce a complete VC. After this normalization, this complete VC appears as the F⋆ requires clause for the call to vc_sound_norm. F⋆'s own VC generation sends whatever appears in a requires clause to the SMT solver. In this case, this means that F⋆ sends the complete VC generated by vc_gen to the SMT

solver, which is exactly we want. (We rely on F*'s normalization reducing terms inside bodies of quantifiers and lambdas, rather than just evaluating to a weak head normal form. Otherwise, it would stop short of generating all the inner quantifiers in our desired VC.)

In fact, naively applying normalize to vc_gen does not quite produce our ideal VC, because normalize actually normalizes too much, unfolding all function definitions and thereby replacing basic operators like $\wedge$ with their underlying calculus-of-constructions representations, which makes the resulting SMT query needlessly verbose. Fortunately, F* programs can pass additional arguments to the normalization function that configure the behavior of normalization, specifying, for example, exactly which function definitions to unfold, and our extended Vale tool automatically generates the necessary normalization arguments for F*.

The normalization process allows us to take full advantage of F*'s computation, in the spirit of proof by reflection. For example, the inefficient query in Section 3.2 forces the SMT solver to prove that the operand OReg Rbx is a valid destination (OReg? (OReg Rbx)). By contrast, the normalization process simply runs the function OReg?, normalizing it to a value of True, which F* optimizes away when constructing the SMT query. Similarly, to handle the modifies clause, the normalizer evaluates state_eq sM (update_state Rax sM (update_state Rbx sM s0))) to True, which is again optimized away when constructing the SMT query.

More significantly, normalization symbolically executes updates to states. For example, the update_reg s0 dst.r x expression in wp_Add updates the state s0 so that the operand dst contains the symbol x. When Triple writes to the operand OReg Rax, the normalizer symbolically updates the register map so that Rax points to the symbol x. The next time the VC generation fetches a value from Rax (using eval_operand), the normalization process returns the symbol x. This produces an efficient SMT query with expressions like "$x_3 == x_1 + x_2$", written in terms of symbols like x_1 and x_2 rather than in terms of inefficient map update and select operations.

In addition to generating lemma_Triple, Vale/F* also generates a quickCode value for Triple, just like the quick_Add value from Fig. 5. Other procedures can then build on Triple just as Triple builds on Add and Move.

3.4 State Records

The technique described above extends easily to more complicated state types, such as the state record from Fig. 2. There's one slight complication: normalization will try to reduce field selection operations applied to datatype values, but this only succeeds when the value is actually a datatype value, not a variable. Therefore, vc_sound_norm unpacks and repacks the state s0 to eta-expand it into an equal value whose head symbol is Mkstate. Other dependently typed languages, e.g., Agda, internalize such eta expansions for single constructor types; F* does not and so we need to invoke it explicitly.

3.5 Error Reporting

To report errors in terms of the Vale source code rather than the Vale/F*-generated F* code, we augment our quickCodes datatype with error messages, as shown in red:

```
type quickCodes : list code → Type =
  | QEmpty: quickCodes []
  | QSeq: #c:code → #cs:list code → msg:string → quickCode c → quickCodes cs → quickCodes (c::cs)
  | QLemma: #cs:list code → msg:string → pre:Type → post:Type →
              lem:(unit → Lemma (requires pre) (ensures post)) → quickCodes cs → quickCodes cs
```

We then extend vc_gen to create VCs instrumented with source code locations report errors back to the user in case the VC fails:

```
let rec vc_gen (cs:list code) (qcs:quickCodes cs) (k:state → Type) : state → Type = λ(s0:state) →
  match qcs with | QEmpty → k s0
                 | QSeq  msg  qc qcs →  labeled msg  (qc.wp (vc_gen cs.tl qcs k) s0)
                 | QLemma  msg  pre post _ qcs →  labeled msg  pre ∧ (post ⟹ vc_gen cs qcs k s0)
```

F* recognizes `labeled msg` as a hint to print `msg` as part of reporting a verification error. Our extended Vale tool introduces messages like:

```
"POSTCONDITION␣NOT␣MET␣AT␣line␣264␣column␣53␣of␣file␣./code/crypto/aes/x64/X64.GCTR.vaf"
```

which F* then prints in case of an error.

3.6 Control Constructs

Given the framework for `vc_gen` described above, adding support for the `Block`, `IfElse`, and `While` control constructs is straightforward. In fact, it does not even require adding more cases to `quickCodes`. Instead, `QProc` values can encode `Block` and `IfElse`, as in this encoding of if-else:

```
let wp_If (#c1 #c2:code) (b:cmp) (qc1:quickCode c1) (qc2:quickCode c2) (k:state → Type) (s0:state) : Type =
  valid_cmp b s0 ∧ (eval_cmp s0 b ⟹ qc1.wp k s0) ∧ (not (eval_cmp s0 b) ⟹ qc2.wp k s0)

let quickIf (#c1 #c2:code) (b:cmp) (qc1:quickCode c1) (qc2:quickCode c2) :
  quickCode (IfElse (cmp_to_ocmp b) c1 c2) =
  QProc (IfElse (cmp_to_ocmp b) c1 c2) (wp_If b qc1 qc2) (qIf_hasWp b qc1 qc2)
```

For a while loop, our extended Vale tool uses `vc_gen` on the loop body, but encodes the proof about the loop itself as an inductive proof using F*'s `let rec` construct. This allows the encoding to take direct advantage of F*'s own termination checking for `let rec` to check the termination of the loop.

3.7 Binding Ghost Variables

In addition to manipulating physical registers and memory, Vale procedures can manipulate ghost values, which assist in proving preconditions and postconditions but do not appear in the actual assembly language represented by the `code` type. For example, our AES-GCM implementation defines an abstract datatype `poly` for Galois field $GF(2^n)$ polynomials, and stores concrete representations of these polynomials in machine registers. The following polynomial multiply routines express their preconditions and postconditions in terms of ghost values of type `poly`, connected to XMM registers via the `to_quad32` function of type `poly → quad32`:

```
procedure Clmul128(ghost ab:poly, ghost cd:poly) returns(ghost lo:poly, ghost hi:poly)
    modifies flags; r12; xmm1; xmm2; xmm3; xmm4; xmm5;
    requires degree(ab) ≤ 127 ∧ degree(cd) ≤ 127;
             xmm1 == to_quad32(ab) ∧ xmm2 == to_quad32(cd);
    ensures degree(lo) ≤ 127 ∧ degree(hi) < 127;
             xmm1 == to_quad32(lo) ∧ xmm2 == to_quad32(hi);
             mul(ab, cd) == add(shift(hi, 128), lo);

procedure ClmulRev128(ghost ab:poly, ghost cd:poly) returns(ghost lo:poly, ghost hi:poly) . . .{
    lo, hi := Clmul128(reverse(ab, 127), reverse(cd, 127));
    ShiftLeft128(lo, hi);
. . .}
```

When our extended Vale tool constructs a `quickCodes` value for `ClmulRev128`, it constructs a `QSeq` value for the `ShiftLeft128(lo, hi);` statement, and the variables `lo` and `hi` must be in scope at this point in the `quickCodes` value. We were initially worried that ghost values would require some sort of deep

embedding, perhaps with integer names for the ghost variables. Fortunately, a shallower encoding, in the style of higher-order abstract syntax [Pfenning and Elliott 1988] sufficed. Specifically, we write new definitions of quickCode and quickCodes in a monadic style, where QEmpty becomes the monadic unit operation that returns a ghost value of type a, and we add an additional constructor in quickCodes, a bind operation QBind, with the type below.

```
type quickCode (a:Type) : code → Type =
  | QProc: c:code → wp:t_wp a → hasWp:has_wp c wp → quickCode a c

type quickCodes (a:Type) : list code → Type =
  | QEmpty: a → quickCodes a []
  ...
  | QBind: #b:Type → #c:code → #cs:list code → msg:string → quickCode b c →
           (b → state → quickCodes a cs) → quickCodes a (c::cs)
```

QBind combines a quickCode returning type b with a function transforming the program state and a b value into quickCodes for type a to produce a final quickCodes for type a.

