Vale: Verifying High-Performance Cryptographic Assembly Code (draft)

Abstract

High-performance cryptographic code often relies on complex hand-tuned assembly language that is customized for individual hardware platforms. Such code is difficult to understand or analyze. We introduce a new programming language and tool called Vale that supports foundational, automated verification of high-performance assembly code. The Vale tool transforms annotated assembly language into an abstract syntax tree (AST), while also generating proofs about the AST that are verified via an SMT solver. Since the AST is a first-class proof term, it can be further analyzed and manipulated by proven-correct code before being extracted into standard assembly. For example, we have developed a novel, proven-correct taint-analysis engine that verifies the code's freedom from digital side-channels. Using these tools, we verify the correctness and security of implementations of SHA-256 on x86 and ARM, Poly1305 on x64, and hardware-accelerated AES-CBC on x86. Several implementations meet or beat the performance of unverified, state-of-the-art cryptographic libraries.

1 Introduction

The security of the Internet rests on the correctness of the cryptographic code used by popular TLS/SSL implementations such as OpenSSL [60]. Because this cryptographic code is critical to TLS performance, implementations often use hand-tuned assembly, or even a mix of assembly, C preprocessor macros, and Perl scripts. For example, the Perl subroutine in Figure 1 generates optimized ARM code for OpenSSL's SHA-256 inner loop.

Unfortunately, while the flexibility of script-generated assembly leads to excellent performance (§5.1) and helps support dozens of different platforms, it makes the cryptographic code difficult to read, understand, or analyze. It also makes the cryptographic code more prone to inadvertent bugs or maliciously inserted backdoors. For instance, in less than a month last year, three separate bugs were found just in OpenSSL's assembly implementation of the MAC algorithm Poly1305 [63–65].

Since cryptographic code is so critical for security, we argue that it ought to be *verifiably* correct and leakage-free.

Existing approaches to verifying assembly code fall roughly into two camps. On one side, frameworks like Bedrock [19], CertiKOS [23], and x86proved [41] are built on very expressive higher-order logical frameworks like Coq [22]. This allows great flexibility in how the assembly is generated and verified, as well as high assurance that the verification matches the semantics of the assembly language. On the other side, systems like BoogieX86 [36, 75], VCC [55], and various assembly language analysis tools [10] are built on satisfiability-modulo-theories (SMT) solvers like Z3 [25]. Such solvers can potentially blast their way through large blocks of assembly and tricky bitwise reasoning, making verification faster and easier.

```
sub BODY_00_15 {
my ($i,$a,$b,$c,$d,$e,$f,$g,$h) = @_;
$code.=<<___ if ($i<16);
#if __ARM_ARCH__>=7
 @ ldr  $t1,[$inp],#4  @ $i
# if $i==15
 str $inp,[sp,#17*4]  @ make room for $t4
# endif
 eor $t0,$e,$e,ror#‘$Sigma1[1]-$Sigma1[0]‘
 add $a,$a,$t2  @ h+=Maj(a,b,c) from the past
 eor $t0,$t0,$e,ror#‘$Sigma1[2]-$Sigma1[0]‘@Sigma1(e)
# ifndef __ARMEB__
 rev $t1,$t1
# endif
#else
 @ ldrb  $t1,[$inp,#3]  @ $i
 add  $a,$a,$t2  @ h+=Maj(a,b,c) from the past
 ldrb $t2,[$inp,#2]
 ldrb $t0,[$inp,#1]
 orr  $t1,$t1,$t2,lsl#8
 ldrb $t2,[$inp],#4
 orr  $t1,$t1,$t0,lsl#16
# if $i==15
 str $inp,[sp,#17*4]  @ make room for $t4
# endif
 eor $t0,$e,$e,ror#‘$Sigma1[1]-$Sigma1[0]‘
 orr $t1,$t1,$t2,lsl#24
 eor $t0,$t0,$e,ror#‘$Sigma1[2]-$Sigma1[0]‘@Sigma1(e)
#endif
```

FIGURE 1—***Snippet of a SHA-256 code-generating Perl script from OpenSSL, with spacing adjusted to fit in one column. §4.1 helps decode and explain the motivations for this style.***

In this paper, we present Vale, a new language for expressing and verifying high-performance assembly code that strives to combine the advantages of both approaches; i.e., it combines flexible generation of high-performance assembly with automated, rigorous, machine-checked verification. For any assembly program written in Vale, the Vale tool constructs an abstract syntax tree (AST) representing the program's code, and produces a proof that this AST obeys a desired specification for any possible evaluation of the code (Figure 2). Both the AST and the proofs are currently expressed in Dafny [49], an off-the-shelf logical framework supporting SMT-based verification, higher-order reasoning, datatypes, functions, lemmas, and extraction of executable code. Dafny uses Z3 to verify the

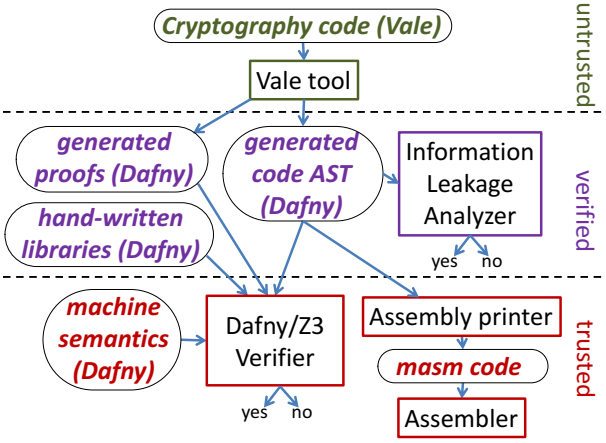

FIGURE 2—***Verifying cryptographic code with Vale and Dafny.***

proofs generated by Vale.

After verification, the AST is available in the logical framework for further analysis and manipulation. As a powerful example of this post-analysis, we have developed a verified analyzer that checks Vale ASTs for potential information leakage through timing and memory-access channels.

Although we trust Dafny and the assembly language semantics to be correct, as shown in Figure 2, neither Vale nor the information leakage analyzer is part of the trusted computing base. The former merely produces ASTs and proofs that are then checked by Dafny; the latter is written directly in Dafny and verified once and for all to be correct for all possible ASTs. Furthermore, working directly on assembly language means we trust an assembler but not a higher-level compiler, so we need not worry about compilers that introduce information leaks into cryptographic code [42].

Contributions In summary, this paper makes the following contributions.

- The design and implementation of Vale (§2), which combines flexible generation of high-performance assembly with automated machine-checked verification.
- A machine-verified analyzer that checks verified Vale programs for side channels based on a novel combination of dataflow analysis and Hoare-style proofs (§3).
- A series of case studies applying Vale to standard algorithms like Poly1305, AES, and SHA on x86, x64, and ARM platforms with support for both GCC and MASM (§4). They show that Vale is flexible enough to express and verify even highly scripted code generation like that in Figure 1. In particular, it replaces the use of chaotic Perl scripts with a principled approach based on verification and partial evaluation.
- An evaluation demonstrating that, because Vale can match the expressiveness of OpenSSL's code-generation techniques, the verified assembly code generated by Vale can match the performance of highly-optimized implementations like OpenSSL (§5). Hence, verification does not require compromising on performance. We believe that Vale is the first system to demonstrate formally verified assembly language cryptographic code whose performance matches that of comparable OpenSSL code.

All Vale code and case studies are available on GitHub.[1]

2 Vale Design and Implementation

Vale is currently targeted towards verifying cryptographic assembly language code, which tends to have simple structured control flow and heavy inlining. Therefore, the Vale language includes control constructs such as instructions, inline procedures, conditionals, and loops. Note that these control constructs are independent of any particular features in the underlying logical framework. For example, even when using Dafny as a logical framework, Vale procedures are not executable Dafny methods, Vale while loops are not executable Dafny while loops, and executable Vale code is not compiled with Dafny's compiler (which compiles Dafny's own control constructs to C#). Instead, Vale relies on the logical framework mainly for mathematical reasoning in proofs, and uses executable Dafny code only for specialized tasks like printing assembly language code and static analysis of Vale code.

The Vale language does not contain anything specific to a particular architecture such as x86 or ARM or to a particular assembler such as GCC or MASM. Instead, programmers write Dafny code that defines the syntax and semantics for the architecture of their choice, and then use Vale to manipulate the syntax and semantics. §2.1 introduces examples of such Dafny declarations. §2.2 and §2.3 then present Vale code, demonstrating the flexibility and expressiveness of the Vale language. Finally, §2.4 describes how Vale generates Dafny proofs that make the best use of Dafny and Z3. Figures in these subsections present examples of Dafny and Vale code. Although the syntax for both languages is similar, Vale's syntax (see appendix) is independent from Dafny's syntax; the figure captions specify which code is in which language.

