Cryptographic Primitives Implemented Efficiently, Correctly and Securely

Nik Swamy, OPLSS 2021

Thanks to Jonathan Protzenko and Chris Hawblitzel for these slides. Errors are mine

Crypto code is hard to get right

But is critical for system security

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

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

AES-GCM

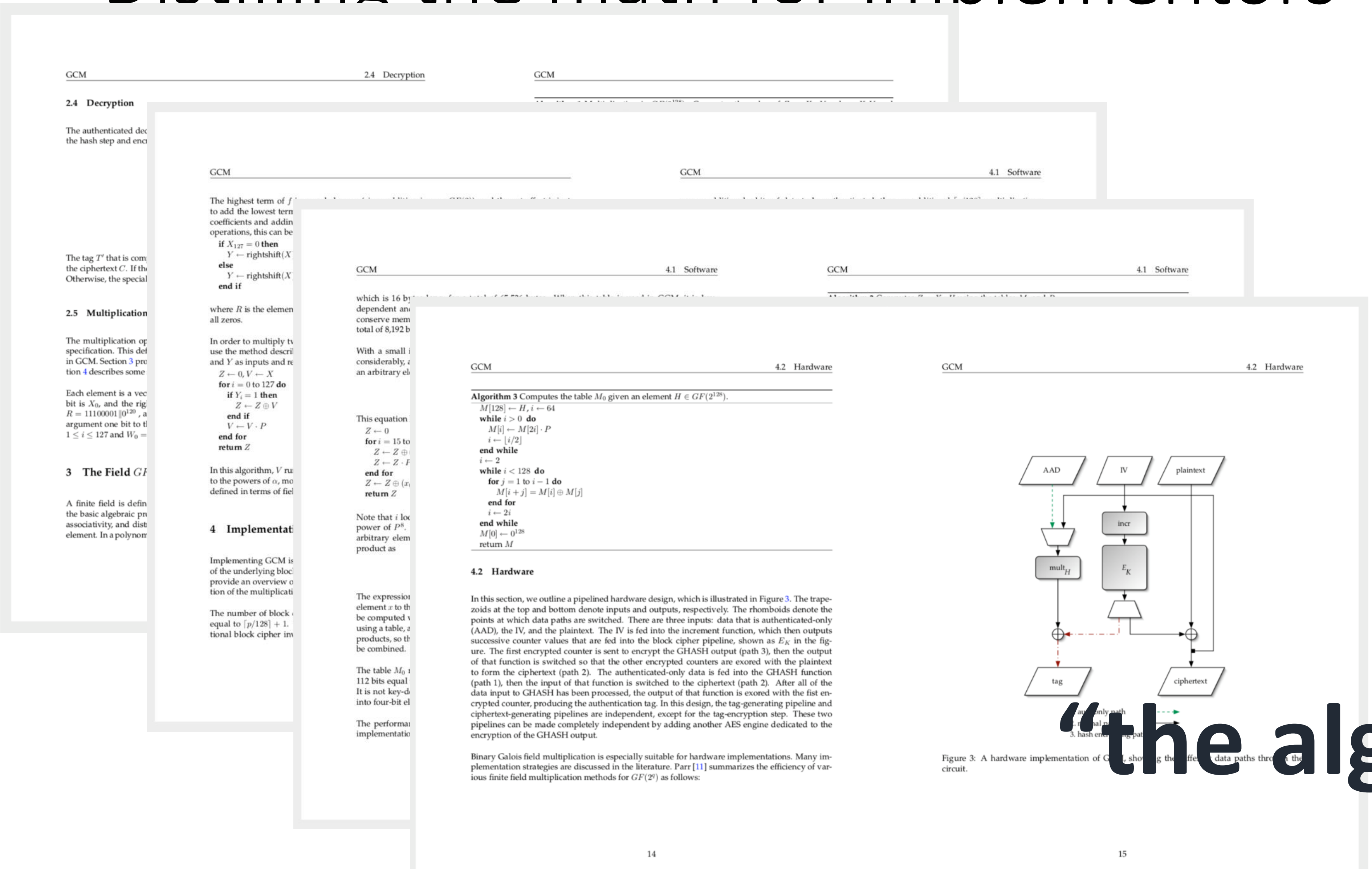
Evaluate polynomials in this field to get an **authentication code!** (see also: Poly1305)

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] ← H, i ← 64
while i > 0 do
  M[i] ← M[2i] · P
  i ← ⌊i/2⌋
end while
i ← 2
while i < 128 do
  for j = 1 to i − 1 do
    M[i + j] = M[i] ⊕ M[j]
  end for
  i ← 2i
end while
M[0] ← 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 the…ffe… data paths thro…h the circuit.