Our extended Vale tool generates QBind values for statements that introduce ghost variables. For example, the tool generates a quickCodes value for `ClmulRev128` that looks roughly like:

```
let qcodes_ClmulRev128 (ab:poly) (cd:poly) : quickCodes (poly * poly) codes_ClmulRev128 =
  QBind ...(quick_Clmul128 (reverse ab 127) (reverse cd 127)) (λ (lo, hi) s1 →
  QSeq ...(quick_ShiftLeft128 lo hi) ...
```

The proof of soundness for the vc_gen remains short, even with the new QBind case:

```
let rec vc_gen (#a:Type) (cs:list code) (qcs:quickCodes a cs) (k:a → state → Type) =
...
    | QBind qc f_qcs → qc.wp (vc_gen_Bind cs.tl f_qcs k) s0
and vc_gen_Bind (#a #b:Type) (cs:list code) (f_qcs:b → state → quickCodes a cs) (k:a → state → Type) =
  λ(g:b) (s0:state) → vc_gen cs (f_qcs g s0) k s0

let rec vc_sound (#a:Type) (cs:list code) (qcs:quickCodes a cs) : has_wp a (Block cs) (vc_gen cs qcs) =
...
    | QBind msg qc f_qcs → let (gM, sM, fM) = qc.hasWp (vc_gen_Bind cs.tl f_qcs k) s0 in
                           let (gN, sN, fN) = vc_sound cs.tl (f_qcs gM sM) k sM in
                           let fN' = lemma_merge c.hd cs.tl s0 fM sM fN sN in
                           (gN, sN, fN')
```

4 CALLING ASSEMBLY FROM C: CORRECT AND SECURE DSL INTEROPERATION

In the realm of unverified software, interoperation between languages is frequent. For instance, C-like code is convenient for writing efficient low-level code, and it is the standard choice to develop protocols such as TLS [Rescorla 2018]. But for maximum performance, hand-tuned assembly is required and is the *de facto* standard for high-performance, state-of-the-art cryptographic libraries such as OpenSSL [Bond et al. 2017]. Indeed, at the assembly level, one can manually apply various optimizations that may be difficult for a compiler, and directly benefit from hardware features such as vectorized instructions or AES-NI [Gueron 2012]. Hence, cryptographic libraries are typically hybrid programs that contain C routines that periodically call in to assembly for higher performance.

Since we ultimately aim to support full verification of such hybrid programs, we provide a way to allow programs written in a subset of F* (called Low* [Protzenko et al. 2017]) to call optimized assembly routines in Vale, while checking that the specifications used by the two DSLs compose

well and preserve verification guarantees across the boundary. In addition to DSL interoperation, we would like, when possible, to verify Vale programs at a level of abstraction that is closer to Low⋆ than the lowest-level machine state. Of course, this will not always be desirable, since we explicitly want to make use of architectural features exposed only at the assembly level.

Supporting the kind of DSL integration we have in mind poses several challenges. First, Low⋆ and Vale have very different memory models. Low⋆ models memory as a well-typed, structured heap (similar to CompCert's [Leroy et al. 2016]), while the machine model in Vale maps integer addresses to bytes. Second, calls between C and assembly are mediated by a calling convention specific to the operating system and hardware used. We want to make assumptions about these conventions explicit at the boundary between our DSLs. Automating the generation of these assumptions seems wise, since our past experience has shown that they are particularly error-prone when done manually. Note, we do not aim for our DSL interoperation layer to model inline assembly, focusing instead only on calls from C into assembly. We also do not model callbacks from assembly to C. Finally, given our interest in security-related applications, we would like measures for side-channel resistance adopted by Vale and Low⋆ to also compose well.

In the rest of this section, we describe our solutions to these challenges in the context of small illustrative examples. Section 5 then uses our interoperability framework at a larger scale to build proofs of verified cryptography implemented in a hybrid style that mixes Low⋆ and Vale.

4.1 Background on Low⋆

Low⋆ [Protzenko et al. 2017] is a shallow embedding of a small, well-behaved subset of C in F⋆. It models the memory of a C program as a set of disjoint logical regions, allowing modular verification based on separation properties. It also supports allocation and deallocation of memory. Well-typed programs written in Low⋆ are guaranteed to be memory safe; i.e., they never access out-of-bounds or deallocated memory, or attempt to repeatedly free the same memory. Further, since Low⋆ programs are just F⋆ programs, beyond simple memory safety, they also enjoy type abstraction and assertion safety. F⋆ provides a tool, KreMLin, that compiles Low⋆ code to C. KreMLin emits idiomatic, human readable code that is suitable for manual review. Proofs on paper ensure that safety, correctness, and security guarantees proven at the F⋆ level are preserved in the generated C code. The proofs relate the semantics of Low⋆ to CompCert's Clight.

The state of a Low⋆ program is modeled by an F⋆ type HS.mem (referred to as a hyper-stack by Protzenko et al. [2017]). Briefly, hyper-stacks provide a region-based memory model [Grossman et al. 2002; Tofte and Talpin 1997], distinguishing *heap regions* from *stack regions*. Each region in a hyper-stack maps abstract memory addresses to typed values, e.g., fixed width integers (nat8, nat64 etc.), or mutable arrays of values. Imperative Low⋆ computations are encapsulated by F⋆'s effect system in a *computation type* ST a (requires pre) (ensures post), a stateful computation which when run in an initial state h_0:HS.mem satisfying the precondition pre h_0, returns a v:a in final state h_1 satisfying postcondition post h_0 v h_1 (similar to Hoare Type Theory [Nanevski et al. 2008]).

Arrays. Low⋆ models an array t as a reference to a sequence of t. Reading from an array, for example, has the following specification from F⋆'s library:

```
let get (#t:Type) (m:HS.mem) (x:array t) (i:nat32{i < x.length}) :Ghost t = Seq.index (m[x.content]) i
val index: #t:Type → x:array t → i:nat32{i < x.length} → ST t
  (requires (λ m → live x m))
  (ensures (λ m0 r m1 → m0 == m1 ∧ r = get m0 x i))
```

The imperative operation index on mutable arrays (x:array t) is specified in terms of a pure operation get on an immutable sequence in a given memory (m[x.content] : seq t). The precondition on index and the refinement type on its argument i enforce temporal and spatial safety, respectively.

4.2 Reconciling Memory Models Between Low⋆ and Vale

To support hybrid programs, we must enable Low⋆ and Vale programs to share selected regions of memory that correspond to the storage referred to by mutable references in Low⋆. However, aspects of a C program's memory that are not observable from Low⋆ must remain inaccessible from Vale. For instance, although a Vale program should be able to access stack data that was explicitly allocated in Low⋆, we do not wish to allow a Vale program to access the control stack of a Low⋆ program, as otherwise it could undermine the Low⋆ metatheory that relates its semantics to C's semantics. In what follows, we focus on shared access to arrays between Low⋆ and Vale.

