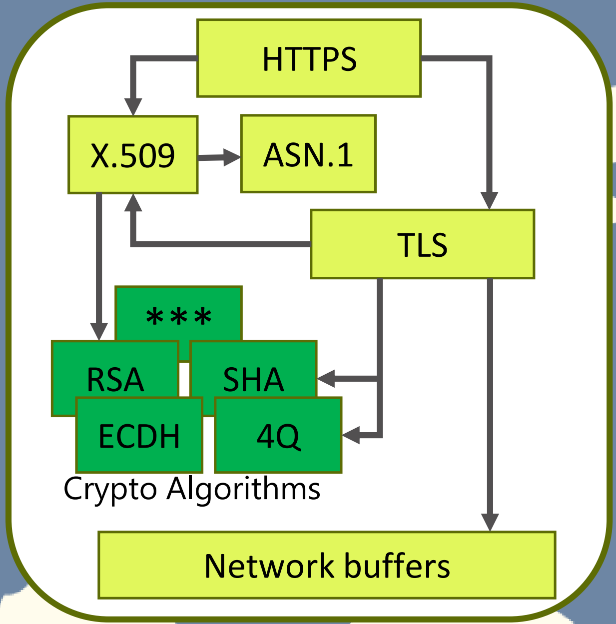
Verified High-Assurance Crypto Libraries



Sample crypto algorithm in OpenSSL

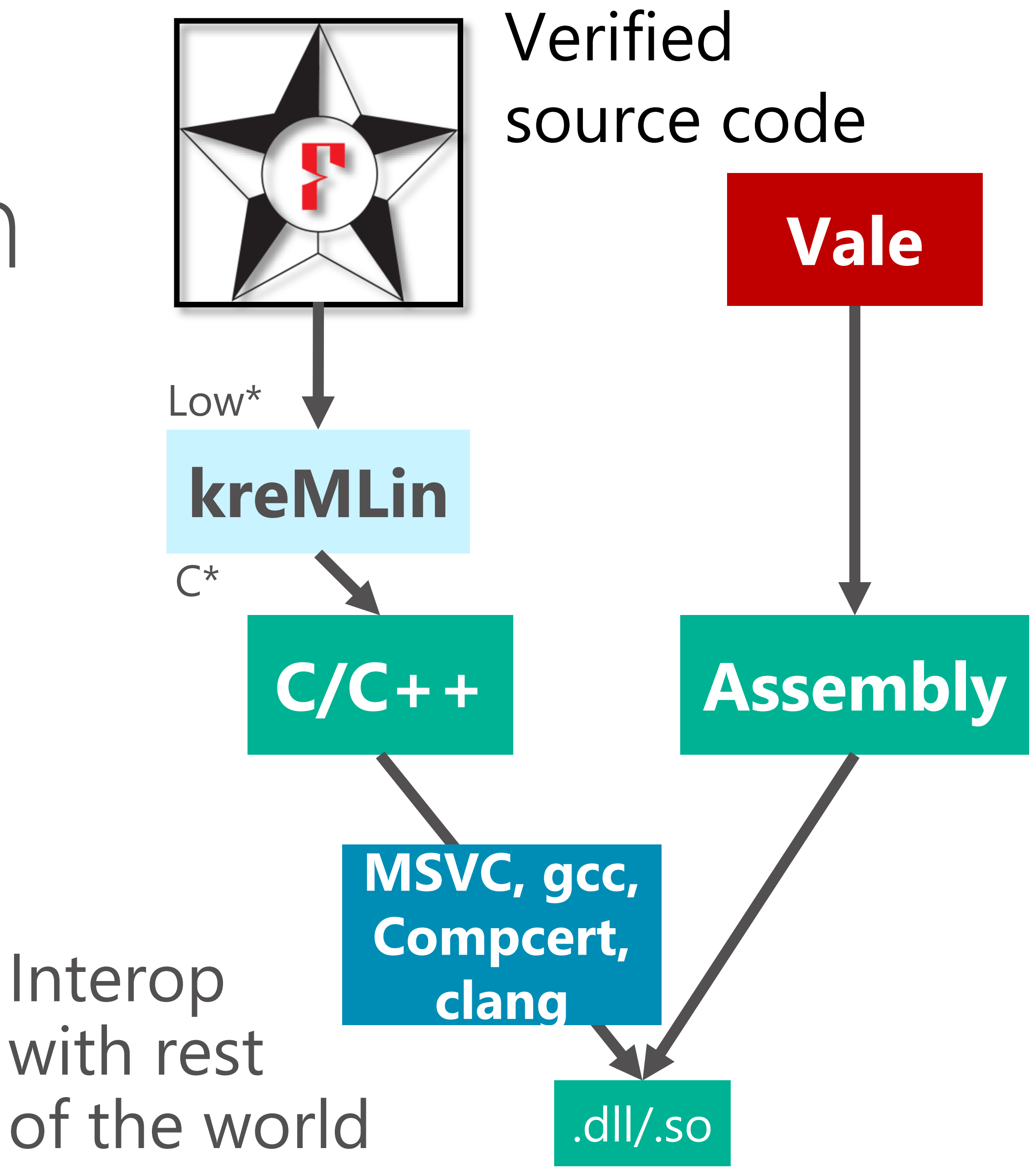
- Hand-crafted mix of Perl and assembly
- Customized for 50+ hardware platforms
- Why?