15

“the algorithm”

Writing the actual code

A long way from the math

“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)
```

$$GF(2^{128}) = GF(2)[X]/(x^{128} + x^7 + x^2 + x + 1)$$

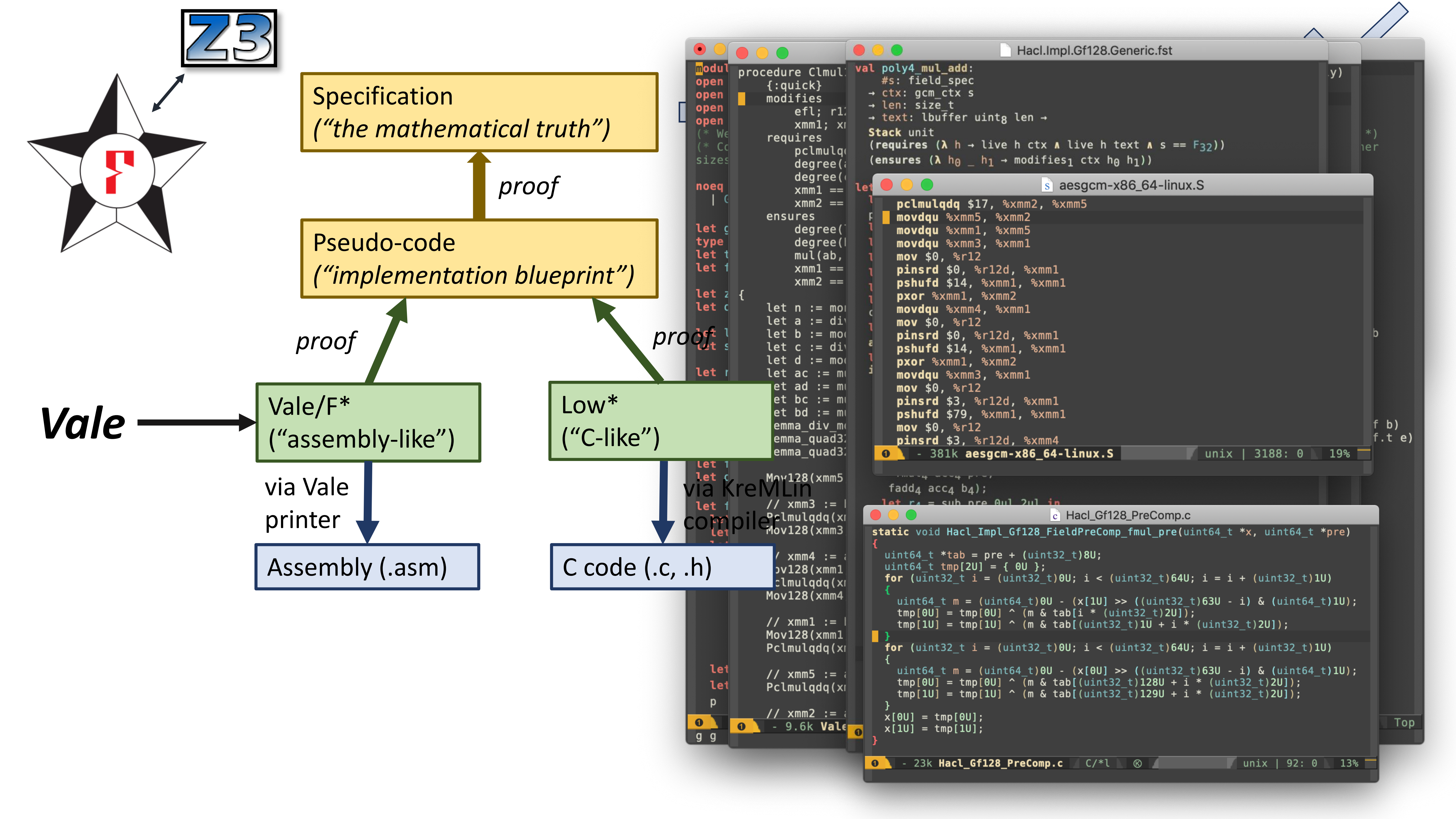
refines

Algorithm 1 Multiplication in $GF(2^{128})$. $Z \in GF(2^{128})$.

> $Z \leftarrow 0, V \leftarrow X$
> **for** $i = 0$ to 127 **do**
>   **if** $Y_i = 1$ **then**
> $Z \leftarrow Z \oplus V$
>   **end if**
>   **if** $V_{127} = 0$ **then**
> $V \leftarrow \text{rightshift}(V)$
>   **else**
> $V \leftarrow \text{rightshift}(V) \oplus R$
>   **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
val poly4_mul_add:
#s: field_spec
→ ctx: gcm_ctx s
→ len: size_t
→ text: lbuffer uint8 len →
Stack unit