In contrast to Low⋆'s hyper-stack, Vale models memory as a map from physical addresses to bytes, i.e., map int nat8. To unify both memory models, we need to make manifest the layout of regions of memory in the hyper-stack in Vale's flat memory model and show that valid memory accesses at corresponding addresses in each level manipulate the same underlying value.

Specifically, we define the following relation to state that a (fragment of a) Low⋆ memory (heap) is simulated by a Vale memory (mem).

```
let correct_simulation (addrs:addr_map) (ptrs:list (array nat8)) (heap:HS.mem) (mem:map int nat8) =
  ∀(b:array nat8). List.member b ptrs ∧ live b heap ⟹ (
    ∀(i:nat{i < b.length}). get heap b i == mem.[addrs b + i])
```

One can see correct_simulation addrs ptrs : HS.mem → map int nat8 → Type as a relation between the two memories indexed by (1) a function addr_map that maps live, disjoint abstract addresses in Low⋆ to disjoint, valid address ranges in the Vale memory model; and (2) a list of array references ptrs that are to be shared between Low⋆ and Vale. The definition states that all live arrays in ptrs have the same values in both heap (at their abstract address) and in mem (at their corresponding concrete address chosen by addr_map). When calling from Low⋆ into Vale, the initial state of the Vale procedure is *assumed*[1] to be in correct_simulation relation with the state of the Low⋆ program at the time of the call. At the time the call returns, the state of the Low⋆ program is proven to be in correct_simulation relation with the final state of the Vale procedure.

4.2.1 An Intermediate Machine Semantics With a Structured View of Memory. In theory, to implement a Low⋆ *extern* function, it would suffice to write assembly code manipulating the low-level bytes, and report these changes back to Low⋆. In practice, preserving an *array-like view* of memory inside of Vale is often more convenient. An array-like view allows us to express pre- and post-conditions on arrays directly and to leverage the Low⋆ array library.

We implement such a view inside of Vale to reason about all memory operations. Rather than working with a Hoare logic defined directly on top of the machine semantics that uses a map int nat8 memory, we define an untrusted intermediate-level machine semantics on top of the lowest-level machine state with a new structure to represent memory (see Figure 6). This view of memory, Vale.mem, consists of three components, as shown below:

[1] A technicality: rather than assuming outright that the initial hyper-stack at the time of the call is in correct_simulation with the Vale memory, it suffices to assume just the existence of an addr_map. From this, one can simply compute a Vale memory that is in correct_simulation relation with the hyper-stack. Removing the assumption about the existence of an addr_map does not seem possible, since Low⋆ abstract addresses are drawn from an infinite address space, while Vale imposes a restriction that memory addresses fit in the machine words used to represent registers (recall Fig. 1).

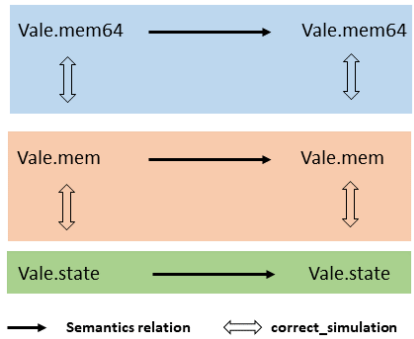

Fig. 6. A Structured View on Low-Level Memory. To make programming in Vale/F⋆ more pleasant and productive, we lift the trusted byte-level semantics defined for Vale.state to an *array-like view* (Vale.mem) and prove that the lifted semantics remain in a correct_simulation relation with the byte-level semantics. Next, we lift the semantics from arrays of bytes to arrays of various wider types, such as 64-bit integers (Vale.mem64), and prove that the correct_simulation relation is preserved.

```
type mem = {
  hs: HS.mem;
  ptrs: list (array nat8){∀ (x:array nat8). List.mem x ptrs ⟹ live hs x};
  addrs: addr_map;
}

let valid_state (h:Vale.mem) (m:map int nat8) = correct_simulation h.addrs h.ptrs h.hs m
```

It contains a Low⋆ hyper-stack, a list of live arrays in scope that we call ptrs, and the addr_map function which appeared previously as an index to the correct_simulation relation.

We verify that our intermediate machine semantics based on Vale.mem correctly simulate the memory at the bytes-level by proving that the correct_simulation relation is preserved between two corresponding executions of the intermediate-level and lowest-level (map int nat8) machine semantics. This ensures that our view of memory is consistent with the Vale/F⋆ model.

4.2.2 Beyond Byte Arrays: Views That Provide Wider Types. To keep our trusted correct_simulation relation as simple as possible, we relate the Vale bytes-based memory model to arrays of bytes, i.e., the shared pointers in the ptrs field is just a list (array nat8). Nevertheless, it is often convenient to reason and program with larger integers. For example, Intel x64 stores 64-bit integers in registers, so we find it easier to reason about arrays of nat64 in memory. Furthermore, when verifying some cryptographic algorithms, official specifications are sometimes expressed in terms of 128-bit integers that we prefer to consider as four 32-bit numbers that we call a quad32. Specifying, and reasoning about ghost arrays also enables a lighter coding style.

To have this flexibility, we build bijections to treat arrays of bytes (of appropriate length) as arrays of nat64 (a.k.a., array64) or arrays of quad32 (a.k.a., array128). To safely consider an array of bytes x as an array64, the length of x must be divisible by 8, as 8 bytes form a nat64. We have a similar condition on array128. With this restriction, we implement and prove correct bijections via the obvious mathematical mapping.

With our bijections in place, we can lift relevant operations on the byte arrays to corresponding operations on their views. There are three main memory operations available at the bytes level: Validity check, access, and update. Their analogs for array64 are shown below (types only).

```
val valid_mem64: ptr:int → mem → Ghost bool (* is there a 64–bit word at address ptr? *)
val load_mem64: ptr:int → mem → Ghost nat64 (* the 64–bit word at ptr (if valid_mem64 holds) *)
val store_mem64: ptr:int → v:nat64 → mem → Ghost mem (* if valid, update the 64–bit word at ptr *)
```

We prove that when a memory update through store_mem64 or store_mem128 occurs, the relation correct_simulation is preserved. This ensures that the relation holds at all times and that our array view of memory correctly simulates the bytes model. Using these facilities, a programmer can implement procedures in Vale while manipulating typed arrays rather than raw, low-level memory.

4.3 Calling Conventions

Calling conventions describe how subroutines interact with their caller. They specify for instance how parameters are passed, in which registers they are stored, and which registers must be preserved by the callee. Calling conventions vary heavily based on the operating system, architecture, and compiler used: For instance, Microsoft systems on x86-64 pass the first integer argument of a function in the RCX register, while Linux systems pass their first argument in the RDI register. Rules are complex and following them is error-prone: a programmer can easily make assumptions about them that do not hold in the system they are using.

To address this issue, we implemented a simple (trusted) tool called CCWrap that wraps a Vale function with a Low⋆ caller while following the calling conventions of a user-specified system. Given an abstract function signature, the operating system, and the hardware under use, CCWrap automatically generates both Low⋆ and Vale signatures. Additionally, if a user provides a verified Vale procedure p satisfying the generated signature, it generates the corresponding Low⋆ wrapper implementation f and a proof that both the Low⋆ and Vale functions interoperate seamlessly. Other Low⋆ functions calling f need never know that f is implemented using a call into an assembly-language routine.

Currently CCWrap provides support for Windows and Linux on Intel x64. It can also be easily extended to other calling conventions: a user only needs to specify the registers in which parameters are passed and returned, and which registers must be saved by the callee.

