Vale: Verifying High-Performance Cryptographic Assembly Code

Barry Bond[1], Chris Hawblitzel[1], Manos Kapritsos[2], K. Rustan M. Leino[1], Jacob R. Lorch[1], Bryan Parno[3], Ashay Rane[4], Srinath Setty[1], Laure Thompson[5]

[1]Microsoft Research, [2]University of Michigan, [3]CMU, [4]UT Austin, [5]Cornell University

Cryptography for Information Security

Strong cryptography is critical for security in various domains.

Web traffic

Data at rest

Cryptocurrency

Cryptographic Implementation Requirements

Difficult to meet all three goals.

Correct

Formally prove that implementation matches specification

Secure

Correct control flow and free from leakage and side channels

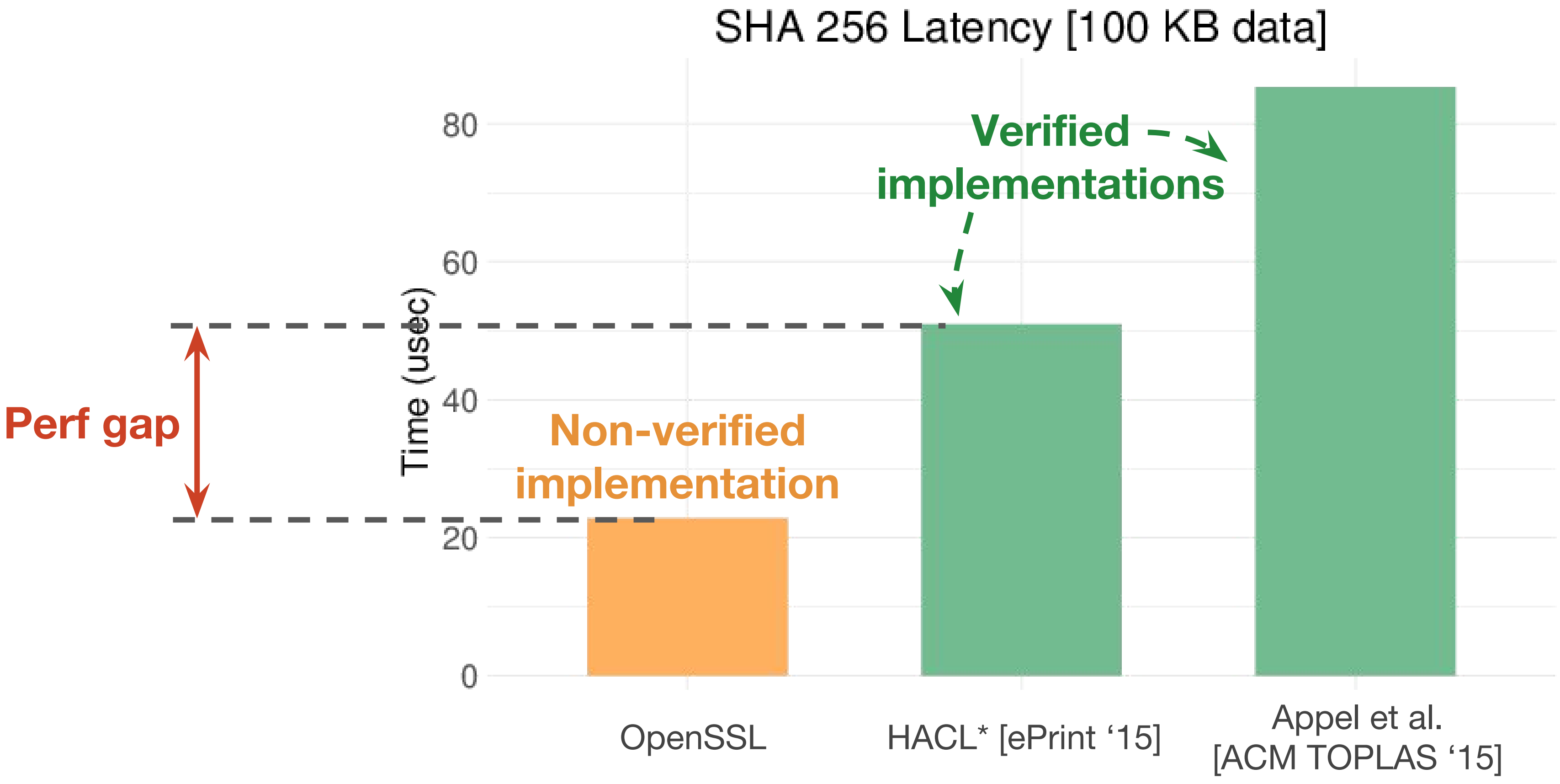
Fast

Platform-agnostic & platform-specific optimizations

Result: Crypto implementations usually fall into one of two camps.

Fast but non-verified
crypto implementations

Verified but slow
crypto implementations


SHA 256 Latency [100 KB data]
Time (usec)
80
60
40
20
0
Verified implementations
Non-verified implementation
Perf gap
OpenSSL
HACL* [ePrint '15]
Appel et al. [ACM TOPLAS '15]

OpenSSL Performance Tricks

Mix of ASM + Perl

```
sub BODY_00_15 {
  $code .= <<END
    #if __ARM_ARCH__>=7
      @ ldr $t1,[$inp],#4
      #if $i==15
        ...
      #endif
  END
}
```

Assembly code is a Perl string

C macros for target instruction selection

C macros for code specialization

OpenSSL Performance Tricks

Perl variables for register names

```
@V = ("r4", "r5", "r6", "r7", "r8", "r9", "r10", "r11");

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

Code expansion using loops

Register selection using Perl arrays

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

> Result: Code becomes **difficult to understand, debug, and formally verify** for correctness and security.

Our Contribution: **Vale**

Flexible framework for writing high-performance, proven correct and secure assembly code.

Correct

Secure

Fast

Our Contribution: **Vale**

Flexible framework for writing high-performance, proven correct and secure assembly code.

Flexible Syntax

Vale supports constructs for expressing functionality as well as optimizations.

High Performance

Code generated by Vale matches or exceeds OpenSSL's performance.

High Assurance

Vale can be used to prove functional correctness and correct information flow.

Vale is a work in progress. Not a complete replacement to OpenSSL.

Key **Language Constructs** in Vale

Assembly Instructions

e.g. `Mov`, `Rev`, and `AesKeygenAssist`

Vary according to the target platform

Structured Control Flow

e.g. `if`, `while`, and `procedure`

Enable proof composition

Optimization Constructs

Customize code generation

Optimization Using `inline if` Statements

Vale supports inline `if` statements, which are evaluated **during code generation**, not during code execution.

Useful for selecting instructions and for unrolling loops.

Target Instruction Selection
(**Platform-dependent** optimization)