  Performance!
  several bytes/cycle

```
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
```

Crypto verification & compilation Toolchain

1. Compile restricted subset of verified source code to **efficient C/C++** ; or
2. Use a DSL for **portable verified assembly code**



Sample crypto algorithm: poly1305

$$MAC(k, m, \vec{w}) = m + \sum_{i=1..|\vec{w}|} w_i * k^i$$

Authenticate data by

1. Encoding it as a polynomial in the prime field $2^{130} - 5$
2. Evaluating it at a random point: the first part of the key k
3. Masking the result using the second part of the key m

Sample crypto algorithm: poly1305

$$MAC(k, m, \vec{w}) = m + \sum_{i=1..|\vec{w}|} w_i * k^i$$

Security?

If the sender and the receiver disagree on the data $\vec{w}$ then the difference of their polynomials is not null.

Its evaluation at a random k is 0 with probability $\approx \dfrac{|\vec{w}|}{2^{130}}$

Sample crypto algorithm: poly1305

$$MAC(k, m, \vec{w}) = m + \sum_{i=1..|\vec{w}|} w_i * k^i$$

A typical 64-bit arithmetic implementation:

1. Represent elements of the prime field for $\mathbf{p} = 2^{130} - 5$ using **3 limbs** holding 42 + 44 + 44 bits in 64-bit registers
2. Use $\left(a.2^{130} + b\right) \% \ p = (a + 4a + b) \% \, p$ for reductions
3. Unfold loop

Specifying, programming & verifying poly1305

Sample F* code:
the **spec** for the
multiplicative MAC
used in TLS 1.3

Its verified optimized
implementation for x64
takes 3K+ LOCs

Spec.Poly1305.fst

File Edit Options Buffers Tools Help

```
module Spec.Poly1305

(* Mathematical specification of multiplicative
   hash in the prime field 2^130 - 5  *)

let prime = 2^130 - 5

type elem = e:ℕ{e < prime}

let a +@ b = (a + b) % prime
let a  *@ b = (a × b) % prime

let encode (word:bytes {length w ≤ 8}): elem =
  2^(8 × length word) +@ little_endian word

let rec poly (text: seq bytes) (r: elem): elem =
  if Seq.length text = 0 then 0
  else encode (Seq.head text)) +@ poly (Seq.tail text) r) *@ r
```

-**- Spec.Poly1305.fst   Top L19   Git-dev (F✪ +3)

Why verify poly1305?

- Bugs happen: 3 fresh ones just in OpenSSL's poly1305.