2.1 Dafny declarations

As a running example, the Dafny declarations in Figure 3 define registers, operands, instructions, and structured code for a simplified subset of ARM code. The state of the simplified ARM machine consists of registers and memory. To assist with the operational seman-

[1] URL omitted for double-blind submission.

```
datatype reg = R0 | R1 | R2 | R3 | R4 | R5
| R6 | R7 | R8 | R9 | R10 | R11 | R12 | LR
datatype op =
  op_reg(r:reg)
| op_const(n:uint32)
datatype ins =
  Add(addDst:op, addSrc1:op, addSrc2:op)
| Ldr(ldrDst:op, ldrBase:op, ldrOffset:op)
| Str(strSrc:op, strBase:op, strOffset:op)
datatype cmp = Lt(o1:op, o2:op) | Le(o3:op, o4:op)
datatype code =
  Ins(ins:ins)
| Block(block:codes)
| IfElse(ifCond:cmp, ifTrue:code, ifFalse:code)
| While(whileCond:cmp, whileBody:code)
type codes = list<code>
datatype state = State(ok:bool,
                       regs:map<reg, uint32>,
                       mem:map<int, uint32>)
function evalCode(c:code, s1:state, s2:state):bool
{   match c case Ins(ins) => ...
            case Block(block) => ...
            case IfElse(cond, ifT, ifF) => ...
            case While(cond, body) => ...
}
method PrintCode(c:code) { ... }
```

FIGURE 3—***Example Dafny definitions for simplified ARM.***

tics definitions, the state also contains an `ok` flag to indicate whether the state is considered good (`ok = true`) or crashed (`ok = false`). The operational semantics are defined as a relation `evalCode` that specifies all possible states `s2` that code `c` can reach in a finite number of steps from `s1`. Since any crash happens in a finite number of steps, showing $\forall$`s2 . evalCode(c, s1, s2)` $\Rightarrow$ `s2.ok` proves that `c` starting in state `s1` cannot crash.

Notice that neither the logical framework nor Vale need to know anything about particular assembly language architectures. In fact, the logical framework need not know anything about assembly language at all. This allows Vale to take advantage of existing logical frameworks like Dafny. It also ensures portability across architectures; supporting a new architecture means writing a new set of declarations like those in Figure 3, with no modifications needed to Vale or Dafny. Furthermore, proofs about program behavior are foundational, in the sense of being proven directly in terms of the semantics of assembly language, without having to trust the Vale language or tool. Although Vale builds on standard Hoare rules for conditional statements, while loops, and procedure framing, these rules are expressed as a hand-written library of lemmas, verified relative to the assembly language semantics, as depicted in Figure 2.

Given a machine's semantics, a Dafny programmer could, in theory, construct ASTs for assembly-language programs and try to prove properties about their evaluation. Figure 4 shows an example. It creates a block of code consisting of three Add instructions; proves that the final state is good; proves that one effect of the code is to

```
method Main()
{
  var code := Block(
    cons(Add(op_reg(R7), op_reg(R7), op_const(1))
   ,cons(Add(op_reg(R7), op_reg(R7), op_const(1))
   ,cons(Add(op_reg(R7), op_reg(R7), op_const(1))
   ,nil))));

  ...proof about code...

  assert forall s1:state, s2:state ::
       s1.ok && evalCode(code, s1, s2)
    && s1.regs[R7] < 0xfffffffd
    ==> s2.ok && s2.regs[R7] == s1.regs[R7] + 3;
  PrintCode(code);
}
```

FIGURE 4—***Example Main method in Dafny.***

add three to register R7; and prints the code.

However, `cons(...)` is an awkward syntax for writing assembly language. Furthermore, it's useful to have language support for constructing proofs about complex code. Vale can be thought of as an assistant that generates the `code` variable in the example above and fills in the missing "...proof about code...". Dafny checks the proof against the code, so any mistakes in the proof will be caught by Dafny. This means that Vale is not part of the trusted computing base on which the correctness and security of the code depend.

2.2 Vale procedures

Figure 5 shows some simple Vale procedures. The global variables `ok`, `r0`...`r12`, `lr`, and `mem` represent distinct components of the state type declared in Figure 3. Each procedure declares which of these components it can read (using `reads` clauses) or read and modify (using `modifies` clauses). The effect on the state is expressed using preconditions (`requires` clauses) and postconditions (`ensures` clauses). The `Add3ToR7` procedure, for example, promises to add three to register `r7` under the condition that the initial value or `r7` isn't so big that adding three would cause overflow.

In addition to preconditions and postconditions, Vale also requires loop invariants for loops:

```
while (r7 <= 100)  // example loop in Vale
    invariant r7 <= 103;  // loop invariant
{
    Add3ToR7();
}
```

For each procedure, the Vale tool generates a Dafny function that produces an AST value of type `code`. For example, for the code in Figure 5, it will generate the following Dafny code:

```
function method{:opaque} code_AddOne(r:op):code {
    Block(cons(code_ADD(r, r, op_const(1)), nil()))
}
function method{:opaque} code_Add3ToR7():code {
    Block(cons(code_AddOne(op_reg(R7)),
```

```
var{:state ok()} ok:bool;
var{:state reg(R0)} r0:uint32;
var{:state reg(R1)} r1:uint32;
...
var{:state reg(R12)} r12:uint32;
var{:state reg(LR)} lr:uint32;
var{:state mem()} mem:map(int, uint32);

procedure AddOne(inout operand r:uint32)
    requires r < 0xffffffff;
    ensures  r == old(r) + 1;
{
    ADD(r, r, 1);
}

procedure Add3ToR7()
    modifies r7;
    requires r7 < 0xfffffffd;
    ensures  r7 == old(r7) + 3;
{
    AddOne(r7);
    AddOne(r7);
    AddOne(r7);
}
```

FIGURE 5—*Examples of state declarations and inline procedure declarations in Vale.*

```
procedure{:instruction Ins(Add(dst,src1,src2))}
  ADDW(out operand dst:uint32,
      operand src1:uint32, operand src2:uint32)
  ensures  dst == (src1 + src2) % 0x100000000;

procedure{:instruction Ins(Add(dst,src1,src2))}
  ADD(out operand dst:uint32,
      operand src1:uint32, operand src2:uint32)
  requires 0 <= src1 + src2 < 0x100000000;
  ensures  dst == src1 + src2;

procedure{:instruction Ins(Ldr(dst,base,offset))}
  LDR(out operand dst:uint32,
      operand base:uint32, operand offset:uint32)
  reads mem;
  requires InMem(base + offset, mem);
  ensures  dst == mem[base + offset];

procedure{:instruction Ins(Str(src,base,offset))}
  STR(operand src:uint32,
      operand base:uint32, operand offset:uint32)
  modifies mem;
  requires InMem(base + offset, mem);
  ensures  mem == old(mem)[base + offset := src];
```

FIGURE 6—*Example Vale instruction declarations, including two for the same instruction: both a wrapping (`ADDW`) and non-wrapping (`ADD`) specification of the `Add` instruction.*

```
        cons(code_AddOne(op_reg(R7)),
        cons(code_AddOne(op_reg(R7)), nil())))
}
```

Here, `function method` is Dafny's syntax for a function whose code can be extracted and executed. Both Vale and Dafny use the syntax `{:...}` for attributes. The `opaque` attribute indicates that the function definition will be hidden during proofs except where explicitly revealed.

The leaves of the AST are the individual instructions declared in Figure 3. Programmers declare instructions as Vale procedures with specifications of their choice. They must prove that the specifications are sound with respect to the semantics given by `evalCode`, so these specifications do not have to be trusted.

Multiple procedures with different specifications may be given for the same instruction if different specifications will be more convenient in different situations. For example, the `ADDW` (wrapping add) and `ADD` (non-wrapping add) procedures in Figure 6 both have the same body, a single `Add` instruction. However, `ADD` restricts its input operands so it can provide a simpler postcondition that need not consider the consequences of overflow. This hiding often makes code easier to verify in cases when wrapping is not intended.

The generation of first-class AST values allows programmers to customize the analysis and processing of assembly language code. For example, the `PrintCode` method in Figure 3 can be customized to print assembly language in various formats; our current `PrintCode` emits either GCC or MASM assembly code, depending on a command-line argument. This makes Vale more flexible than tools like BoogieX86 [75] and VCC [55] that hardwire the generation of assembly language output. Indeed, Vale initially only supported MASM output, but adding support for GCC took less than two hours. §3 pushes this flexibility even further, implementing an entire verified information leakage analysis with no modifications to Vale.

2.3 Operands, ghost variables, and inline variables

Parameters to procedures may be operands, ghost variables, or inline variables. The `AddOne` procedure in Figure 5, for example, takes an operand `r` as a parameter. Operands are marked as `in`, `out`, or `inout` to indicate whether the operand is read, written, or both, where `in` is the default. These labels are used in place of `reads` and `modifies` clauses. Register and CISC-style memory operands may be read and/or written, while constant operands may only be read.

Ghost parameters may be used to make specifications about the state easier to express. For example, the ReadA procedure in Figure 7 uses a ghost parameter `a` to help express the memory pointed to by register `r0`. In this case, each 4-byte word of the memory contains one element of the sequence `a`. Ghost parameters are used in the proofs (but not the ASTs) that Vale generates.