For example, here is an abstract function signature (using CCWrap-specific datatype constructors like TNat64 to specify the types for arguments) that CCWrap can use to generate a Low⋆ wrapper for an optimized memcpy implementation:

```
let memcpy = ("memcpy", // Name of the function
  [("dst", TBuffer TNat64); ("src", TBuffer TNat64); ("len", TBase TNat64)]) // List of arguments
```

From this description, CCWrap generates the code in Fig. 7 (hand-edited slightly for clarity). The code contains four free variables: pre_cond and post_cond are F⋆ specifications that the programmer can choose for their generated code; implies_pre and implies_post are free variables for lemmas the programmer can use to relate their chosen specs to the proven specifications of the Vale code they wish to call. In most cases, if the Vale code is already proven with respect to the array-view, the choice of these variables from the Low⋆ side is trivial.

To implement the interoperation, we need to call the verified Vale procedure on an initial Vale state corresponding to the initial Low⋆ state. To prove that the interoperation is correct, we must ensure that the memory views in both the Vale and the Low⋆ states are consistent and that the calling conventions are respected. As explained previously, assuming the existence of an addr_map, we can build a valid Vale memory in correct_simulation with the initial hyper-stack using the to_bytes_mem function (line 5). To enforce calling conventions, our translator stores the function arguments in the correct registers according to the calling conventions (line 6) and ensures through a post-condition that specific registers are correctly saved by the callee (line 19).

```
1  let addrs = addrs_map_axiom() (* assume the existence of a map from abstract to machine addresses *)
2
3  let create_initial_trusted_state (dst:array nat8) (src:array nat8) (len:nat64) (h0:HS.mem) =
4    (* create a map of bytes satisfying the correct_simulation relation from a hyper-stack *)
5    let (mem:map int nat8) = to_bytes_mem addrs [dst; src] h0 in
6    let regs = init_regs[Rdi <- addrs dst; Rsi <- addrs src; Rdx <- len] (* calling convention for Linux *)
7    {ok = true; regs = regs; xmms = init_xmms; flags = 0; mem = mem} (* initial machine state *)
8
9  val memcpy: (dst:array nat8{length dst % 8 == 0}) -> (src:array nat8{length src % 8 == 0}) ->
10       (len:nat64) -> (h0:HS.mem) -> Ghost (state * nat * HS.mem)
11       (requires pre_cond h0 dst src len)
12       (ensures λ(s1, f1, h1) ->
13         let s0 = create_initial_trusted_state dst src h0 in (* initial machine state *)
14         s1 == eval_code (va_code_memcpy ()) s0 ∧ (* The final state satisfies the machine semantics *)
15         post_cond h0 h1 dst src len ∧ (* The post_condition is verified *)
16         s1.ok ∧ (* The execution didn't crash *)
17         (* The final HS.mem is in correct_simulation relation with the final state of the Vale procedure *)
18         correct_simulation h1 addrs [dst;src] s1.mem ∧
19         callee_saved_registers s0 s1 (* The callee_saved registers are correct *) )
20
21 let memcpy dst src len h0 =
22   let s0 = create_initial_trusted_state dst src len h0 in (* initial machine state *)
23   implies_pre h0 dst src len;
24   (* va_lemma_memcpy: Lemma in the style of Section 3 attesting correctness of (va_code_memcpy()) *)
25   let s1, f1 = va_lemma_memcpy (va_code_memcpy ()) s0 dst src len in
26   implies_post s0 s1 f1 dst src len;
27   (* Propagates changes to the memory back to the hyper-stack to satisfy correct_simulation relation *)
28   let h1 = from_bytes_mem addrs [dst; src] s1.mem h0
29   s1, f1, h1
```

Fig. 7. Excerpt from the generated Low★ implementation of memcpy for Linux on Intel x64. Lines 1-19 are trusted, while the implementation of memcpy (l. 21-29) is verified against its specification

Lastly, we ensure that the final state is the result of calling the trusted machine interpreter on the given code through eval_code (line 14) and that the execution was safe (line 16). This allows us to keep the Vale language and its F★ code generator out of our TCB: Any mistake made in translating Vale to F★ will manifest as a verification failure in the code shown in Fig. 7.

Finally, we provide the trusted function stput (shown below) so that Low★ code can interoperate with the implementation generated by CCWrap. This stateful (ST) function allows Low★ code to replace the current state (h0:HS.mem) underlying the effect monad with a new state (h1 == f h) if that state is accompanied by a witness function (f) that computes the new state from the old state (as long as the old state satisfies precondition pre). For example, if Low★ code wishes to invoke the memcpy function from Fig. 7, it passes memcpy as f.

Hence, the TCB for this generated wrapper consists of three components: the creation of an initial Vale state respecting argument-passing conventions, a signature for the wrapper ensuring that the Low★ and Vale memory views are consistent and that the execution followed the trusted machine semantics, and the st_put function:

```
assume val st_put (pre:HS.mem → Type) (f:(h0:HS.mem{pre h0}) → GTot HS.mem)
  :ST unit (λ h0 → pre h0) (λ h0 _ h1 → h1 == f h0)
```

This, together with the trusted definitions from Section 4.2, constitutes the TCB for correct x64-C interoperation.

4.4 Proving Hybrid Programs Secure From Information Leakage

When implementing cryptographic algorithms, ensuring that the code is correct is not sufficient: it must be proven secure as well. More precisely, since cryptographic code operates on secrets, we must prove the absence of *leakage*. In particular, secrets may leak via a *side channel*. A side-channel attack leverages additional information from the program's behavior, such as cache access patterns [Aciiçmez et al. 2010; Percival 2005] or execution time [Andrysco et al. 2015; Brumley and Boneh 2003], in order to extract cryptographic keys or other sensitive data. Although physical side channels are an issue [Gandolfi et al. 2001; Masti et al. 2015], digital side channels such as memory accesses [Aciiçmez et al. 2010; Yarom et al. 2010] or program execution time [Brumley and Boneh 2003; Kocher 1996] are a bigger concern, as they can be exploited by a remote attacker.

When proving side-channel resistance, some techniques are more convenient than others, depending on the language and level of abstraction used. For instance, in Vale/F*, we use proof by reflection to demonstrate that our assembly code is secure (Section 4.4.1), while Low* relies on type abstraction (Section 4.4.2). Hence, when moving between the two languages, we need to unify both styles to provide verified end-to-end security guarantees (Section 4.4.3).

4.4.1 Background: Proving Assembly Code Free from Information Leakage. In Vale/F*, we adopt the approach used in Vale/Dafny [Bond et al. 2017], in order to prove our assembly code is free of information leakage. To enable such proofs, we model a strong attacker capable of observing detailed digital side-channel information. The attacker can see every instruction executed, every memory address accessed, and every element of the machine state that is not explicitly declared secret. To capture this model, we augment our machine state with a trace field, and we augment our machine semantics to record adversarial observations in this trace. We then define leakage freedom, as shown in Fig. 8, as a classic non-interference property [Goguen and Meseguer 1982]. A procedure (code) is leakage free if, for all states s1 and s2 such that the traces and memory locations marked as public in s1 and s2 are initially identical, the traces are identical in the states r1 and r2 computed by successful executions of code.

```
type observation = | BranchPredicate: pred:bool → observation | MemAccess: addr:nat64 → observation
type state = {. . .; trace : list observation}

let isLeakageFree (code:code) (isPub:loc → bool) =
  ∀(s1 s2:state) fuel.
    let r1 = eval_code code fuel s1 and r2 = eval_code code fuel s2 in
    s1.trace == s2.trace ∧ (∀ x. isPub x ⟹ s1[x] == s2[x]) ⟹
    r1.trace == r2.trace
```

Fig. 8. Side Channels. We extend the machine state with adversarial observations that capture digital side channels, and we define the absence of leakage in a program code via the isLeakageFree predicate.