```
inline if(platform == x86_AESNI) {
  ...
}
```

Loop Unrolling
(**Platform-independent** optimization)

```
inline if (n > 0) {
  ...
  recurse(n - 1);
}
```

Example Vale Code

Example
Vale Code

```
procedure Incr_By_N(inline n:nat) {
    inline if (n > 0) {
        ADD(r5, r5, 1);
        Incr_By_N(n - 1);
    }
}

Incr_By_N(100);
```

Example Vale Code

Example Vale Code

```
procedure Incr_By_N(inline n:nat) {
    inline if (n > 0) {
        ADD(r5, r5, 1);
        Incr_By_N(n - 1);
    }
}

Incr_By_N(100);
```

→

Expanded Vale AST

```
ADD(r5, r5, 1)
ADD(r5, r5, 1)
ADD(r5, r5, 1)
ADD(r5, r5, 1)
...

Total 100 ADD
instructions
```

Example Vale Code

Example Vale Code

```
procedure Incr_By_N(inline n:nat) {
    inline if (n > 0) {
        ADD(r5, r5, 1);
        Incr_By_N(n - 1);
    }
}

Incr_By_N(100);
```

Generated Assembly Code

```
add r5, r5, 1
add r5, r5, 1
add r5, r5, 1
add r5, r5, 1
...