Inline parameters, on the other hand, do appear in the ASTs and may be used to specialize the generated code before passing it to `PrintCode`, as seen in Figure 7. The `ReadA` procedure uses an inline `bool` to generate code to load from `r0 + 0` if `b = true`, and to load from `r0 + 4` if `b = false`. The `AddNToR7` procedure uses an inline natural number `n` to repeat the `AddOne` instruction `n` times,

```
procedure ReadA(ghost a:seq(uint32),inline b:bool)
    reads r0; mem;
    modifies r1;
    requires
        length(a) >= 3;
        a[0] <= 100;
        a[1] <= 100;
        forall i :: 0 <= i < length(a) ==>
            InMem(r0 + 4 * i, mem)
         && mem[r0 + 4 * i] == a[i];
    ensures
        b  ==> r1 == a[0] + 1;
        !b ==> r1 == a[1] + 1;
{
    inline if (b) {
        LDR(r1, r0, 0); //load memory [r0+0] into r1
        AddOne(r1);
    } else {
        LDR(r1, r0, 4); //load memory [r0+4] into r1
        AddOne(r1);
    }
}
procedure{:recursive} AddNToR7(inline n:nat)
    modifies r7;
    requires r7 + n <= 0xffffffff;
    ensures  r7 == old(r7) + n;
{
    inline if (n > 0) {
        AddOne(r7);
        AddNToR7(n - 1);
    }
}
```

FIGURE 7—***Ghost and inline parameters in Vale.***

```
function method{:opaque} code_ReadA(b:bool):code
{
    Block(cons((
        if b then Block(cons(code_LDR(
            op_reg(R1), op_reg(R0), op_const(0))
            ...
        else ... )))
}
function method{:opaque} code_AddNToR7(n:nat):code
{
    Block(cons((
        if (n > 0) then Block(cons(
            code_AddOne(sp_op_reg(R7)) ...
        else ...)))
}
```

FIGURE 8—***Dafny code that generates varying assembly code.***

generating a completely unrolled loop.

From these, Vale generates functions parameterized over the inline `b` and `n` variables, so that each `b` and `n` produces a possibly different AST (see Figure 8). In these functions, **`inline if`** statements turn into conditional expressions that generate different code for different inline variable assignments, in contrast to ordinary **`if`** statements that turn into `IfElse` nodes in the AST. Nevertheless, the proofs generated by Vale verify the correctness of the procedures for all possible `b` and `n`. From the proof's perspective, inline variables are no different from ordinary variables and **`inline if`** statements are no different from traditional **`if`** statements. The proofs are checked before picking particular `b` and `n` values to generate and print the code. This may be thought of as a simple form of partial evaluation, analogous to systems like MetaML [71] that type-check a program before partially evaluating it. This sort of partial evaluation provides a principled replacement for Perl scripts and ifdefs; §4.1 describes how these features are used to express and verify the code in Figure 1.

2.4 Vale proofs

Although Vale ultimately proves properties in terms of the underlying machine semantics, it still structures its proofs to take advantage of the automated SMT-based reasoning provided by Dafny and Z3. For each procedure `p`, the Vale tool generates a Dafny lemma which proves that if `p`'s preconditions hold then it does not crash and its postconditions hold. A Dafny lemma is quite similar to a Dafny method: the desired property is declared as the postcondition (i.e., via `ensures` clauses) and proof assumptions are declared as preconditions (i.e., via `requires` clauses). In its simplest form, a Vale proof looks much like the assertion in Figure 4; i.e., it consists of a Dafny lemma

- taking an initial state `s1` and a final state `s2` as parameters,
- requiring `evalCode(p.code, s1, s2)`,
- requiring `s1.ok`,
- requiring that all of `p`'s preconditions hold for `s1`,
- ensuring that all of `p`'s postconditions hold for `s2`, and
- ensuring that any state not mentioned in a `modifies` clause remains the same from `s1` to `s2`.

The lemma's proof (i.e., the body of the lemma) consists largely of calls to the lemmas for other procedures; for example, the proof of Figure 5's lemma for `Add3ToR7` consists mainly of three calls to the lemma for `AddOne`, whose proof consists mainly of one call to the lemma for `ADD`. Vale stitches these calls together by adding additional calls to library lemmas, written in Dafny, for sequential composition, **`if`**/**`else`**, and **`while`** loops.

For some procedures, this simple proof form led to slower-than-expected proof verification by Dafny and Z3. After some investigation, the primary culprit turned out to be Z3's reasoning about long chains of updates to the `state` type and its components `state.regs` and `state.mem`. By itself, reasoning about updates was acceptably fast, but the combination of updates and complex specifications led to slow reasoning.

Therefore, Vale can also generate more sophisticated proofs that factor reasoning about updates and reasoning about specifications into two separate lemmas. An outer lemma reasons about the updates and the well-formedness of a procedure `p`'s states and instructions, but does not

attempt to reason about p's preconditions and postconditions. Instead, the outer lemma calls an inner lemma to do this reasoning. Conversely, the state is never exposed to the inner lemma; instead, the inner lemma reasons only about the components of the state at each step of the evaluation. This frees the inner lemma from reasoning about long chains of updates to the state. The optimized proof form sped up verification of some procedures, such as the Vale code for the SHA ARM code from Figure 1, by as much as a factor of three.

3 Information Leakage Analysis

Since cryptographic code typically operates on secrets, proving it secure requires more than functional correctness; it requires proving the absence of *leakage*. Historically, attackers have exploited two broad categories of leakage: leakage via state and leakage via side channels. Leakage via state occurs when a program leaves secrets or secret-dependent data in registers or memory [20]. Leakage via side channels occurs when an adversary learns secrets by observing aspects of the program's behavior. While physical side channels are a concern [31, 47, 54], digital side channels are typically more problematic, since they can often be exploited by a remote attacker. These side channels include program execution time [11, 17, 46] and (particularly in shared tenancy deployments, such as the cloud) memory accesses [8, 14, 29, 39, 67, 76]. Eliminating such side channels at the source-code level can be difficult, as compilers may optimize away defensive code, or even introduce new side channels [16, 42, 48].

To prove the absence of digital leakage in Vale programs, we developed a novel approach that combines functional verification with a taint-based analyzer proven correct against a simple specification (§3.1). This analyzer, written in Dafny, makes use of Vale's ability to reason about ASTs in a high-level logical framework (§3.2). It also leverages existing specifications (e.g., framing conditions) and invariants from the code's functional verification to greatly simplify analysis (§3.3).

As we discuss in detail in §4, we ran our analyzer on our various cryptographic implementations to prove them free of digital leakage. In the process, we discovered state leakage in OpenSSL.

Overall, because our analyzer is formally verified against a small spec (§5.2), it has far fewer lines of trusted code than prior compiler-aided approaches to detecting side channels [56, 68, 69, 77]. Additionally, since we directly analyze assembly programs, our approach does not suffer from the compiler-introduced side channels discussed above. Compared with prior approaches to formally proving the absence of side channels (e.g., [10]), we invest a one-time effort in verifying our analyzer, which we can then run on an arbitrary number of Vale programs, rather than formally reasoning about side channels for every Vale program we write. Furthermore, previous work struggled with alias analysis and hence resorted to manually inserted assumptions [10], whereas our alias analysis is machine-checked (§3.3).

3.1 Specifying leakage freedom

Below, we first provide some intuition for what it means for a method to be leakage free. We then conclude the section with our formal definition of leakage freedom, a definition based on non-interference [32].

Secret inputs. A method is leakage free if it does not leak secret inputs to an adversary. Thus, part of the specification of leakage freedom is a specification of which inputs are secret. To be conservative, we have the programmer specify the *opposite*: the set of locations PubStartLocs she is sure contain non-secret information. We then treat all other locations as containing secrets.

State leakage. Secrets leak when they can be deduced from architectural state visible to the adversary when the method terminates. Thus, part of the specification of leakage freedom is a specification of which outputs are visible to the adversary. For this, we have the programmer specify a set of locations PubEndLocs that are visible to the adversary upon method termination. To prove leakage freedom, we must prove that these locations' final contents do not depend on secrets.

The programmer may omit from PubEndLocs any locations whose final contents are fully determined by the functional-correctness specification. One useful application of this principle is declassification; e.g., we leave the 32-byte hash computed by SHA-256 out of PubEndLocs since it is a fully specified function of the hash's input. Another common application of this principle is framing, i.e., when the calling convention for a method specifies that it must leave a location unchanged. In this case, there is no need to check for leakage via this location since functional correctness already prevents it.

Cache-based side channels. As shown by Barthe et al., a program is free of cache-based side channels if it does not branch on secrets, and if it performs no secret-dependent memory accesses [13]. Thus, to prove freedom from cache-based side channels, it is sufficient to show the following: if an execution trace records all branches and memory-access locations, then that trace does not depend on secret inputs.

