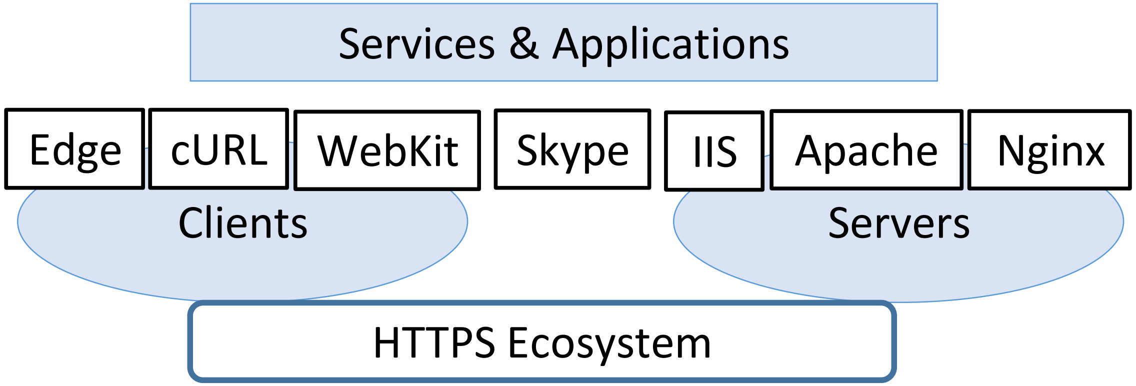
Developing Fast, Mechanically-Verified Cryptographic Code

Bryan Parno

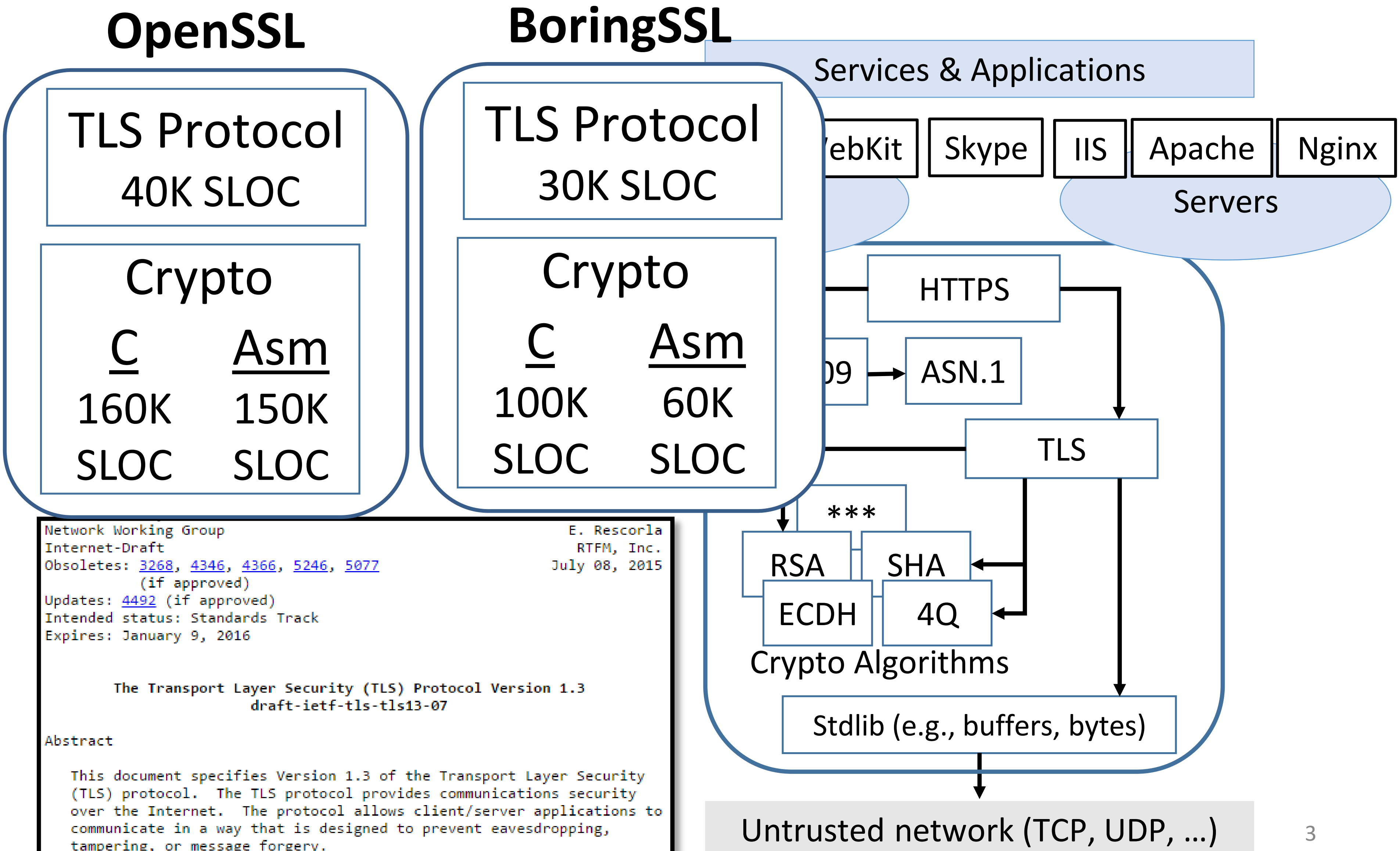
Carnegie Mellon University

The HTTPS Ecosystem is critical



- Most widely deployed security protocol?
  - 40% all Internet traffic (+40%/year)
- Web, cloud, email, VoIP, 802.1x, VPNs, …

The HTTPS Ecosystem is complex



Network Working Group E. Rescorla
Internet-Draft RTFM, Inc.
Obsoletes: 3268, 4346, 4366, 5246, 5077 July 08, 2015
(if approved)
Updates: 4492 (if approved)
Intended status: Standards Track
Expires: January 9, 2016

The Transport Layer Security (TLS) Protocol Version 1.3
draft-ietf-tls-tls13-07

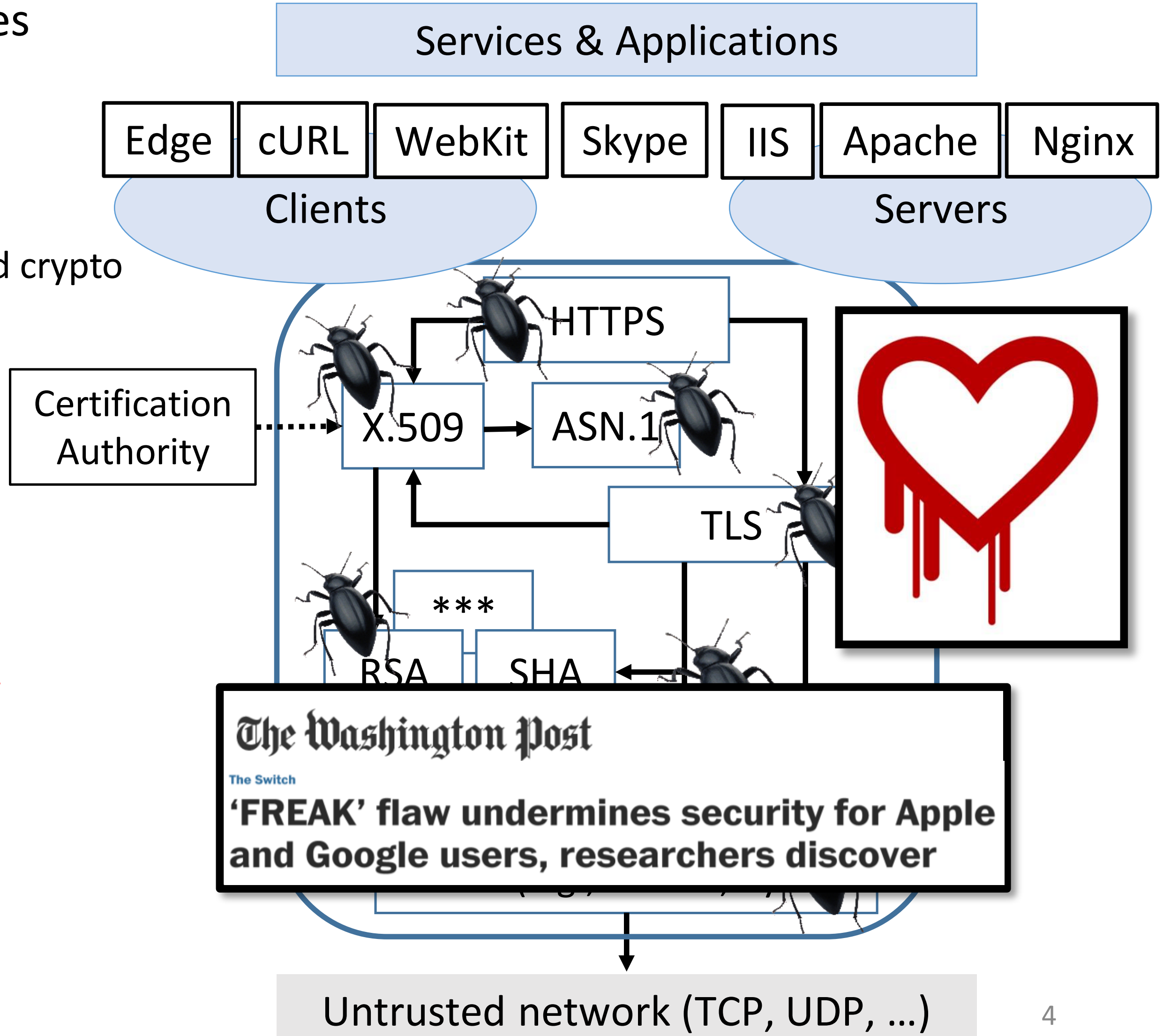
Abstract

This document specifies Version 1.3 of the Transport Layer Security (TLS) protocol. The TLS protocol provides communications security over the Internet. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery.

Status of This Memo

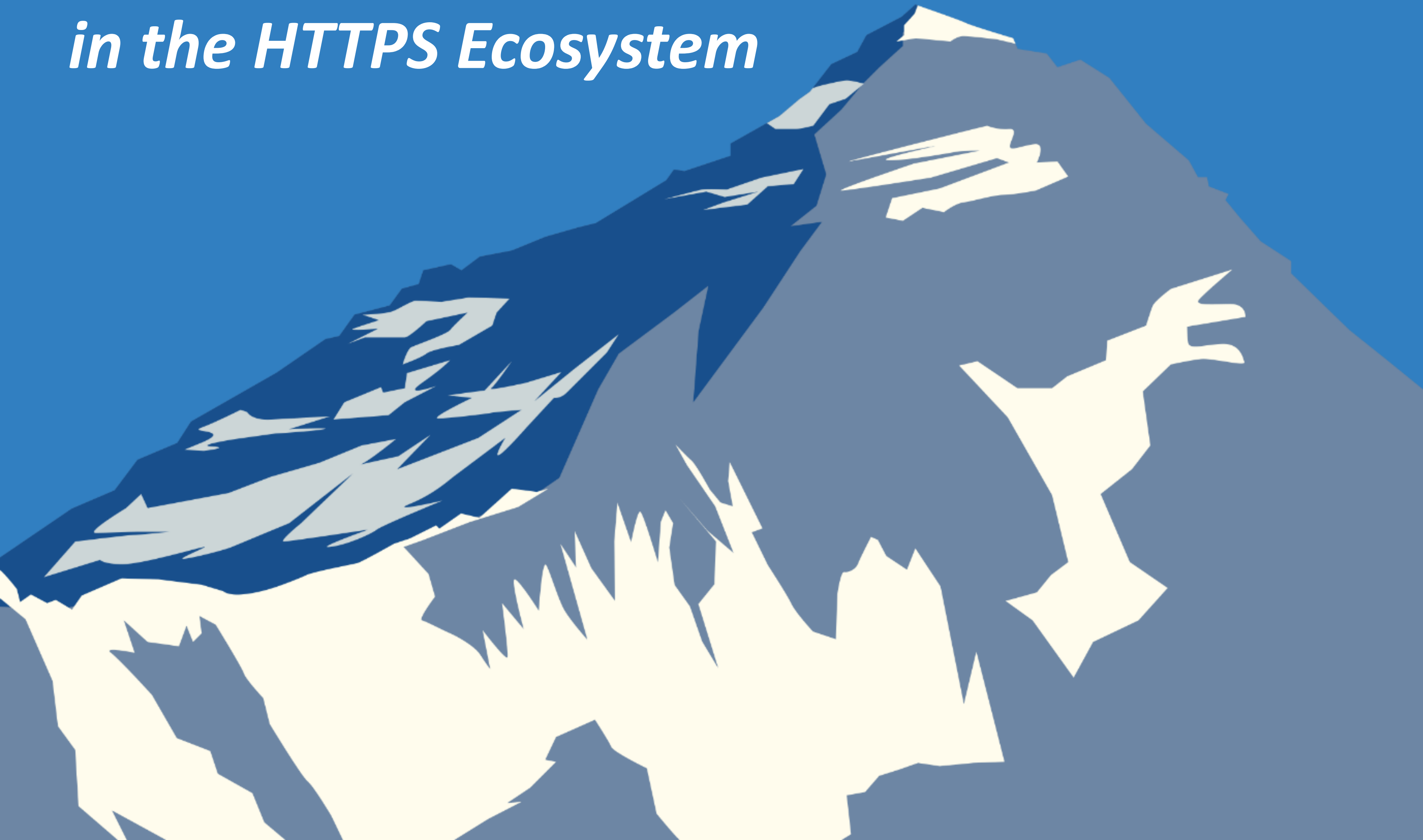
The HTTPS Ecosystem is buggy

- 20 years of attacks & fixes
  - Buffer overflows
  - Memory management
  - Incorrect state machines
  - Lax certificate parsing
  - Weakly or badly implemented crypto
  - Side channels
  - Error-inducing APIs
  - Flawed standards
  - …
- Many implementations
  - OpenSSL, Schannel, NSS, …
  - *Still patched every month!*



Everest:

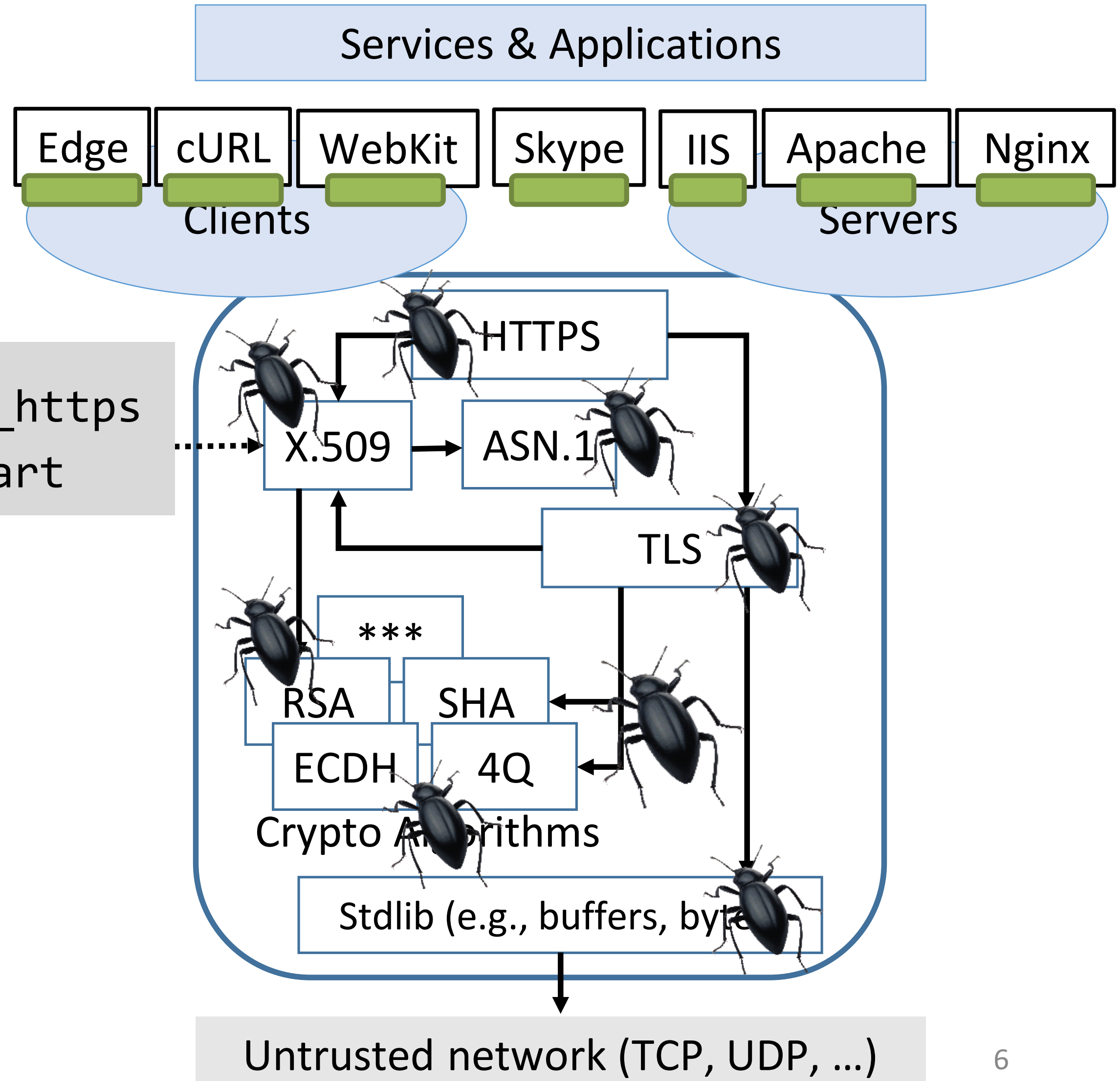
Deploying Verified-Secure Implementations in the HTTPS Ecosystem



Everest Goals

- Fully verified replacement
- Widespread deployment
- Trustworthy, usable tools

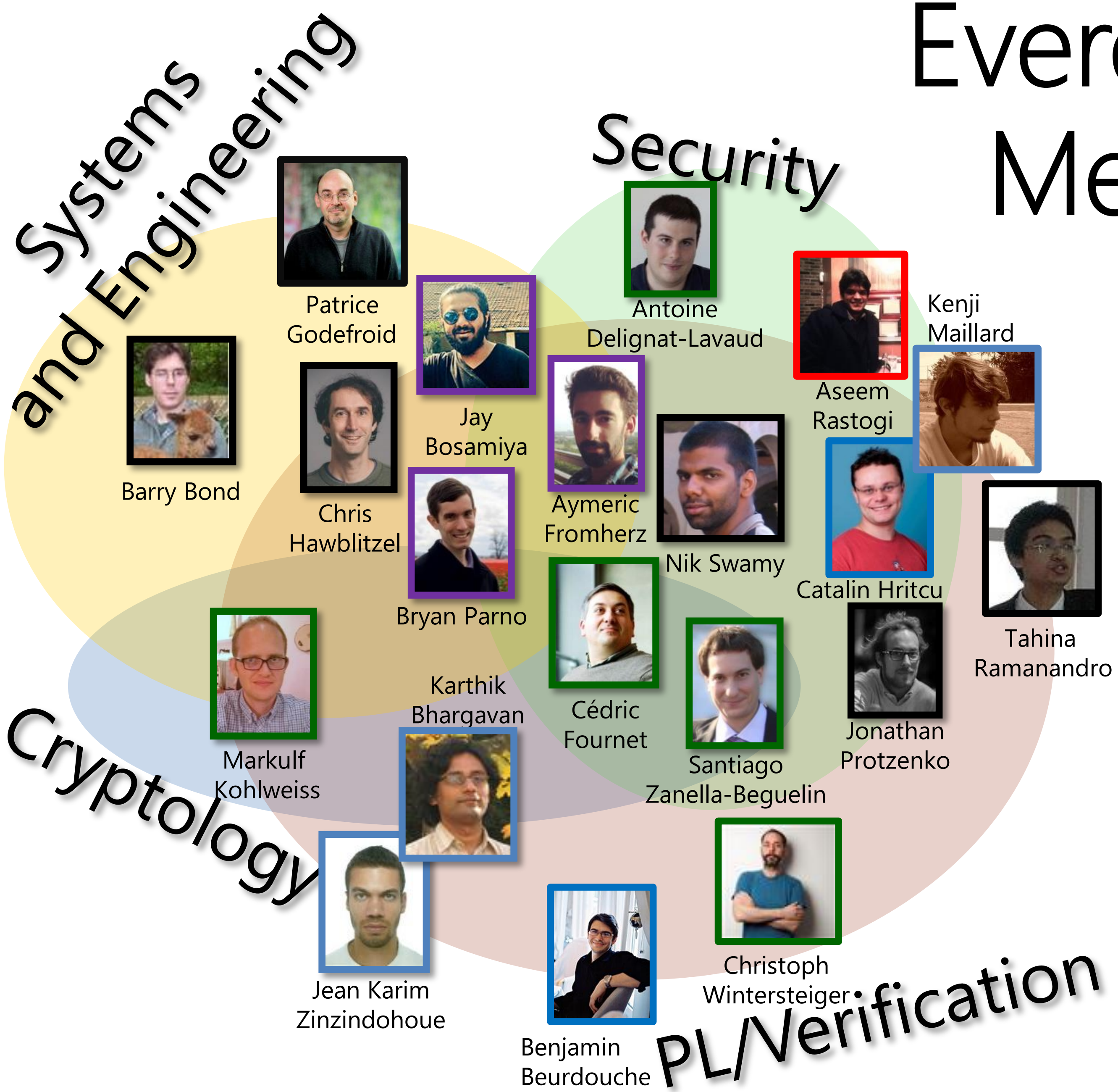
```
$ apt-get install verified_https
$ /etc/init.d/apache2 restart
```



Research Questions

- How do we decide whether new protocols are secure?
  - Especially when interoperating with insecure protocols
- Can we make verified systems as fast as unverified?
- How do we handle advanced threats?
  - Ex: Side channels
- Why should we trust automated verification tools?
- How can verification be more accessible?
  - Especially to non-experts in verification

Everest Team Members



MSR-Cambridge

MSR-Bangalore

MSR-Redmond

INRIA

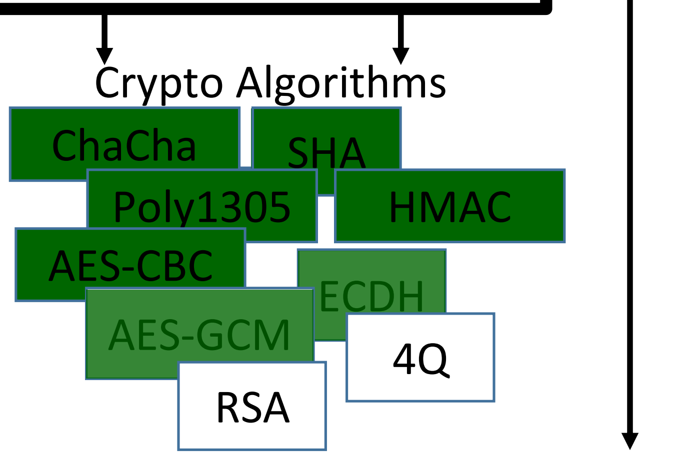
CMU

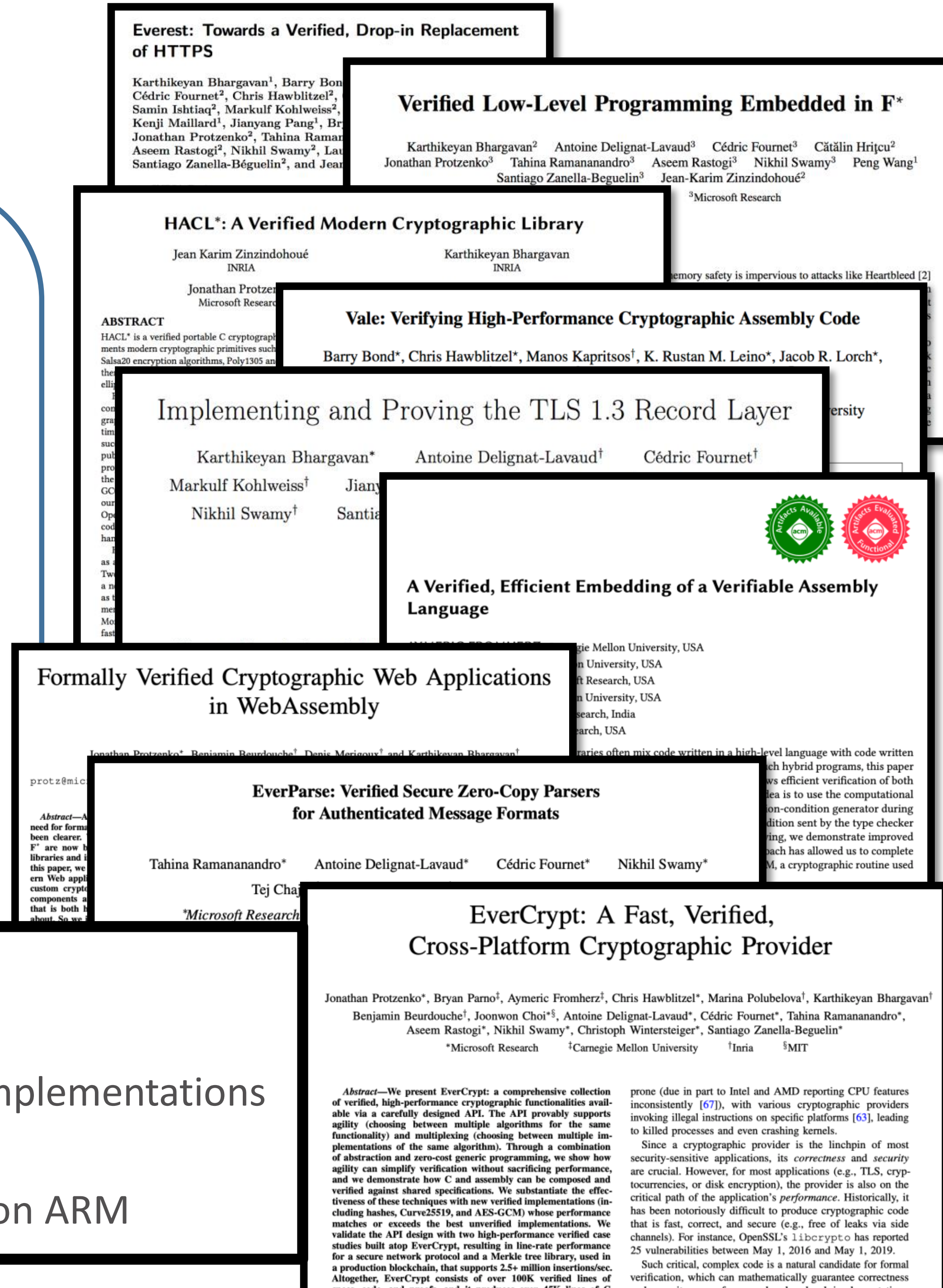
+ interns and many other collaborators…

Current Status

(Partial) Deployments

- Microsoft
- Tezos blockchain
- WireGuard VPN
- Mozilla Firefox

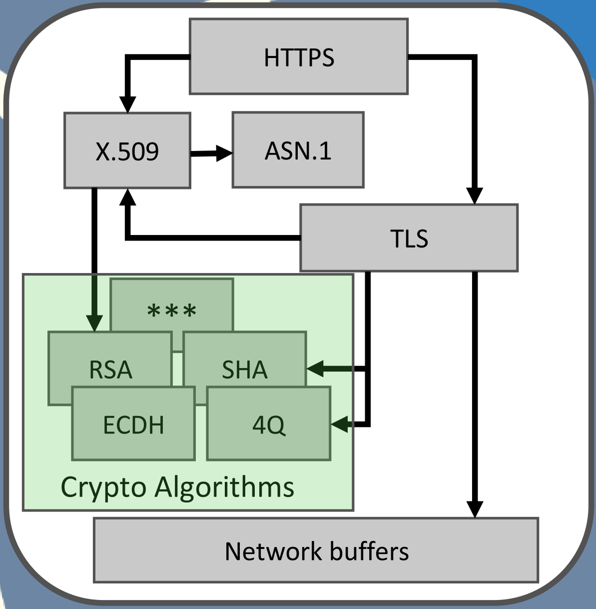




Spinoffs

- QUIC prototypes
- Verified TLS models and reference implementations
- TLS 1.3 RFC fixes and improvements
- Komodo: Verified SGX-like enclaves on ARM

EverCrypt: A Verified Crypto Provider



Why Verify Crypto?

