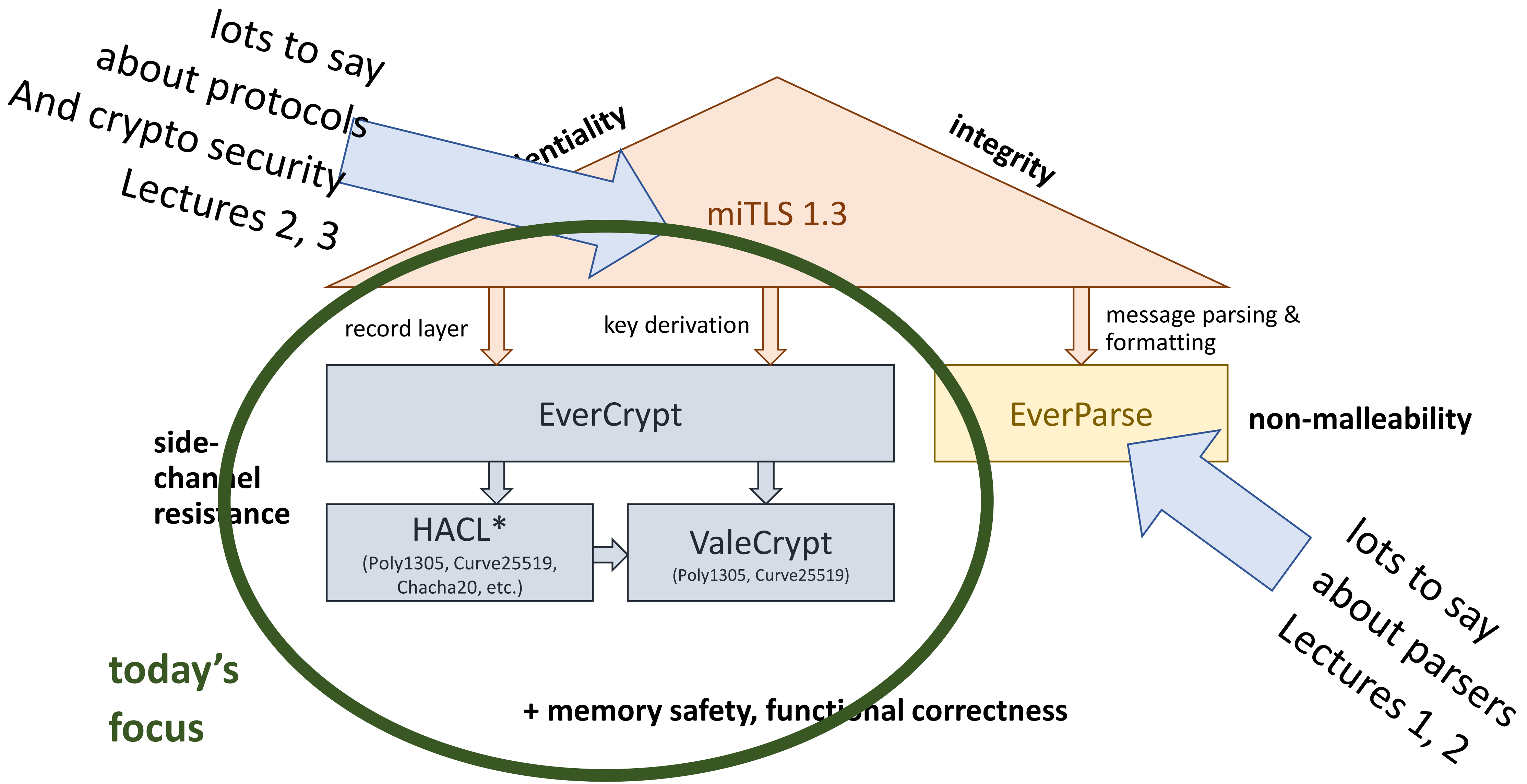
EVERCRYPT

Thanks to Jonathan Protzenko and Chris Hawblitzel for these slides.

Errors are mine

Nik Swamy, OPLSS 2019



lots to say
about protocols
And crypto security
Lectures 2, 3
entiality
integrity
miTLS 1.3
record layer
key derivation
message parsing &
formatting
EverCrypt
EverParse
non-malleability
side-
channel
resistance
HACL*
(Poly1305, Curve25519,
Chacha20, etc.)
ValeCrypt
(Poly1305, Curve25519)
lots to say
about parsers
Lectures 1, 2
today's
focus
+ memory safety, functional correctness

What is a cryptographic *provider*?

A *COLLECTION* OF ALGORITHMS **(EXHAUSTIVE)**

SEVERAL *IMPLEMENTATIONS* **(MULTIPLEXING)**

APIS GROUPED BY *FAMILY* **(AGILITY)**

EASY-TO-USE API **(CPU AUTO-DETECTION)**

An essential piece of software

A cryptographic provider is useful **beyond secure communications**, e.g.

- file encryption
- secure enclaves
- document signatures
- cryptocurrencies
- any modern piece of software

What is a cryptographic *provider?*

libsodium

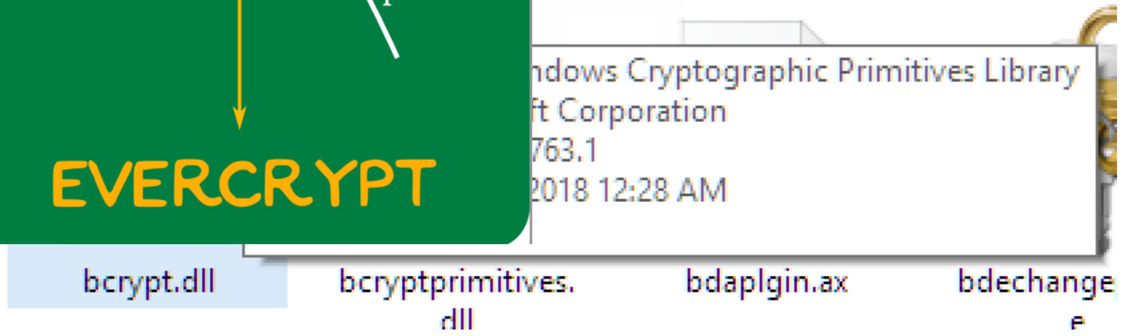
OpenSSL

Cryptography and SSL/TLS Toolkit

(libcrypto)

arm MBED

EVERCRYPT



A brief reminder: why verify cryptographic algorithms?

AES-GCM

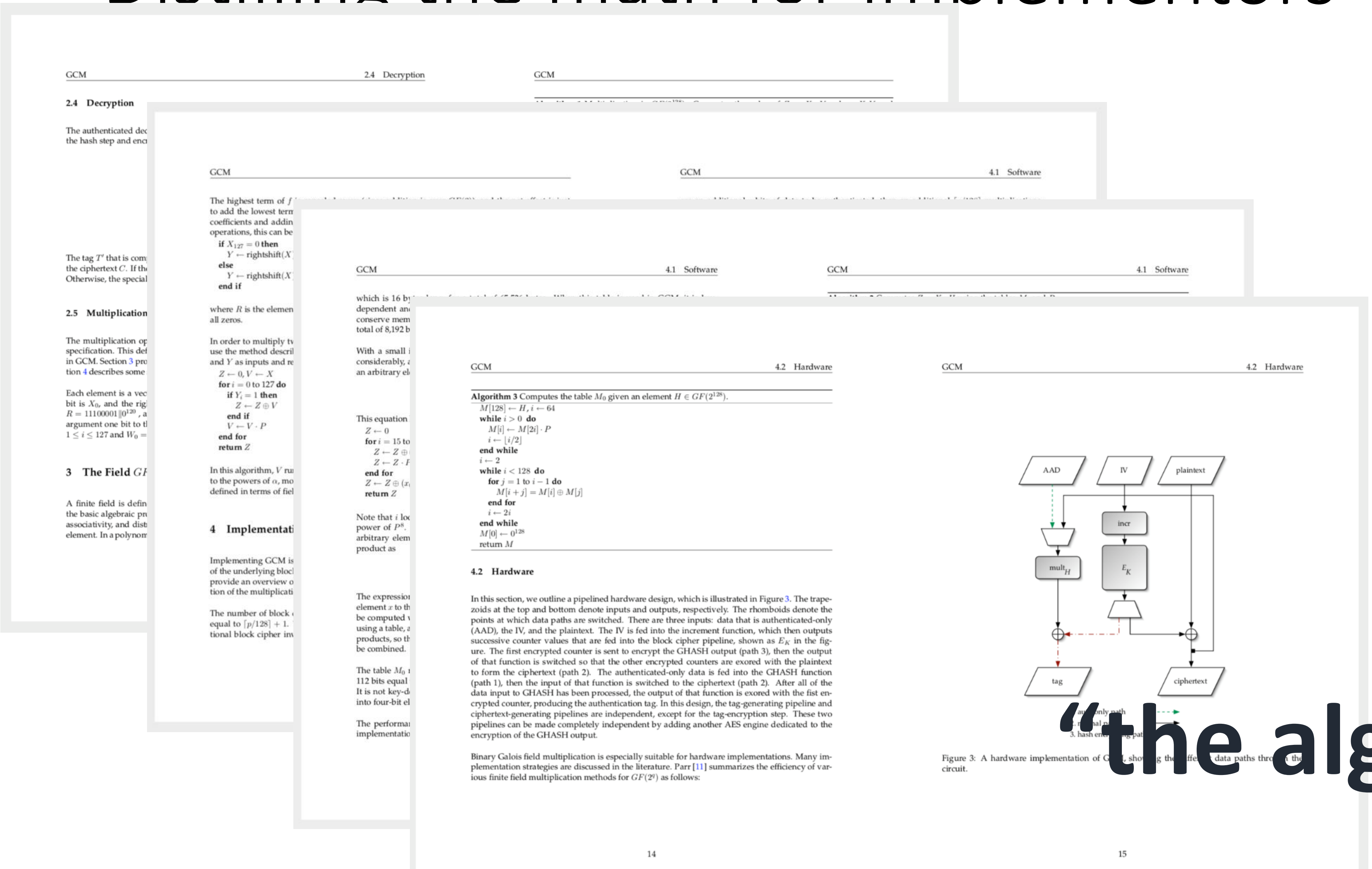
Evaluate polynomials in this field to get an **authentication code!** (see also: Poly1305)