To enable machine-checked verification of cache-based side-channel freedom in Vale, we expand the architectural model of the state with an additional ghost field `trace` that represents the execution trace defined above. We also update our machine semantics to ensure the execution trace captures all branches and memory access locations. For instance, we ensure that a store instruction appends the accessed memory address to the execution trace.

```
predicate isLeakageFree(
    code:code, pubStartLocs:set<location>,
    pubEndLocs:set<location>) {
  forall s, t, s', t' ::
       (  evalCode(code, s, s')
       && evalCode(code, t, t')
       && (forall loc :: loc in pubStartLocs
                      ==> s[loc] == t[loc])
       && s.trace == t.trace )
   ==> (  s'.trace == t'.trace
       && (forall loc :: loc in pubEndLocs
                      ==> s'[loc] == t'[loc]) )
}
```

FIGURE 9—***Correctness specification for leakage freedom.***

Timing-based side channels. Closing cache-based side channels is an important step in closing timing-based side channels, but it is not sufficient. We must also show that inputs to any variable-latency instructions do not depend on secrets [68]. Thus, we update our machine semantics to also capture in `trace` all inputs to variable-latency instructions. This way, if we prove that the trace is independent of secrets then we also prove that the running time is independent of secrets.

Formal definition of leakage freedom. We now present a formal definition of leakage freedom; for its encoding in Dafny, see Figure 9. A Vale method with code Code is leakage free if, for any two successful runs of Code, the following two conditions:

- the two initial states, `s` and `t`, match in every location in PubStartLocs; and
- the execution traces in those initial states, `s.trace` and `t.trace`, are identical

imply the following two outcomes:

- the two final states, `s'` and `t'`, match in every location in PubEndLocs; and
- the execution traces in those final states are identical.

This is an intuitive specification for leakage freedom: for any pair of executions of the program with the same public values but potentially different secrets, the timing and cache behavior of the program are the same in both executions. Hence, any adversary's observations must be independent of the secret values. It is reasonable to only consider successful runs since our functional verification proves that the code always executes successfully.

3.2 Analyzer implementation

Rather than directly proving that each Vale program satisfies our leakage specification, we invest in a one-time effort to write and prove correct a leakage analyzer that can run on any Vale program. Our analyzer takes as input:

- a Vale code value (§2) Code,
- a set of locations (e.g., register names) PubStartLocs assumed to be free of secrets when the code begins, and
- a set of locations PubEndLocs that must be free of secrets when the code ends.

It outputs a Boolean LeakageFree indicating whether the code is leakage free under these conditions.

The analyzer's top-level specification states that the analysis is sound (though it may not be complete); i.e., when the analyzer claims that Code is leakage free, then it satisfies isLeakageFree (Figure 9). More formally, we prove that the analyzer satisfies the following postcondition:

$$\mathsf{LeakageFree} \Rightarrow \mathsf{isLeakageFree}(\mathsf{Code}, \mathsf{PubStartLocs}, \mathsf{PubEndLocs}).$$

We prove this correctness property via machine-checked proofs in Vale's underlying logical framework Dafny [49].

The analyzer's implementation is a straightforward dataflow analysis [44] in the tradition of Denning et al. [27]. The main novelties are that we formally verify the implementation relative to a succinct correctness condition, and that we leverage the knowledge of aliasing present in the functional verification of the Vale programs, as described further in §3.3.

The dataflow analysis checks the code one instruction at a time, keeping track of the set of untainted locations PubLocs. In other words, it maintains the invariant that each location in PubLocs contains only public information. Initially, it sets PubLocs to PubStartLocs. If at any point it concludes that the execution trace may depend on state outside of PubLocs, the analyzer returns `False` to indicate it cannot guarantee leakage freedom. This may happen if a branch predicate might use a location not in PubLocs, or a memory dereference might use the contents of a register not in PubLocs as its base address or offset. Loops are iterated until PubLocs reaches a fixed point. If the analyzer reaches the end of Code, it returns `True` if $\mathsf{PubEndLocs} \subseteq \mathsf{PubLocs}$.

3.3 Memory taint analysis

The main challenge for taint analysis is tracking the taint associated with memory locations. However, given our focus on proving functional correctness of cryptographic code, we observe that we can carefully leverage the work already done to prove functional correctness, to drastically simplify memory taint analysis.

Memory taint analysis is challenging because typically one cannot simply look at an instruction and determine which memory address it will read or write: the address used will depend on the particular dynamic values that happen to be in the registers used as the base and/or offset of the access. Thus, existing tools for analyzing leakage of assembly language code depend on alias analysis, which is often too conservative to verify existing cryptographic code without making potentially unsound assumptions [10].

Our approach to memory taint analysis carefully leverages the work already done to prove functional correctness, since some of that work requires reasoning about the flow of information to and from memory. After all, a program cannot be correct unless it manages that flow correctly. For example, the developer cannot prove SHA-256 correct without proving that the output hash buffer does not overlap with the unprocessed input.

We can push some of the work of memory taint analysis to the developer by relying on her to provide lightweight annotations in the code. In addition to specifying which addresses are expected to contain public information on entry and exit, she must make a similar annotation for each load and store. This annotation consists of a bit indicating whether she expects the instruction to access public or secret information. For CISC instructions where each operand may implicitly specify a load or store, she must annotate each such memory-accessing operand with a bit. Crucially, however, we do not rely on the correctness of these annotations for security. If the developer makes a mistake, it will be caught during either functional-correctness verification or during leakage analysis.

These annotations make our analyzer's handling of memory taint straightforward. The analyzer simply labels any value resulting from a load public or secret based on the load instruction's annotation. The analyzer also checks, for each store annotated as public, that the value being stored is actually public.

To ensure that annotation errors will be caught during functional correctness verification, we expand the architectural model of the state with an additional ghost field `pubaddrs`, representing the set of addresses currently containing public information. A store adds its address to, or removes it from, `pubaddrs`, depending on whether the annotation bit indicates the access is public or secret. A load annotated as public fails (i.e., causes the state's `ok` field to become `False`) if the accessed address is not in `pubaddrs`.

Thus, the developer is obligated to prove, before performing a load, that the accessed address is in `pubaddrs`. She can do this by adding it as a precondition, or by storing public information into that address. She must also prove that any intervening store of secret information does not overwrite the public information; in other words, she must perform her own alias analysis. Note, however, that she must already perform such alias analysis to prove her code correct, so we are not asking her to do more work than she would already have had to perform, given our goal of functional correctness.

4 Case Studies

To illustrate Vale's capabilities, we present four case studies of high-performance cryptographic code we built with it.

4.1 OpenSSL SHA-256 on ARM

As we describe in more detail in §5.1, to identify a baseline for our performance, we measured the performance of six popular cryptographic libraries. For the platforms and algorithms we evaluated, OpenSSL consistently proved to be the fastest.

To achieve this performance, OpenSSL code tends to be highly complex, as illustrated by the SHA-256 code snippet in Figure 1. Note that this code is not written directly in a standard assembly language, but is instead expressed as a Perl subroutine that generates assembly code. This lets OpenSSL improve performance by calling the subroutine 16 times to unroll the loop:

```
for($i=0;$i<16;$i++) {
  &BODY_00_15($i,@V); unshift(@V,pop(@V));
}
```

Furthermore, each unrolled loop iteration is customized with a different mapping from SHA variables a...h to ARM registers r4...r11 (stored in the `@V` list). This reduces register-to-register moves and further increases performance. Finally, a combination of Perl-based run-time checks (`if ($i < 16)`) and C preprocessor macros are used to select the most efficient available instructions on various versions of the ARM platform, as well as to further customize the last loop iteration (`i = 15`).

OpenSSL's use of Perl scripts is not limited to SHA on ARM. It implements dozens of cryptographic algorithms on at least 50 different platforms, including many permutations of x86 and x64 (with and without SSE, SSE2, AVX, etc.) and similarly for various versions and permutations of ARM (e.g., with and without NEON support). Many of these implementations rely on similar mixes of assembly, C preprocessor macros, and Perl scripts. The difficulty of understanding such code-generating code is arguably a factor in the prevalence of security vulnerabilities in OpenSSL.

To demonstrate Vale's ability to reason about such complex code, we ported all of OpenSSL's Perl and assembly code for SHA-256 on ARMv7 to Vale. This code takes the current digest state and an arbitrary array of plaintext blocks, and compresses those blocks into the digest. C code handles the padding of partial blocks.

Porting the Perl and assembly code itself was relatively straightforward and mostly involved minor syntactic changes. The primary challenge was recreating in our minds the invariants that the developers of the code presumably had in theirs. As Figure 1 shows, the code comments are minimalist and often cryptic, e.g.,

```
eor  $t3,$B,$C    @ magic
ldr  $t1,[sp,#`($i+2)%16`*4]    @ from future BODY_16_xx
```

In the second line, the odd syntax with the backticks is used when the Perl code makes a second pass over the string representing the assembly code, this time acting as an interpreter to perform various mathematical operations,

like `($i+2)%16`.

As discussed above, OpenSSL's code generation relies on many Perl-level tricks that we replicated in Vale. For example, we use inline parameters to unroll loops and conditionally include certain code snippets, similar to how the Perl code does. The Perl code also carefully renames the Perl variables that label ARM registers in each unrolled loop in order to minimize data movement. To support this, our corresponding Vale procedure takes an inline loop iteration parameter i, and eight generic operand arguments. The mapping from operand to ARM register is then added as a statically verified function of i, e.g.,

```
requires @h == OReg(4+(7-(i%8))%8)
```

This requires the `h` operand ("`@h`" refers to the operand `h`, as opposed to the value in the operand `h`) to be R11 on the first iteration, R10 on the next iteration, etc., which essentially shifts the contents of the SHA variable h in iteration i into the SHA variable g in iteration $i+1$ (and similarly for the other state variables, some of which are updated in more complex ways).