- Bugs are real, and potentially devastating!
- 24 vulnerabilities in OpenSSL's `libcrypto` in ~3 years!

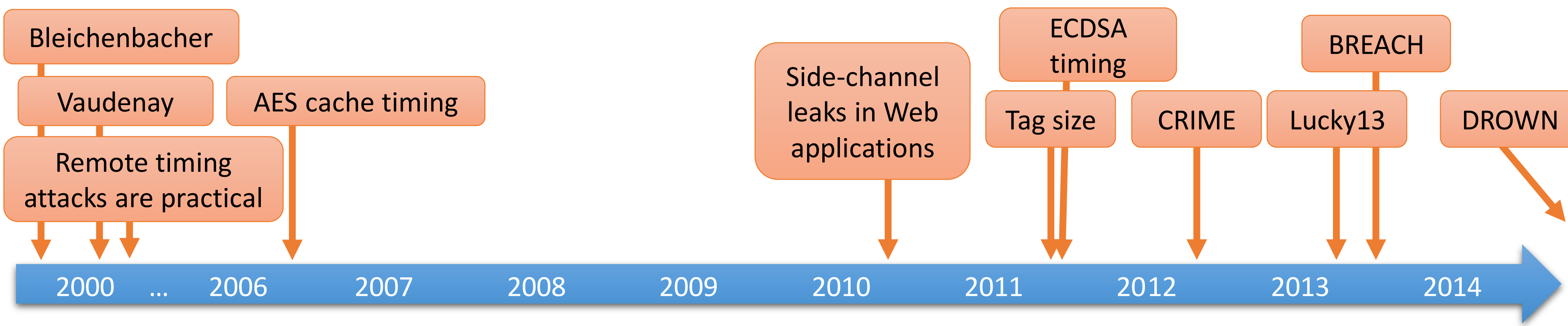
> "These produce wrong results. The first example does so only on 32 bit, the other three also on 64 bit."
>
> "I believe this affects both the SSE2 and AVX2 code. It does seem to be dependent on this input pattern."
>
> "I'm probably going to write something to generate random inputs and stress all your other poly1305 code paths against a reference implementation."

```
These produce wrong results. The first example d
the other three also on 64 bit.
```

```
You know the drill. See the attached poly1305_test2.c.

$ OPENSSL_ia32cap=0 ./poly1305_test2
PASS
$ ./poly1305_test2
Poly1305 test failed.
got:      2637408fe03086ea73f971e3425e2820
expected: 2637408fe13086ea73f971e3425e2820

I believe this affects both the SSE2 and AVX2 code. It does seem to be
dependent on this input pattern.

This was found because a run of our SSL tests happened to find a
problematic input. I've trimmed it down to the first block where they
```

Side Channel Challenge (Attacks)

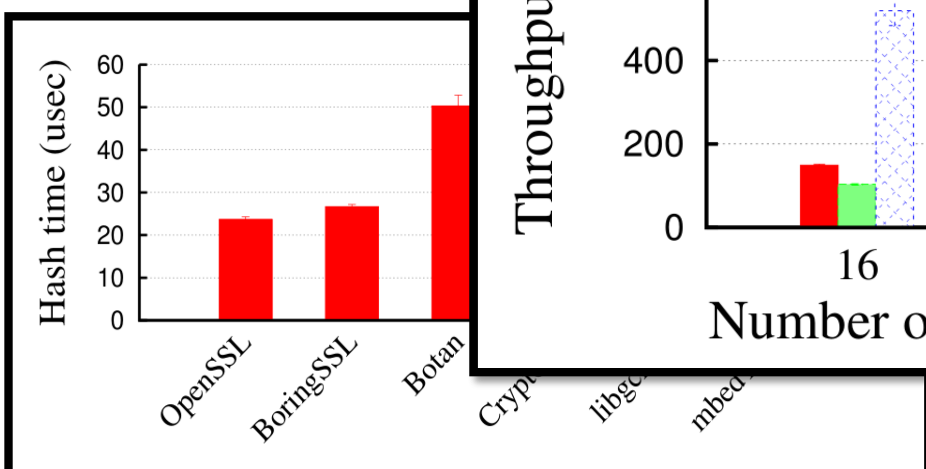
Protocol-level side channels	Traffic analysis	Timing attacks against cryptographic primitives	Memory & Cache
TLS messages may reveal information about the internal protocol state or the application data	Combined analysis of the time and length distributions of packets leaks information about the application	A remote attacker may learn information about crypto secrets by timing execution time for various inputs	Memory access patterns may expose secrets, in particular because caching may expose sensitive data (e.g. by timing)
• Hello message contents (e.g. time in nonces, SNI) • Alerts (e.g. decryption vs. padding alerts) • Record headers	• CRIME/BREACH (adaptive chosen plaintext attack) • User tracking • Auto-complete input theft	• Bleichenbacher attacks against PKCS#1 decryption and signatures • Timing attacks against RC4 (Lucky 13)	• OpenSSL key recovery in virtual machines • Cache timing attacks against AES

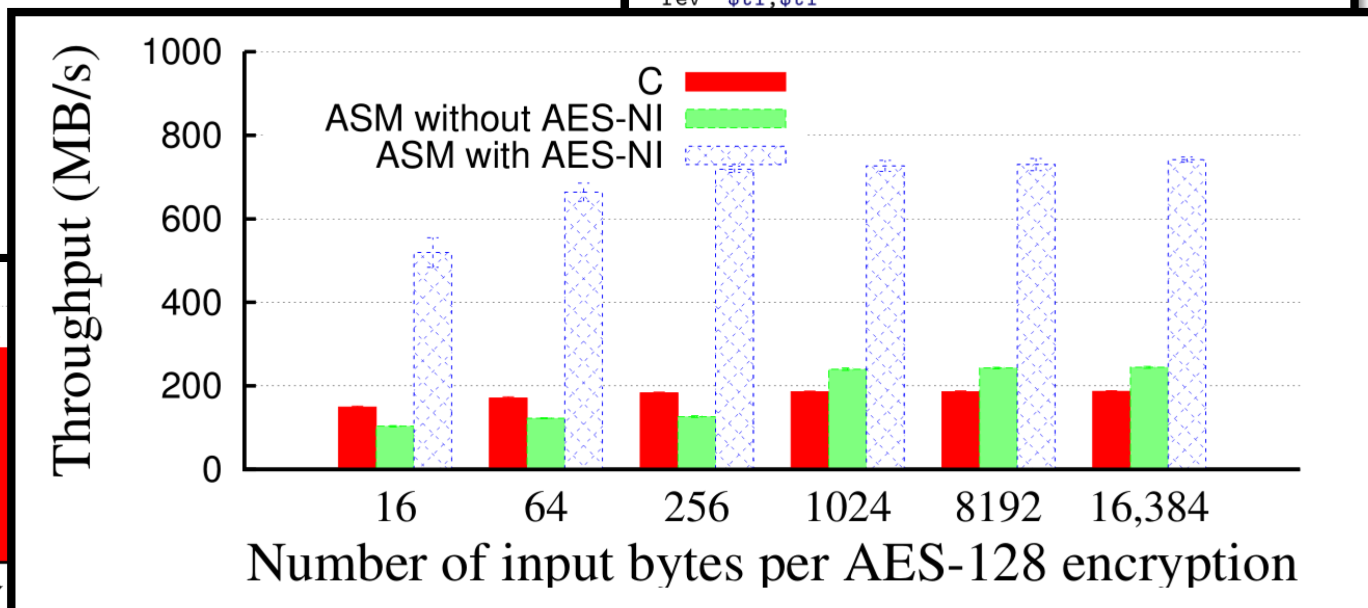


Current State of the Art: OpenSSL

- Hand-written mix of Perl and assembly
- Customized for 50+ hardware platforms
- Why?
  - Performance!

```
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
```





Features of an Ideal Library (programmer)

- **Usable**
  - preferably in C or ASM, not “exotic” languages
- **Comprehensive**
  - one algorithm per processor generation / bitsize
- **Auto-configurable multiplexing**
  - best algorithm picked automatically
- **Agility**
  - clients deal with a unified API for each family

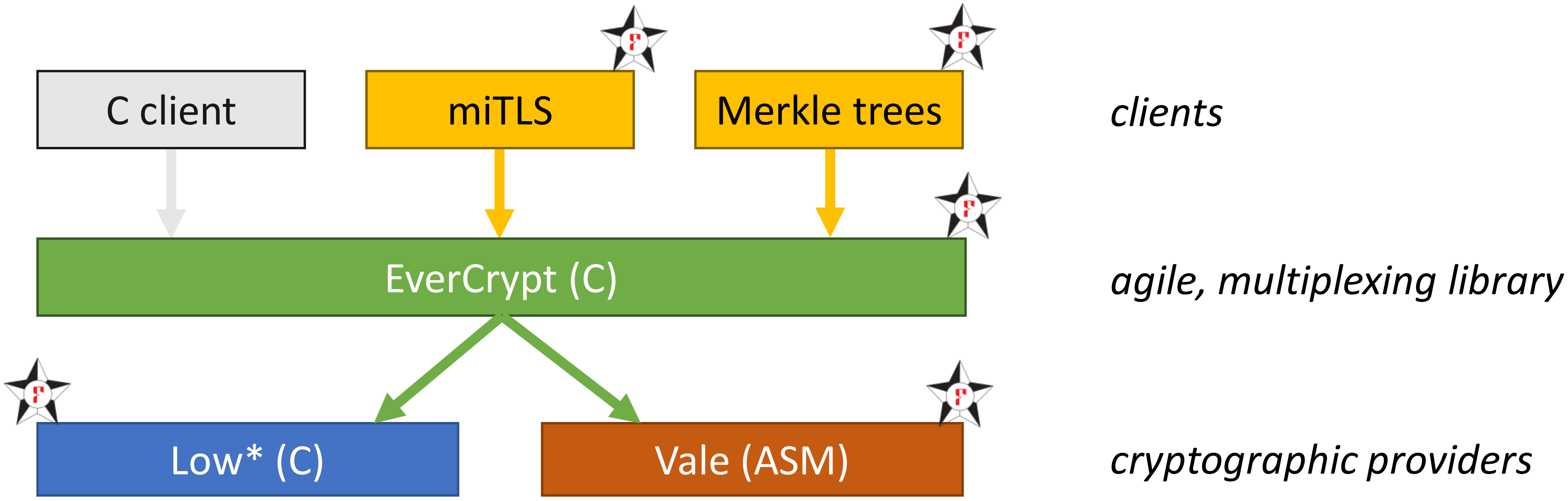
Features of an Ideal Library (researcher)

- **Verifiable**
  - written in a language amenable to verification
- **Programmer productivity**
  - share as much code as possible / agile
- **Auto-configurable**
  - doesn't blue-screen with "missing instruction"
- **Deep integration**
  - each implementation verifies against the same spec
- **Abstraction**
  - clients need not know any implementation details

<u>EverCrypt provides a comprehensive verification result without compromising performance</u>

EverCrypt Internals

EverCrypt mediates between (possibly verified) clients and different implementations



<u>EverCrypt Features</u>

- **Agility**
  - same functionality (e.g., hash), multiple algorithms
- **Multiplexing**
  - same algorithm (e.g., SHA2_256), multiple implementations
- **Abstraction**
  - clients verify against a single spec and an abstract footprint

EverCrypt is Comprehensive

Algorithm	C version	Targeted ASM version
AEAD		
AES-GCM		AES-NI + PCLMULQDQ + AVX
Chacha-Poly	yes	
High-level APIs		
Box	yes	
SecretBox	yes	
Hashes		
MD5	yes	
SHA1	yes	
SHA2	yes	SHA-EXT (for SHA2-224+SHA2-256)
MACS		
HMAC	yes	agile over hash
Poly1305	yes	X64
Key Derivation		
HKDF	yes	agile over hash
ECC		
Curve25519	yes	BMI2 + ADX
Ed25519	yes	
Ciphers		
ChaCha20	yes	
AES128, 256		AES NI + AVX
AES-CTR		AES NI + AVX

Talk Overview

1. Introduction to Everest and EverCrypt
2. Verifying Assembly
3. Verifying C + interop
4. Verifying Cryptographic Constructions
5. Achieving Agility and No-Cost Abstraction
6. Verified Applications

Cryptographic Implementation Requirements

Difficult to meet all three goals.