GHASH (AES-GCM):

- $p = 2^{128}$ $(q = 2, n = 128)$
- P = $x^{128} + x^7 + x^2 + x + 1$

“the math”

Distilling the math for implementors



GCM 4.2 Hardware

Algorithm 3 Computes the table M_0 given an element $H \in GF(2^{128})$.

$M[128] \leftarrow H, i \leftarrow 64$
while $i > 0$ **do**
  $M[i] \leftarrow M[2i] \cdot P$
  $i \leftarrow \lfloor i/2 \rfloor$
end while
$i \leftarrow 2$
while $i < 128$ **do**
  for $j = 1$ to $i - 1$ **do**
    $M[i + j] = M[i] \oplus M[j]$
  end for
  $i \leftarrow 2i$
end while
$M[0] \leftarrow 0^{128}$
return M

4.2 Hardware

In this section, we outline a pipelined hardware design, which is illustrated in Figure 3. The trapezoids at the top and bottom denote inputs and outputs, respectively. The rhomboids denote the points at which data paths are switched. There are three inputs: data that is authenticated-only (AAD), the IV, and the plaintext. The IV is fed into the increment function, which then outputs successive counter values that are fed into the block cipher pipeline, shown as E_K in the figure. The first encrypted counter is sent to encrypt the GHASH output (path 3), then the output of that function is switched so that the other encrypted counters are exored with the plaintext to form the ciphertext (path 2). The authenticated-only data is fed into the GHASH function (path 1), then the input of that function is switched to the ciphertext (path 2). After all of the data input to GHASH has been processed, the output of that function is exored with the fist encrypted counter, producing the authentication tag. In this design, the tag-generating pipeline and ciphertext-generating pipelines are independent, except for the tag-encryption step. These two pipelines can be made completely independent by adding another AES engine dedicated to the encryption of the GHASH output.

Binary Galois field multiplication is especially suitable for hardware implementations. Many implementation strategies are discussed in the literature. Parr [11] summarizes the efficiency of various finite field multiplication methods for $GF(2^q)$ as follows:

14

GCM 4.2 Hardware

Figure 3: A hardware implementation of G[...], sho[...]g th[...] [...] data paths thro[...] the circuit.

15

“the algorithm”

Writing the actual code

A long way from quotienting a ring by an ideal

“the reality”

```
.Lgcm_dec_body:
___
$code.=<<___;
	vzeroupper

	vmovdqu		($ivp),$T1		# input counter value
	add		\$-128,%rsp
	mov		12($ivp),$counter
	lea		.Lbswap_mask(%rip),$const
	lea		-0x80($key),$in0	# borrow $in0
	mov		\$0xf80,$end0		# borrow $end0
	vmovdqu		($Xip),$Xi		# load Xi
	and		\$-128,%rsp		# ensure stack alignment
	vmovdqu		($const),$Ii		# borrow $Ii for .Lbswap_mask
	lea		0x80($key),$key		# size optimization
	lea		0x20+0x20($Xip),$Xip	# size optimization
	mov		0xf0-0x80($key),$rounds
	vpshufb		$Ii,$Xi,$Xi

	and		$end0,$in0
	and		%rsp,$end0
	sub		$in0,$end0
	jc		.Ldec_no_key_aliasing
	cmp		\$768,$end0
	jnc		.Ldec_no_key_aliasing
	sub		$end0,%rsp		# avoid aliasing with key
.Ldec_no_key_aliasing:

	vmovdqu		0x50($inp),$Z3		# I[5]
	lea		($inp),$in0
	vmovdqu		0x40($inp),$Z0
	lea		-0xc0($inp,$len),$end0
	vmovdqu		0x30($inp),$Z1
	shr		\$4,$len
	xor		$ret,$ret
	vmovdqu		0x20($inp),$Z2
	 vpshufb	$Ii,$Z3,$Z3		# passed to _aesni_ctr32_ghash_6x
	vmovdqu		0x10($inp),$T2
	 vpshufb	$Ii,$Z0,$Z0
	vmovdqu		($inp),$Hkey
	 vpshufb	$Ii,$Z1,$Z1
	vmovdqu		$Z0,0x30(%rsp)
	 vpshufb	$Ii,$Z2,$Z2
	vmovdqu		$Z1,0x40(%rsp)
	 vpshufb	$Ii,$T2,$T2
	vmovdqu		$Z2,0x50(%rsp)
	 vpshufb	$Ii,$Hkey,$Hkey
	vmovdqu		$T2,0x60(%rsp)
	vmovdqu		$Hkey,0x70(%rsp)

	call		_aesni_ctr32_ghash_6x

	vmovups		$inout0,-0x60($out)	# save output
	vmovups		$inout1,-0x50($out)
```

What could possibly go wrong?

Many bugs in Curve25519 **implementations**

(C and assembly)

agl / **curve25519-donna**

Watch

<> Code | Issues 2 | Pull requests 7 | Projects 0 | Wiki | Insights

Correct bounds in 32-bit code.

```
The 32-bit code was illustrative of the tricks used in the original
curve25519 paper rather than rigorous. However, it has proven quite
popular.

This change fixes an issue that Robert Ransom found where outputs between
2^255-19 and 2^255-1 weren't correctly reduced in fcontract. This
appears to leak a small fraction of a bit of security of private keys.

Additionally, the code has been cleaned up to reflect the real-world
needs. The ref10 code also exists for 32-bit, generic C but is somewhat
slower and objections around the lack of qhasm availibility have been
raised.
```

master 1.3

Curve25519-donna

agl committed on Jun 9, 2014 — 1 parent

```
sv pack25519(u8 *o
{
  int i,j,b;
  gf m,t;
  FOR(i,16) t[i]=n
  car25519(t);
  car25519(t);
  car25519(t);
  FOR(j,2) {
    m[0]=t[0]-0xff
    for(i=1;i<15;i
      m[i]=t[i]-0x
      m[i-1]&=0xff
    }
    m[15]=t[15]-0x
    b=(m[15]>>16)&
    m[15]&=0xffff;
    sel25519(t,m,1-b);
  }
  FOR(i,16) {
    o[2*i]=t[i]&0xff;
    o[2*i+1]=t[i]>>8;
  }
}
```

TweetNaCl

This bug is triggered when the last limb `n[15]` of the input argument `n` of this function is greater or equal than `0xffff`. In these cases the result of the scalar multiplication is not reduced as expected resulting in a wrong packed value. This code can be fixed simply by replacing `m[15]&=0xffff;` by `m[14]&=0xffff;`.

Ed25519 amd64 bug

gistfile1.md — Raw

NaCl (asm)

While visiting 30c3, I attended the You-broke-the-Internet workshop on NaCl.

One thing mentioned in the talk was that auditing crypto code is a lot of work, and that this is one of the reasons why Ed25519 isn't included in NaCl yet (they promised a version including it for 2014). The speakers mentioned a bug in the amd64 assembly implementation of Ed25519 as an example of a bug that can only be found by auditing, not by randomized tests. This bug is caused by a carry being added in the wrong place, but since that carry is usually zero, the bug is hard to fint (occurs with probability 2^{-60} or so).

The TweetNaCl paper briefly mentions this bug as well:

> Partial audits have revealed a bug in this software (`r1 += 0 + carry` should be `r2 += 0 + carry` in `amd64-64-24k`) that would not be caught by random tests; this illustrates the importance of audits.

Searching for this string in the SUPERCOP source code turns up four matches:

```
crypto_scalarmult\curve25519\amd64-64\fe25519_mul.s
crypto_scalarmult\curve25519\amd64-64\fe25519_square.s
crypto_sign\ed25519\amd64-64-24k\fe25519_mul.s
crypto_sign\ed25519\amd64-64-24k\fe25519_square.s
```

So it appears like the `amd64-64` implementation of both Curve25519 and Ed25519 is affected.