To prove the functional correctness of our code, we verified it against the Dafny SHA-256 specification from the Ironclad project [36], itself based off the FIPS 180-4 standard [59]. This proof required several auxiliary lemmas, typically to help guide Z3 when instantiating quantifiers, or to reveal certain function definitions that we hide by default to improve verifier performance. We also took advantage of Z3's bit-vector theory to automatically discharge relations like:

$$(x\&y) \oplus (\sim x\&z) == ((y \oplus z)\&x) \oplus z$$

which allow OpenSSL's code to optimize various SHA steps (e.g., the relation above saves an instruction by computing the right-hand side).

To demonstrate that our implementation is not only correct but side-channel and leakage free, we ran our verified analysis tool (§3) on the Vale-generated AST. To our surprise (given that the code is a direct translation of OpenSSL's implementation), the tool reported that the implementation was free of leakage via side channels, but not via state. Indeed, further investigation showed that while OpenSSL's C implementation carefully scrubs its intermediate state, after OpenSSL's assembly implementation returns, the stack still contains most of the caller's registers and 16 words from the expanded version of the final block of hash input. We do not know of an attack to exploit this leakage, but in general, leakage like this can undermine security [20].

Discussions with the OpenSSL security team indicate that while they aim to always scrub key material from memory, the remainder of their scrubbing efforts are ad hoc due to their unusual threat model [66]. On the one hand, OpenSSL usually runs in process with an application, and hence everything in the address space is trusted; nonetheless, they feel an instinctual need to scrub memory when they can do so without too much performance overhead. Because they do not have a precise and systematic way to identify "tainted" memory and scrub it efficiently, leaks like the one we identified are tolerated.

Tools like Vale offer one approach to systematically track leakage and provably and efficiently block it. Indeed, after we added the appropriate stack scrubbing to our implementation, our analyzer confirmed that it was free of both side channels and leakage.

4.2 SHA-256 on x86

To demonstrate Vale's generality across platforms, we also used it to write an x86 version of SHA-256's core. This required writing a trusted semantics for a subset of Intel's architecture, a trusted printer to translate instructions into assembly code, and a verified proof library (which in many cases differs very little from our corresponding ARM library). For the SHA implementation, we wrote the code from scratch, rather than copying the algorithm from OpenSSL. At no point did we need to change Vale itself.

One of the benefits of Vale's platform generality is that we can write and use high-level lemmas that are platform-independent. For example, we reused most of the lemmas from §4.1, since each describes a series of operations necessary to prove that a SHA step was correctly taken. We also used the same specification for SHA as we did on ARM.

When we ran our verified analysis tool on our code, it confirmed that it was leakage free.

4.3 Poly1305 on x64

We also ported the 64-bit non-SIMD code for Poly1305 [15] from OpenSSL to Vale. OpenSSL's Poly1305 is a mix of C and assembly language code. We began by writing a trusted semantics for a subset of x64. We then verified the OpenSSL assembly language code for the Poly1305 main loop and added our own initialization and finalization code in assembly language to replace the C code, resulting in a complete Vale implementation of Poly1305. Except for an extra instruction for the while-loop condition, our main loop code is identical to the OpenSSL code. This forced us to verify the mathematical tricks that underlie OpenSSL's efficient 130-bit multiplication and mod implementations. For this verification, Z3's automated reasoning about linear integer arithmetic was quite useful, helping us, for example, to maintain invariants on the size of intermediate values that sometimes exceed 130 bits by various amounts. These invariants are crucial to establish that numbers are eventually reduced all the way to their 130-bit form and that enough carry bits are propagated through larger interme-

diate values. In fact, a bug fixed in March 2016 [64] was due to not propagating carry bits through enough digits; Vale's verification, of course, catches this bug.

4.4 AES-CBC using x86 AES-NI instructions

Our final case study illustrates Vale's ability to support complex, specialized instructions. Specifically, we used Vale to implement the AES-128 CBC encryption mode using the AES-NI instructions provided by recent Intel CPUs [35]. AES is a block cipher that takes a fixed amount of plaintext; cipher-block chaining (CBC) is an encryption mode that applies AES repeatedly in order to encrypt an arbitrary amount of plaintext. In 2008, Intel introduced AES-NI instructions to both increase the performance of the core AES block cipher and make it easier to write side-channel free code, since software implementations of AES typically rely on in-memory lookup tables which are expensive to make side-channel free. As we quantify in §5.1, implementations that take advantage of this hardware support are easily 3.5-4.0× faster than traditional hand-tuned assembly that does not.

For this case study, we extended our x86 model from §4.2 by adding support for 128-bit XMM registers, definitions for Intel's six AES-support instructions [34], and four generic XMM instructions [38]. None of these extensions required changes to Vale. We also wrote a formal specification for AES-CBC based on the official FIPS specification [58].

Our implementation follows Intel's recommendations for how to perform AES-128 [34]. However, unlike the code provided by Intel, our code includes a proof of its correctness. We also ran our verified analysis tool on the code to confirm that it was leakage free.

The implementation involves an elaborate sequence of AES-NI instructions interwoven with generic XMM instructions. Proving it correct is non-trivial, particularly since Intel's specifications for its instructions assume various properties of the AES specification that we had to prove as part of our efforts. For example, we had to prove that the AES `RotWord` step commutes with its `SubWord` step.

5 Evaluation

In our evaluation, we aim to answer two questions: **(1)** Can our verified code meet or beat the performance of state-of-the-art unverified cryptographic libraries? **(2)** How much time and effort is required to verify our cryptographic code?

5.1 Comparative performance

To compare our performance to a state-of-the-art implementation, we first measure the performance of six popular cryptographic libraries: BoringSSL [1], Botan [2], Crypto++ [3], GNU libgcrypt [5], ARM mbedTLS (for-

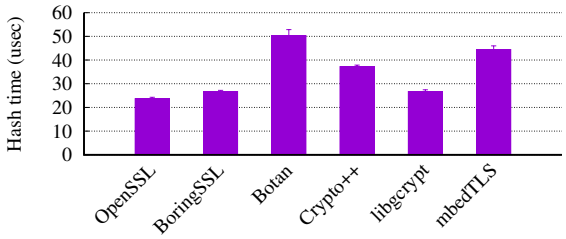

FIGURE 10—***Time for various libraries to compute the SHA-256 hash of 10 KB of random data. Each data point averages 100 runs.***

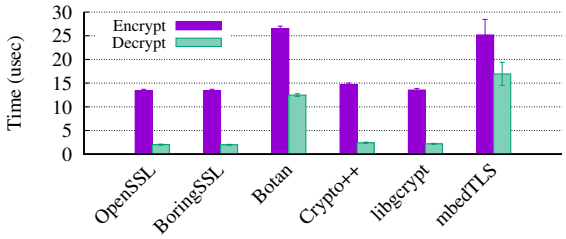

FIGURE 11—***Time for various libraries to encrypt and decrypt 10 KB of random data using AES-CBC. Decryption is generally faster because it is more parallelizable. Each data point averages 100 runs.***

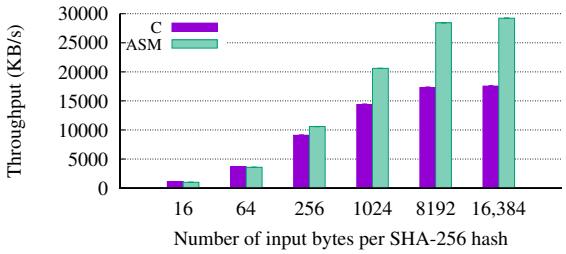

FIGURE 12—***Throughput comparison of two OpenSSL SHA-256 routines for ARM: one written in C and one written in hand-tuned assembly. Each data point averages 10 runs.***

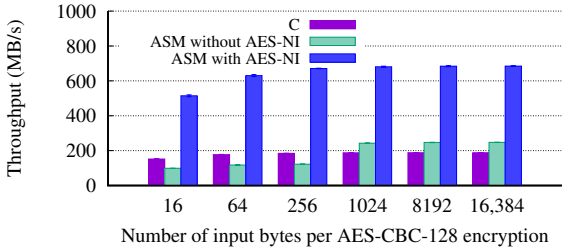

FIGURE 13—***Throughput comparison of three OpenSSL AES-CBC-128 routines for x86: one written in C, one written in hand-tuned assembly with only scalar instructions, and one written in hand-tuned assembly using SSE and AES-NI instructions. Each data point averages 10 runs.***