Correct

Formally prove that implementation matches specification

Secure

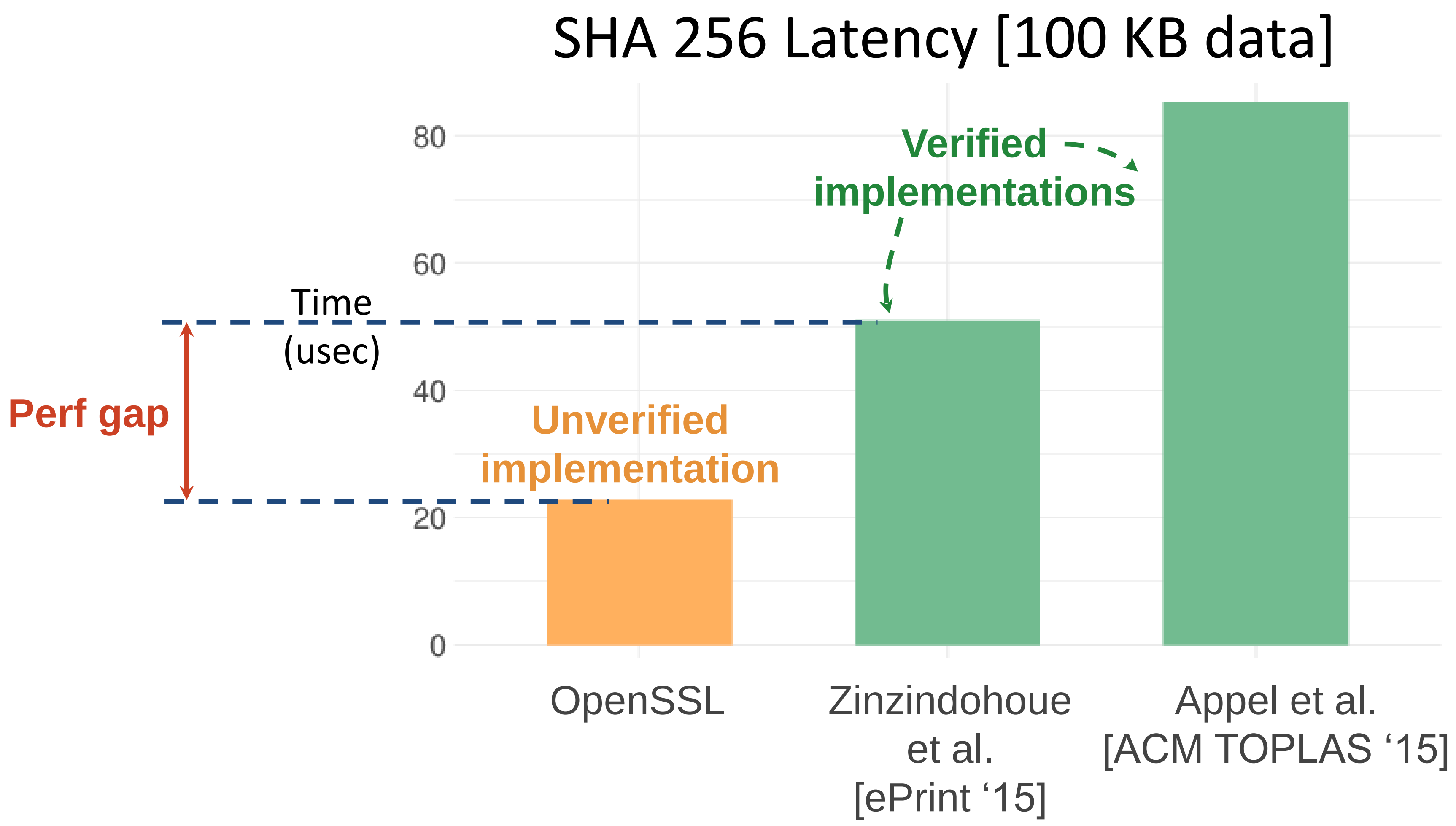
Correct control flow and free from leakage and side channels