> "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."

```
Poly1305 functions of openssl.

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

```
Hi folks,

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

$ OPENSSL_ia32cap=0 ./poly1305_test2
PASS
```

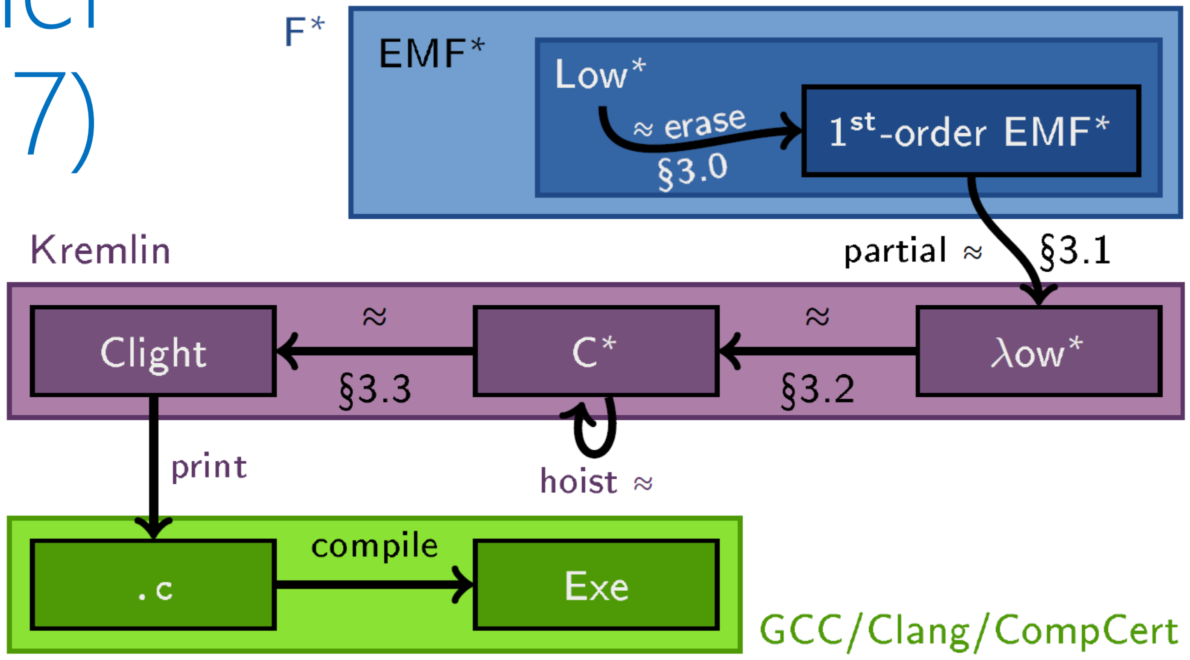
Low*: a subset of F* for safe C-style programming

Supports compilation to C, in nearly 1-1 correspondence,
for auditability of our generated code

Features a C-like view of memory (pointer arithmetic with verified safety)

KreMLin: a new compiler from Low* to C (ICFP'17)

- Semantics preserving from Low* to CompCert Clight
- Also: does not introduce memory-based side channels
- Then compile C using mainstream compilers
- Or, CompCert



Low*: low-level programming in F*

We must get to Low* **after** typing, erasure, and much inlining

- Compile-time error otherwise
- Goal: zero implicit heap allocations
- Non-goal: bootstrapping and high-level modelling (we have F*/OCaml for that)

Machine arithmetic

- Static checks for overflows
- Explicit coercions

Not the usual ML memory

Infix pointer arithmetic (erased lengths)

Static tracking of

- Liveness & index ranges
- Stack allocation
- Manual allocation
- Regions

No F* hack! Just libraries.

KreMLin: from F* to Low* to C* to C

- Why C/C++ ???

  Performance, portability
  Predictability (GC vs side channels)
  Interop (mix'n match)

  Readability, transparency (code review)

  Adoption, maintenance

- Formal translations
- Various backends

  Clang/LLVM; gcc

  Compcert, with verified translation from C* to Clight

- What KreMLin does

  Monomorphization of dependent types
  Data types to flat tagged unions
  Compilation of pattern matching
  From expressions to statements (hoisting)
  Name-disambiguation (C's block-scoping)
  Inlining (in-scope closures, stackInline)

- Early results for **HACL***: high assurance crypto library

  15 KLOCs of type-safe, partially-verified elliptic curves, symmetric encryption…

  Up to 150x speedup/ocamlopt

  Down by 50% vs C/C++ libraries

Hacl.Impl.Poly1305_64.fst

File Edit Options Buffers Tools F✪ Help