(requires (λ h → live h ctx ∧ live h text ∧ s == F32))
(ensures (λ h0 _ h1 → modifies1 ctx h0 h1))
procedure Clmul
{:quick}
modifies
requires
ensures
aesgcm-x86_64-linux.S
pclmulqdq $17, %xmm2, %xmm5
movdqu %xmm5, %xmm2
movdqu %xmm1, %xmm5
movdqu %xmm3, %xmm1
mov $0, %r12
pinsrd $0, %r12d, %xmm1
pshufd $14, %xmm1, %xmm1
pxor %xmm1, %xmm2
movdqu %xmm4, %xmm1
mov $0, %r12
pinsrd $0, %r12d, %xmm1
pshufd $14, %xmm1, %xmm1
pxor %xmm1, %xmm2
movdqu %xmm3, %xmm1
mov $0, %r12
pinsrd $3, %r12d, %xmm1
pshufd $79, %xmm1, %xmm1
mov $0, %r12
pinsrd $3, %r12d, %xmm4
- 381k aesgcm-x86_64-linux.S unix | 3188: 0 19%
Hacl_Gf128_PreComp.c
static void Hacl_Impl_Gf128_FieldPreComp_fmul_pre(uint64_t *x, uint64_t *pre)
{
uint64_t *tab = pre + (uint32_t)8U;