Total 100 ADD
instructions
```

Cryptographic Implementation Requirements

Fast

Code generated by Vale matches or exceeds OpenSSL's performance.

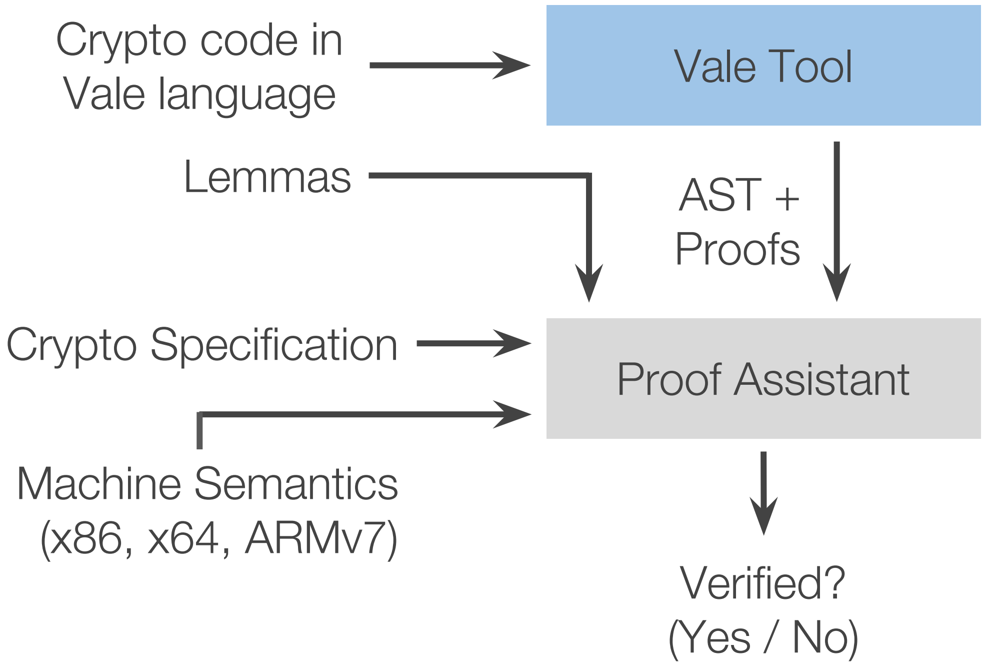
Cryptographic Implementation Requirements

Correct

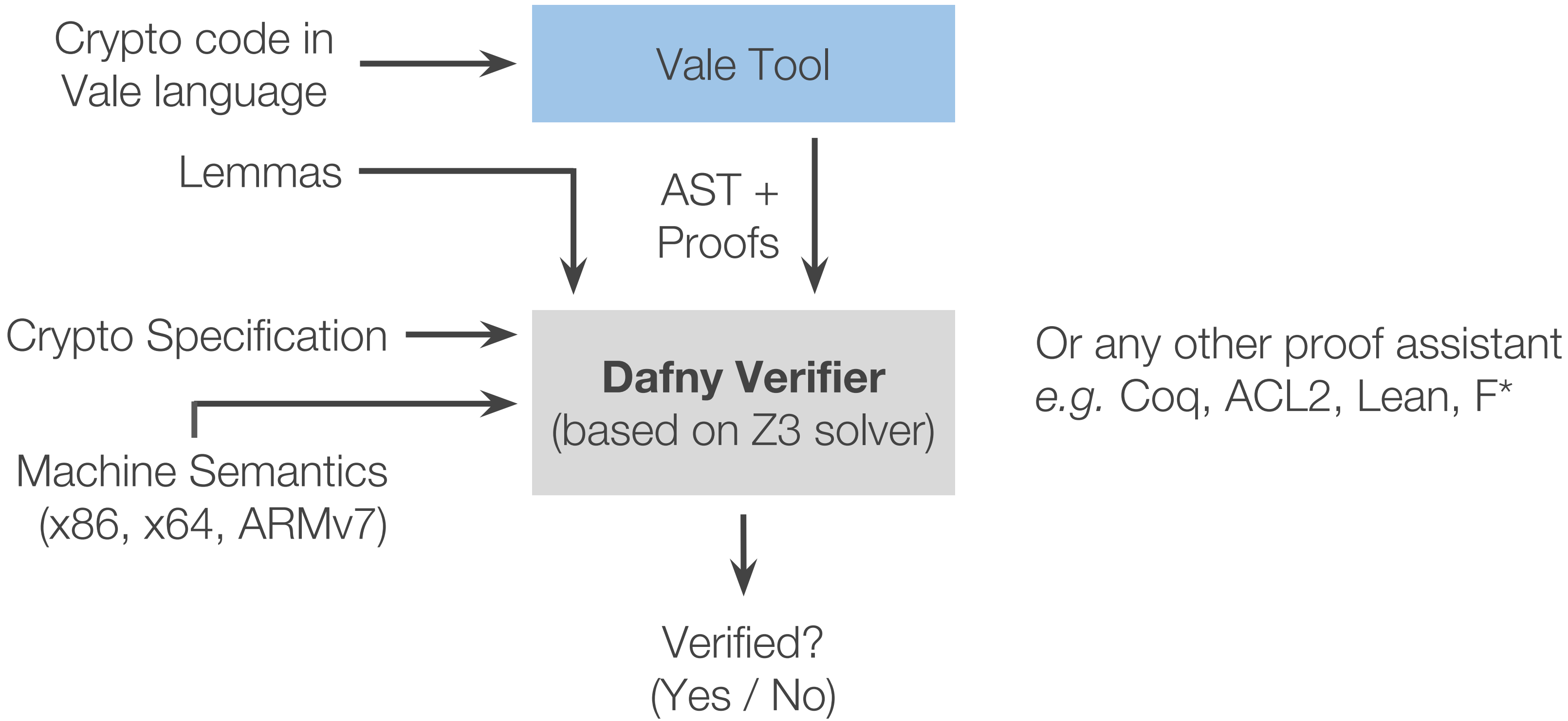
Fast

Code generated by Vale matches or exceeds OpenSSL's performance.

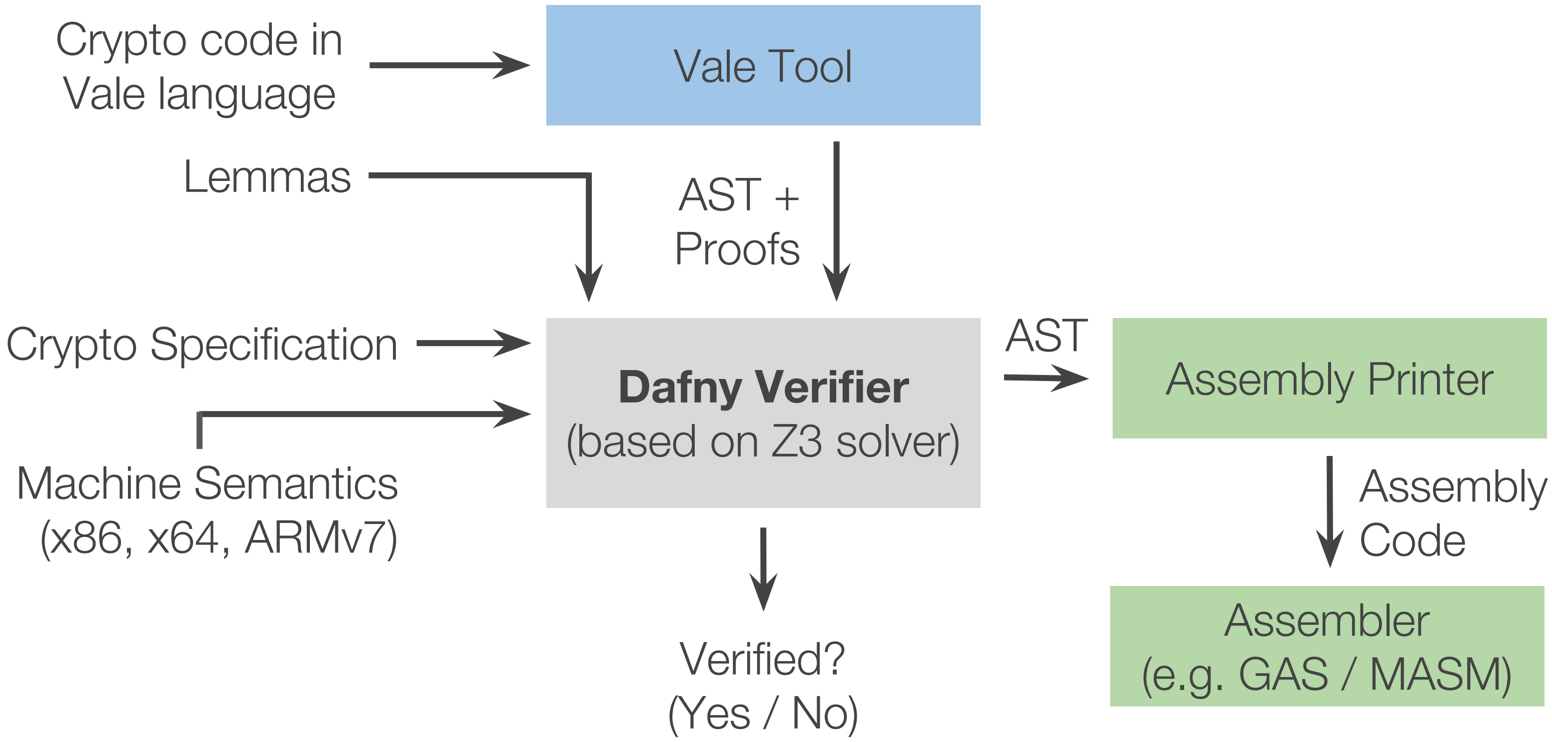
Vale Architecture

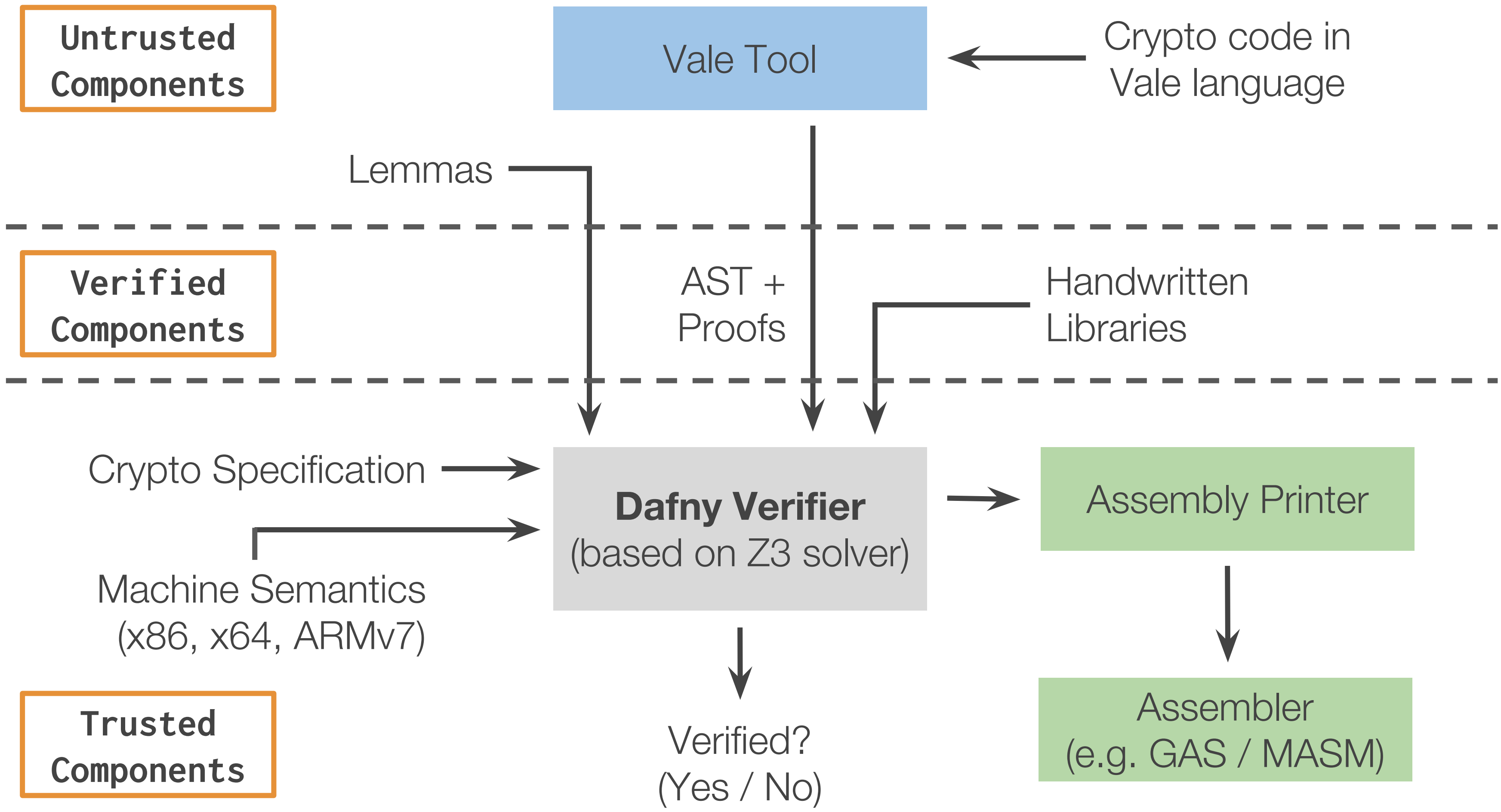


Vale Architecture



Vale Architecture





Untrusted Components
Vale Tool
Crypto code in Vale language
Lemmas
Verified Components
AST + Proofs
Handwritten Libraries
Crypto Specification
Dafny Verifier
(based on Z3 solver)
Assembly Printer
Machine Semantics (x86, x64, ARMv7)
Trusted Components
Verified? (Yes / No)
Assembler (e.g. GAS / MASM)

$\{p\}\ C\ \{q\}$

$$\{p\}\ C\ \{q\}$$

Pre-Conditions
'requires' keyword

Example Vale Code