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)
Perf gap
Unverified
implementation
Verified
implementations
80
60
40
20
0
OpenSSL
Zinzindohoue
et al.
[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
	eor	$t0,$t0,$e,ror#`$Sigma1[2]-$Sigma1[0]`	@
Sigma1(e)
#endif
END
```

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

Vale: A Firmer Foundation

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

Correct **Secure** **Fast**

Vale: A Firmer Foundation

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.

Key **Language Constructs** in Vale

Assembly Instructions	**Structured Control Flow**	**Optimization Constructs**
e.g. Mov, Rev, and AesKeygenAssist	*e.g.* if, while, and procedure	
Vary according to the target platform	Enable proof composition	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.

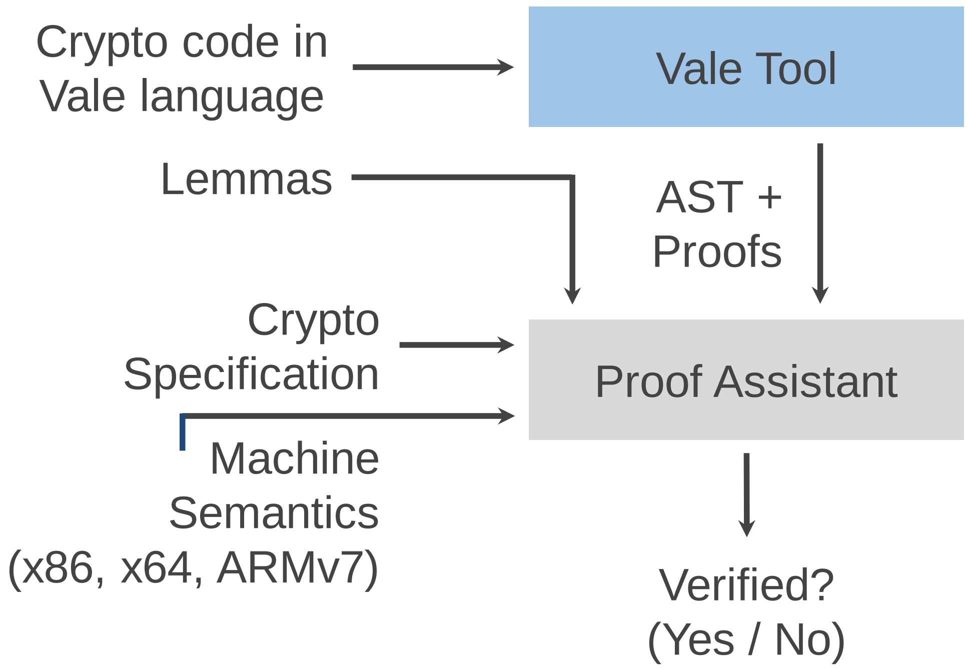
Cryptographic Implementation Requirements

Correct

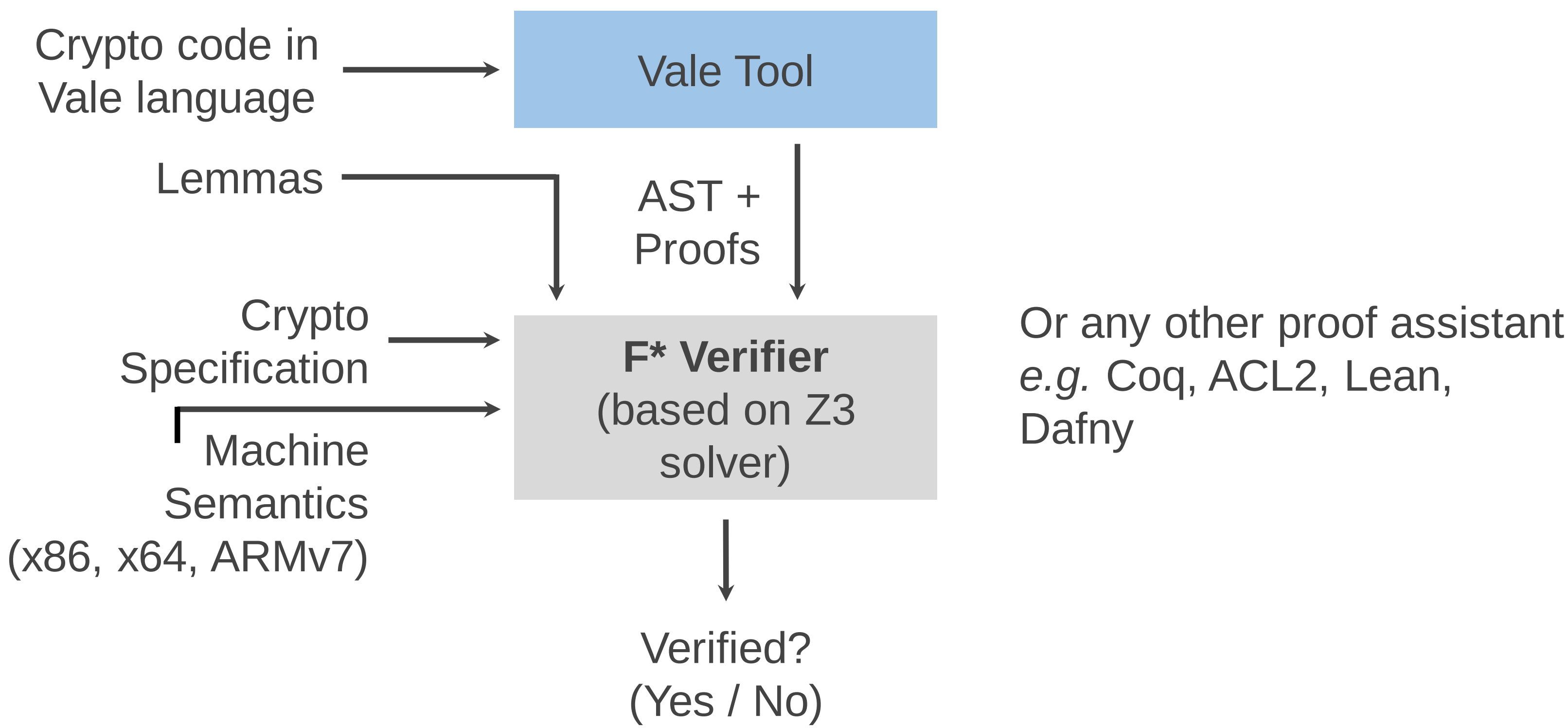
Fast

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

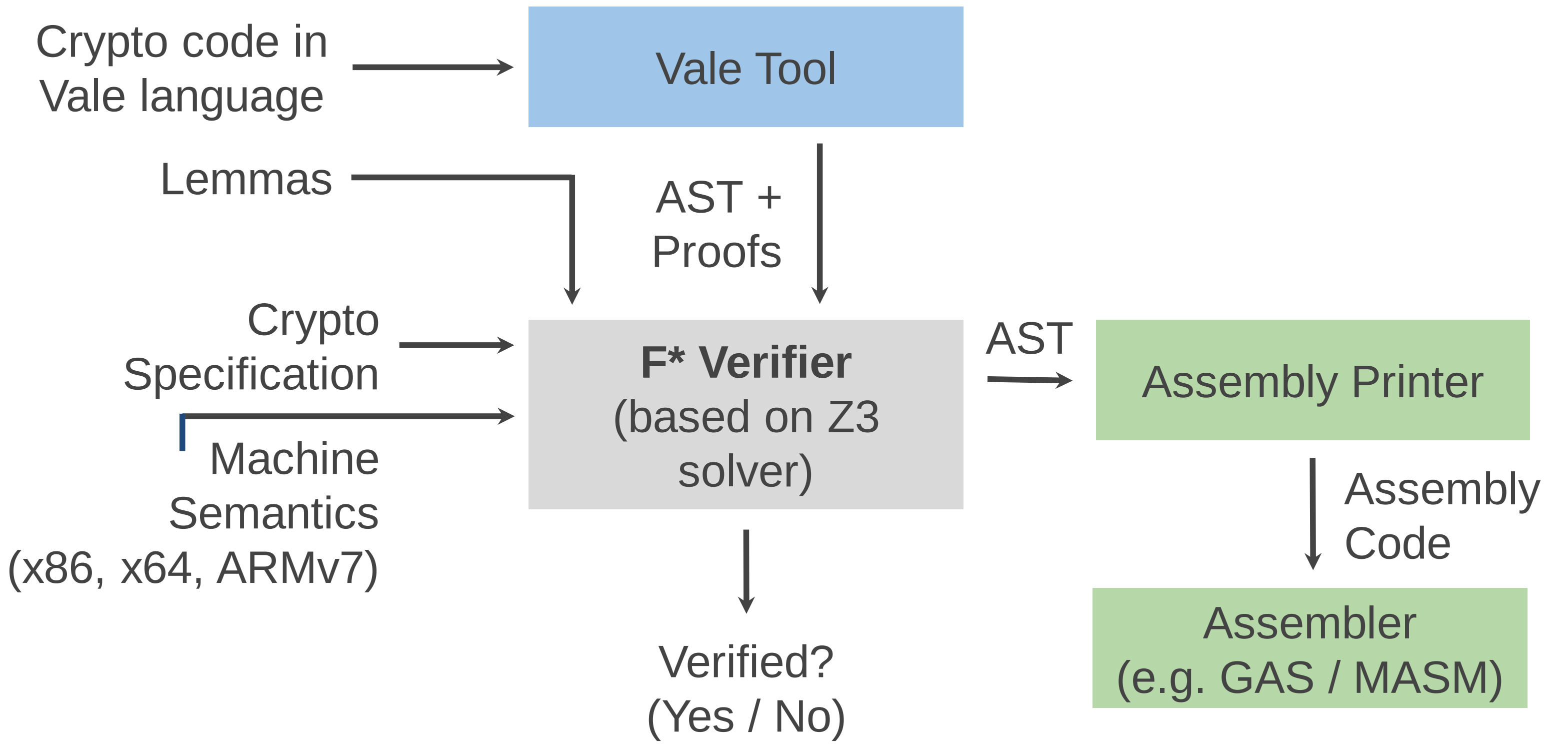
Vale Architecture

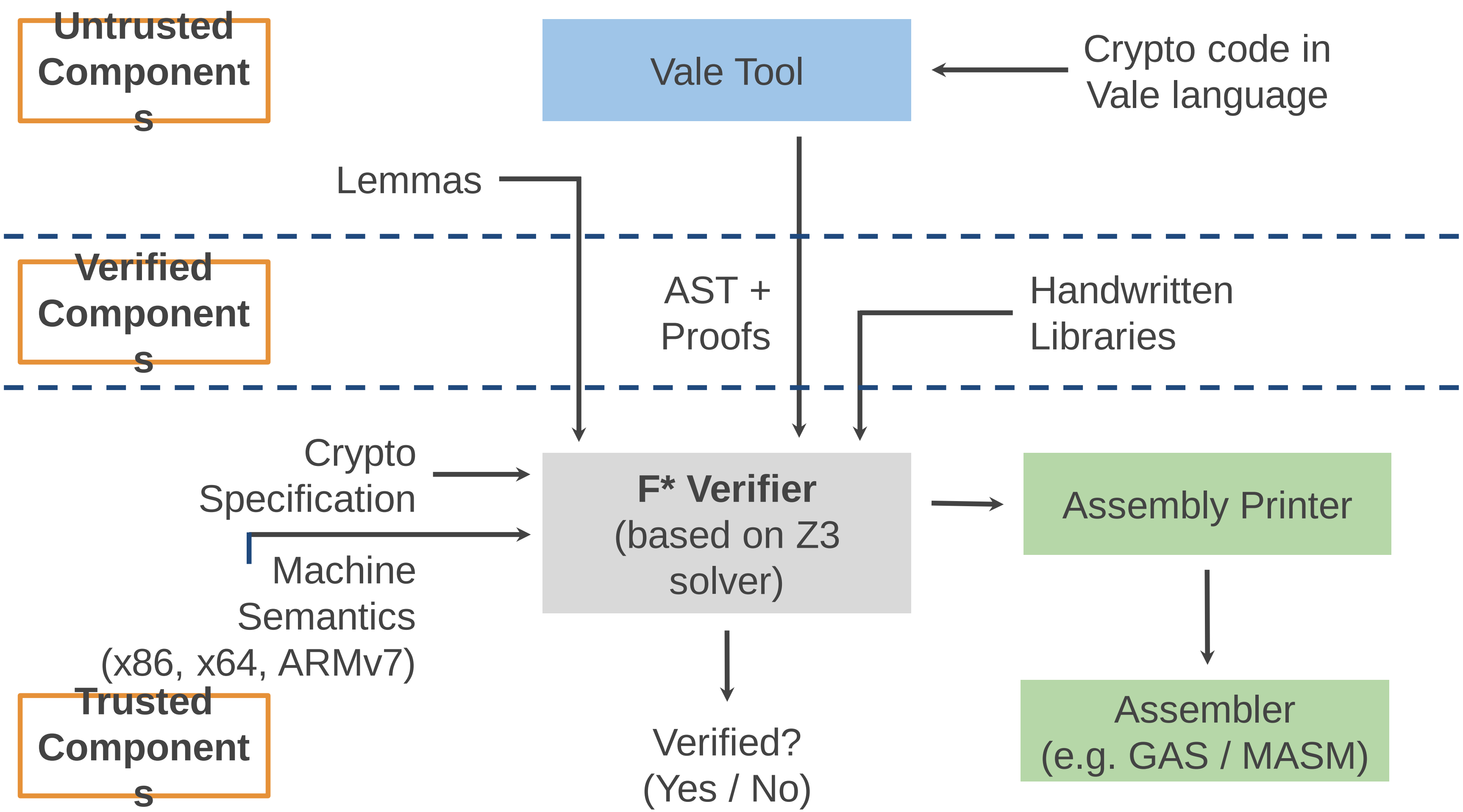


Vale Architecture



Vale Architecture




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

What is it like to verify software?

Demo!

Cryptographic Implementation Requirements

Correct

Vale supports assertions that are checked by F*

Fast

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

Cryptographic Implementation Requirements

Correct

Vale supports assertions that are checked by F*

Secure (Leakage Free)

Fast

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

Secret Information Leakage

Secrets should not leak through:

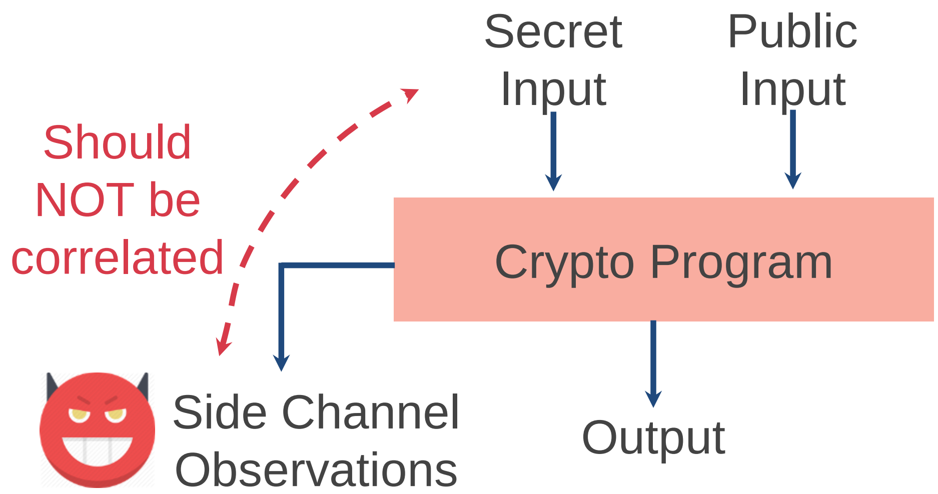
➔ **Digital Side Channels:** Observations of program behavior through cache usage, timing, memory accesses, etc.

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

Secret Information Leakage

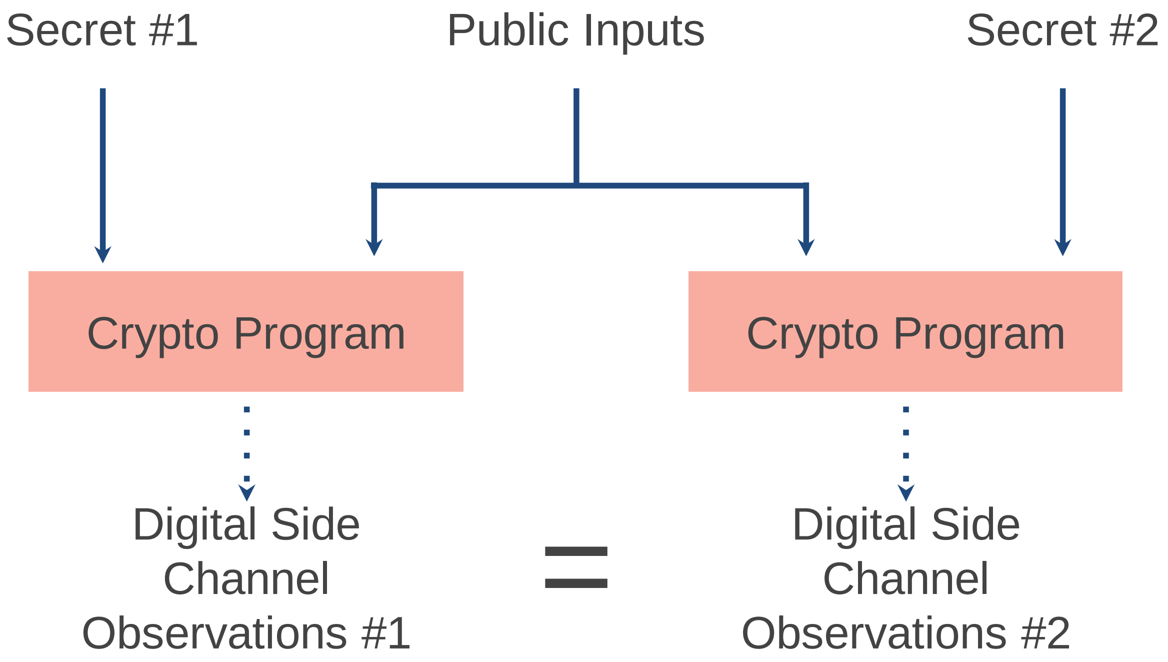
Secrets should not leak through:

➔ **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

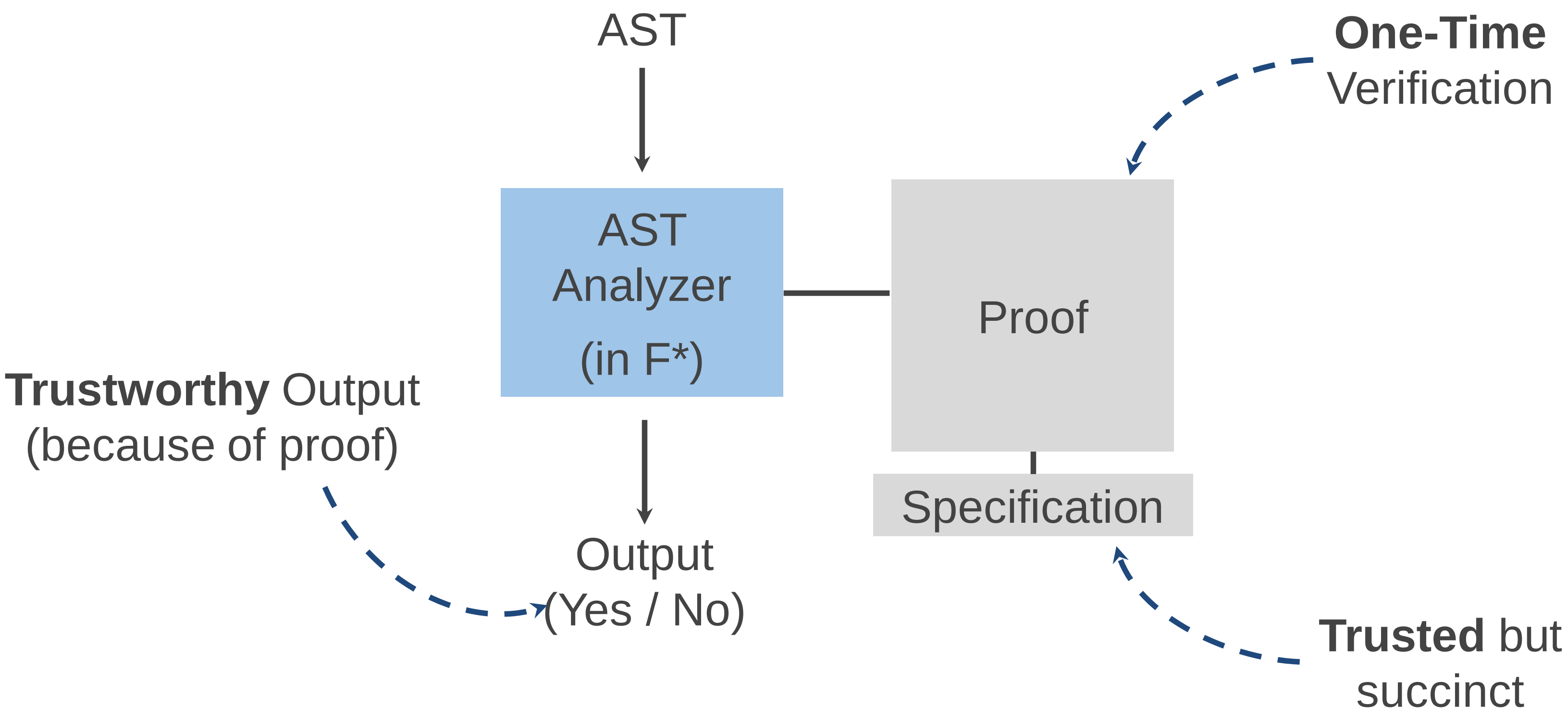
Formally, for a crypto program C,

$\forall$ 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

AES AST / **Poly-1305** AST / **SHA-256** AST / …

↓

Verified Leakage Analyzer

↓

Leakage Free?
(Yes / No)

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 analysis will be imprecise
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.

Examples of Using Vale

A few examples of the many cryptographic programs verified in Vale:

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 and AES-GCM (with AESNI) on x64

After fixing the issues, all 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 the verifer**.

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

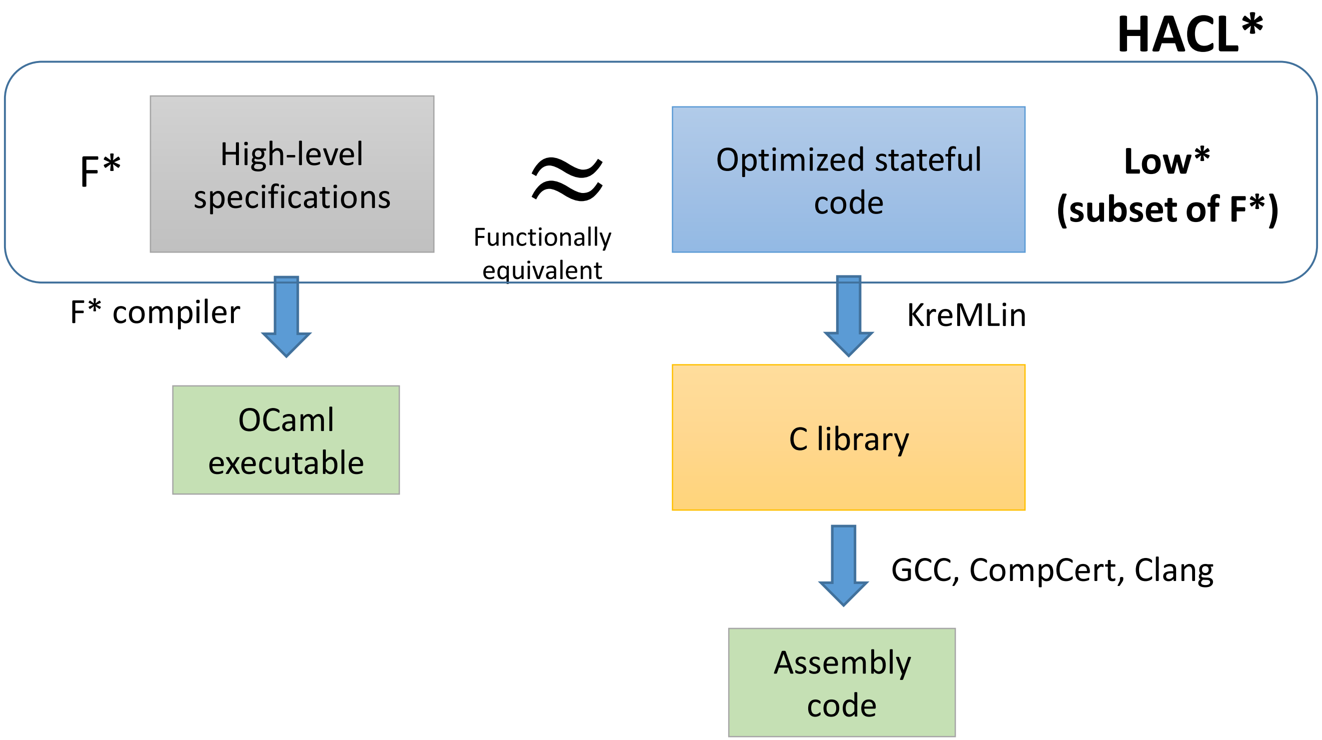
Vale Summary

- 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.

Talk Overview

1. Introduction to Everest and EverCrypt
2. Verifying Assembly
3. Verifying C + interop
4. Verifying Cryptographic Constructions
5. Achieving Agility and No-Cost Abstraction
6. Verified Applications

Verified C With the HACL* Architecture



HACL* SHA example