uint64_t tmp[2U] = { 0U };
for (uint32_t i = (uint32_t)0U; i < (uint32_t)64U; i = i + (uint32_t)1U)
{
uint64_t m = (uint64_t)0U - (x[1U] >> ((uint32_t)63U - i) & (uint64_t)1U);
tmp[0U] = tmp[0U] ^ (m & tab[i * (uint32_t)2U]);
tmp[1U] = tmp[1U] ^ (m & tab[(uint32_t)1U + i * (uint32_t)2U]);
}
for (uint32_t i = (uint32_t)0U; i < (uint32_t)64U; i = i + (uint32_t)1U)
{
uint64_t m = (uint64_t)0U - (x[0U] >> ((uint32_t)63U - i) & (uint64_t)1U);
tmp[0U] = tmp[0U] ^ (m & tab[(uint32_t)128U + i * (uint32_t)2U]);
tmp[1U] = tmp[1U] ^ (m & tab[(uint32_t)129U + i * (uint32_t)2U]);
}
x[0U] = tmp[0U];
x[1U] = tmp[1U];
}
- 23k Hacl_Gf128_PreComp.c unix | 92: 0 13%

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.

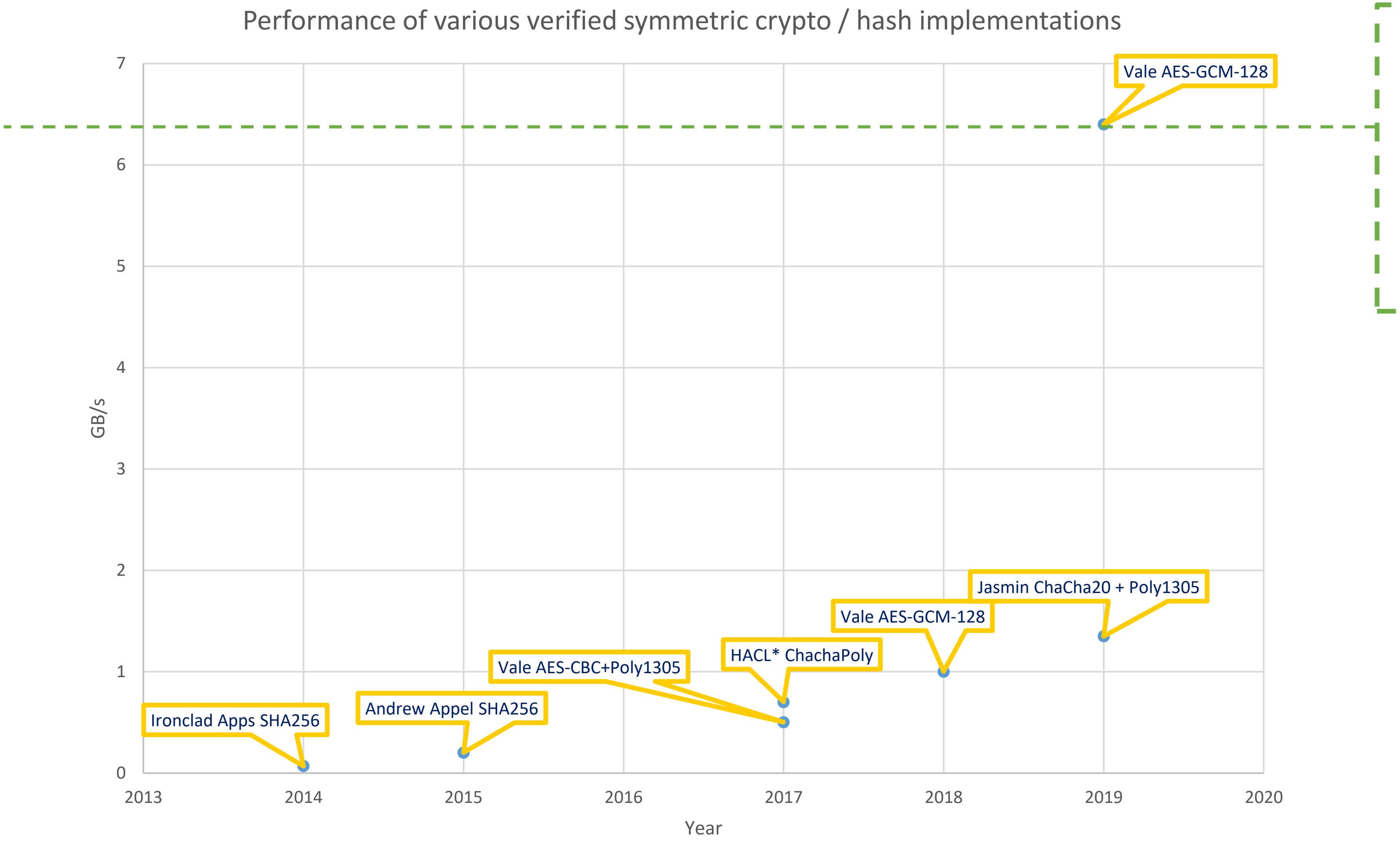