It seems difficult to exploit this when used for key generation or signing since the attacker cannot influence the data. Key-exchange and signature verification might be a problem.

3 Bugs in OpenSSL implementation of Poly1305 last year

OpenSSL Security Advisory [10 Nov 2016]

[openssl-dev] [openssl.org #4439] poly1305-x86.pl produces incorrect output

"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."

recommend doing the same in your own test harness, to make sure there aren't others of these bugs lurking around.

Implementation bug in AES-GCM

The fragility of AES-GCM authentication algorithm

Shay Gueron[1,2], Vlad Krasnov[2]

[1] Department of Mathematics, University of Haifa, Israel
[2] Intel Corporation, Israel Development Center, Haifa, Israel

March 15, 2013

Abstract. A new implementation of the GHASH function has been recently committed to a Git version of OpenSSL, to speed up AES-GCM. We identified a bug in that implementation, and made sure it was quickly fixed before trickling into an official OpenSSL trunk. Here, we use this (already fixed) bug

Implementation bug in Windows SymCrypt

Potential DDOS.

https://bugs.chromium.org/p/project-zero/issues/detail?id=1804

Issue 1804: cryptoapi: SymCrypt modular inverse algorithm

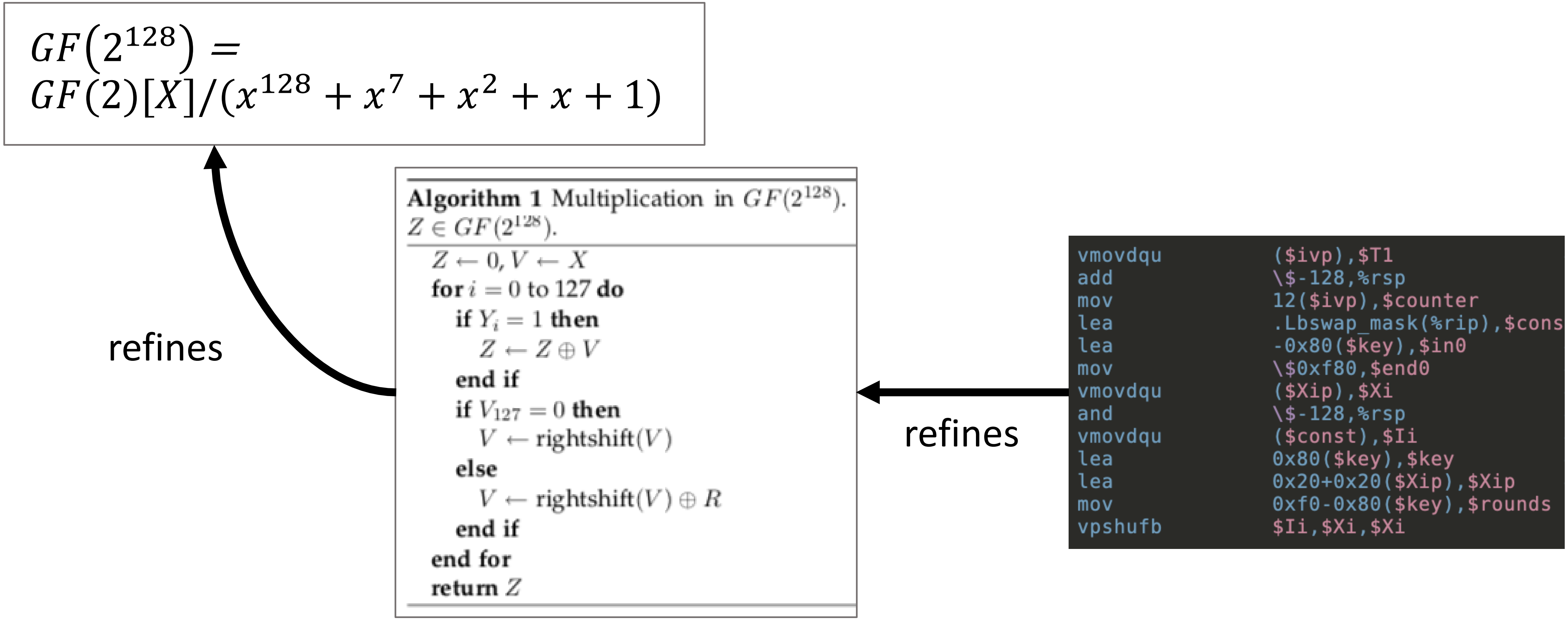
Reported by taviso@google.com on Tue, Mar 12, 2019, 9:15 PM PDT **Project Member**

Code ‹ Prev

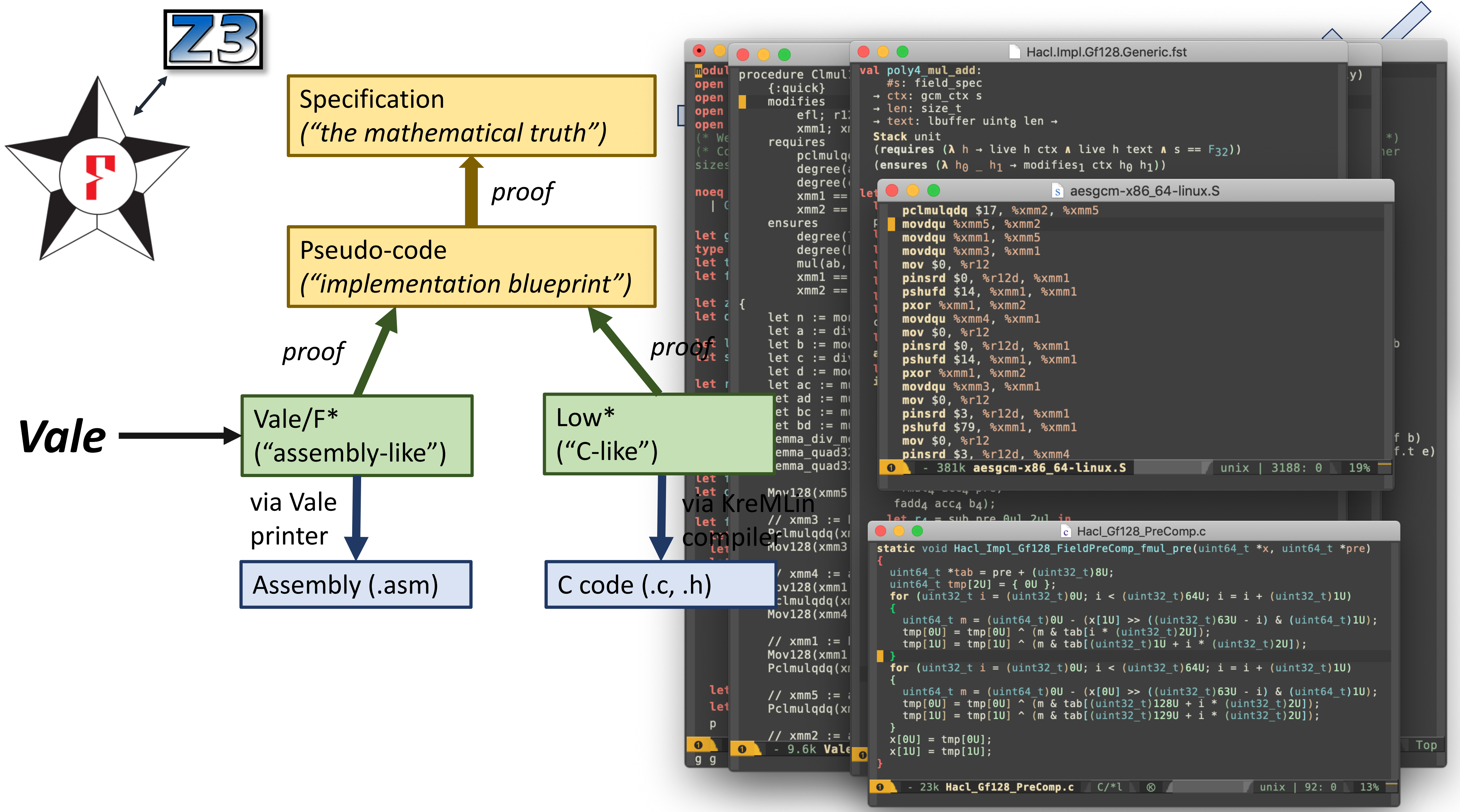
There's a bug in the SymCrypt multi-precision arithmetic routines that can cause an infinite loop when calculating the modular inverse on specific bit patterns with bcryptprimitives!SymCryptFdefModInvGeneric.

I've been able to construct an X.509 certificate that triggers the bug. I've found that embedding the certificate in an S/MIME message, authenticode signature, schannel connection, and so on will effectively DoS any windows server (e.g. ipsec, iis, exchange, etc) and (depending on the context) may require the machine to be rebooted. Obviously, lots of software that processes untrusted content (like antivirus) call these routines on untrusted data, and this will cause them to deadlock.