```
procedure Incr_By_N(inline n:nat)
  requires r5+n < 0x1_0000_0000


{
    inline if (n > 0) {
        ADD(r5, r5, 1);
        Incr_By_N(n - 1);
    }
}
```

$$\{p\}\ C\ \{q\}$$

Pre-Conditions
'requires' keyword

Post-Conditions
'ensures' keyword

Example Vale Code

```
procedure Incr_By_N(inline n:nat)
  requires r5+n < 0x1_0000_0000
  ensures  r5 == old(r5) + n

{
    inline if (n > 0) {
        ADD(r5, r5, 1);
        Incr_By_N(n - 1);
    }
}
```

State Manipulation
'`modifies`' keyword

$$\{p\}\ C\ \{q\}$$

Pre-Conditions
'`requires`' keyword

Post-Conditions
'`ensures`' keyword

Example Vale Code

```
procedure Incr_By_N(inline n:nat)
  requires r5+n < 0x1_0000_0000
  ensures  r5 == old(r5) + n
  modifies r5
{
    inline if (n > 0) {
        ADD(r5, r5, 1);
        Incr_By_N(n - 1);
    }
}
```

Code is verified **before** expansion of inline-if statement.

Cryptographic Implementation Requirements

Correct

Vale supports assertions that are checked by Dafny

Fast

Code generated by Vale matches or exceeds OpenSSL's performance.

Cryptographic Implementation Requirements

Correct

Vale supports assertions that are checked by Dafny

Secure (Leakage Free)

Fast

Code generated by Vale matches or exceeds OpenSSL's performance.

Secret Information Leakage

Secrets should not leak through:

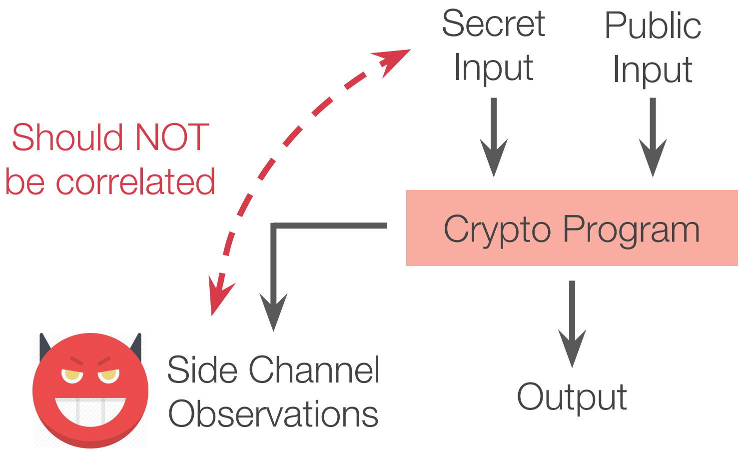
➔ **Digital Side Channels:** Observations of program behavior through cache usage, timing, memory accesses, etc.

➔ **Residual Program State:** Secrets left in registers or memory after termination of program

Secret Information Leakage

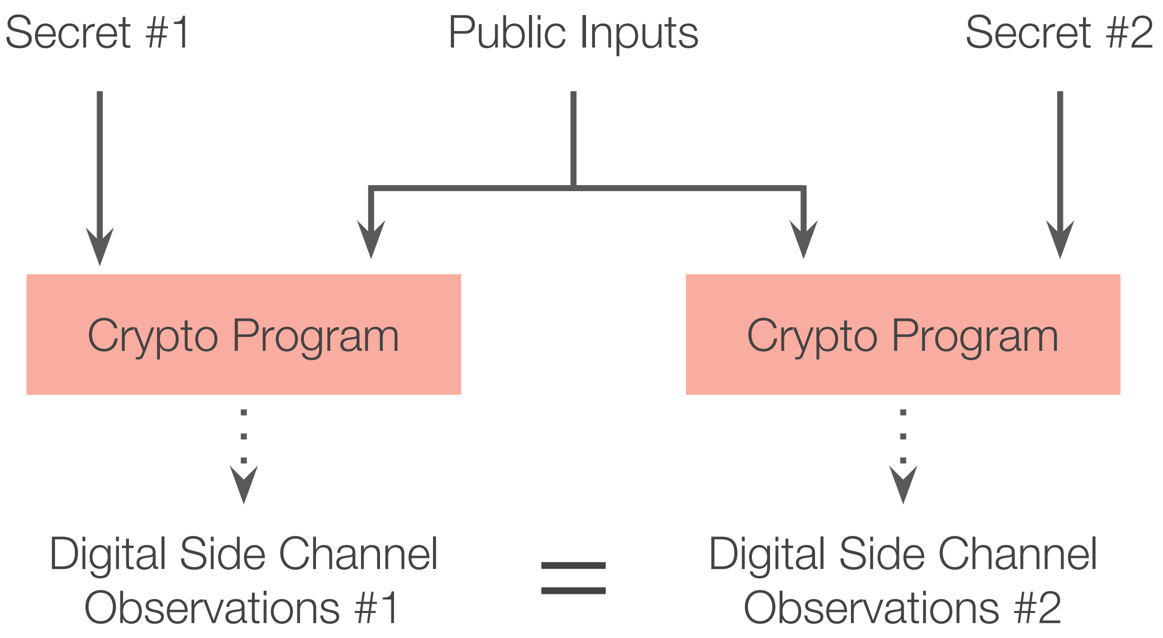
Secrets should not leak through:

➔ **Digital Side Channels:** Observations of program behavior through cache usage, timing, memory accesses, etc.



Information Leakage Specification

Based on Non-Interference



Information Leakage Specification

Based on Non-Interference

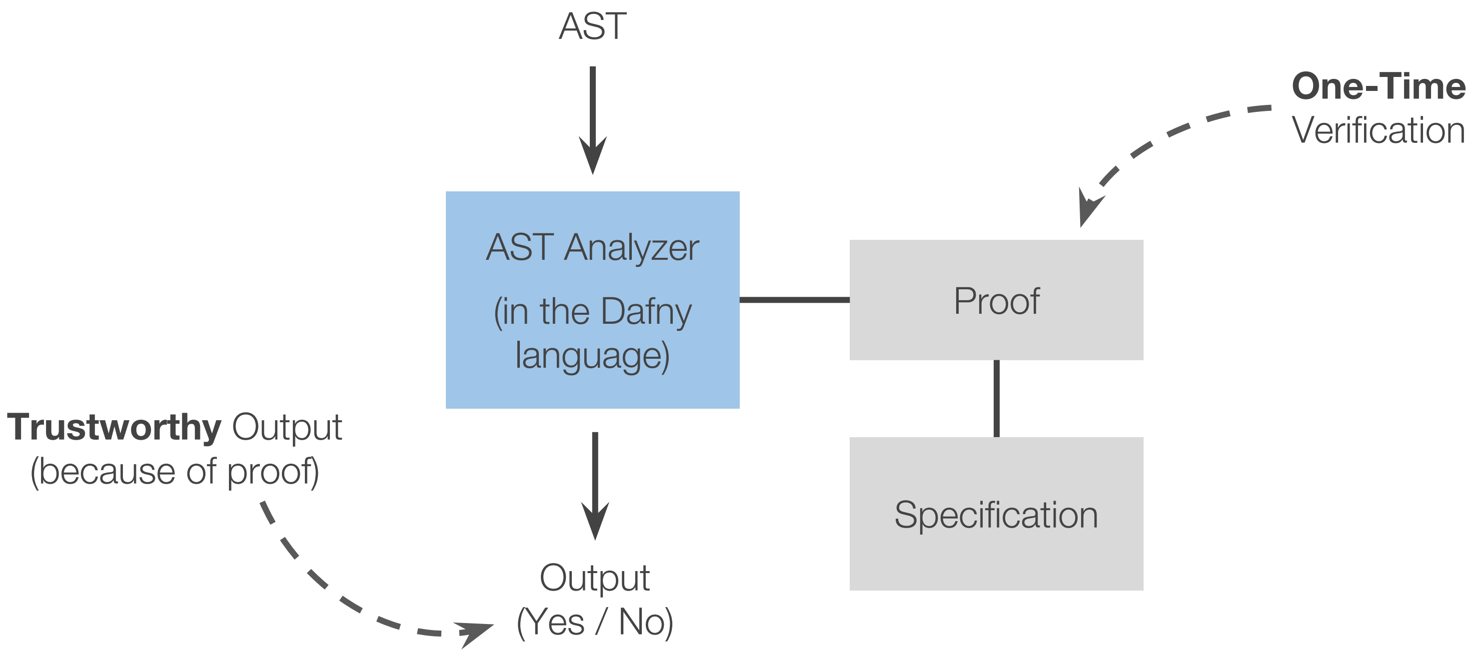
Formally, for a crypto program C,

$\forall$ pairs of secrets s_1 and s_2

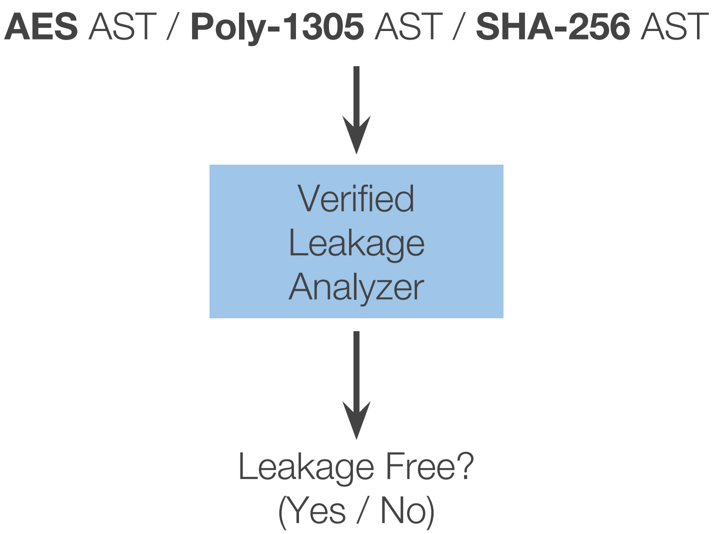
$\forall$ public values p,

$$obs(C, p, s1) = obs(C, p, s2)$$

Solution: Verified Analysis



Verified Leakage Analysis



Leakage Analysis Using Taint Tracking