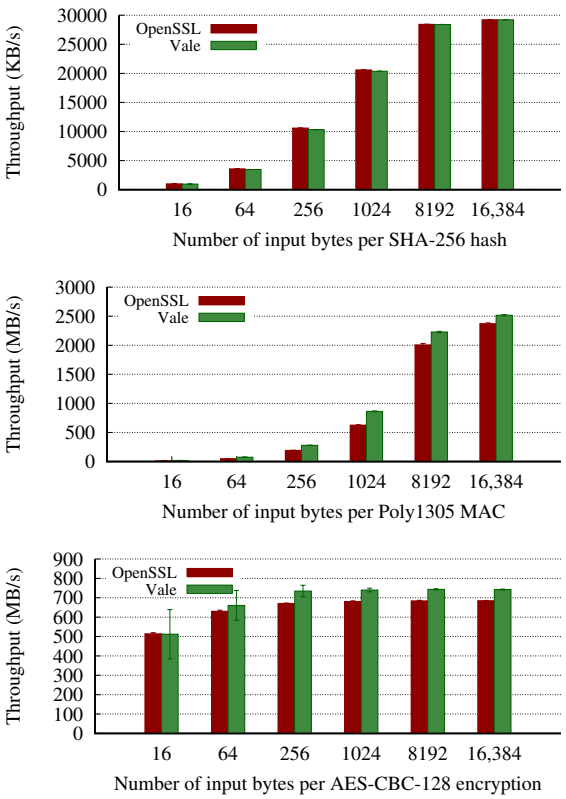

FIGURE 14—***Comparing Vale implementations to OpenSSL's, for SHA-256 on ARM, for Poly1305 on x64, and for AES-CBC-128 on x86. Each data point averages 10 runs.***

merly PolarSSL) [6], and OpenSSL [7]. We collect the measurements on a G5 Azure virtual machine running Ubuntu 16.04 on an Intel Xeon E5 v3 CPU and configured with enough dedicated CPUs to ensure sole tenancy. Each reported measurement is the average from 100 runs and, as is the case in all figures in this paper, error bars indicate 95% confidence intervals.

As shown in Figures 10 and 11, our results support the anecdotal belief that, in addition to being one of the most popular TLS libraries [60], OpenSSL's cryptographic implementations are the fastest available. Hence, in our remaining evaluation, we compare our performance with OpenSSL's. These strong OpenSSL performance results suggest that OpenSSL's Byzantine mix of Perl and hand-written assembly code (recall Figure 1) does result in noticeable performance improvements compared to the competition. As further support for the need for hand-written assembly code, Figures 12 and 13 compare the performance of OpenSSL's C implementations (compiled with full optimizations) to that of its hand-written assembly routines. We see that OpenSSL's assembly code for SHA-256 on ARM gets up to 67% more throughput than its C code, and its assembly code for AES-CBC-128 on x86 gets 247–300% more throughput than its C code due to the use of SSE and AES-NI instructions.

To accurately compare our performance with OpenSSL's, we make use of its built-in benchmarking tool `openssl speed` and its support for extensible cryptographic engines. We register our verified Vale routines as a new engine and link against our static library. Surprisingly, in collecting our initial measurements, we discovered that OpenSSL's benchmarking tool does not actually conduct a fair comparison between the built-in algorithms and those added via an engine. Calls via an engine perform several expensive heap allocations that the built-in path does not. Hence, the "null" engine that returns immediately actually runs slower than OpenSSL's hashing routine! To get a fair comparison, we create a second engine that simply wraps OpenSSL's built-in routines. We report comparisons between this engine and ours.

We compare Vale's performance with OpenSSL's on three platforms. We compare our ARM implementation (§4.1) with OpenSSL's by running them on Linux (Raspbian Jessie) on a Raspberry Pi with a 900MHz quad-core ARM Cortex-A7 CPU and 1GB of RAM. For this, we compile OpenSSL to target ARMv7 and above, but without support for NEON, ARM's SIMD instructions. For Intel x64, we compare our Poly1305 (§4.3) implementation with OpenSSL's with SIMD disabled. Finally, to show that we can take advantage of advanced instructions, on Intel x86, we measure our AES-CBC-128 (§4.4) implementation against OpenSSL's with full optimizations enabled, including the use of AES-NI and SIMD instructions. We collect the x86/x64 measurements on Windows Server 2016 Datacenter using the same Azure instance as in §5.1.

Figure 14 summarizes our comparative results. They show that, for SHA-256 and AES-CBC-128, Vale's performance is nearly identical to OpenSSL's. Indeed, our AES-CBC-128 implementation is up to 9% faster than OpenSSL's, due to our more aggressive loop unrolling. Our Poly1305 implementation also slightly outperforms OpenSSL, largely due to using a complete assembly implementation rather than a mix of C and assembly. These positive results should be taken with a grain of salt, however. For real TLS/SSL connections, for instance, OpenSSL typically calls into an encryption mode that computes four AES-CBC ciphertexts in parallel (to support, e.g., multiple outbound TLS connections) to better utilize the processor's SIMD instructions. Our Vale implementation does not yet support a similar mode.

5.2 Verification time and code size

Table 1 summarizes statistics about our code. In the table, specification lines include our definitions of ARM and Intel semantics, as well as our formal specification for the cryptographic algorithms. Implementation lines consist of assembly instructions and control-flow code we write

Component	Spec	Impl	Proof	ASM	Verification time (min)
	(Source lines of code)				
ARM	873	170	650	–	0.6
x86	1565	256	1000	–	2.1
x64	1999	377	1392	–	3.8
Common Libraries	1100	252	4302	–	1.2
SHA-256 (ARM)	237	330	1424	2085	6.6
SHA-256 (x86)		598	2265	4345	5.7
Poly1305 (x64)	47	325	1155	202	2.7
AES-CBC (x86)	413	432	2296	311	8.2
Taint analysis	217	1276	2116	–	4.3
Total	6451	4016	16600	6943	35.3

TABLE 1—***System Line Counts and Verification Times.***

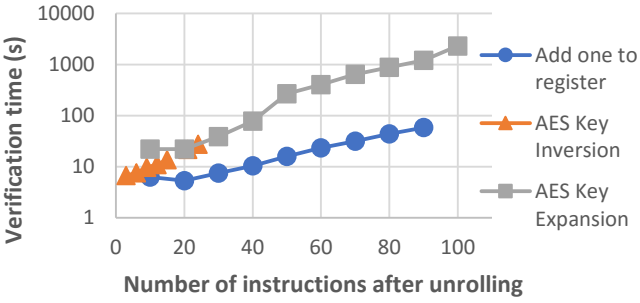

FIGURE 15—***Time to verify various loops, if verification is done after unrolling.***

in Vale itself, whereas ASM counts the number of assembly instructions emitted by our verified code. Proof lines count all annotations added to help the verifier check our code, e.g., pre- and post-conditions, loop invariants, assertions, and lemmas. Vale itself is 5,277 lines of untrusted F# code.

The overall verification time for all the hand-written Dafny code and Vale-generated Dafny code is about 56 minutes, with the bulk of the time in the procedures constituting the cryptographic code. Most procedures took no more than around 10 seconds to verify, with the most complex procedures taking on the order of a minute. The reasonably fast turnaround time for individual procedures is important, because the developer spends considerable time repeatedly running the verifier on each procedure when developing and debugging it.

A key design decision in Vale is the verification of inlined procedures and unrolled loops before the inlining and unrolling occurs. Furthermore, as discussed in §4.1, Vale supports operand renaming for inlined procedures and unrolled loops, allowing us to match OpenSSL's Perl-based register renaming. Figure 15 quantifies the benefits of this decision by showing the cost of verifying code *after* unrolling (note the log scale). The three lines show:

- the cost of verifying an unrolled loop consisting entirely of x86 `add eax, 1` instructions, ranging from 10 to 100 total instructions;
- the cost of verifying an unrolled loop of x86 AES key inversions, up to the maximum 9 iterations performed by AES, where each iteration consists of 3 instructions; and
- the cost of verifying an unrolled loop of x86 AES key expansions, up to the maximum 10 iterations performed by AES, where each iteration consists of 10 instructions.

If 100 adds are unrolled before verification, verification takes 58 seconds, which is tolerable, though much slower than verifying before unrolling. If all 10 AES key expansion iterations are unrolled, verification takes 2300 seconds, compared to 105 seconds for verifying before unrolling. Finally, Dafny/Z3 failed to verify the AES key inversion for 6 unrolled iterations and 9 unrolled iterations, indicating that SMT solvers like Z3 are still occasionally unpredictable. Verifying code before inlining and unrolling helps mitigate this unpredictability and speeds up verification.

Vale	Taint	1st SHA	AES-CBC	SHA port	Poly1305
12	6	6	5	.75	0.5

TABLE 2—***Person months per component.***

5.3 Verification effort

Table 2 summarizes the approximate amount of time we spent building Vale, our verified taint analyzer, and our case studies. Our first implementations of SHA-256 and of AES-CBC were developed in parallel with Vale itself. They helped push the tool to evolve, but they also required multiple rewrites as Vale changed. Once Vale stabilized, porting SHA-256 to other architectures and implementing Poly1305 from scratch (including specifications, code, and proof) went much more rapidly, though the effort required is still non-trivial.

As further evidence for Vale's usability, an independent project is using Vale to develop a microkernel; several researchers in that project are new to verification and yet are able to make progress using Vale. Their efforts have also proceeded without the need to modify Vale.

6 Related Work

Other projects have verified correctness and security properties of cryptographic implementations written in C or other high-level languages, either using Coq [12] or SMT solvers [18, 30, 78]. We hope to connect verified assembly language code to such high-level languages in the future.