```
// F* code
let _Ch x y z =
  H32.logxor (H32.logand x y)
             (H32.logand (H32.lognot x) z)
…
let shuffle_core hash block ws k t =
  …
  let e = hash.(4ul) in
  let f = hash.(5ul) in
  let g = hash.(6ul) in
  …
  let t1 = …(_Ch e f g)… in
  let t2 = … in
```

```
// C code
…
uint32_t e = hash_0[4];
uint32_t f1 = hash_0[5];
uint32_t g = hash_0[6];
…
uint32_t t1 =  …(e & f1 ^ ~e & g)…;
uint32_t t2 = …;
```

Verified Interoperation Between C and Assembly

- Low* can be extracted to C
- Vale verifies assembly code
- We **verifiably** interoperate between C and assembly
- **Challenges**:
  - Different memory models
  - Calling conventions vary based on hardware, OS, compiler
  - Different security mechanisms for preventing side channels

Verified Interoperation Between C and Assembly

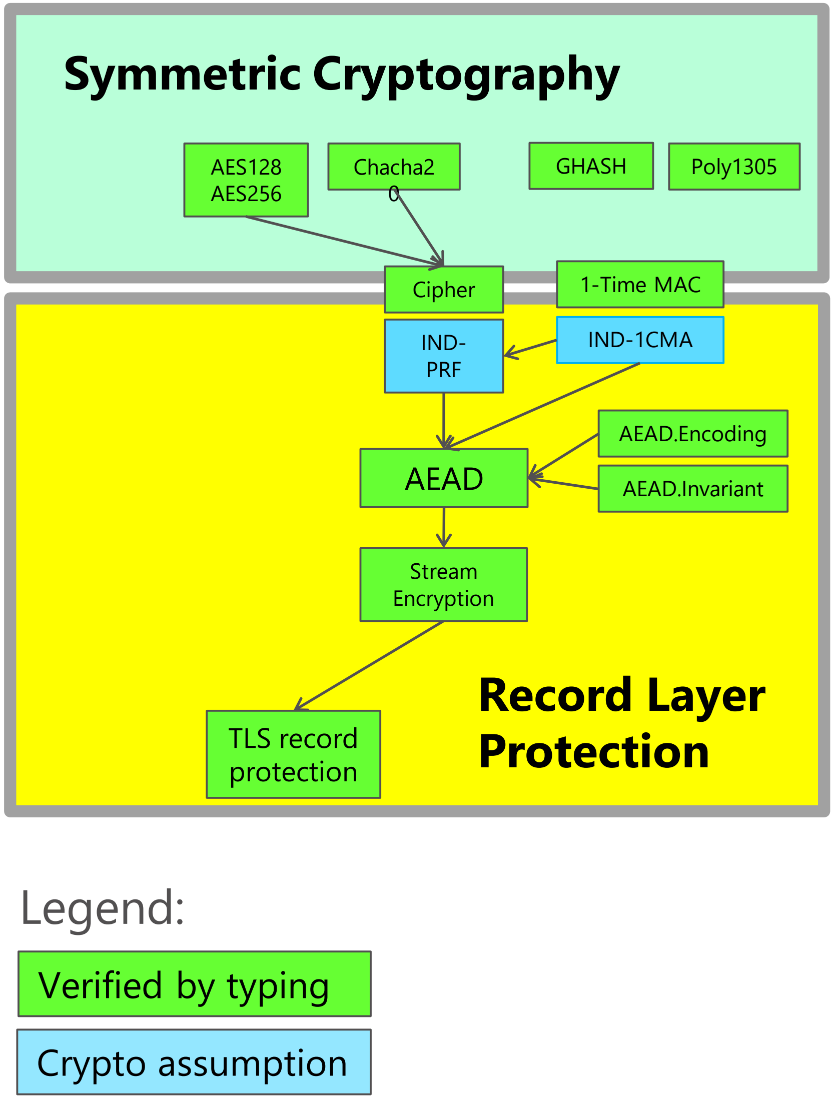
- Reconciling Memory
  - A **map** from the Low* memory model to Vale's
  - A library of **views** that capture the layout of arrays
- Calling Conventions
  - A generic **trusted** wrapper sets up the initial register state
  - A combinator **captures** that a Vale procedure (`mem -> mem`) can "morally" be executed with a suitable effect when in Low*
- Security
  - (Paper) proof unifying sequences of Low* and Vale observations

Talk Overview

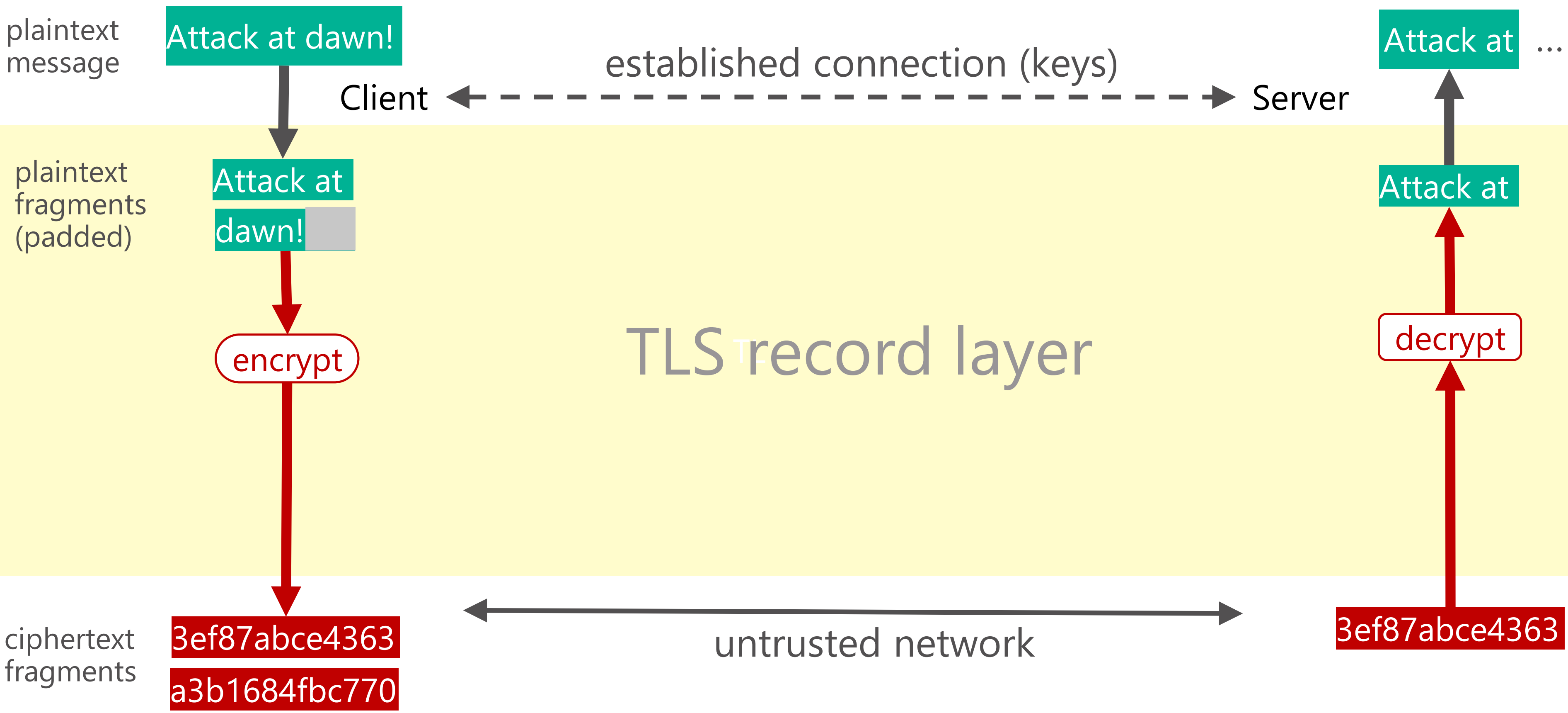
1. Introduction to Everest and EverCrypt
2. Verifying Assembly
3. Verifying C + interop
4. Verifying Cryptographic Constructions
5. Achieving Agility and No-Cost Abstraction
6. Verified Applications

Illustrate crypto construction verification on TLS 1.3 record layer

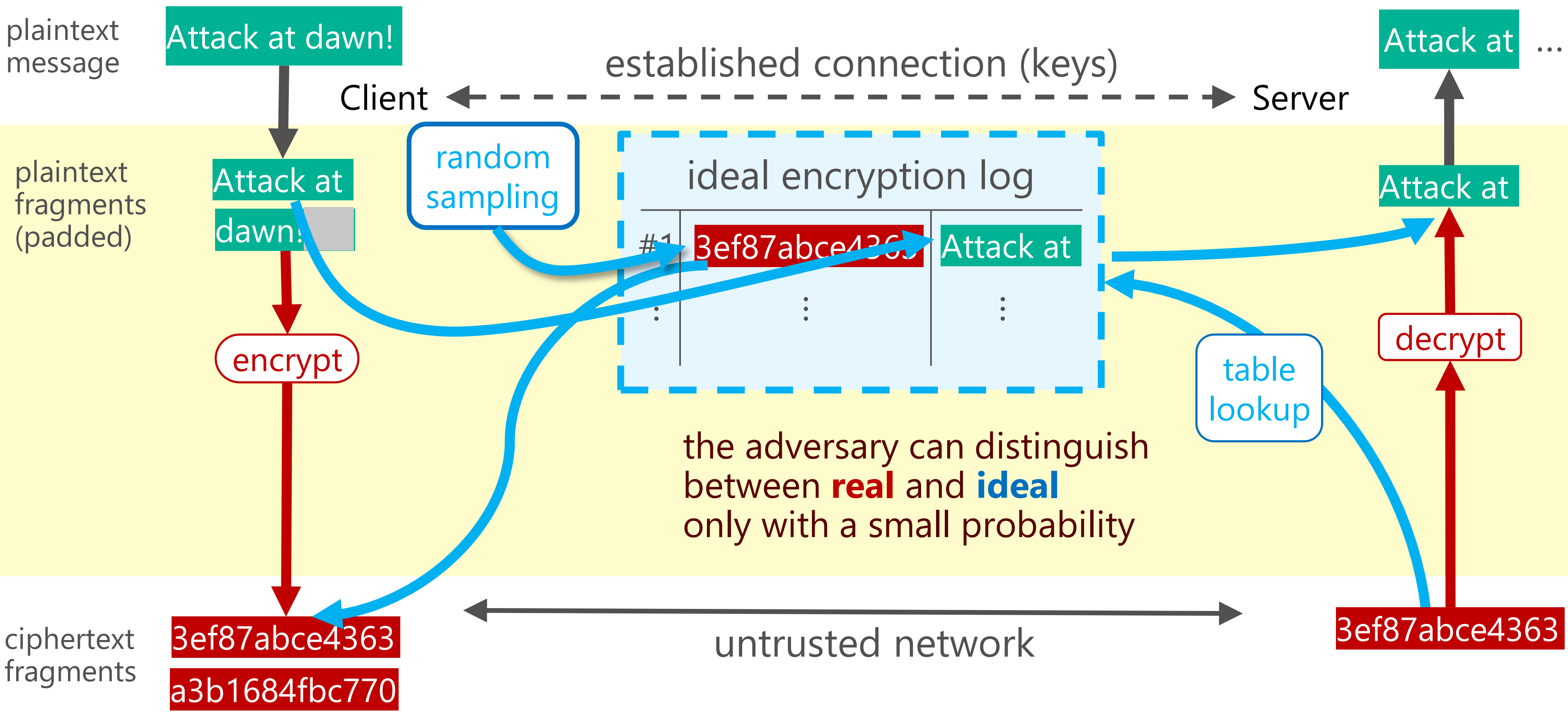
- Security definition
- New constructions
- Concrete security bounds
- Verification



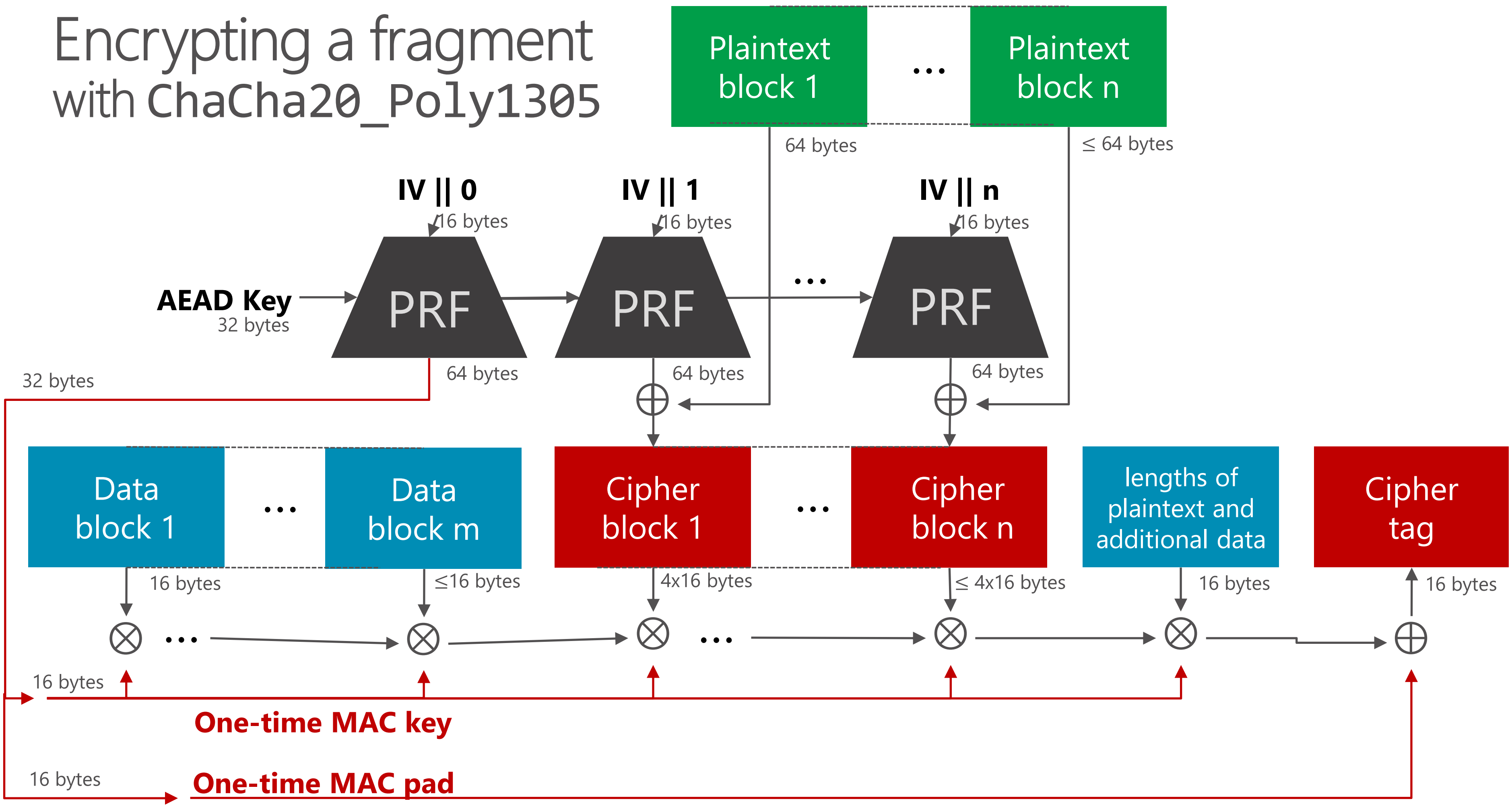
Stream Encryption: Security Definition



Stream Encryption: Security Definition



Encrypting a fragment with ChaCha20_Poly1305



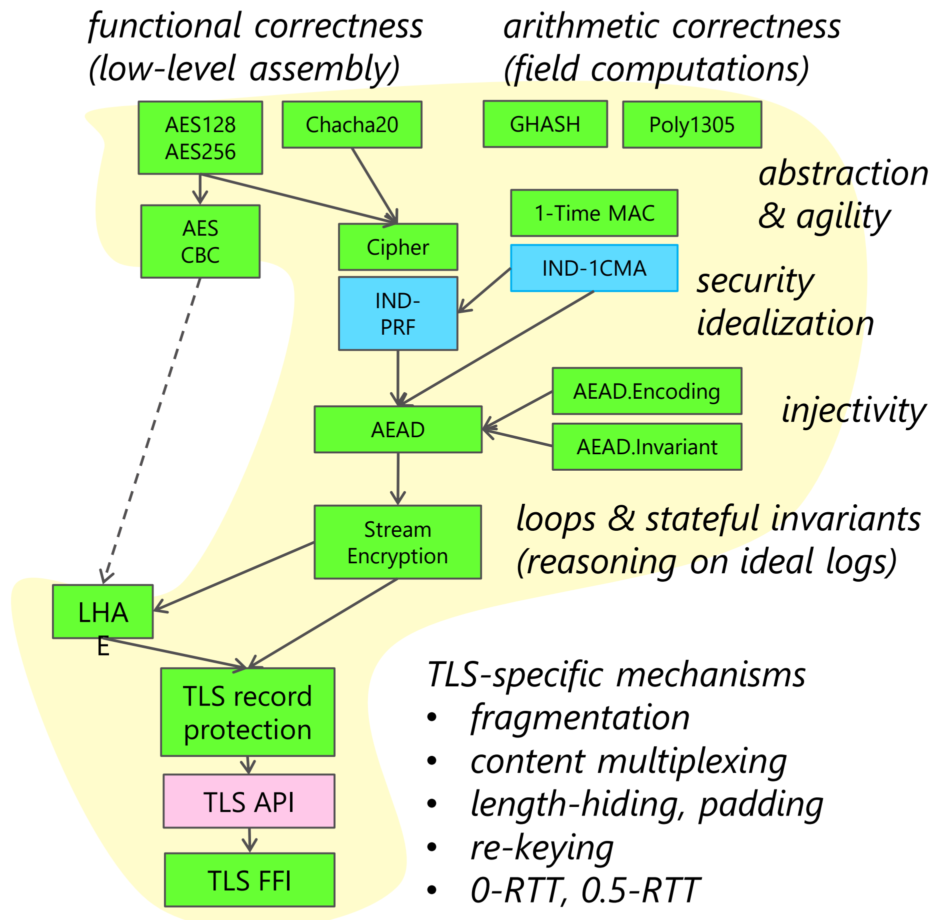
Stream Encryption: Construction

many kinds of proofs
not just code safety!

Given
- a block cipher, modelled as
  a pseudo-random function
- a field for computing one-time MACs
- injective message encodings

We program and verify a generic **authenticated stream encryption with associated data.**

We show
- functional correctness
- security (reduction to PRF assumption)
- concrete security bounds for the
  3 main record ciphersuites of TLS



Stream Encryption: Concrete Bounds

Theorem: the 3 main AEAD ciphersuites are secure for TLS 1.2 and 1.3 except with probabilities

Ciphersuite	$\epsilon_{\mathsf{Lhse}}(\mathcal{A}[q_e, q_d]) \leq$
General bound	$\epsilon_{\mathsf{Prf}}(\mathcal{B}[q_e(1 + \lceil (2^{14} + 1)/\ell_b \rceil) + q_d + j_0]) + \epsilon_{\mathsf{MMac1}}(\mathcal{C}[2^{14} + 1 + 46, q_d, q_e + q_d])$
ChaCha20-Poly1305	$\epsilon_{\mathsf{Prf}}\left(\mathcal{B}\left[q_e\left(1 + \left\lceil \frac{(2^{14}+1)}{64} \right\rceil\right) + q_d\right]\right) + \frac{q_d}{2^{93}}$
AES128-GCM AES256-GCM	$\epsilon_{\mathsf{Prp}}(\mathcal{B}[q_b]) + \frac{q_b^2}{2^{129}} + \frac{q_d}{2^{118}}$ where $q_b = q_e(1 + \lceil (2^{14} + 1)/16 \rceil) + q_d + 1$
AES128-GCM AES128-GCM	$\frac{q_e}{2^{24.5}}\left(\epsilon_{\mathsf{Prp}}(\mathcal{B}[2^{34.5}]) + \frac{1}{2^{60}} + \frac{1}{2^{56}}\right)$ with re-keying every $2^{24.5}$ records (counting q_b for all streams, and $q_d \leq 2^{60}$ per stream)

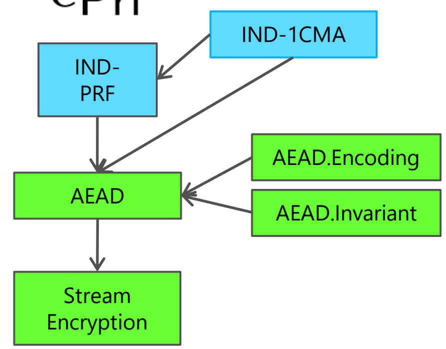
q_e is the number of encrypted records;
q_d is the number of chosen-ciphertext decryptions;
q_b is the total number of blocks for the PRF

Standard crypto assumption

ϵ_{Prf}

Probabilistic proof (on paper) in abstract field + F verification*

$$\epsilon_{\mathsf{MMac1}} = \frac{d.\tau.q_v}{|R|}$$



$$\epsilon_{\mathsf{Lhse}}(\mathcal{A}[q_e, q_d]) = \epsilon_{\mathsf{Prf}} + \epsilon_{\mathsf{MMac1}}$$

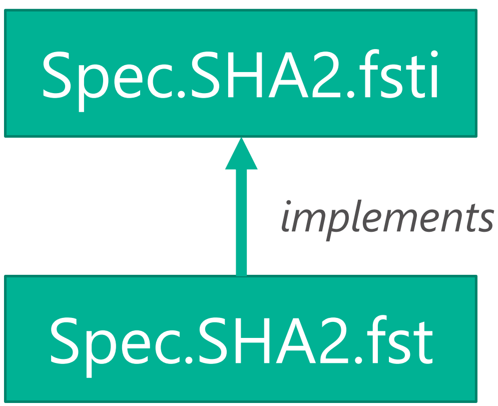
F type-based verification on code formalizing game-based reduction*

Talk Overview

1. Introduction to Everest and EverCrypt
2. Verifying Assembly
3. Verifying C + interop
4. Verifying Cryptographic Constructions
5. Achieving Agility and No-Cost Abstraction
6. Verified Applications

Abstract, Agile Specifications

- One **key challenge** in SMT-backed software verification: the context
- Introducing **abstractions** is essential, even at the level of the **specs**