Like Vale/Dafny, our strategy for establishing that a Vale/F* program is leakage free involves a classic use of proof by reflection. We implement in F* a *taint analyser* that consumes our (deeply embedded) syntax of assembly language. We prove, once and for all, that our taint analysis algorithm

conservatively decides the isLeakageFree property, by proving it sound with respect to our trace-augmented semantics. The F⋆ signature of our taint analysis is shown below:

```
val taint_analysis: c:code → isPub:(loc → bool) → b:bool{b ⟹ isLeakageFree c isPub}
```

To achieve a more precise analysis compared to typical taint analysers, we use memory aliasing information that is verified during the verification of the code's functional correctness.

To run the taint analysis, we extract taintAnalysis to OCaml (using F⋆'s existing extraction capabilities) and then concretely run OCaml code on the syntax of the program and secret labeling, checking that it returns true.

4.4.2 Background: Proving C Code Free from Information Leakage. Unlike the deep embedding we use for assembly, Low⋆ is a shallow embedding and uses type abstraction to prove side channel resistance. We briefly review the Low⋆ approach, which we have significantly simplified for exposition; we refer the reader to the supplementary material for a complete formalization of Low⋆ and our extensions to it.

When programming with secrets, Low⋆ programs are written against an interface that provides secrets at an abstract type. The type system then ensures that the programs cannot, for example, branch on secrets or use secrets as array indices. The interface also provides functions to operate on the secrets (e.g., to add two secret integers to get a new secret integer). Assuming that the implementations of such functions are secret independent, Low⋆ then proves a secret-independence (meta) theorem [Protzenko et al. 2017, Theorem 1].

To formalize secret independence, Low⋆ semantics are instrumented with traces that reflect the branching behavior (brT and brF) and the memory access patterns (read(b, n) and write(b, n)):

$$\text{Trace} \quad \ell \quad ::= \quad \cdot \mid \mathsf{read}(b, n) \mid \mathsf{write}(b, n) \mid \mathsf{brT} \mid \mathsf{brF} \mid \ell_1, \ell_2$$

Protzenko et al. define an equivalence relation between Low⋆ memories and Low⋆ expressions ($H_1 \equiv_\Gamma H_2$ and $e_1 \equiv_\Gamma e_2$), which relates two memories and expressions that are equal except in subterms that have abstract types (per the type environment Γ). Their theorem then states that equivalent Low⋆ configurations (pairs of memories and expressions (H, e) that are point-wise equivalent) produce equal traces and equivalent configurations.

4.4.3 Proving Hybrid Programs Free from Information Leakage. To prove the security of hybrid programs, we must reconcile the notion of leakage in Vale/F⋆ and in Low⋆. To this end, we extend the Low⋆ secret-independence theorem to account for interoperation with Vale programs by, first, extending Low⋆'s formal syntax with an extern c expression form, that denotes the Vale code c embedded in Low⋆ code. We model the Vale semantics using an opaque relation that transforms Low⋆ memory, emitting an (abstract) trace: $(H, \mathsf{extern}\ c) \longrightarrow_z H'$, along with the following extensions to the Low⋆ syntax:

$$\begin{array}{rrcl} \text{Expression} & e & ::= & \cdots \mid \mathsf{let}\ \_ = \mathsf{extern}\ c\ \mathsf{in}\ e \\ \text{Extern trace} & z & & \\ \text{Trace} & \ell & ::= & \cdots \mid z \end{array}$$

The second and the key component of the extension is lifting Vale's static taint-analyser (Section 4.4.1) to the meta-level in Low⋆– in particular the isLeakageFree property from Fig. 8.

Proposition 4.1 (Meta property about the Vale/F⋆ taint analyser). *Let* $\Gamma \vdash \mathsf{extern}\ c$. *If* $\mathsf{taint\_analyse}(\Gamma, c) = \mathsf{true}$, *then for two well-typed heaps* H_1 *and* H_2 *s.t.* $H_1 \equiv_\Gamma H_2$, *we have* $(H_1, c) \longrightarrow_{z_1} H_1'$, $(H_2, c) \longrightarrow_{z_2} H_2'$, H_1' *and* H_2' *are well-typed in* Γ, $z_1 = z_2$, *and* $H_1' \equiv_\Gamma H_2'$.

Using the proposition above, it is straightforward to extend the following secret-independence theorem from Protzenko et al. [2017] to include the new extern expression form. The detailed proof is available online. [2]

Theorem 4.2 (Secret Independence for Hybrid Low⋆/Vale programs). *Given configurations* (H_1, e_1) *and* (H_2, e_2)*, where* $\Gamma \vdash (H_1, e_1) : \tau$, $\Gamma \vdash (H_2, e_2) : \tau$, $H_1 \equiv_\Gamma H_2$ *and* $e_1 \equiv_\Gamma e_2$*, and a secret independent implementation of the secret interface* P_s*, either both the configurations cannot reduce further, or* $\exists\, \Gamma' \supseteq \Gamma$ *s.t.* $P_s \vdash (H_1, e_1) \rightarrow^+_{\ell_1} (H_1', e_1')$, $P_s \vdash (H_2, e_2) \rightarrow^+_{\ell_2} (H_2', e_2')$, $\Gamma' \vdash (H_1', e_1') : \tau$, $\Gamma' \vdash (H_2', e_2') : \tau$, $\ell_1 = \ell_2$, $H_1' \equiv_{\Gamma'} H_2'$, *and* $e_1' \equiv_{\Gamma'} e_2'$,

In our concrete implementation, CCWrap relates secret types in Low⋆ to the labeling function (isPub from Section 4.4) needed for the taint analysis to run in Vale.

5 CASE STUDIES

We illustrate Vale/F⋆'s functionality by porting a previously verified cryptographic algorithm (Poly1305, Section 5.1), and we demonstrate Vale/F⋆'s scalability by verifying a freshly developed, complex cryptographic algorithm (AES-GCM, Section 5.1). These are two of the three algorithms mandated by the upcoming TLS 1.3 standard [Rescorla 2018].

5.1 Porting Poly1305

To demonstrate that Vale/F⋆ retains the functionality of Vale/Dafny, we ported Bond et al.'s [2017] verified implementation of Poly1305, which they previously ported from OpenSSL's 64-bit non-SIMD implementation. Poly1305 [Bernstein 2005] is a popular algorithm for computing a message authentication code (MAC), which provides integrity for messages sent between parties who share a symmetric key.

Porting Poly1305 to Vale/F⋆ required a small number of syntactic changes to accommodate minor differences in Vale/F⋆'s parser. More effort was required to port the supporting Dafny lemmas to F⋆; this process was tedious but largely straightforward. The lemmas help prove the soundness of the various mathematical tricks that OpenSSL employs to compute operations within a 130-bit field on a 64-bit architecture. Our verification rules out the types of bugs that have cropped up in OpenSSL's implementation of Poly1305 [OpenSSL 2016a,b,c].

Section 6.1 reports on the verification time for the ported version of Poly1305 vs. the original.

5.2 A High-Speed Implementation of AES-GCM

To illustrate Vale/F⋆'s ability to support developer-friendly verification and its ability to scale to large, complex implementations, we implement and verify a high-performance implementation of AES-GCM from scratch.

AES-GCM [NIST 2007] is one of the most widely used cryptographic algorithms in the world (e.g., one study shows 91% of secure web traffic using AES-GCM [Mozilla 2018]), thanks to its strong cryptographic properties and real-world efficiency. It provides authenticated encryption with additional data (AEAD), which means that it guarantees both secrecy and integrity for plaintext messages, and it can provide integrity protection for additional, unencrypted data (e.g., one can use an AEAD algorithm to protect the secrecy of a network packet's contents and the integrity of its header information).