Program verification!



GF(2^128) =
GF(2)[X]/(x^128 + x^7 + x^2 + x + 1)
refines
Algorithm 1 Multiplication in GF(2^128).
Z ∈ GF(2^128).
Z ← 0, V ← X
for i = 0 to 127 do
if Y_i = 1 then
Z ← Z ⊕ V
end if
if V_127 = 0 then
V ← rightshift(V)
else
V ← rightshift(V) ⊕ R
end if
end for
return Z
refines
vmovdqu ($ivp),$T1
add \$-128,%rsp
mov 12($ivp),$counter
lea .Lbswap_mask(%rip),$cons
lea -0x80($key),$in0
mov \$0xf80,$end0
vmovdqu ($Xip),$Xi
and \$-128,%rsp
vmovdqu ($const),$Ii
lea 0x80($key),$key
lea 0x20+0x20($Xip),$Xip
mov 0xf0-0x80($key),$rounds
vpshufb $Ii,$Xi,$Xi



Z3
Specification
("the mathematical truth")
proof
Pseudo-code
("implementation blueprint")
proof
proof
Vale
Vale/F*
("assembly-like")
Low*
("C-like")
via Vale
printer
via KreMLin
compiler
Assembly (.asm)
C code (.c, .h)
Hacl.Impl.Gf128.Generic.fst
aesgcm-x86_64-linux.S
Hacl_Gf128_PreComp.c

What is verified?

What do we verify?

Safety

Memory- and type-safety. Mitigates buffer overruns, dangling pointers, code injections. No undefined behavior.

Functional correctness

Our fast implementations behave precisely as our simpler specifications.

Secrecy

Access to secrets, including crypto keys and private app data is restricted according to design.

Our specifications and implementations are written together, in one language (F*)
Drift between spec and implementation cannot happen.

Each application can do custom proofs beyond functional correctness and safety:

- non malleability (parsers)
- crypto games (TLS)
- security reduction (Merkle Trees)
- etc. etc.

COMPUTER SECURITY

Cryptography That Can't Be Hacked

20

Researchers have just released hacker-proof cryptographic code — programs with the same level of invincibility as a mathematical proof.

nope

/nōp/

exclamation INFORMAL

variant of no.

"“Have you seen it?” “Nope.”"

Schneier on Security

Blog | Newsletter | Books | Essays | News | Talks | Academic | About Me

Blog >

Unhackable Cryptography?

A recent article overhyped the release of EverCrypt, a cryptography library created using formal methods to prove security against specific attacks.

The *Quanta* magazine article sets off a series of "snake-oil" alarm bells. The author's Github README is more measured and accurate, and illustrates what a cool project this really is. But it's not "hacker-proof cryptographic code."

Tags: cryptography, encryption, hacking, snake oil

same sentiment on: Hacker News, Reddit, Slashdot, twitter, etc.

Will Everest be perfectly secure? No.

Our models make assumptions, e.g.

- The private signing key must remain private and not used in other protocols
- We assume security for core crypto algorithms, based on hard problems.

Our models may not be complete

- Our detailed models are designed to exclude all known attacks,
  but may be blind to new classes of attack (hardware faults,…)

Our verification toolchain may be buggy

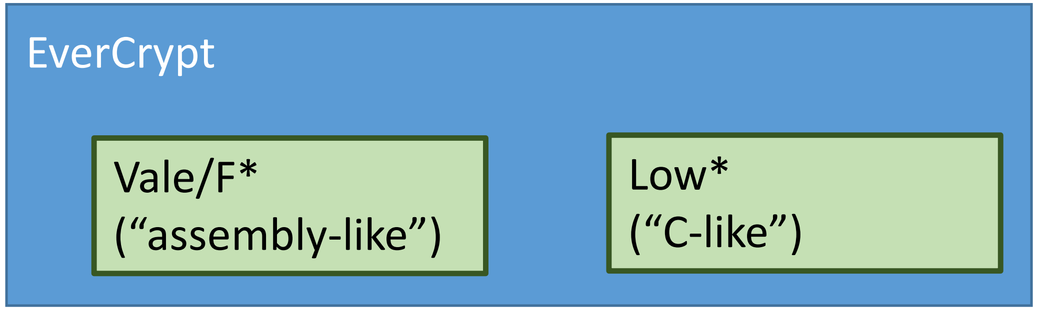
- Our TCB includes Z3, Kremlin, C compilers… Efforts to reduce it are under way.

Computer-aided verification also has advantages: once in place, proof verification is

- automated (but takes hours)
- compositional (we can re-use verified component as building blocks for others)
- maintainable (we can extend or modify our code, and re-check everything as part of CI).

The Essence of EverCrypt

EverCrypt: no excuses industrial-grade crypto library, with full verification



- A **single artifact** for clients to use
- **State-of-the-art** performance
- A **single verification** result (Vale or Low*)
- **Deep integration** for seamless interop
- **Total abstraction** for clients

Algorithm	C version	ASM version	Agile API
AES-GCM		✓ (AESNI)	✓
ChachaPoly	✓		✓
MD5, SHA1	✓		✓
SHA2	✓	✓ (SHAEXT)	✓
SHA3	✓		
Blake2	✓		
HMAC	✓		✓
HKDF	✓		✓
Curve25519	✓	✓ (BMI2 + ADX)	
Ed25519	✓		
Chacha20	✓		
AES 128, 256		✓	
AES-CTR		✓	
Poly1305	✓ (+ AVX + AVX2)	✓ (X64)	

One algorithm, several implementations *(multiplexing)*

- Verifies multiple implementations (Vale & Low*) against **one specification**
- **Isolates** clients from processor and target details
- **Auto-detects** static & dynamic features
- Eliminates **illegal instruction errors**
- Expected by an **industrial-grade library**

EverCrypt_Poly1305.c

```
*/
void EverCrypt_Poly1305_poly1305(uint8_t *dst, uint8_t *src, uint32_t len1, uint8_t *key)
{
  bool avx2 = EverCrypt_AutoConfig2_has_avx2();
  bool avx = EverCrypt_AutoConfig2_has_avx();
  bool vale1 = EverCrypt_AutoConfig2_wants_vale();
  #if EVERCRYPT_TARGETCONFIG_X64
  if (avx2)
  {
    Hacl_Poly1305_256_poly1305_mac(dst, len1, src, key);
    return;
  }
  #endif
  #if EVERCRYPT_TARGETCONFIG_X64
  if (avx)
  {
    Hacl_Poly1305_128_poly1305_mac(dst, len1, src, key);
    return;
  }
  #endif
  #if EVERCRYPT_TARGETCONFIG_X64
  if (vale1)
  {
    EverCrypt_Poly1305_poly1305_vale(dst, src, len1, key);
    return;
  }
  #endif
  Hacl_Poly1305_32_poly1305_mac(dst, len1, src, key);
}
```

1 - 19k EverCrypt_Poly1305.c unix | ??: 0 Bottom

Several algorithms, one API *(agility)*

- Verifies that multiple algorithms fit the same **family of specifications**
- Allows clients to **switch** between algorithms (crucial for TLS)
- Uses F* meta-programming to **templatize** the code
- Expected by an **industrial-grade library**

EverCrypt_Hash.c

```c
void EverCrypt_Hash_init(EverCrypt_Hash_state_s *s)
{
  EverCrypt_Hash_state_s scrut = *s;
  if (scrut.tag == EverCrypt_Hash_MD5_s)
  {
    uint32_t *p1 = scrut.case_MD5_s;
    Hacl_Hash_Core_MD5_init(p1);
  }
  else if (scrut.tag == EverCrypt_Hash_SHA1_s)
  {
    uint32_t *p1 = scrut.case_SHA1_s;
    Hacl_Hash_Core_SHA1_init(p1);
  }
  else if (scrut.tag == EverCrypt_Hash_SHA2_224_s)
  {
    uint32_t *p1 = scrut.case_SHA2_224_s;
    Hacl_Hash_Core_SHA2_init_224(p1);
  }
  else if (scrut.tag == EverCrypt_Hash_SHA2_256_s)
  {
    uint32_t *p1 = scrut.case_SHA2_256_s;
    Hacl_Hash_Core_SHA2_init_256(p1);
  }
  else if (scrut.tag == EverCrypt_Hash_SHA2_384_s)
  {
    uint64_t *p1 = scrut.case_SHA2_384_s;
    Hacl_Hash_Core_SHA2_init_384(p1);
  }
  else if (scrut.tag == EverCrypt_Hash_SHA2_512_s)
  {
    uint64_t *p1 = scrut.case_SHA2_512_s;
    Hacl_Hash_Core_SHA2_init_512(p1);
  }
  else
```

- 62k EverCrypt_Hash.c   C/*l   unix | 359: 0   39%

Deep integration between C and ASM
(speed)