```
[@"substitute"]
val poly1305_last_pass_:
  acc:felem →
  Stack unit
    (requires (λ h → live h acc ∧ bounds (as_seq h acc) p44 p44 p42))
    (ensures (λ h0 _ h1 → live h0 acc ∧ bounds (as_seq h0 acc) p44 p44 p42
      ∧ live h1 acc ∧ bounds (as_seq h1 acc) p44 p44 p42
      ∧ modifies_1 acc h0 h1
      ∧ as_seq h1 acc == Hacl.Spec.Poly1305_64.poly1305_last_pass_spec_ (as_seq h0 acc)))

[@"substitute"]
let poly1305_last_pass_ acc =
  let a0 = acc.(0ul) in
  let a1 = acc.(1ul) in
  let a2 = acc.(2ul) in
  let open Hacl.Bignum.Limb in
  let mask0 = gte_mask a0 Hacl.Spec.Poly1305_64.p44m5 in
  let mask1 = eq_mask a1 Hacl.Spec.Poly1305_64.p44m1 in
  let mask2 = eq_mask a2 Hacl.Spec.Poly1305_64.p42m1 in
  let mask  = mask0 &^ mask1 &^ mask2 in
  UInt.logand_lemma_1 (v mask0); UInt.logand_lemma_1 (v mask1); UInt.logand_lemma_1 (v mask2);
  UInt.logand_lemma_2 (v mask0); UInt.logand_lemma_2 (v mask1); UInt.logand_lemma_2 (v mask2);
  UInt.logand_associative (v mask0) (v mask1) (v mask2);
  cut (v mask = UInt.ones 64 ⟹ (v a0 ≥ pow2 44 - 5 ∧ v a1 = pow2 44 - 1 ∧ v a2 = pow2 42 - 1));
  UInt.logand_lemma_1 (v Hacl.Spec.Poly1305_64.p44m5); UInt.logand_lemma_1 (v Hacl.Spec.Poly1305_64.p44m1);
  UInt.logand_lemma_1 (v Hacl.Spec.Poly1305_64.p42m1); UInt.logand_lemma_2 (v Hacl.Spec.Poly1305_64.p44m5);
  UInt.logand_lemma_2 (v Hacl.Spec.Poly1305_64.p44m1); UInt.logand_lemma_2 (v Hacl.Spec.Poly1305_64.p42m1);
  let a0' = a0 -^ (Hacl.Spec.Poly1305_64.p44m5 &^ mask) in
  let a1' = a1 -^ (Hacl.Spec.Poly1305_64.p44m1 &^ mask) in
  let a2' = a2 -^ (Hacl.Spec.Poly1305_64.p42m1 &^ mask) in
  upd_3 acc a0' a1' a2'
```

-:**- **Hacl.Impl.Poly1305_64.fst** 55% L394 Git-master (F✪ FlyC- company ElDoc Wrap)

Poly1305_64.c

File Edit Options Buffers Tools C Help