AES-GCM is considerably more complex than the underlying block cipher, AES [NIST 2001]. The latter is simply a pseudorandom permutation on a fixed number (128) of bits, whereas AES-GCM can process an (almost) arbitrarily long plaintext and additional data, providing both secrecy and

[2] https://github.com/project-everest/vale/blob/popl_artifact_submit/proof.txt

integrity. Internally, it employs a variant of counter-mode encryption to provide secrecy, and Galois/Counter Mode (GCM) [McGrew and Viega 2004] for integrity.

Given AES-GCM's ubiquity, Intel added dedicated hardware instructions to support efficient and side-channel free implementations [Gueron 2012]. Specifically, they added six instructions (called "AESNI") to support AES computations, and one to support GCM computations. The former operate on Intel's extended register sets (e.g., 128-bit XMM registers) and must be combined with additional vectorized instructions to achieve maximum performance. The GCM-support instruction (PCLMULQDQ) accelerates one step in the GCM algorithm, but considerable work is required to complete the entire algorithm, let alone prove it correct. The algorithm processes its input in 128-bit chunks viewing each chunk as an element of the Galois field $GF(2^{128})$, and the core operations are additions and multiplications within this field. PCLMULQDQ performs a carry-less multiply, but the GCM implementation must then compute a polynomial reduction on the result in order to remain within the field.

Hence, proving the correctness of our AES-GCM implementation required reasoning about vectorized instructions and their effects on SIMD registers represented as bit-vectors, mathematical integers, and Galois field elements. The latter required developing an F* library for reasoning about Galois fields in general, and the AES-GCM field in particular.

Furthermore, while it would be simpler to reason about an AES-GCM implementation by first computing the encryption pass (using AES in counter mode), and then computing an authentication pass (using GCM), for maximum performance, our implementation, like any other performant implementation, interleaves the two passes, so that we only read the input data once.

Our AES-GCM implementation has two versions; the first keeps the initialization, and high-level control flow in Low*, making targeted calls to 13 different Vale routines to leverage dedicated hardware instructions or hand-coded routines for critical operations (e.g., computing a polynomial multiplication and reduction in $GF(2^{128})$). To maximize performance, we also developed a second version that performs the entire algorithm in Vale, but exposes a Low*/C interface generated by CCWrap. It also takes advantage of SIMD parallelism for the AES (but not yet the GCM) portions of the computation. The two versions provably meet the same cryptographic spec and share nearly all of their lemmas.

Altogether, counting comments and white space, our verified AES-GCM implementation requires 339 lines of specification, 2020 lines of proof libraries, 73 Vale procedures, over 1100 lines of Low* code, and more than 4400 lines of Vale code. Compared to, say, Bond et al.'s [2017] AES-CBC implementation, which only required 26 procedures, our AES-GCM implementation is considerably more complex. Section 6.2 shows that it also achieves gigabyte-per-second performance, putting it orders of magnitude ahead of previously verified implementations.

6 EVALUATION

This section presents the verification and run-time performance of Vale/F* on AES-GCM and Poly1305. All measurements were taken on a single core of an Intel i7-8700 processor.

6.1 Verification Performance

Developing a Vale procedure is typically an iterative process; the programmer writes the code, adds necessary preconditions and the desired postconditions, and checks for correctness. If verification fails (an all too typical occurrence), the programmer can add static assertions to better understand what can and cannot be verified, or they can adjust one or more of the preconditions, postconditions, or the code itself. Another verification is attempted and the process repeats until verification succeeds.

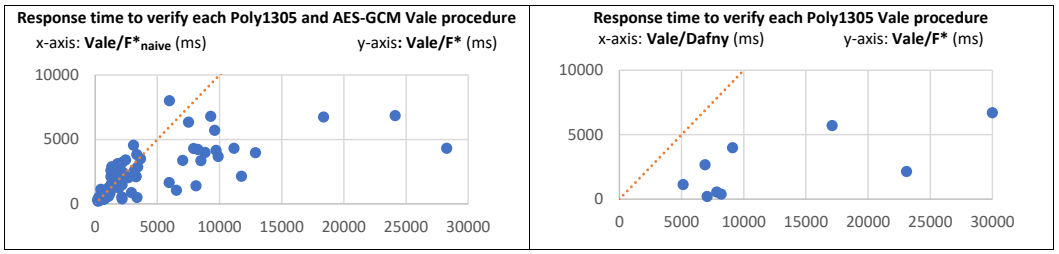

Fig. 9. Verification time: Vale/F* vs. Vale/F$^\star_{naive}$ and Vale/F* vs. Dafny. All dots below and to the right of the dotted $y = x$ line indicate Vale/F* outperforming the baseline.

Vale/F*'s optimized VC generation is designed to reduce verification time, which is the crucial bottleneck in the debug cycle outlined above. Minimizing verification time is crucial to keeping developers productive; in our experience, verification times under approximately 10 seconds keep the debug cycle tolerable, while longer verification times (e.g., over 30 seconds) are vastly more annoying since they disrupt the programmer's train of thought (e.g., by allowing time to check email, or get coffee). A short debug cycle is even more critical for larger procedures, since it typically takes more iterations to get all of the preconditions and postconditions right.

To assess the impact of Vale/F*'s improvements, we compare its performance to both Vale/F$^\star_{naive}$ and Vale/Dafny, the most directly related systems. A comparison with other systems, e.g., developed in Coq, would be difficult without redeveloping many libraries and cryptographic implementations in Coq. The performance of such a development would be heavily affected by how we developed the Coq code and which tactics we chose. Anecdotally, other Coq projects have reported large verification times; e.g., Chlipala reports an hour of total verification time, with many case studies running in minutes "at best" [Chlipala 2013].

The left chart in Fig. 9 shows the time taken by F* and Z3 to verify the "lemma_..." functions generated by Vale/F* and Vale/F$^\star_{naive}$ for each procedure in Poly1305 and AES-GCM, defined to be the time taken to run F*'s normalization plus the time taken by Z3. The right chart shows the same for Vale/F* and Vale/Dafny for Poly1305 (AES-GCM has not been implemented in Vale/Dafny).

For small procedures, Vale/F$^\star_{naive}$ was fast enough (less than 5 seconds); for these, Vale/F* neither helped much nor hurt much. Larger procedures, though, benefited significantly from Vale/F*'s optimized VC generation. Overall, six procedures took more than 10 seconds with Vale/F$^\star_{naive}$, whereas all procedures took less than 10 seconds with Vale/F*. For all procedures, Vale/F* was significantly faster than Vale/Dafny (note that the rightmost Vale/Dafny point actually took longer than two minutes; we show the point at the right edge to fit it on the chart).

As discussed above, these results do not necessarily indicate that Vale/F* is faster than alternate verification approaches. Instead, they suggest that our techniques from Section 3 do significantly speed up the SMT solver's performance relative to more naive approaches, and hence it is worth putting effort into optimizing SMT queries when constructing embedded languages for verified code.

6.2 Cryptographic Case Studies: Run-Time Performance

Fig. 10 shows the performance of our verified case studies (Section 5), including both of our implementations of AES-GCM (one with a mix of Low* and Vale, the other primarily written in Vale), relative to OpenSSL. We use OpenSSL as our representative of state-of-the-art unverified cryptographic code, as prior work shows that OpenSSL consistently matches or exceeds the performance of other mainstream libraries [Bond et al. 2017]. We compile both OpenSSL and our

Low*-generated C code with gcc. For verified implementations, we compare against implementations from Vale/Dafny [Bond et al. 2017] and HACL* [Zinzindohoué et al. 2017], since other cryptographic verification efforts have focused on other algorithms (primarily elliptic curves) [Erbsen et al. 2019; Tsai et al. 2017].