Implementation	Radix	Language	CPU cy.
donna64 [2]	51	64-bit C	159634
fiat-crypto [31]	51	64-bit C	145248
amd64-64 [21]	51	Intel x86_64 asm	143302
sandy2x [22]	25.5	Intel AVX asm	135660
EverCrypt portable (*this paper*)	51	64-bit C	135636
openssl* [5]	64	Intel ADX asm	118604
Oliveira et al. [52]	64	Intel ADX asm	115122
EverCrypt targeted (*this paper*)	64	64-bit C + Intel ADX asm	113614

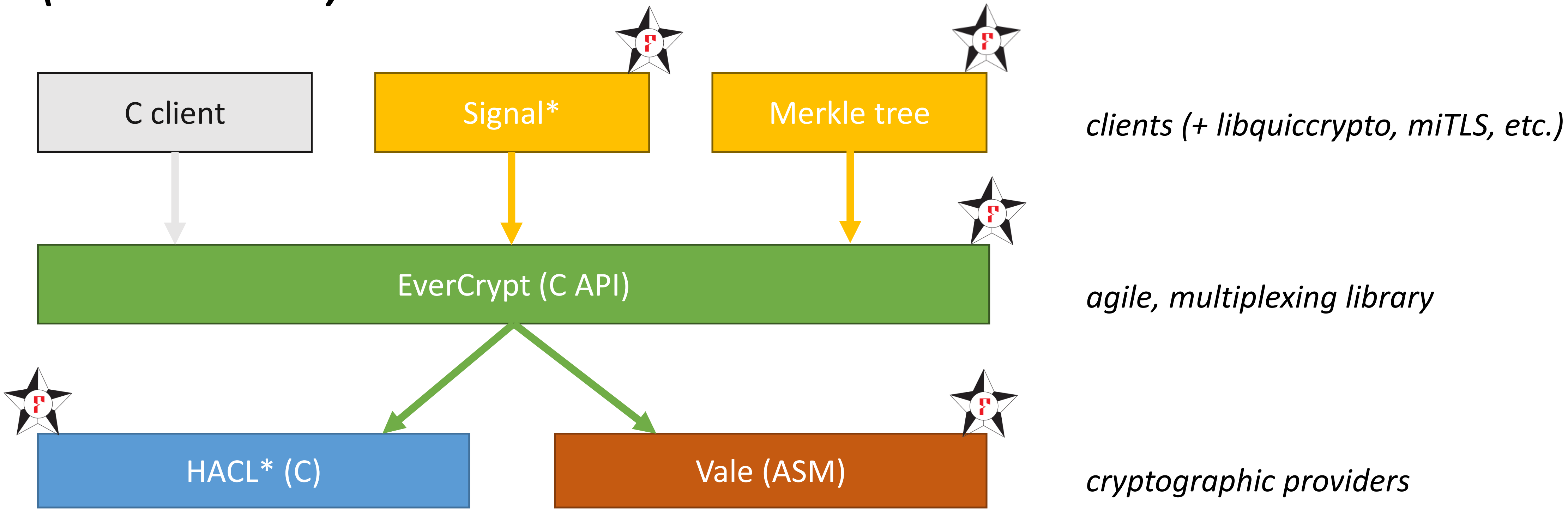
Figure 10. ***Performance comparison between Curve25519 Implementations.***

Verification allows **more optimizations** and does **not compromise** speed.

Mundane parts of the algorithms are written in Low* while critical bits are in Vale.

A new verified interop layer ensures **sound interoperation** between two languages.

A foundation for verified apps *(abstraction)*



EverCrypt **seals** the abstraction, meaning verified clients are **shielded from underlying verification details**.

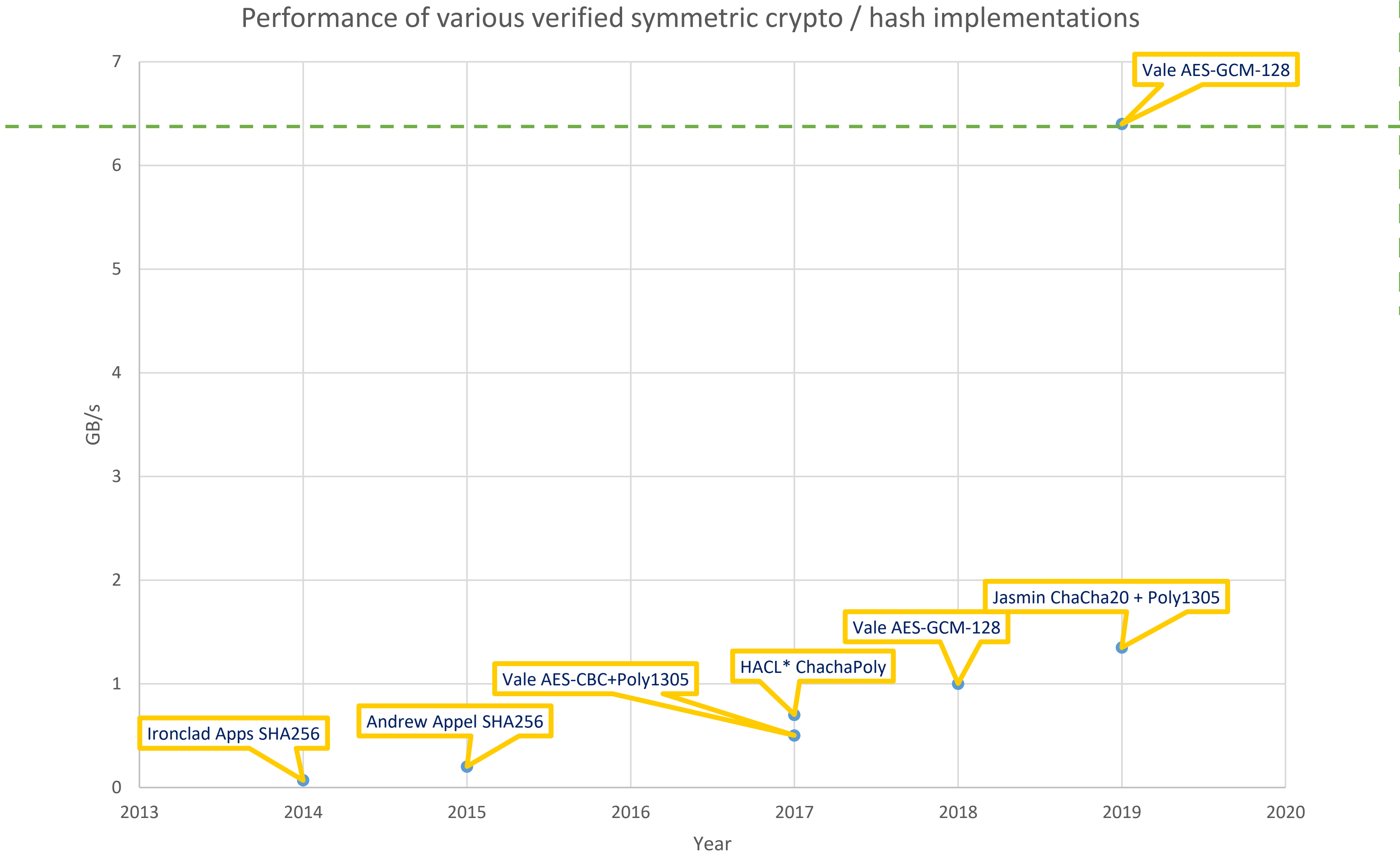
A significant verification effort

Components	LOC
All specifications	8009
Low* support libraries	6066
Low* algorithms	17637
Vale libraries and interop (F*)	37127
Vale algorithms (Vale)	25467
EverCrypt layer	4412
Merkle tree	6505
QUIC transport cryptography	2282
Total (hand-written F* and Vale)	107505
Vale algorithms (F* code generated from Vale files)	76685
Compiled code (.c files)	23400
Compiled code (.h files)	4236
Compiled code (ASM files)	18046

Figure 11. ***System Line Counts.***

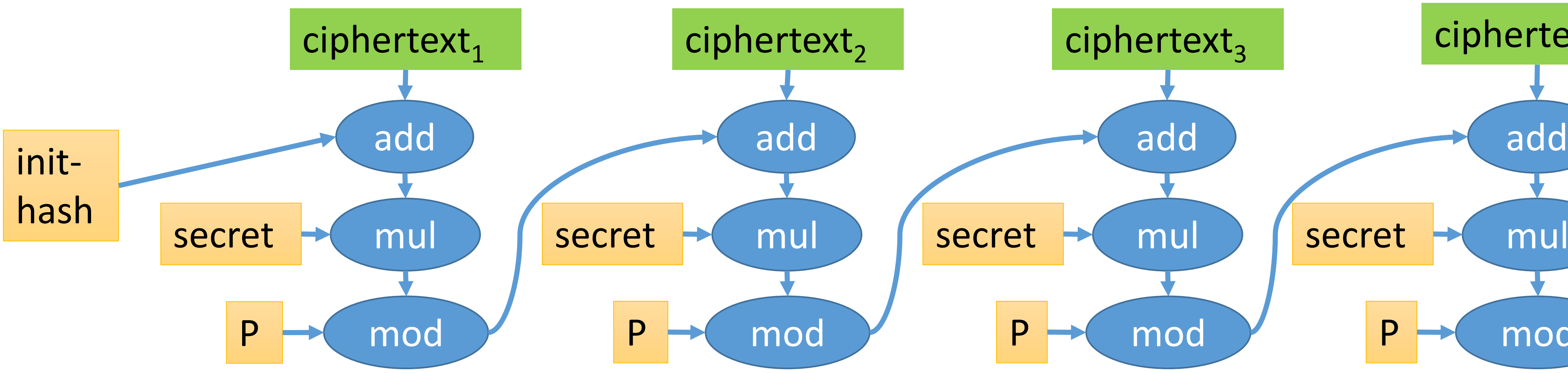
Verified Assembly Language in Vale / F*

We have a *fast* verified AES-GCM



Fastest OpenSSL assembly code

Optimizing AES-GCM



Important optimizations:

- delay mod operations
- parallelize add/mul operations
- math+bitwise tricks for mod
- careful instruction scheduling

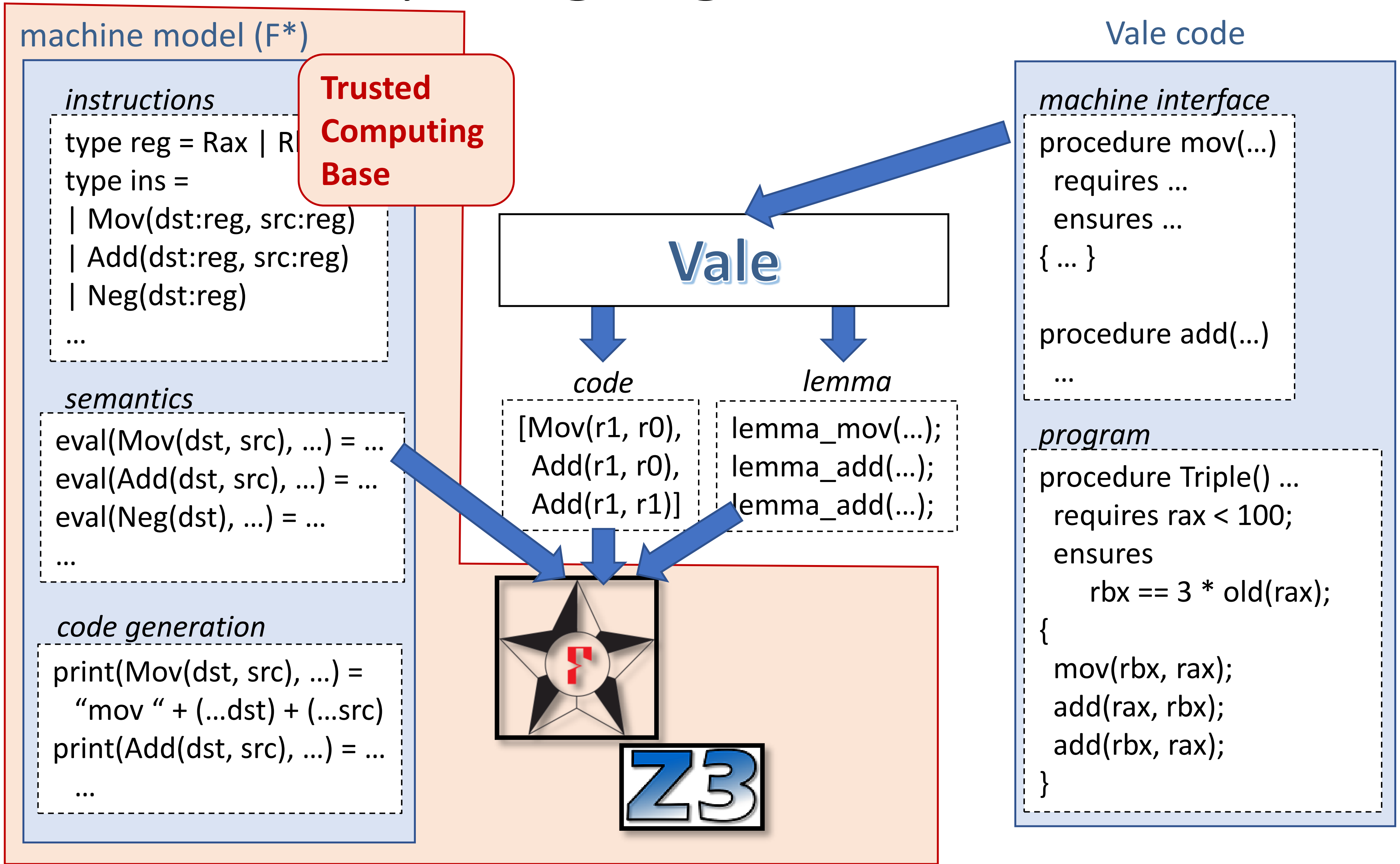
$(((init + c_1) * s \% P + c_2) * s \% P + c_3) * s \% P$

$\rightarrow (((init + c_1) * s + c_2) * s + c_3) * s \% P$

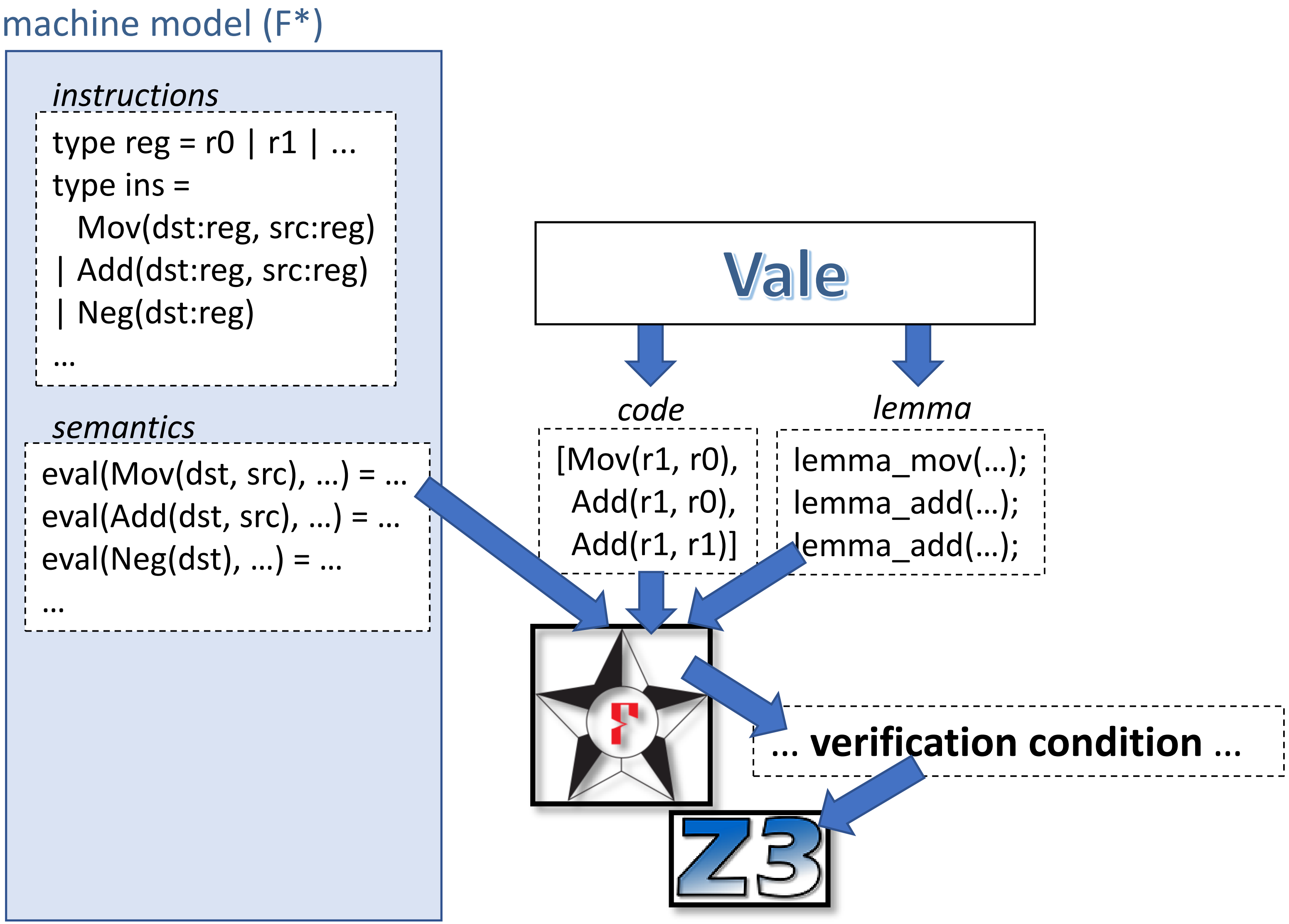
$\rightarrow ((init + c_1) * s^3 + c_2 * s^2 + c_3 * s^1) \% P$