```c
static void Hacl_Impl_Poly1305_64_poly1305_last_pass(uint64_t *acc)
{
  Hacl_Bignum_Fproduct_carry_limb_(acc);
  Hacl_Bignum_Modulo_carry_top(acc);
  uint64_t a0 = acc[0];
  uint64_t a10 = acc[1];
  uint64_t a20 = acc[2];
  uint64_t a0_ = a0 & (uint64_t )0xfffffffffff;
  uint64_t r0 = a0 >> (uint32_t )44;
  uint64_t a1_ = (a10 + r0) & (uint64_t )0xfffffffffff;
  uint64_t r1 = (a10 + r0) >> (uint32_t )44;
  uint64_t a2_ = a20 + r1;
  acc[0] = a0_;
  acc[1] = a1_;
  acc[2] = a2_;
  Hacl_Bignum_Modulo_carry_top(acc);
  uint64_t i0 = acc[0];
  uint64_t i1 = acc[1];
  uint64_t i0_ = i0 & (((uint64_t )1 << (uint32_t )44) - (uint64_t )1);
  uint64_t i1_ = i1 + (i0 >> (uint32_t )44);
  acc[0] = i0_;
  acc[1] = i1_;
  uint64_t a00 = acc[0];
  uint64_t a1 = acc[1];
  uint64_t a2 = acc[2];
  uint64_t mask0 = FStar_UInt64_gte_mask(a00, (uint64_t )0xffffffffffb);
  uint64_t mask1 = FStar_UInt64_eq_mask(a1, (uint64_t )0xfffffffffff);
  uint64_t mask2 = FStar_UInt64_eq_mask(a2, (uint64_t )0x3ffffffffff);
  uint64_t mask = mask0 & mask1 & mask2;
  uint64_t a0_0 = a00 - ((uint64_t )0xffffffffffb & mask);
  uint64_t a1_0 = a1 - ((uint64_t )0xfffffffffff & mask);
  uint64_t a2_0 = a2 - ((uint64_t )0x3ffffffffff & mask);
  acc[0] = a0_0;
  acc[1] = a1_0;
  acc[2] = a2_0;
}
```

-:**- **Poly1305_64.c** 49% L272 Git-master (C/l company A

Hacl* Examples

F* spec, F* code, C code for

- Chacha20
- Poly1305

Performance for verified C code compiled from F*

As fast as best hand-written portable C implementations

Algorithm	HACL*	OpenSSL
ChaCha20	6.17 cy/B	8.04 cy/B
Poly1305	2.07 cy/B	2.16 cy/B
Curve25519	157k cy/mul	359k cy/mul

Still slower than best hand-written assembly language implementations

Mozilla Security Blog

SEP 13 2017

Verified cryptography for Firefox 57


Benjamin Beurdouche

Traditionally, software is produced in this way: write some code, maybe do some code review, run unit-tests, and then hope it is correct. Hard experience shows that it is very hard for programmers to write bug-free software. These bugs are sometimes caught in manual testing, but many bugs still are exposed to users, and then must be fixed in patches or subsequent versions. This works for most software, but it's not a great way to write cryptographic software;

Benjamin Beurdouche

Mozillian INRIA Paris - Prosecco team

More from Benjamin Beurdouche »

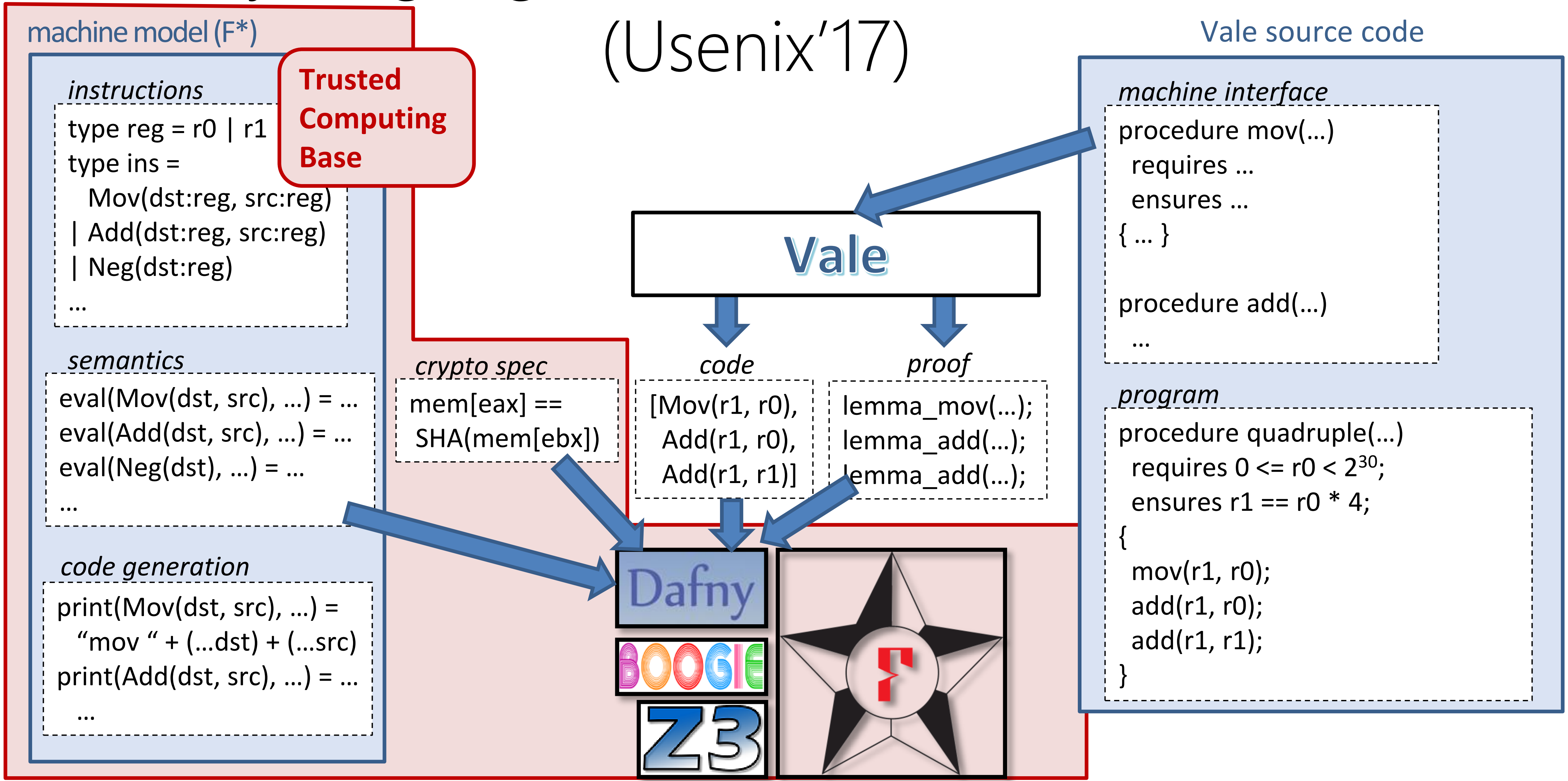
Categories

Announcements

Automated Testing

Vale: extensible, automated assembly language verification (Usenix'17)

functional correctness & side-channel protection



Vale Poly1305

OpenSSL Poly1305

raw.githubusercontent.c

raw.githubusercontent.com/openssl/openssl/mast