```
val compress:
  a:sha_alg -> state a -> bytes -> state a
```

- **Agile** specifications limit code duplication!
- **Abstract** specifications tame context prolifera

This maximizes spec compactness

Generic Programming + Partial Evaluation

This is not Low*:

```
val compress:
  a:sha_alg → state a → array u8 → Stack unit
```

Reason:

```
let state a = function
  | SHA2_224 | SHA2_256 -> array u32
  | SHA2_384 | SHA2_512 -> array u64
```

This *could* be compiled as a union.
However, this is not **idiomatic or efficient.**

Instead, we rely on **partial evaluation:**

```
let compress_224 = compress SHA2_224
let compress_256 = compress SHA2_256
let compress_384 = compress SHA2_384
let compress_512 = compress SHA2_512
```

Connecting Vale and HACL* for Implementation Multiplexing

```
let multiplexed_compress_blocks_sha2_256
  (s: state SHA2_256)
  (blocks: array u8)
  (n: u32)
=
  if StaticConfig.has_vale && AutoConfig.has_shaext then
    Vale.Interop.SHA2.compress_256 s blocks n
  else
    Hacl.SHA2.compress_256 s blocks n
```

This uses static and dynamic configuration

- **On the Low* side:**

```
extern void Vale_Interop_SHA2_compress256(uint32 *s, uint8 *blocks, uint32 n)
```

- **On the Vale side:**