$\rightarrow ((init + c_1) * (s^3 \% P) + c_2 * (s^2 \% P) + c_3 * s^1) \% P$

Vale: extensible, automated assembly language verification



Vale: extensible, automated assembly language verification



Verification condition

```
   procedure Triple()
      requires rax < 100;
      ensures
         rbx == 3 * rax;
   {
1     Move(rbx, rax); // --> rbx1
2     Add(rax, rbx);   // --> rax2
3     Add(rbx, rax);   // --> rbx3
   }
```

verification condition

$$
\begin{array}{l}
rax_0 < 100 \\
\vdash \\
(rbx_1 == rax_0 \implies \\
rax_0 + rbx_1 < 2^{64} \wedge (rax_2 == rax_0 + rbx_1 \implies \\
rbx_1 + rax_2 < 2^{64} \wedge (rbx_3 == rbx_1 + rax_2 \implies \\
rbx_3 == 3 * rax_0)))
\end{array}
$$

Ugh! Default SMT query looks awful!

verification condition we want:

……………………… $(rax_2 == rax_0 + rbx_1 ==>$

$rbx_1 + rax_2 < 2^{64}$ ………………………………….

verification condition we get:

```
…
(forall (ghost_result_0:(state * fuel)).
  (let (s3, fc3) = ghost_result_0 in
    eval_code (Ins (Add64 (OReg (Rax)) (OReg (Rbx)))) fc3 s2 == Some s3 /\
    eval_operand (OReg Rax) s3 == eval_operand (OReg Rax) s2 + eval_operand (OReg Rbx) s2 /\
    s3 == update_state (OReg Rax).r s3 s2) ==>
  lemma_Add s2 (OReg Rax) (OReg Rbx) == ghost_result_0 ==>
  (forall (s3:state) (fc3:fuel). lemma_Add s2 (OReg Rax) (OReg Rbx) == Mktuple2 s3 fc3 ==>
    Cons? codes_Triple.tl /\
    (forall (any_result0:list code). codes_Triple.tl == any_result0 ==>
      (forall (any_result1:list code). codes_Triple.tl.tl == any_result1 ==>
        OReg? (OReg Rbx) /\ eval_operand (OReg Rbx) s3 + eval_operand (OReg Rax) s3 < 2^64
…
```

Let's write our own VC generator!

- ??? Maybe like this: ???

procedure Triple() …
…

Our own Vale VC generator

verification condition we want:

………………………. $(rax_2 == rax_0 + rbx_1 ==>$
$rbx_1 + rax_2 < 2^{64}$ ………………………………….

I'm lonely and sad.

- But won't it be part of TCB?
- And how do we interact with F*?
- Can we reuse F* features and libraries?

Let's write our own VC generator!

- Like this!

procedure Triple() …
…

Our own Vale
VC generator,
written in F*,
run by F*'s interpreter during type checking

I'm happy.

verification condition we want:
………………………. ($rax_2 == rax_0 + rbx_1 ==>$
$rbx_1 + rax_2 < 2^{64}$ ………………………………….

- Part of TCB? ***No -- we verify its soundness in F****
- Interact with F*? ***Yes***
- Reuse F* features and libraries? ***Yes***

Let's write our own VC generator!

procedure Triple() ...
...

A b~~…~~ing?

A datatype:

```
type quickCode = ...
type quickCodes =
| QEmpty
| QSeq of quickCode * quickCodes ...
| QLemma of ... (Lemma pre post) * ...
```

Like our earlier code AST,
but with assertions, lemma calls,
ghost variables, etc.

Our own Vale
VC generator,
written in F*,
run by F*'s interpreter

verification condition we want:

.....................$(rax_2 == rax_0 + rbx_1 ==>$
$rbx_1 + rax_2 < 2^{64}$ ....

A b~~…~~ing?

A d~~…~~pe?

An F* term:

$(forall\ rbx_1.\ rbx_1 == rax_0 ==>$
$rax_0 + rbx_1 < 2^{64}$ /\
$(forall\ rax_2.\ rax_2 == rax_0 + rbx_1 ==>$
$rbx_1 + rax_2 < 2^{64}$ /\ ...

Demo

- Verification condition generation for Vale

Optimizing Curve25519

Low*

```
match s with
 | M51 -> F51.fmul1 out f1 f2
 | M64 -> F64.fmul1 out f1 f2
```

Interop

```
val fmul1 (dst:u256) (a:u256) (b:uint64{v f2 < pow2 17}) :
  Stack unit
    (requires fun h -> adx_enabled /\ bmi2_enabled /\ ...)
    (ensures ...)
```

Vale

```
procedure fmul1(...)...
    lets dst_ptr @= rdi; inA_ptr @= rsi; b @= rdx;
    requires adx_enabled && bmi2_enabled && ...
    ensures ...
{
    fast_mul1(0, inA_b); ... Mov64(b, 38);
    carry_pass(false, 0, dst_b);
}
```

Demo: Interop between Vale and Low*

Conclusions

- We've verified fast assembly language crypto implementations:
  - SHA
  - Poly1305
  - AES-GCM
  - Curve25519
- Expressive logics + SMT automation
  - We wrote our own domain-specific VC generator
    - We proved it sound
    - We run it from with F*'s type checker, and verification is fast
  - What other opportunities are there?

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

Deployments and applications

Level 1: cherry-pick approach

```
poly1305-simd is among the failing algorithms because it loses carry bits when
handling long "all 0xff bytes" inputs.  poly1305-avx2-x86_64.S is definitely
broken, and poly1305-sse2-x86_64.S *might* be too.  I am working on a patch...
```

Example: Linux Kernel (ZINC).
- Kernel already has multiplexing and CPU auto-detection facilities.

- Taking EverCrypt Curve25519 (C/ASM)
- Also took Fiat crypto
- They want algorithms we don't yet have

Also in that category: Firefox

The latter project takes the approach of modeling the algorithm in F and proving the model correct, which F* is designed to optimize.* ***Then — in a term of art which never fails to make me think of Arnold Schwarzenegger's Terminator descending into a bath of molten metal — the model is "lowered into" C (or in some cases, all the way into assembly language).*** *According to Donenfeld, this produces C which, though slightly non-idiomatic, is surprisingly readable, and much more likely to be bug-free than human-written code. It also produces some of the fastest C implementations that exist, which he suspects is because the formal verification process removes certain things that are not obviously removable when you're working the mathematics out by hand.*

Level 2: the whole library

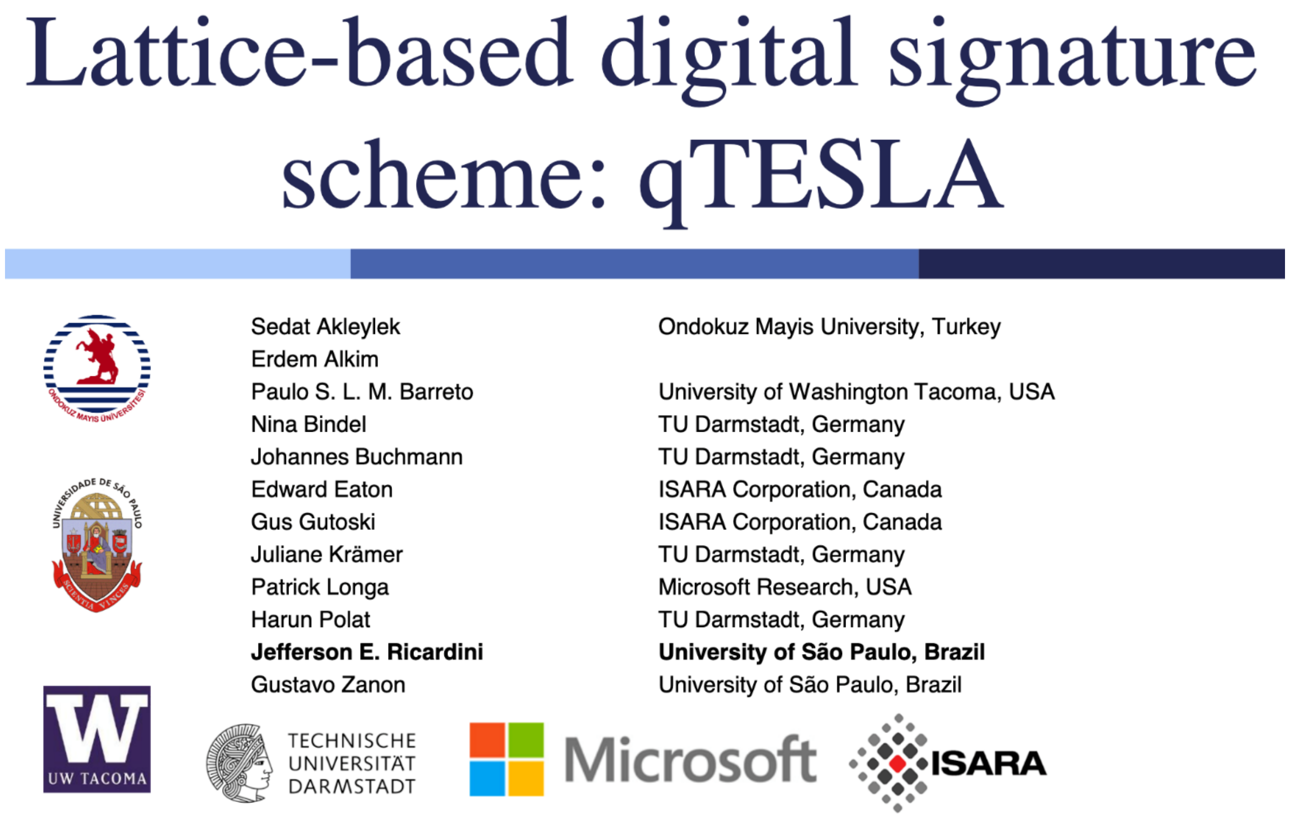
- Easiest approach: just take the whole directory
- Expectations are higher for security-related applications
- Beneficial peer pressure

<u>Examples:</u> Concordium & Tezos blockchains, remote attestation (UC Irvine)

> network-based attacks from compromising CIDER. We shield the remaining core CIDER code from the adversary through isolation in time and by checking the integrity of all inputs using the formally verified High-Assurance Cryptographic Library (HACL) [34].