Our specifications and implementations are written together, in one language (F*)
Drift between spec and implementation cannot happen.

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

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

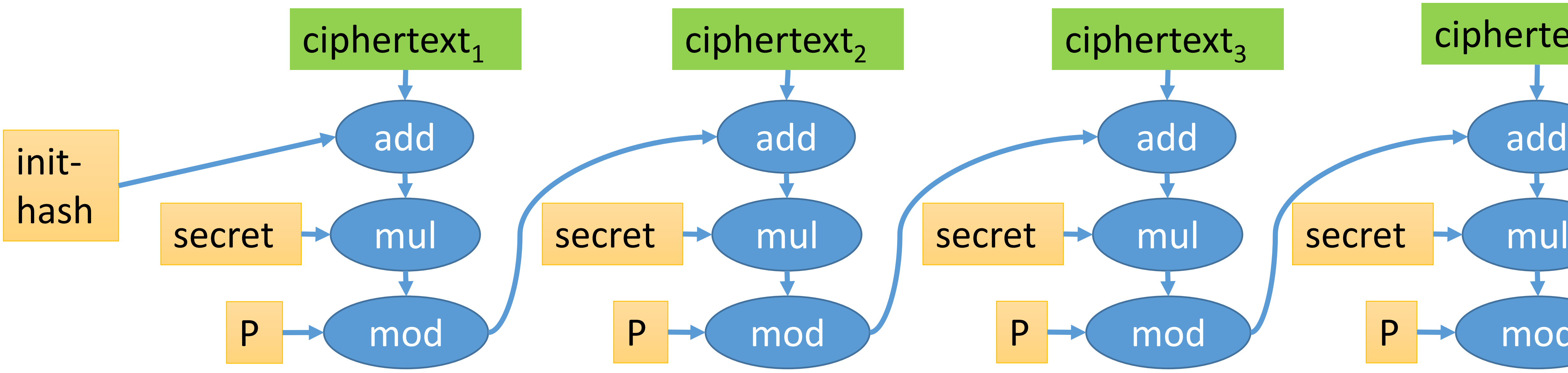
Verified Assembly Language in Vale / F*

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

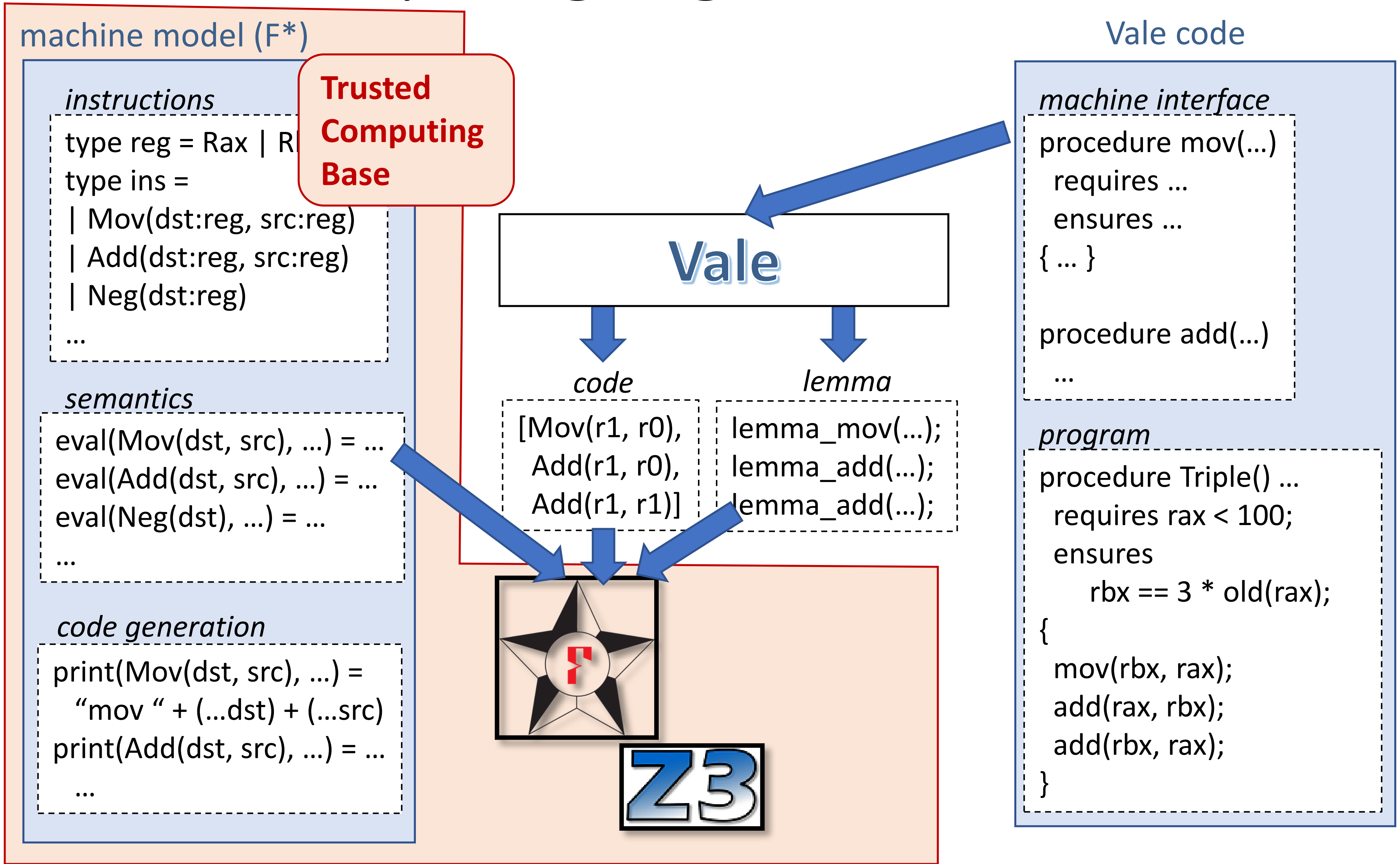
$(((\text{init} + c_1) * s \% P + c_2) * s \% P + c_3) * s \% P$

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

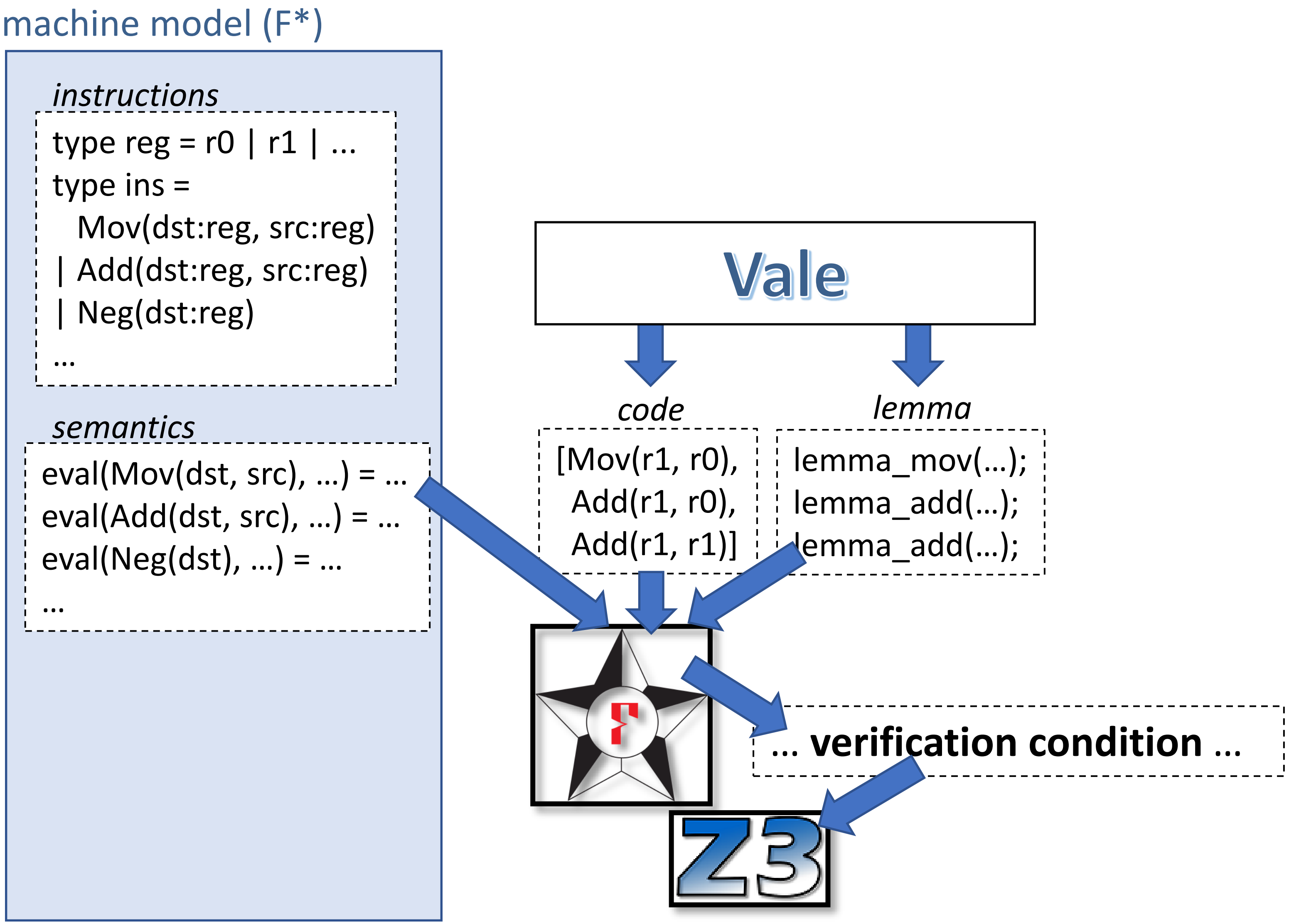
$\rightarrow ((\text{init} + c_1) * s^3 + c_2 * s^2 + c_3 * s^1) \% P$

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

Vale: extensible, automated assembly language verification



Vale: extensible, automated assembly language verification



Verification condition

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

verification condition

$rax_0 < 100$

$\vdash$

$(rbx_1 == rax_0 \implies$

$rax_0 + rbx_1 < 2^{64} \wedge (rax_2 == rax_0 + rbx_1 \implies$

$rbx_1 + rax_2 < 2^{64} \wedge (rbx_3 == rbx_1 + rax_2 \implies$

$rbx_3 == 3 * rax_0)))$

Demo

- Verification condition generation for Vale

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

I'm lonely and sad.

verification condition we want:

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

- 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:

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