Since the Vale/F* Poly1305 code is the same assembly code as OpenSSL's scalar code and that of Vale/Dafny, the Vale/F* Poly1305 performance matches OpenSSL's scalar code and Vale/Dafny's code. Both versions of our Vale/F* AES-GCM code are faster than the non-AESNI/PCLMULQDQ OpenSSL implementations. OpenSSL's absolute best code, however, is much more aggressively optimized; it also takes advantage of SIMD parallelism for the GCM computations, as well as several number-theoretic tricks that we have not yet implemented. Doing so would require more code and proof development, but no changes to Vale or our machine specifications. Note that our mostly Vale version of AES-GCM outperforms our hybrid implementation, at the cost of writing and maintaining more assembly code.

We also compared our AES-GCM performance to the only other verified AES-GCM implementation we could find, which consisted of experimental reference implementations from HACL* [Zinzindohoué et al. 2017] of AES-128 and GCM, neither optimized for performance (in contrast to other, more optimized and less experimental HACL* algorithms). We compiled each one to C and glued them together to produce a comparison for AES-GCM-128; not surprisingly, without access to AESNI/PCLMULQDQ, the performance was much slower that that of Vale/F*. Nevertheless, HACL* supports many more cryptographic algorithms (all written in F*) than Vale, so we hope that Vale/F* can be used to provide optimized assembly language to speed up existing HACL* code.

	Verified	Poly1305	AES-GCM-128	AES-GCM-256
OpenSSL (scalar)	–	3791	143	110
OpenSSL (SIMD)	–	8353	332	282
OpenSSL (SIMD, AESNI/PCLMULQDQ)	–	–	6414	4730
Vale/Dafny (scalar)	✓	3815	–	–
HACL* (C)	✓	2013	0.27	–
Vale/F* (scalar)	✓	3803	–	–
Vale/F* (SIMD, AESNI/PCLMULQDQ)	✓	–	613 991	935

Fig. 10. Cryptographic performance (MB/s for 8192-byte input data). Larger is better. '–' indicates the corresponding implementation does not exist. For AES-GCM-128 in Vale/F*, we give the performance for both our Low*/Vale hybrid (upper) and our primarily Vale (lower) implementations.

7 RELATED WORK

Other projects have verified VC generators, including partially-mechanized proofs (all the way back in 1973 [Ragland]), and more recent fully mechanized proofs [Gordon 1989; Herms et al. 2012; Homeier and Martin 1995]. Gordon [1989] doesn't define and verify a VC generator, but rather provides a library of verified primitives that tactics can use to construct VCs. Homeier and Martin [1995] verify a VC generator for a small imperative language with assertions, but without bound variables (in contrast to Vale, which has bound ghost variables). Herms et al. [2012] verify a VC generator by deeply embedding both the programming language and the logic (via De Bruijn indices), allowing them "to extract a standalone executable, and consequently to discharge VCs using external provers like SMT solvers". Our approach, by contrast, discharges VCs to an SMT solver by running the VC generator as part of F*'s type checking phase, taking advantage of F*'s

existing SMT support. We believe that this is novel, as is our monadic approach to handling bound ghost variables. A prior version of F⋆'s typechecker [Strub et al. 2012] was certified to produce VCs for SMT solvers in a provably sound way, bootstrapping trust in the system based on trust in Coq. A similar approach applied to the current version of F⋆ would allow us to remove it from the TCB.

Bedrock [Chlipala 2013] builds a Hoare-style verification framework on top of an assembly-level language. Like Vale/F⋆, Bedrock verifies the verification condition generation for the Hoare logic. However, the Bedrock VC generators construct existential and universal quantifiers over entire states, rather than over small atomic values as in Vale/F⋆. This means that Bedrock cannot simply normalize the VC generation as Vale/F⋆ can – the normalizer would block when trying to look up register values inside universally or existentially quantified state variables, since the state variables are just names, not actual record or map values. In addition, the Bedrock ghost variables [Chlipala 2013] can only be declared in preconditions and postconditions. Vale/F⋆, by contrast, uses a monadic approach to allow procedure bodies to contain ghost declarations, each of which can be initialized with values computed from earlier ghost declarations. Similarly, Myreen et al. [Myreen 2009; Myreen and Curello 2013] apply Hoare reasoning to assembly language via a "decompilation" process, although this decompiler does not exploit the fast proof-by-reflection/normalization techniques central to Vale/F⋆ nor does it appear to support ghost variables (as in our monadic approach).

Earlier work on Vale [Bond et al. 2017, Section 2.4] mentions "slower-than-expected proof verification by Dafny and Z3" and implements an optimization that splits each procedure's lemma into "inner" and "outer" lemmas that are checked separately. However, the separated lemmas still go through Dafny's VC generation, and the result is still much slower than Vale/F⋆, as seen in Fig. 9. In particular, Vale/F⋆ generates a more optimal VC for the "inner" part of the lemma, and eliminates the SMT query for the "outer" part of the lemma entirely by performing computation during F⋆ type checking.

We believe that it is important for verified assembly language to interoperate with verified higher-level code. We are not the first to do this; the CertiKOS project [Costanzo et al. 2016; Gu et al. 2016, 2018], for example, also contains a mixture of verified x86 code interoperating with verified C code. CertiKOS's approach embeds *both* C and assembly language into Coq, based on CompCert's embeddings of C and assembly. This gives CertiKOS a very small TCB, although it is closely tied to the CompCert compiler. Our approach to interoperation is a lighter-weight point in the design space, where assembly language is embedded, but C is simply extracted from the outer language (F⋆) directly by an existing tool (KreMLin/Low⋆) and compiled with off-the-shelf compilers like gcc, which provide more optimizations than CompCert. FunTAL [Patterson et al. 2017] is another system supporting interoperation between assembly language and higher-level code. FunTAL supports callbacks from assembly to a higher-level language, which Vale does not yet support. Overall, Vale/F⋆ targets Hoare-style proofs of assembly language program correctness, and proofs by reflection for security, whereas FunTAL is aimed more at proofs of program equivalence needed for formal verification of compilers transformations. Nevertheless, program equivalence could in principle be used to verify applications like cryptographic implementations as well.

Several recent projects have verified high-performance cryptography code, including bvCryptoLine [Tsai et al. 2017], HACL* [Zinzindohoué et al. 2017], and Fiat-Crypto [Erbsen et al. 2019]. Of these, only bvCryptoLine verifies code at the level of assembly language (using an SMT solver and computer algebra system), although they verify code in an idealized language extracted from assembly language that omits details about memory and calling conventions. Like Vale/F⋆, HACL* code is verified using F⋆ and Z3, but unlike Vale, is extracted to C code using KreMLin. Fiat-Crypto develops correct-by-construction implementations of cryptographic algorithms through a series of verified transformations. The transformations don't yet go all the way down to assembly language, but are instead extracted to C. Likewise, LMS-Verify [Amin and Rompf 2017] focuses on generating

lower-level C code from a higher-level language, using translation validation to verify the generated C code. For some cryptographic algorithms, the fastest C implementations are almost as fast as the fastest assembly language implementations, but for others, assembly language is faster by a factor of 2 or more (see the nistz256 +ADX results for P-256 in Erbsen et al. [2019], for example). To cover both these cases, correct-by-construction C code and Hoare-style verification of existing hand-written assembly are both useful and may complement each other: the former is easier, while the latter gives more control, when necessary, over the exact assembly code. Jasmin [Almeida et al. 2017] also addresses formal reasoning about low-level cryptography, though with an emphasis on verifying the translation to assembly language, rather than verification of hand-written assembly language. Except for HACL*'s experimental code, none of these projects have verified AES-GCM.