```
procedure foo(public :nat,
  secret :nat) {
    // r5 := secret + 1
    ADD(r5, secret, 1);

    if (r5 < 10) {
        bar();
    }
}
```

Step 1: Developer marks regs and mem that contain non-secret information.

Leakage Analysis Using Taint Tracking

```
procedure foo(public :nat,
  secret :nat) {
    // r5 := secret + 1
    ADD(r5, secret, 1);

    if (r5 < 10) {
        bar();
    }
}
```

Step 1: Developer marks regs and mem that contain non-secret information.

Step 2: Analysis conservatively assumes that all other locations contain secrets.

Leakage Analysis Using Taint Tracking

```
procedure foo(public :nat,
  secret :nat) {
    // r5 := secret + 1
    ADD(r5, secret, 1);

    if (r5 < 10) {
        bar();
    }
}
```

Step 1: Developer marks regs and mem that contain non-secret information.

Step 2: Analysis conservatively assumes that all other locations contain secrets.

Step 3: Analysis tracks secrets through registers and memory locations.

Leakage Analysis Using Taint Tracking

```
procedure foo(public :nat,
  secret :nat) {
    // r5 := secret + 1
    ADD(r5, secret, 1);

    if (r5 < 10) {
        bar();
    }
}
```

Step 1: Developer marks regs and mem that contain non-secret information.

Step 2: Analysis conservatively assumes that all other locations contain secrets.

Step 3: Analysis tracks secrets through registers and memory locations.

Step 4: Report violation if secret used in branch predicate, memory address, or as input to variable-latency instruction.

Problems Caused by Aliasing

```
store [rbx] <- 0
store [rax] <- 10
load  rcx <- [rbx]
```

Does `rcx` contain `0` or `10`?

Difficult to answer without knowing whether `rax` = `rbx`.

Alias Analysis is a Difficult Problem

Existing alternatives:

1. Analyze source code in a high level language but compiler may introduce new side channels.
2. Implement pointer analysis for assembly code but imprecise analysis.
3. Assume no aliases, but this is an unsafe assumption.

Vale is uniquely suited to use a different approach:

Reuse developer's effort from proof of correctness.

Reusing Effort from Proof of Correctness

Functional verification requires precisely identifying information flow.

Specification	Implementation

‘`output`’ should be equal to `0`	`store [rbx] <- 0`
	`store [rax] <- 10`
	`load output <- [rbx]`

To prove that `output` = `0` and not `10`, developer should prove that `rax` ≠ `rbx`.

Lightweight Annotations for Memory Taint

Vale requires the developer to mark memory operands that contain secrets:

```
load rax <- [rdx] @secret
```

Easy for developer since proving correctness requires identifying all information flows.

Since these **annotations are checked by the verifier, they are untrusted**.

Cryptographic Implementation Requirements

Correct

Vale supports assertions that are checked by Dafny

Secure

Vale checks for leakage via state and digital side channels.

Fast

Code generated by Vale matches or exceeds OpenSSL's performance.

Case Studies Using Vale

Using Vale, we developed four verified cryptographic programs:

1. SHA-256 on ARMv7 (ported from OpenSSL) **Discovered leakage on stack.**
2. Poly1305 on x64 (ported from OpenSSL) **Confirmed a previously known bug.**
3. SHA-256 on x86
4. AES-CBC (with AESNI) on x64

After fixing the issues, all four programs were proved correct and secure using Vale.

Key Lessons

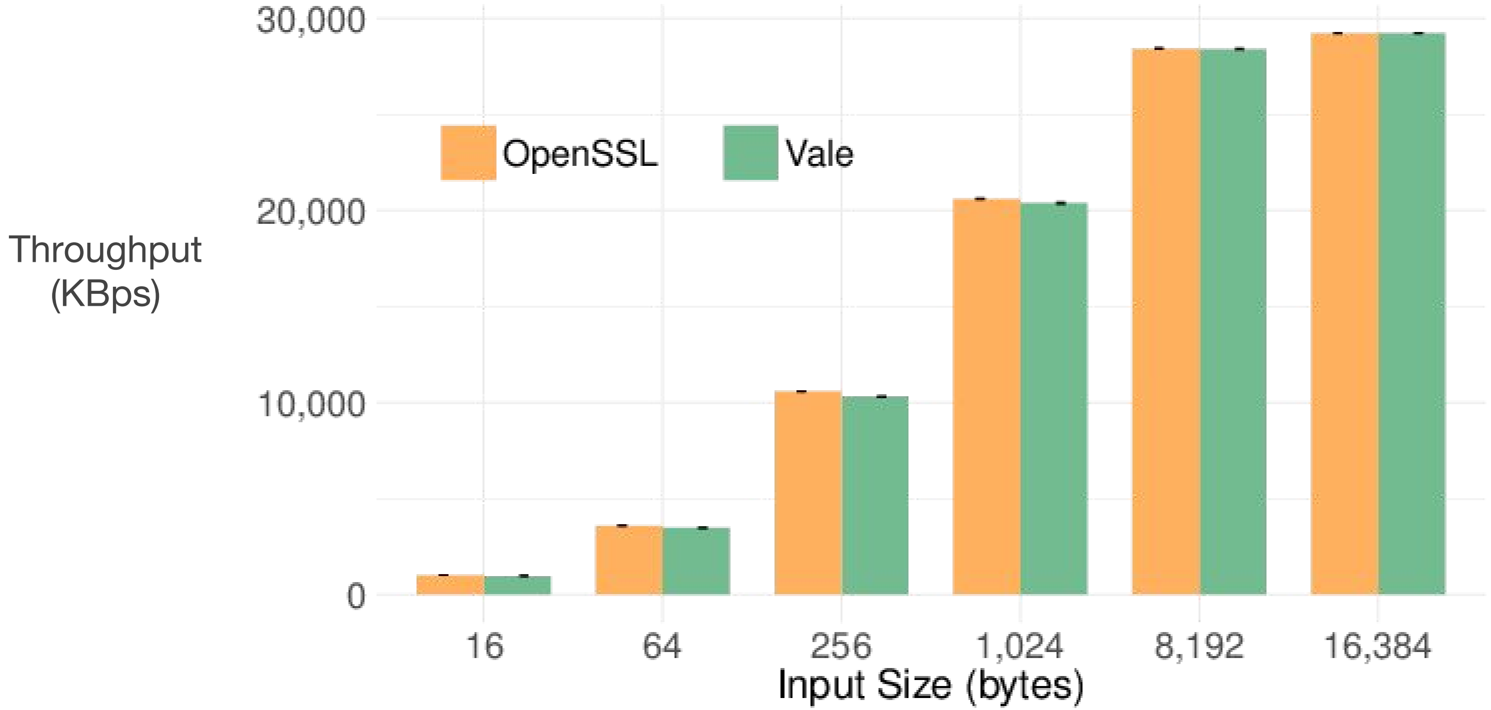
1. Vale’s specifications + lemmas were **reusable across platforms** (x86, x64, ARM).

2. Porting OpenSSL’s Perl tricks required understanding and proving invariants.

   Some of OpenSSL’s optimizations were **automatically proved by Dafny**.

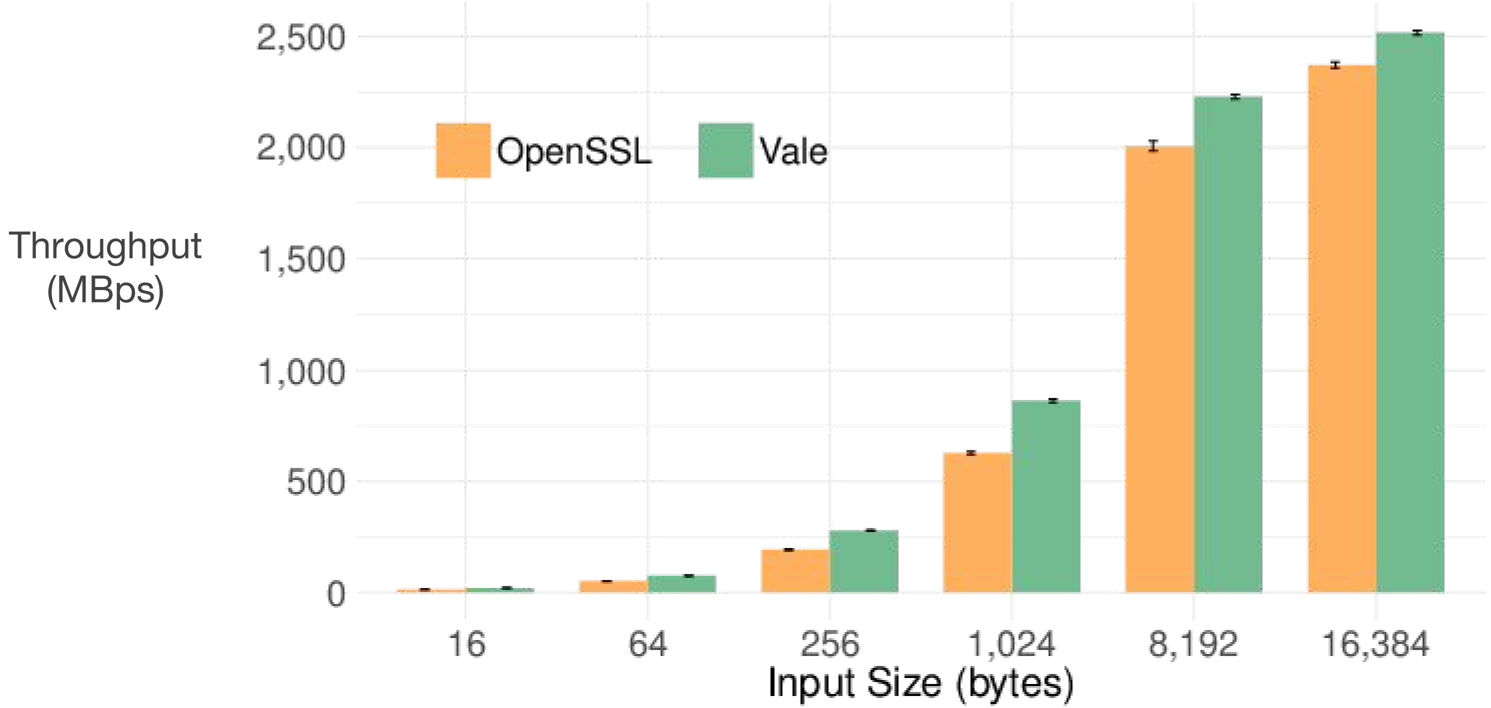
Vale versus OpenSSL: **SHA-256**

x64 assembly code



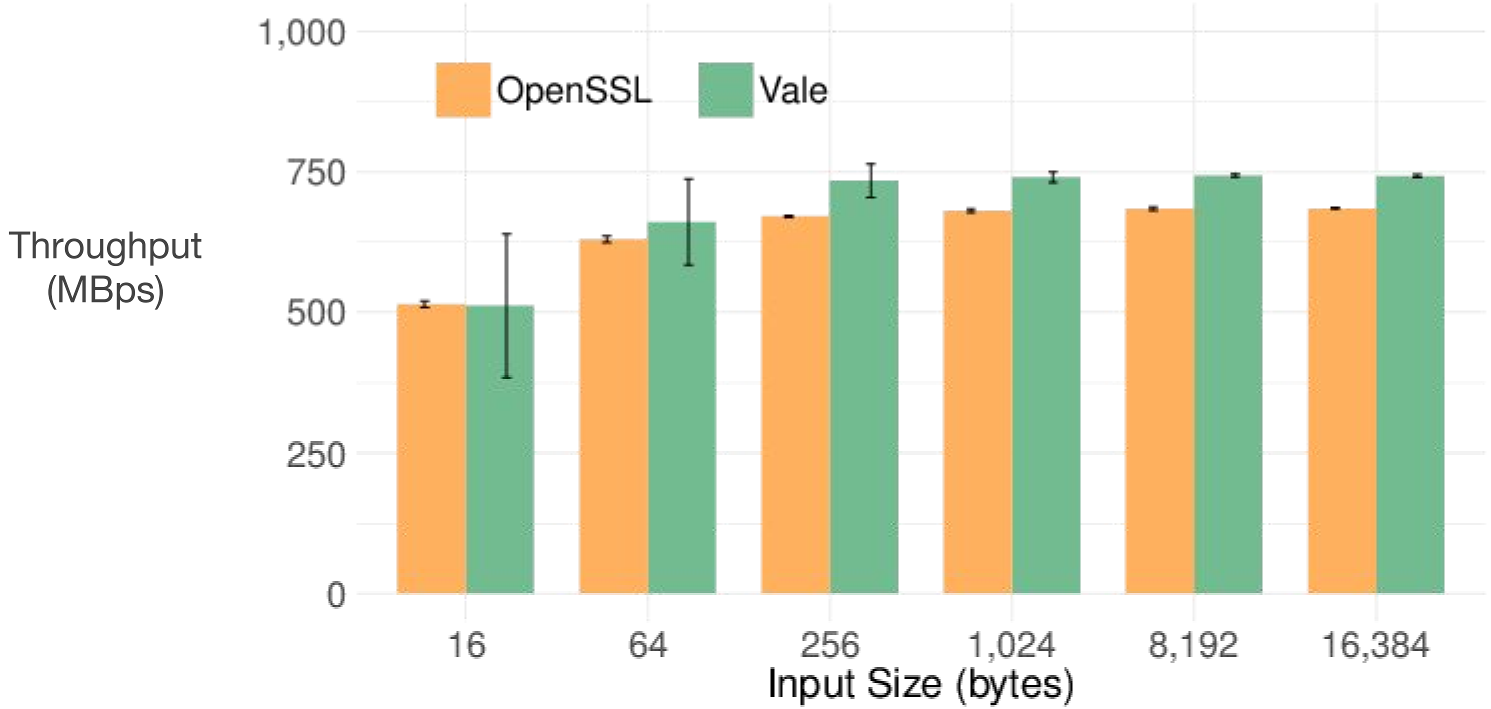
Vale versus OpenSSL: **Poly-1305**

64-bit non-SIMD assembly code



Vale versus OpenSSL: **AES-CBC-128**