The Vale language follows in the path of earlier work on Bedrock [19] and x86proved [41, 43], which used Coq to build assembly language macros for various control constructs like IF, WHILE, and CASE. Vale attempts to make these approaches easier to use by leveraging an

SMT-based logical framework and providing features like mutable ghost variables and inline variables. Furthermore, although earlier tools have been used to synthesize cryptography code [28], Vale has been used to verify existing OpenSSL assembly language code, where optimizations are nontrivial. Vale also includes a verified leakage and side-channel analysis.

mCertiKOS [23], based on Coq, does address information flow, although they do not consider timing and memory channels. In contrast to the verification done for mCertiKOS, which was targeted at a single difficult system (a small kernel), our information analysis tool can run automatically on many pieces of code. Such a tool is useful for verifying large suites of cryptographic code. Dam et al. [24] do address timing, but they approximate by assuming each instruction takes one machine cycle.

Other assembly language verifiers like BoogieX86 [75], used by Ironclad [36], and VCC's assembly language [55] have built on SMT solvers, but do not expose ASTs and assembly language semantics as first-class constructs, and thus are neither as flexible nor as foundational as Vale, Bedrock, and x86proved. For example, BoogieX86 and Ironclad could not support verified loop unrolling, and they were tied to the x86 architecture, and hence would require tool changes to support ARM and x64.

Both BoogieX86 and Almeida et al. [10] leverage SMT solvers for information flow analysis. BoogieX86's analysis was very flexible, but was considerably slower than Vale's taint-based approach and did not address timing and memory channels. As discussed in more detail in §3, Almeida et al. address side channels, but do not prove correctness of the cryptographic code, and resort to unproven assumptions about aliasing. Furthermore, they analyze intermediate code emitted by the LLVM compiler, whereas Vale verifies assembly code. This distinction is relevant since a compiler may choose to implement an IR-level instruction (e.g., `srem`) using a sequence of variable-latency assembly instructions (e.g., `idiv`). Also, their analysis is tied to the LLVM compiler's code generation strategy, whereas ours is not.

Myreen et al. [57] apply common proofs across similar pieces of code for multiple architectures by decompiling the assembly language code to a common format. We use a different approach to sharing across architectures: write each architecture's code as a separate Vale procedure, but share lemmas about abstract state between the procedures.

Many attacks based on side channels have been demonstrated [8, 9, 11, 14, 17, 29, 33, 37, 39, 40, 67, 74, 76]. Although we focus on eliminating side channels by statically verifying precise constant-time execution, other side channel mitigations exist, such as compiler transformations [4, 21, 56, 68, 69, 77] operating system or hypervisor modifications [45, 53], microarchitectural modifications [51, 52], or new cache designs [72, 73].

7 Conclusions and Future Work

Vale is our programming language and tool for writing and proving properties of high-performance cryptographic assembly code. It is assembler-neutral and platform-neutral in that developers can customize it for any assembler by writing a trusted printer, or for any architecture by writing a trusted semantics. It can thus support even advanced instructions, as we demonstrate with an x86 implementation of AES-128/CBC that leverages SSE and AES-NI instructions. Also, as we have shown with our implementations of SHA-256 on both ARM and x86, developers can reuse proofs and specifications for code across architectures. Vale supports reasoning about extracted code objects in a general-purpose high-level verification language; as an illustration of this style of reasoning, we have built and verified an analyzer that can declare a program free of digital information leaks. This analyzer uses taint tracking with a unique approach to alias analysis: instead of settling for a conservative, unsound, or slow analysis, it leverages the address-tracking proofs that the developer must already write to prove functional correctness of her code.

While Vale uses Dafny as a target verification language, the fact that we could use such an off-the-shelf tool with no customization suggests we can also support others. In the future, we hope to demonstrate this by adding support for targets like F* [70], Lean [26], and Coq [22].

By porting OpenSSL's SHA-256 ARM code and Poly1305 x64 code, we have shown that Vale can prove correctness and security properties for existing code, even if it is highly complex. We plan to continue porting additional variants of these algorithms (e.g., adding SIMD support for SHA and Poly1305 for a performance boost of 23-41% [61, 62]), and many others. Ultimately, we hope Vale will enable the creation of a complete cryptographic library providing fast, provably correct, and provably leakage-free code for a wide variety of platforms.

References

[1] BoringSSL. https://boringssl.googlesource.com/boringssl. Commit 0fc7df55c04e439e765c32a4dd93e43387fe40be.

[2] Botan. https://github.com/randombit/botan.git. Commit 9eda1f09887b8b1ba5d60e1e432ebf7d828726db.

[3] Crypto++. https://github.com/weidai11/cryptopp.git. Commit 432db09b72c2f8159915e818c5f34dca34e7c5ac.

[4] ctgrind: Checking that functions are constant time with Valgrind. `https://github.com/agl/ctgrind`.

[5] GNU Libgcrypt. https://www.gnu.org/software/libgcrypt/. Version 1.7.0.

[6] mbedTLS. https://github.com/ARMmbed/mbedtls.git. Commit 9fa2e86d93b9b6e04c0a797b34aaf7b6066fbb25.

[7] OpenSSL. https://github.com/openssl/openssl. Commit befe31cd3839a7bf9d62b279ace71a0efbdd39b0.

[8] O. Aciiçmez, B. B. Brumley, and P. Grabher. New results on instruction cache attacks. In *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, Aug. 2010.

[9] O. Aciiçmez and J.-P. Seifert. Cheap hardware parallelism implies cheap security. Sept. 2007.

[10] J. B. Almeida, M. Barbosa, and G. Barthe. Verifying constant-time implementations. In *Proceedings of the USENIX Security Symposium*, 2016.

[11] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. On subnormal floating point and abnormal timing. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2015.

[12] A. W. Appel. Verification of a cryptographic primitive: Sha-256. *ACM Trans. Program. Lang. Syst.*, 37(2):7:1–7:31, Apr. 2015.

[13] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie. System-level non-interference for constant-time cryptography. In *Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Nov. 2014.

[14] D. J. Bernstein. Cache-timing attacks on AES. `https://cr.yp.to/papers.html#cachetiming`, 2005.

[15] D. J. Bernstein. The poly1305-aes message-authentication code. In *Proceedings of Fast Software Encryption*, Mar. 2005.

[16] C. L. Biffle. NaCl/x86 appears to leave return addresses unaligned when returning through the springboard. `https://bugs.chromium.org/p/nativeclient/issues/detail?id=245`, Jan. 2010.

[17] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proceedings of the USENIX Security Symposium*, Aug. 2003.

[18] Y.-F. Chen, C.-H. Hsu, H.-H. Lin, P. Schwabe, M.-H. Tsai, B.-Y. Wang, B.-Y. Yang, and S.-Y. Yang. Verifying Curve25519 software. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 299–309, 2014.

[19] A. Chlipala. 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*, 2013.

[20] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the USENIX Security Symposium*, 2004.

[21] J. V. Cleemput, B. Coppens, and B. De Sutter. Compiler mitigations for time attacks on modern x86 processors. Jan. 2012.

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

[23] D. Costanzo, Z. Shao, and R. Gu. 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)*, June 2016.

[24] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Nov. 2013.

[25] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[26] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean theorem prover. In *Proc. 25th International Conference on Automated Deduction (CADE-25)*, 2015.

[27] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[28] A. Erbsen. Cryptographic primitive code generation in Fiat. https://github.com/mit-plv/fiat-crypto.

[29] N. J. A. Fardan and K. G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2013.

[30] Galois. The SAW scripting language. https://github.com/GaloisInc/saw-script.

[31] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, May

2001.

[32] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1982.

[33] J. Großschädl, E. Oswald, D. Page, and M. Tunstall. Side-channel analysis of cryptographic software via early-terminating multiplications. In *Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Dec. 2009.

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

[35] S. Gueron and M. E. Kounavis. New processor instructions for accelerating encryption and authentication algorithms. *Intel Technology Journal*, 13(2), 2009.

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

[37] C. Hunger, M. Kazdagli, A. Rawat, A. Dimakis, S. Vishwanath, and M. Tiwari. Understanding contention-based channels and using them for defense. In *International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2015.

[38] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, Aug. 2012.

[39] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb. 2012.

[40] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2012.

[41] J. B. Jensen, N. Benton, and A. Kennedy. High-level separation logic for low-level code. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, 2013.

[42] T. Kaufmann, H. Pelletier, S. Vaudenay, and K. Villegas. When constant-time source yields variable-time binary: Exploiting Curve25519-donna built with MSVC 2015. In *Proceedings of the International Conference on Cryptology and Network Security (CANS)*, Nov. 2016.

[43] A. Kennedy, N. Benton, J. B. Jensen, and P.-E. Dagand. Coq: The world's best macro assembler? In *Proceedings of the Symposium on Principles and Practice of Declarative Programming (PPDP)*, 2013.

[44] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, 1973.

[45] T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the USENIX Security Symposium*, Aug. 2012.

[46] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of the International Cryptology Conference (CRYPTO)*, pages 104–113, 1996.

[47] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 1999.