8 CONCLUSIONS AND FUTURE WORK

Verifying low-level software isn't easy, especially for software written in multiple languages. Verification frameworks based on VC generation and SMT solving can support programmers as they try to prove assertions and develop invariants. We've shown that such support can be provided not just by tools dedicated to particular languages, but also by custom embedded languages in dependently typed frameworks like F⋆. In particular, we've shown that such embedded languages can provide *efficient* VC generation, resulting in fast SMT queries. Going forward, we hope to leverage this efficient verification to tame the size and complexity of the most aggressively optimized implementations of cryptographic primitives.

ACKNOWLEDGMENTS

The authors are grateful to Jonathan Protzenko for KreMLin support; to Qunyan Mangus, Guido Martínez, and Tahina Ramananandro for F⋆ support; to Jan Hoffmann, Catalin Hritcu, Brian Milnes, and the anonymous POPL reviewers for their helpful comments and suggestions on the paper, and finally to Antoine Delignat-Lavaud, Cédric Fournet, Jonathan Protzenko, Santiago Zanella-Beguelin and the other members of Project Everest for integrating verified code produced from Vale/F⋆, stimulating discussions, and useful 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 a grant from the Alfred P. Sloan Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these sponsors.

REFERENCES

Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. 2010. New Results on Instruction Cache Attacks. In *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES)*.

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/3133956.3134078

Nada Amin and Tiark Rompf. 2017. LMS-Verify: Abstraction Without Regret for Verified Systems Programming. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*.

Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On Subnormal Floating Point and Abnormal Timing. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the Conference on Computer Aided Verification (CAV)*.

Daniel J. Bernstein. 2005. The Poly1305-AES message-authentication code. In *Proceedings of Fast Software Encryption*.

Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *Proceedings of the USENIX Security Symposium*.

David Brumley and Dan Boneh. 2003. Remote Timing Attacks Are Practical. In *Proceedings of the USENIX Security Symposium*.

Adam Chlipala. 2013. The Bedrock Structured Programming System: Combining Generative Metaprogramming and Hoare Logic in an Extensible Program Verifier. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

Adam Chlipala. 2017. Certified Programming with Dependent Types. http://adam.chlipala.net/cpdt/.

Coq Development Team. 2015. The Coq Proof Assistant Reference Manual, version 8.5. https://coq.inria.fr/distrib/current/refman/.

David Costanzo, Zhong Shao, and Ronghui Gu. 2016. End-to-end Verification of Information-flow Security for C and Assembly Programs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*.

L. de Moura and N. Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Jean-Christophe Filliâtre. 2011. Deductive Program Verification. https://www.lri.fr/~filliatr/hdr/memoire.pdf.

Karine Gandolfi, Christophe Mourtel, and Francis Olivier. 2001. Electromagnetic Analysis: Concrete Results. In *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES)*.

J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Michael J. C. Gordon. 1989. Current Trends in Hardware Verification and Automated Theorem Proving. Springer-Verlag New York, Inc., New York, NY, USA, Chapter Mechanizing Programming Logics in Higher Order Logic, 387–439. http://dl.acm.org/citation.cfm?id=106971.107122

Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/512529.512563

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Berkeley, CA, USA, 653–669. http://dl.acm.org/citation.cfm?id=3026877.3026928

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified Concurrent Abstraction Layers. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 646–661. https://doi.org/10.1145/3192366.3192381

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

Paolo Herms, Claude Marché, and Benjamin Monate. 2012. A Certified Multi-prover Verification Condition Generator. In *Proceedings of the International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*. Springer-Verlag, Berlin, Heidelberg, 2–17. https://doi.org/10.1007/978-3-642-27705-4_2

Peter V. Homeier and David F. Martin. 1995. A Mechanically Verified Verification Condition Generator. *Comput. J.* 38, 2 (1995), 131–141. https://doi.org/10.1093/comjnl/38.2.131

Andrew Kennedy, Nick Benton, Jonas B. Jensen, and Pierre-Evariste Dagand. 2013. Coq: The World's Best Macro Assembler?. In *Proceedings of the Symposium on Principles and Practice of Declarative Programming (PPDP)*.

Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the International Cryptology Conference (CRYPTO)*.

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*.

Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert – A Formally Verified Optimizing Compiler. In *Embedded Real Time Software and Systems (ERTS)*. SEE.

Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. 2015. Thermal Covert Channels on Multi-core Platforms. In *Proceedings of the USENIX Security Symposium*.

David A. McGrew and John Viega. 2004. The Security and Performance of the Galois/Counter Mode of Operation. In *Proceedings of the International Conference on Cryptology in India (INDOCRYPT)*.

Mozilla. 2018. Measurement Dashboard. https://mzl.la/2ug9YCH.

Magnus O. Myreen. 2009. Formal verification of machine-code programs. Ph.D. Thesis, University of Cambridge.

Magnus O. Myreen and Gregorio Curello. 2013. Proof Pearl: A Verified Bignum Implementation in x86-64 Machine Code. In *International Conference on Certified Programs and Proofs (CPP)*.

Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. 2008. Hoare type theory, polymorphism and separation. *J. Funct. Program.* 18, 5-6 (2008), 865–911.

NIST. 2001. Announcing the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197.

NIST. 2007. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. NIST Special Publication 800-38D.

OpenSSL. 2016a. Chase overflow bit on x86 and ARM platforms. GitHub commit d-c3c5067cd90f3f2159e5d53c57b92730c687d7e.

OpenSSL. 2016b. Don't break carry chains. GitHub commit 4b8736a22e758c371bc2f8b3534dc0c274acf42c.

OpenSSL. 2016c. Don't loose [sic] 59-th bit. GitHub commit bbe9769ba66ab2512678a87b0d9b266ba970db05.

Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. 2017. FunTAL: Reasonably Mixing a Functional Language with Assembly. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 495–509. https://doi.org/10.1145/3062341.3062347

Colin Percival. 2005. Cache Missing for Fun and Profit. BSDCan. https://www.daemonology.net/papers/htt.pdf.

F. Pfenning and C. Elliott. 1988. Higher-order Abstract Syntax. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 199–208. https://doi.org/10.1145/53990.54010

Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified Low-Level Programming Embedded in F*. *PACMPL* 1, ICFP (Sept. 2017), 17:1–17:29. https://doi.org/10.1145/3110261

L. C. Ragland. 1973. *A Verified Program Verifier*. Technical Report. Austin, TX, USA.

Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3, Draft 28. https://tools.ietf.org/html/draft-ietf-tls-tls13-28.

Pierre-Yves Strub, Nikhil Swamy, Cedric Fournet, and Juan Chen. 2012. Self-Certification: Bootstrapping Certified Typecheckers in F* with Coq. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*. 571–584. https://hal.inria.fr/inria-00628775

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*. ACM, 256–270.

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Inf. Comput.* 132, 2 (Feb. 1997), 109–176. https://doi.org/10.1006/inco.1996.2613

Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2017. Certified Verification of Algebraic Properties on Low-Level Mathematical Constructs in Cryptographic Programs. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. New York, NY, USA. https://doi.org/10.1145/3133956.3134076

Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2010. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. In *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES)*.

Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.