```
.text
.global Vale_Interop_SHA2_compress256
Vale_Interop_SHA2_compress256:
```

Unified Specifications

- From the client's perspective, the algorithmic specification remains **the same**
- It is now **agile** between all algorithms from a given family
- The **specification abstraction** ensures no context pollution occurs
- The library can serve as a **foundation** for higher-level constructions

Talk Overview

1. Introduction to Everest and EverCrypt
2. Verifying Assembly
3. Verifying C + interop
4. Verifying Cryptographic Constructions
5. Achieving Agility and No-Cost Abstraction
6. Verified Applications

Verified Applications

Using **EverCrypt as a foundation**, we built advanced functionalities, such as:

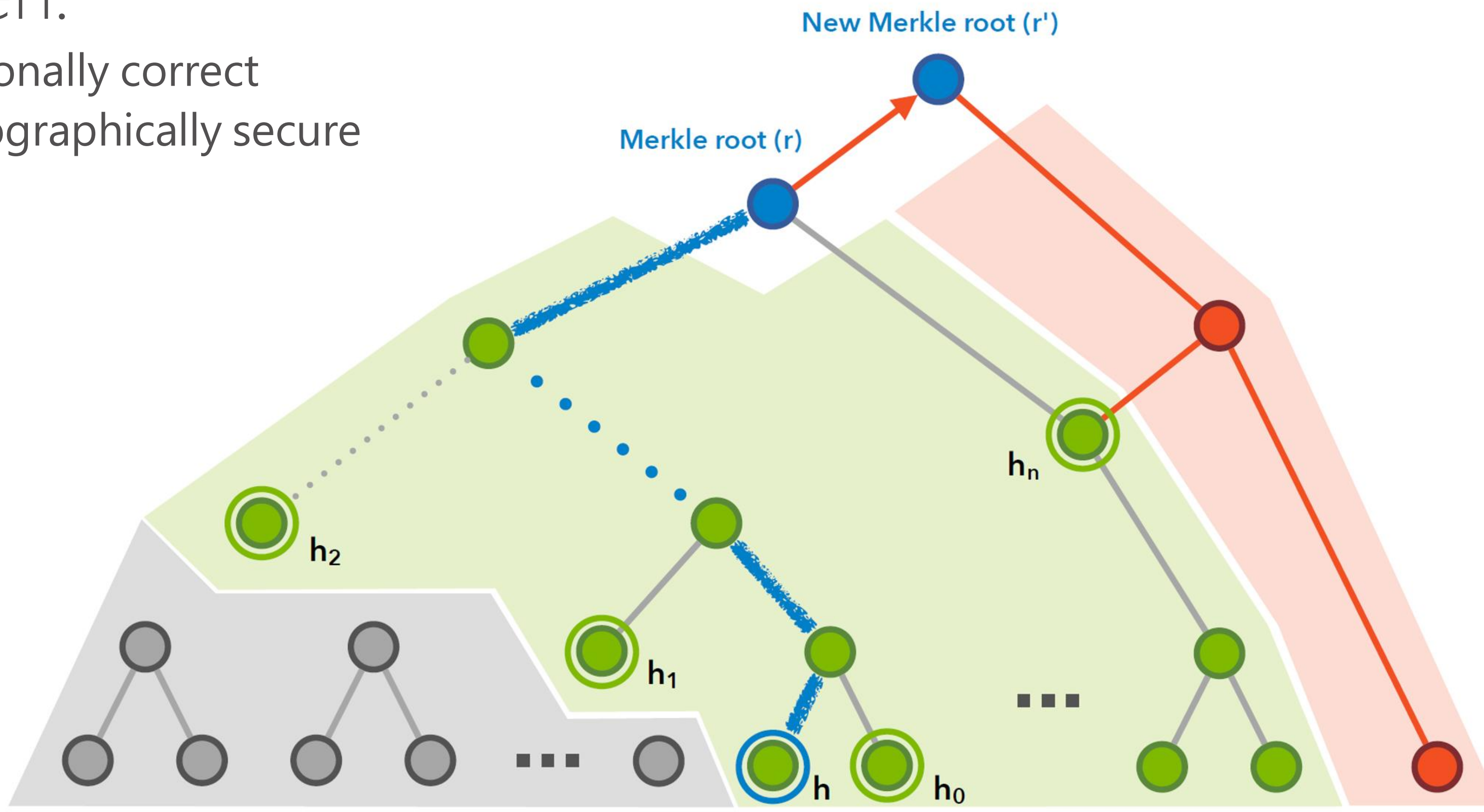
- HMAC
- HKDF
- Merkle trees
- QUIC packet encryption

Each functionality offers a **new layer of abstraction** to further shield its clients from large contexts.

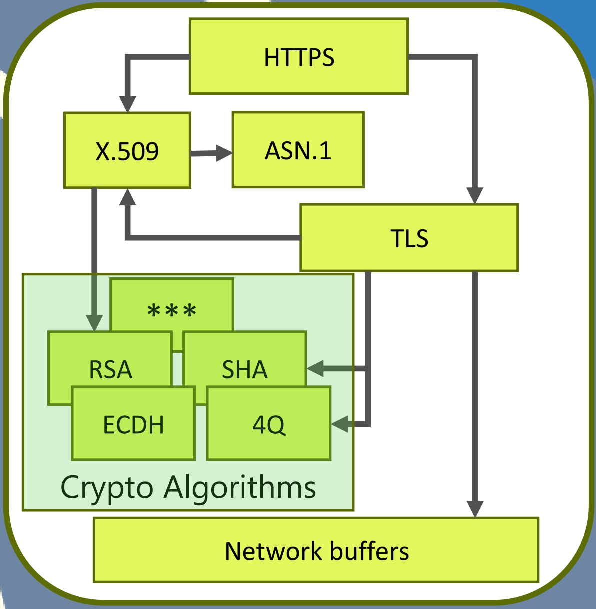
Relying on EverCrypt, each is **naturally agile and multiplexing.**

Example: Merkle trees

- Incremental tree construction
  - Each insert requires 1 hash, on average
- Proven:
  - Functionally correct
  - Cryptographically secure



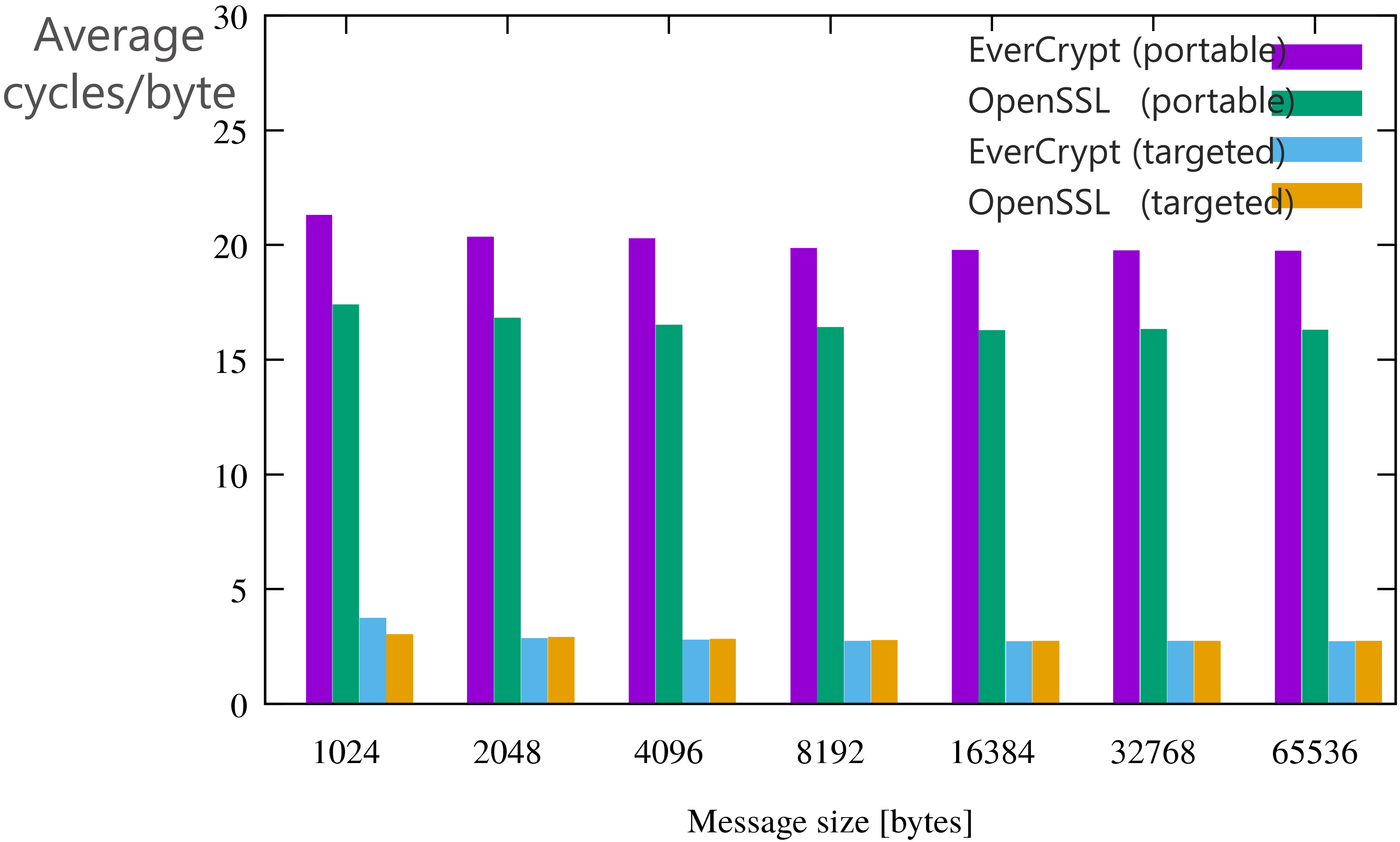
EverCrypt: Performance



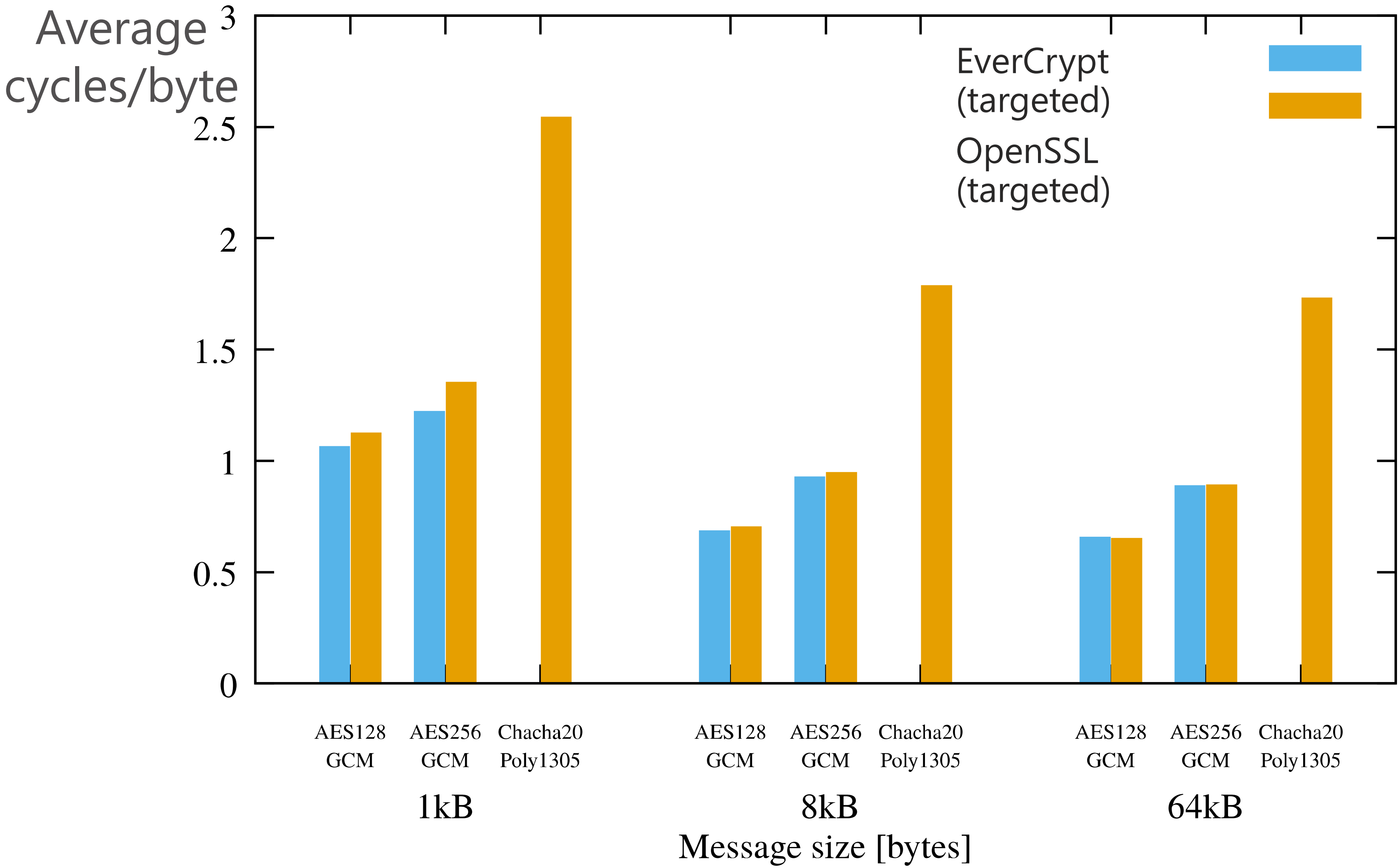
High-Level Summary:

EverCrypt *matches or exceeds* the performance of state-of-the art (verified *or unverified*) implementations!

Performance: SHA-256



Performance: AEAD

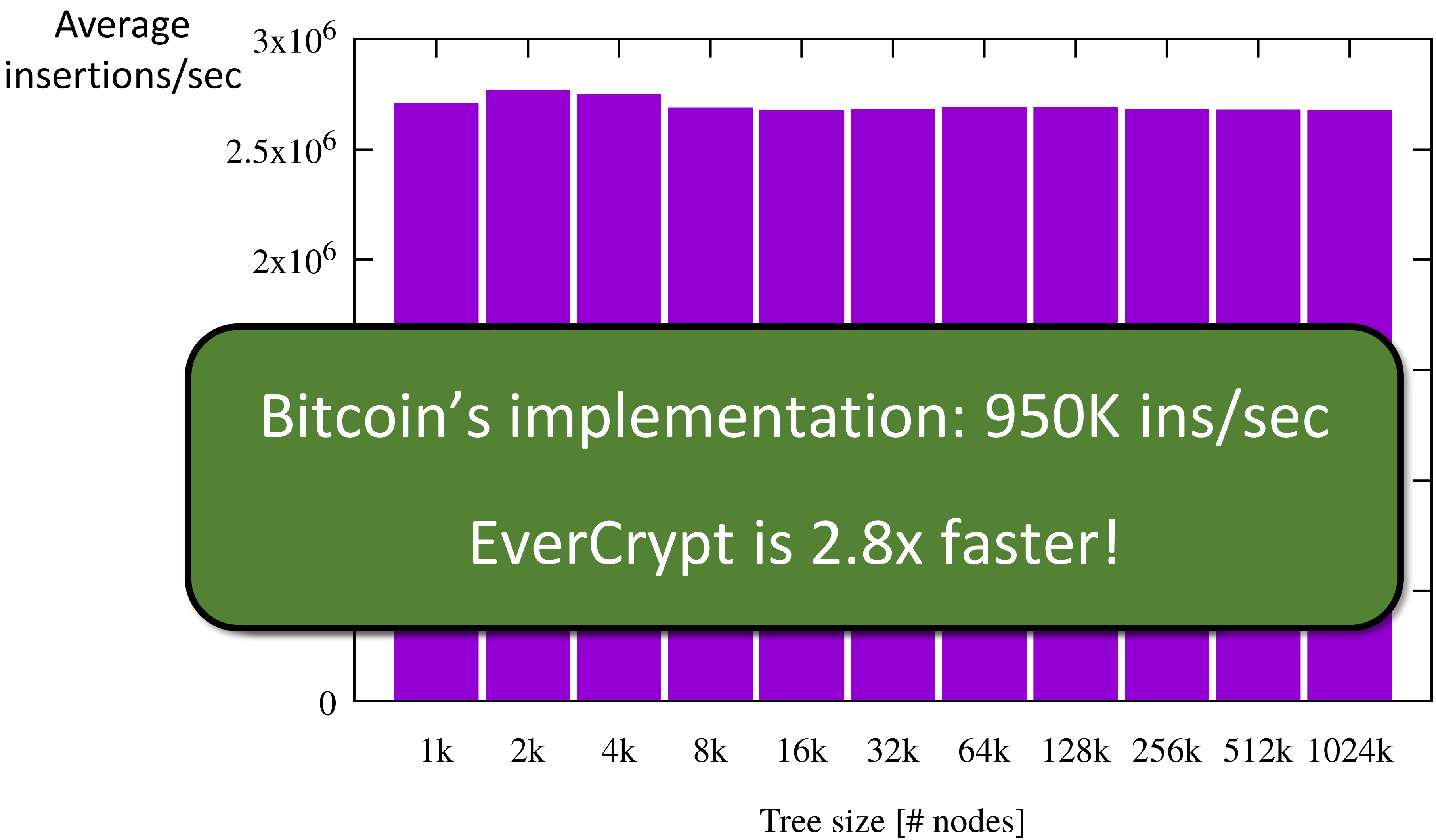


Performance: Curve25519

Unverified
Verified

Implementation	Radix	Language	CPU cycles
donna64	51	C	159634
fiat-crypto	51	C	145248
amd64-64	51	Assembly	143302
sandy2x	25.5	Assembly + AVX	135660
EverCrypt (portable)	51	C	135636
OpenSSL	64	Assembly + ADX	118604
Oliveira et al.	64	Assembly + ADX	115122
EverCrypt (targeted)	64	C + Assembly + ADX	113614

Performance: Merkle tree



Bitcoin’s implementation: 950K ins/sec

EverCrypt is 2.8x faster!

Summary

- Crypto software must be ***fast*** and ***secure***
- New verification tools & techniques make this possible
  - EverCrypt provides verified secure, agile, high-perf crypto
- Everest will showcase the power of verification and its applicability to real-world security problems

https://project-everest.github.io/

Thank you!

parno@cmu.edu