[48] N. Lawson. Optimized memcmp leaks useful timing differences. `https://rdist.root.org/2010/08/05/optimized-memcmp-leaks-useful-timing-differences`, Aug. 2010.

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

[50] K. R. M. Leino and N. Polikarpova. Verified calculations. In E. Cohen and A. Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments — 5th International Conference, VSTTE 2013, Revised Selected Papers*, volume 8164 of *LNCS*, pages 170–190. Springer, 2014.

[51] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. GhostRider: A hardware-software system for memory trace oblivious computation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2015.

[52] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. PHANTOM: Practical oblivious computation in a secure processor. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Nov. 2013.

[53] R. Martin, J. Demme, and S. Sethumadhavan. TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2012.

[54] R. J. Masti, D. Rai, A. Ranganathan, C. Müller, L. Thiele, and S. Capkun. Thermal covert channels on multi-core platforms. In *Proceedings of the USENIX Security Symposium*, Aug. 2015.

[55] S. Maus, M. Moskal, and W. Schulte. Vx86: x86 assembler simulated in C powered by automated the-

orem proving. In *Proceedings of the Conference on Algebraic Methodology and Software Technology*, 2008.

[56] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Dec. 2005.

[57] M. O. Myreen, M. J. C. Gordon, and K. Slind. Decompilation into logic - improved. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Oct. 2012.

[58] National Institute of Standards and Technology. Announcing the ADVANCED ENCRYPTION STANDARD (AES). Federal Information Processing Standards Publication 197, Nov. 2001.

[59] National Institute of Standards and Technology. Secure Hash Standard (SHS), 2012. FIPS PUB 180-4.

[60] Netcraft Ltd. September 2016 Web server survey, Sept. 2016.

[61] OpenSSL. Poly1305-x86_x64. `https://github.com/openssl/openssl/blob/master/crypto/poly1305/asm/poly1305-x86_64.pl`.

[62] OpenSSL. Sha256-armv4. `https://github.com/openssl/openssl/blob/master/crypto/sha/asm/sha256-armv4.pl`.

[63] OpenSSL. Chase overflow bit on x86 and ARM platforms. GitHub commit dc3c5067cd90f3f2159e5d53c57b92730c687d7e, Apr. 2016.

[64] OpenSSL. Don't break carry chains. GitHub commit 4b8736a22e758c371bc2f8b3534dc0c274acf42c, Mar. 2016.

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

[66] OpenSSL Developer Team. Private communication, Jan. 2017.

[67] C. Percival. Cache missing for fun and profit, 2005.

[68] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *Proceedings of the USENIX Security Symposium*, Aug. 2015.

[69] A. Rane, C. Lin, and M. Tiwari. Secure, precise, and fast floating-point operations on x86 processors. In *Proceedings of the USENIX Security Symposium*, Aug. 2016.

[70] N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming (ICFP)*, pages 266–278, 2011.

[71] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, 1997.

[72] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2007.

[73] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Nov. 2008.

[74] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2015.

[75] J. Yang and C. Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[76] Y. Yarom, D. Genkin, and N. Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. In *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, Aug. 2010.

[77] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, June 2012.

[78] J. K. Zinzindohoue, E.-I. Bartzia, and K. Bhargavan. A verified extensible library of elliptic curves. *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, 2016.

A Vale Grammar

For reference, this appendix contains an annotated grammar for the Vale language. We use * to indicate zero or more repetitions, *, to indicate zero or more repetitions separated by commas, $^{+,}$ to indicate one or more repetitions separated by commas, and $^{*;}$ to indicate zero or more repetitions, each terminated by a semi-colon. In most places, vertical bars divide grammar alternatives and square brackets surround optional grammatical components, but in a few places where we think their usage is clear, we abuse this notation and use vertical bars and square brackets to stand for themselves in the grammar. Other punctuation stands for itself. Lower case letters stand for identifiers and are suggestive of what kind of identifiers they represent.

At the top level, a Vale program consists of a some number of declarations.

```
PROGRAM ::=
| DECL*
```

A declaration introduces a variable, function, or procedure, or provides some declarations in the language of the underlying logical framework (Dafny) to be included verbatim.

```
DECL ::=
| var x : TYPE ;
| function f ( FORMAL*, ) : TYPE [:= f] ;
| procedure p ( PFORMAL*, )
    [returns ( PRET*, )] SPEC* { STMT* }
| procedure p ( PFORMAL*, )
    [returns ( PRET*, )] SPEC* extern ;
| procedure p ( PFORMAL*, )
    [returns ( PRET*, )] := p ;
| VERBATIM-DECL-BLOCK
```

A FORMAL represents a formal parameter of a function or a bound variable. It is simply an identifier and an optional type. An attempt is made to infer any omitted types. Procedure parameters are broken down into two categories, PFORMAL and PRET, the latter of which is used for parameters that are only being returned from the procedure.

```
FORMAL ::=
| x [: TYPE]

PFORMAL ::=
| ghost x : TYPE
| inline x : TYPE
| TYPE x : TYPE
| out TYPE x : TYPE
| inout TYPE x : TYPE

PRET ::=
| ghost x : TYPE
| TYPE x : TYPE
```

Types are declared in the underlying logical framework and referred to in Vale by name. Types can be parameterized by other types. In addition, Vale supports tuple types.

```
TYPE ::=
| t
| TYPE ( TYPE*, )
| tuple ( TYPE*, )
| ( TYPE )
```

A procedure can be declared with specification clauses. A **reads** or **modifies** clause says which global variables the procedure may read or write, respectively. The keywords **requires** and **ensures** are used to introduce pre- and postconditions. Since it often happens that a procedure both requires and ensures some invariant condition, there is a specification clause that avoids the syntactic repetition of such conditions.

```
SPEC ::=
| reads x*;
| modifies x*;
| requires LEXP*;
| ensures LEXP*;
| requires / ensures LEXP*;

LEXP ::=
| EXP
| let FORMAL := EXP
```

Procedure bodies have statements.

```
STMT ::=
| assume EXP ;
| assert EXP ;
| assert EXP by { STMT* }
| calc [CALCOP] { CALC* }
| reveal f ;
| p ( EXP , ... , EXP) ;
| ASSIGN ;
| [ghost] var x [: TYPE] [:= EXP] ;
| forall FORMAL*, TRIGGER*
    [:| EXP] :: EXP { STMT* }
| exists FORMAL*, TRIGGER* :: EXP ;
| while ( EXP ) INVARIANT* DECREASE
    { STMT* }
| for ( ASSIGN*, ; EXP ; ASSIGN*, )
    INVARIANT* DECREASE { STMT* }
| [ghost] [inline] if ( EXP ) { STMT* }
    ELSE

ELSE ::=
| else if ( EXP ) { STMT* } ELSE
| [ else { STMT* } ]
```

A proof calculation is a statement that helps guide a proof [50]. CALC represents either an expression or a hint in a calculation.

```
CALC ::=
| [CALCOP] EXP ;
| [CALCOP] { STMT* }

CALCOP ::=
| < | > | <= | >= | ==
| && | || | <== | ==> | <==>
```

Assignment statements are standard.

```
ASSIGN ::=
| x := EXP
| this := EXP
| DESTINATION+, := p ( EXP*, )

DESTINATION ::=
| x
| ( [ghost] var x [: TYPE] )
```

Loops are declared with loop invariants and termination metrics.

```
INVARIANT ::=
| invariant EXP*;

DECREASE ::=
| decreases * ;
| decreases EXP+, ;
```

A matching trigger is a directive for the verifier, a feature useful to experts.

```
TRIGGER ::=
| { EXP+, }
```

Expressions include the expected ones. The productions below show examples of numerical literals and bitvector literals.

```
EXP ::=
| x
| f
| false | true
| 0 | 1 | 2 | 3 | ... | 1_000_000 | ...
| 0.1 | 0.2 | ... | 3.14159 | ...
| 0x0 | 0x1 | ... | 0xdeadBEEF
| 0x1_0000_0000 | ...
| bv1(0) | bv32(0xdeadbeef) | bv64(7) | ...
| "STRING"
| ( - EXP )
| this
| @x
| const(EXP)
| f ( EXP*, )
| EXP [ EXP ]
| EXP [ EXP := EXP ]
| EXP ?[ EXP ]
| EXP . fd
| EXP . ( fd := EXP )
| old ( EXP )
| old [ EXP ] ( EXP )
| seq ( EXP*, )
| set ( EXP*, )
| list ( EXP*, )
| tuple ( EXP*, )
| ! EXP
| EXP * EXP | EXP / EXP | EXP % EXP
| EXP + EXP | EXP - EXP
| EXP < EXP | EXP > EXP
| EXP <= EXP | EXP >= EXP | EXP is c
| EXP == EXP | EXP != EXP
| EXP && EXP
| EXP || EXP
| EXP <== EXP | EXP ==> EXP
| EXP <==> EXP
| if EXP then EXP else EXP
| let FORMAL := EXP in EXP
| forall FORMAL*, TRIGGER* :: EXP
| exists FORMAL*, TRIGGER* :: EXP
| lambda FORMAL*, TRIGGER* :: EXP
| ( EXP )
```