```
and     $d3,%rax
mov     $d3,$h2
shr     \$2,$d3
and     \$3,$h2
add     $d3,%rax
add     %rax,$h0
adc     \$0,$h1
adc     \$0,$h2
```

Bug! This carry was originally missing!

```
procedure poly1305_reduce()
    ...
{
    ...
    And64(rax, d3);
    Mov64(h2, d3);
    Shr64(d3, 2);
    And64(h2, 3);
    Add64Wrap(rax, d3);
    Add64Wrap(h0, rax);
    Adc64Wrap(h1, 0);
    Adc64Wrap(h2, 0);
    ...
}
```

Vale Poly1305

```
procedure poly1305_reduce() returns(ghost hOut:int)
    let
        n := 0x1_0000_0000_0000_0000;
        p := 4 * n * n - 5;
        hIn := (n * n) * d3 + n * h1 + h0;
        d3 @= r10; h0 @= r14; h1 @= rbx; h2 @= rbp;
    modifies
        rax; r10; r14; rbx; rbp; efl;
    requires
        d3 / 4 * 5 < n;
        rax == n - 4;
    ensures
        hOut % p == hIn % p;
        hOut == (n * n) * h2 + n * h1 + h0;
        h2 < 5;
{
    lemma_BitwiseAdd64();
    lemma_poly_bits64();
    And64(rax, d3)...Adc64Wrap(h2, 0);
    ghost var h10 := n * old(h1) + old(h0);
    hOut := h10 + rax + (old(d3) % 4) * (n * n);
    lemma_poly_reduce(n, p, hIn, old(d3), h10, rax, hOut); }
```

```
And64(rax, d3);
Mov64(h2, d3);
Shr64(d3, 2);
And64(h2, 3);
Add64Wrap(rax, d3);
Add64Wrap(h0, rax);
Adc64Wrap(h1, 0);
Adc64Wrap(h2, 0);
```

Performance: OpenSSL vs. Vale

- AES: OpenSSL with SIMD, AES-NI
- Poly1305 and SHA-256: OpenSSL non-SIMD assembly language (same assembly for OpenSSL, Vale)