Level 3: extend

- Formal verification an advantage for standards competitions (NIST)
- Post-quantum algorithms:
  qTESLA, Frodo



Lattice-based digital signature scheme: qTESLA

Sedat Akleylek	Ondokuz Mayis University, Turkey
Erdem Alkim	
Paulo S. L. M. Barreto	University of Washington Tacoma, USA
Nina Bindel	TU Darmstadt, Germany
Johannes Buchmann	TU Darmstadt, Germany
Edward Eaton	ISARA Corporation, Canada
Gus Gutoski	ISARA Corporation, Canada
Juliane Krämer	TU Darmstadt, Germany
Patrick Longa	Microsoft Research, USA
Harun Polat	TU Darmstadt, Germany
Jefferson E. Ricardini	**University of São Paulo, Brazil**
Gustavo Zanon	University of São Paulo, Brazil

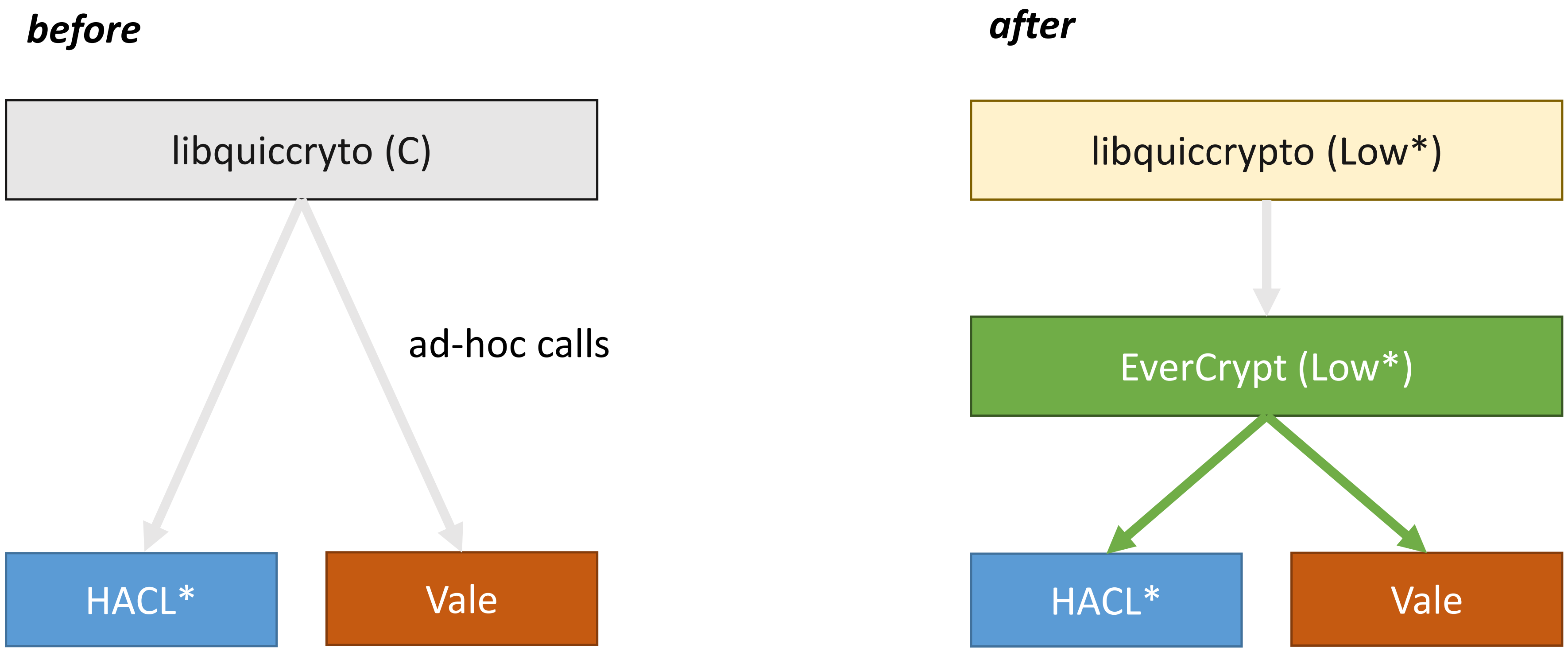
EverCrypt as a **foundation** for verified software

- EverCrypt = a building block
- Why just limit ourselves to TLS?
- **Several artifacts have been developed on top of EverCrypt**

shields clients from conflicting, disparate specifications in favor of crisp, unified cryptographic constructions

A custom provider: libquiccrypto

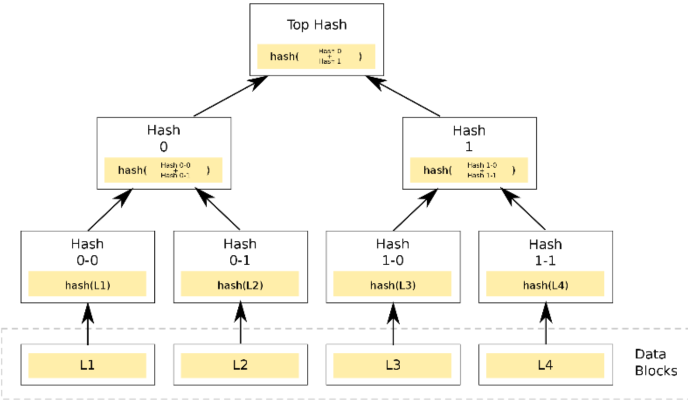
“The cryptographic toolbox one needs to implement QUIC”.



- ☑ memory safety
- ☑ functional correctness
- ☑ cryptographic model

A complete component: Merkle tree

- Used to verify integrity of a large number of blocks
- Needs a hash algorithm
- Needs the fastest hash for the give platform
- Proof of collision resistance by reduction



By Azaghal - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=18157888

CCF uses EverCrypt

(Build 2019)

PASSED | Azure Pipelines succeeded | codecov 81% | docs microsoft.github.io/CCF

The Confidential Consortium Framework

Microsoft Azure

GitHub, Inc. (US) | https://github.com/microsoft/CCF

A full-fledged protocol: Signal*

- Secure communications protocol
- Used by: WhatsApp, Facebook Messenger, Signal, Skype
- Sophisticated cryptography: X3DH, double-ratched
- Forward secrecy, post-compromise security, etc. etc.

A verified implementation compiled to C and ...

A whole new target for EverCrypt: WASM

- Shipped in **all major browsers** (including Edge)
- WASM delivers **portability** and **performance**
- **LLVM** backend (“emscripten”)

Opportunity:
- Desktop applications are running on a **web framework** like Electron (e.g. Skype, Signal, VS Code, Atom, WhatsApp)
- Framework support for cryptography is **lacking** (WebCrypto on the web, node.js crypto on the desktop)

A WASM backend for KreMLin:
- **Auditable** and delivers competitive performance
- An alternative, faster, **less trustworthy** backend: Low* -> C (via KreMLin) -> WASM (via LLVM)
- **EverCrypt for the web:** enables **instant access** to the latest cryptographic primitives on both Desktop & Web

Applications already:
- Use the WASM backend of KreMLin for verified, fast implementation of **messaging protocols, including Signal (IEEE S&P 2019)**

A vision for EverCrypt

- An industrial-grade crypto provider is **now a reality**
  - already adopted
  - demonstrates OpenSSL's libcrypto is no longer inevitable
- **Peer pressure** to use verified code (good)
  - blockchains pushing for formal verification
  - skepticism of crypto is high (backdoors? magic constants? Russian S-BOX?)
  - open-source more nimble (Linux, BoringSSL, Firefox)
- EverCrypt is at the **forefront**
  - breadth and scale of the verification effort
  - With other folks in the same space: MIT, Galois, Amazon
- Prediction: at the five-year horizon, unverified crypto will be a **liability**