AES-NI assembly code



Verification Effort

In person-months

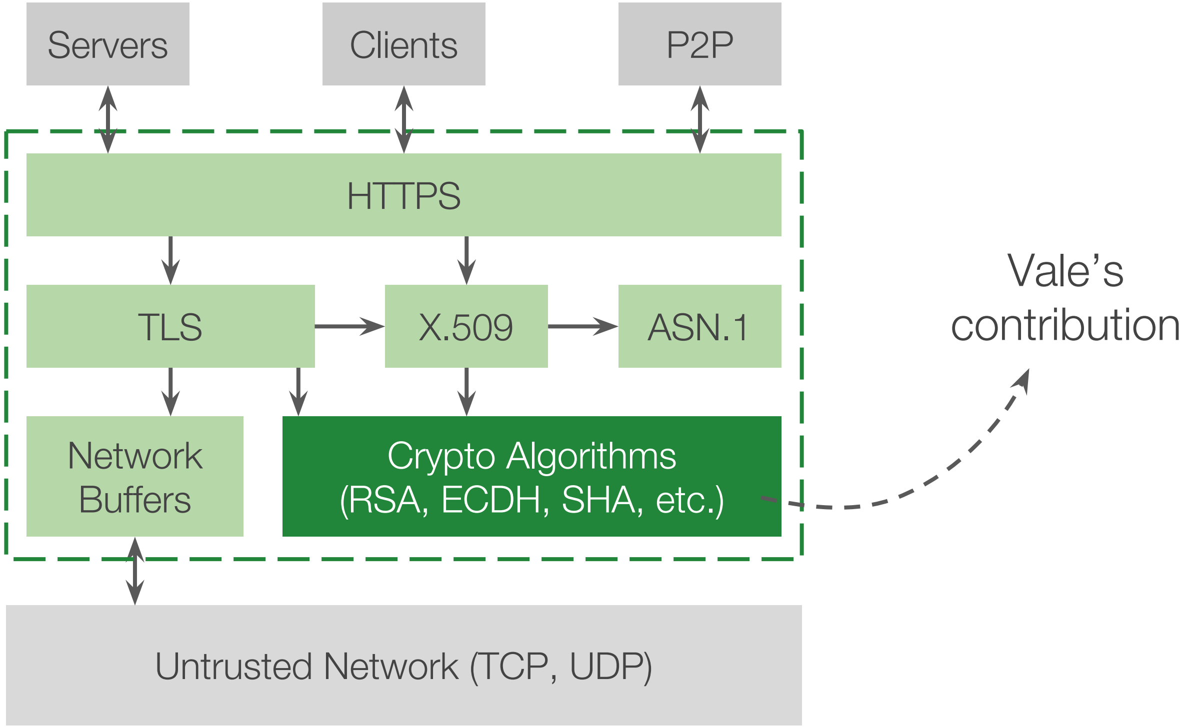
Tool Development

Vale	Leakage Analysis	AES CBC	Poly1305	1st SHA	SHA Port
12	6	5	0.5	6	0.75

Crypto Implementations

The Big Picture: **Project Everest**

Goal: Build and deploy a verified HTTPS stack.



The Big Picture: **Project Everest**

Research Goals:

- Prove the security of new protocols.
- Make verified systems as fast as unverified systems.
- Defend against advanced threats such as side channels.
- Make verification accessible to non-experts.

Conclusion

- Vale is a framework for generating and verifying crypto implementation that is **correct, secure, and fast** for arbitrary architectures.
- Vale's **flexible syntax** allows writing assembly code that OpenSSL expresses using ad-hoc Perl scripts, C preprocessor macros, and custom interpreters.
- Vale supports **verified** analysis of code, e.g., information leakage analysis.
- Vale demonstrates that **verified code can be as fast as highly-optimized, unverified code**.

Future Work

- Verify other crypto implementations and components in the HTTPS stack.
- Build Vale on top of other proof assistants. Ongoing work on using F*.

Vale

A flexible framework for writing high-performance, proven correct, and proven secure assembly code.

https://github.com/project-everest/vale

https://project-everest.github.io