Verified High-Assurance Crypto Libraries

Network buffers

Crypto Algorithms

4Q
ECDH
RSA
ECDH
SHA
***
X.509
ASN.1
TLS
HTTPS
Sample crypto algorithm in OpenSSL

- Hand-crafted mix of Perl and assembly
- Customized for 50+ hardware platforms
- Why?
  
  **Performance!**  
  several bytes/cycle
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

\[ MAC(k, m, \overrightarrow{w}) = m + \sum_{i=1..|\overrightarrow{w}|} w_i \cdot k^i \]

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: \texttt{poly1305}

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

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 \( \approx \frac{|\vec{w}|}{2^{130}} \)
Sample crypto algorithm: poly1305

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

A typical 64-bit arithmetic implementation:
1. Represent elements of the prime field for \( p = 2^{130} - 5 \) using **3 limbs** holding 42 + 44 + 44 bits in 64-bit registers
2. Use \((a \cdot 2^{130} + b) \mod p = (a + 4a + b) \mod p\) 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
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."
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

![Diagram showing the compilation process from F* to Low* to C to Executable]

GCC/Clang/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
```c
static void Hac_Impl_Poly1305_64_poly1305_last_pass(uint64_t *acc)
{
    Hac_Bignum_Fproduct_carry_lim_b(acc);
    Hac_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)0xffffffff;
    uint64_t r0 = a0 >> (uint32_t)44;
    uint64_t a1 = (a10 + r0) & (uint64_t)0xffffffff;
    uint64_t r1 = (a10 + r0) >> (uint32_t)44;
    uint64_t a2 = a20 + r1;
    acc[0] = a0;
    acc[1] = a1;
    acc[2] = a2;
    Hac_Bignum_Modulo_carry_top(acc);
    uint64_t t0 = acc[0];
    uint64_t t1 = acc[1];
    uint64_t t0 = t0 & (((uint64_t)1 << (uint32_t)44) - (uint64_t)1);
    uint64_t t1 = t1 + (t0 >> (uint32_t)44);
    acc[0] = t0;
    acc[1] = t1;
    uint64_t a0 = acc[0];
    uint64_t a1 = acc[1];
    uint64_t a2 = acc[2];
    uint64_t mask0 = FStar_Uint64_gte_mask(a0, (uint64_t)0xffffffff);
    uint64_t mask1 = FStar_Uint64_eq_mask(a1, (uint64_t)0xffffffff);
    uint64_t mask2 = FStar_Uint64_eq_mask(a2, (uint64_t)0xffffffff);
    uint64_t mask0 = mask0 & mask1 & mask2;
    uint64_t a0 = a0 - (uint64_t)0xffffffff & mask);
    uint64_t a1 = a1 - (uint64_t)0xffffffff & mask);
    uint64_t a2 = a2 - (uint64_t)0xffffffff & mask);
    acc[0] = a0;
    acc[1] = a1;
    acc[2] = a2;
```

```fst
val poly1305_last_pass : 
  acc:elem -> 
  Stack unit 
  (requires (λ h → live h acc ∧ bounds (as_seq h acc) p44 p44 p42)) 
  (ensures (λ n0 · 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 h acc == Hac.Spec.Poly1305_64_poly1305_last_pass_spec_. (as_seq h0 acc)))
```
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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>HACL*</th>
<th>OpenSSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChaCha20</td>
<td>6.17 cy/B</td>
<td>8.04 cy/B</td>
</tr>
<tr>
<td>Poly1305</td>
<td>2.07 cy/B</td>
<td>2.16 cy/B</td>
</tr>
<tr>
<td>Curve25519</td>
<td>157k cy/mul</td>
<td>359k cy/mul</td>
</tr>
</tbody>
</table>

Still slower than best hand-written assembly language implementations
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;
Vale: extensible, automated assembly language verification (Usenix’17)

machine model (F*)

instructions

- type reg = r0 | r1
- type ins = Mov(dst:reg, src:reg)
  | Add(dst:reg, src:reg)
  | Neg(dst:reg)
  ...

semantics

- eval(Mov(dst, src), ...) = ...
- eval(Add(dst, src), ...) = ...
- eval(Neg(dst), ...) = ...
  ...

code generation

- print(Mov(dst, src), ...) = "mov\ " + (...dst) + (...src)
- print(Add(dst, src), ...) = ...

trusted computing base

Vale source code

machine interface

- procedure mov(…)
  requires ...
  ensures ...
  { ... }
- procedure add(…)
  ...

program

- procedure quadruple(…)
  requires 0 <= r0 < 2^{30};
  ensures r1 == r0 * 4;
  { mov(r1, r0);
    add(r1, r0);
    add(r1, r1);
  }

crypto spec

- mem[eax] == SHA(mem[ebx])

proof

- lemma_mov(…)
- lemma_add(…);

Dafny

- code
- [Mov(r1, r0), Add(r1, r0), Add(r1, r1)]

Z3

- BOOQGE

trusted computing base

functional correctness & side-channel protection
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);
  ...
}
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); }
Performance: **OpenSSL vs. Vale**

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