Verifying Low-Level Cryptographic Code: Low* and HACL*

Aymeric Fromherz
Inria Paris,

MPRI 2-30

Outline

- Last week:
  - Side-Channel Attacks
  - Statically preventing them through noninterference
- Today:
  - Back to verifying implementations (and F*)
  - Reasoning about low-level code
  - Proof engineering to scale verification
- Exam will be on March 12

Reminder: The F* Proof Assistant

- A functional programming language
- With support for dependent types, refinement types, effects, …
- Semi-automated verification by relying on SMT solving
- Extraction to OCaml, F#, C (under certain conditions)

- Try it online at https://fstar-lang.org/tutorial/
- Or install it locally: https://github.com/FStarLang/FStar

Reminder: The F* Effect System

- Separates between
  - Total functions: Tot t
  - Ghost functions: GTot t
  - Possibly non-terminating functions: Dv t

- Can include refinements for specifications

```
val factorial (n:int) : Pure int (requires n ≥ 0) (ensures fun y -> y ≥ 0)

val append_length (#a:Type) (l1 l2: list a) : Ghost unit
        (requires True)
        (ensures fun _ -> length l1 + length l2 == length (append l1 l2))
```

Stateful F* Programs

- F* provides a built-in effect for modeling and reasoning about stateful programs:

  ST t (requires fun h -> pre h) (ensures fun h0 r h1 -> post h0 r h1)

- Reason about state using a standard Hoare logic (requires/ensures)
- ST models a state with garbage-collected references
- This model is in a partial correctness setting:
  - Pure t <: ST t, and Div t <: ST t

Stateful Programs: An Example

```
val incr (r:ref int) : ST unit
        (requires fun h -> True)
        (ensures fun h0 v h1 -> 1 + sel h0 r == sel h1 r)

let incr (r:ref int) = r := !r + 1
```

Specifying the Heap

```
val heap : Type
val ref (a: Type) : Type

val sel : #a:Type -> heap -> ref a -> GTot a

effect ST (a: Type) (pre: heap -> Type) (post: heap -> a -> heap -> Type) = …
```

Heap Operations

```
val (!) (r:ref int) : ST int
        (requires fun h -> True)
        (ensures fun h0 v h1 -> h0 == h1 ∧ sel h0 r == v)

val (:=) (r:ref int) (v: int) : ST unit
        (requires fun h -> True)
        (ensures fun h0 _ h1 -> sel h1 r == v)
```

Reasoning about Framing

```
val swap (r1 r2:ref int) : ST unit
        (requires fun h -> True)
        (ensures fun h0 _ h1 ->
                sel h0 r1 == sel h1 r2 ∧ sel h0 r2 == sel h1 r1)

let swap r1 r2 =
        let v1 = !r1 in let v2 = !r2 in
        r1 := v2; r2 := v1
```

- Correct? What about aliasing? What part of memory does a write impact?

The Modifies Clause

```
val addr_of : #a:Type -> ref a -> GTot nat

let modifies (s:set nat) (h0 h1 : heap) = forall a (r:ref a).
        not (addr_of r `mem` s) ==> sel h1 r == sel h0 r

val (:=) (r:ref int) (v: int) : ST unit
        (requires fun h -> True)
        (ensures fun h0 _ h1 ->
                modifies {addr_of r} h0 h1 ∧
                sel h1 r == v)
```

The Modifies Clause

```
val addr_of : #a:Type -> ref a -> GTot nat

let modifies (s:set nat) (h0 h1 : heap) = forall a (r:ref a).
        not (addr_of r `mem` s) ==> sel h1 r == sel h0 r

val (:=) (r:ref int) (v: int) : ST unit
        (requires fun h -> True)
        (ensures fun h0 _ h1 ->
              modifies !{r} h0 h1 ∧
              sel h1 r == v)
```

Lighter notation for location sets

Applying the Modifies Clause, Intuitively

```
val swap (r1 r2:ref int) : ST unit
        (requires fun h -> addr_of r1 <> addr_of r2)
        (ensures fun h0 _ h1 ->
                sel h0 r1 == sel h1 r2 ∧ sel h0 r2 == sel h1 r1)


let swap r1 r2 =
        let v1 = !r1 in let v2 = !r2 in // Does not modify memory
        r1 := v2;          // Only modifies location r1, r2 is left unchanged
        r2 := v1           // Only modifies location r2, r1 is left unchanged
```

Monadic Effects

- The ST effect can be seen as a state monad, with a bind operator

```
G |- e1 : ST t1 (requires (fun h0 -> pre h0)) (ensures (fun h0 x1 h1 -> post h0 x1 h1))
G, x1:t1 |- e2 : ST t2
            (requires (fun h1 -> exists h0. post h0 x1 h1))
            (ensures (fun h1 x2 h2 -> post' h1 x2 h2))
 ----------------------------------------------------------------------
 G |- let x1 = e1 in e2 : ST t2
            (requires (fun h0 -> pre h0))
            (ensures (fun h0 x2 h2 -> exists x1 h1. post h0 x1 h1 /\ post' h1 x2 h2))
```

Consequence Rule

```
G |- e1 : ST t1 (requires (fun h0 -> pre h0)) (ensures (fun h0 x1 h1 -> post h0 x1 h1))
G |= forall h0. pre' h0 => pre h0
G |= forall h0 x1 h1. post h0 x1 h1 => post' h0 x1 h1
---------------------------------------------------------------------------------
 G |- e1 : ST t1 (requires (fun h0 -> pre' h0)) (ensures (fun h0 x1 h1 -> post' h0 x1 h1))
```

Monadic Effects, Example

```
let swap r1 r2 =
        let v1 = !r1 in let v2 = !r2 in
        (* Know (P1): exists h1 (v1, v2). h0 == h1 /\ v1 == sel h0 r1 /\ v2 == sel h0 r2 *)
        r1 := v2;
        (* Know (P2): exists h2 (). modifies !{r1} h1 h2 /\ sel h2 r1 == v2 *)
        r2 := v1
        (* Know (P3): exists h3 (). modifies !{r2} h2 h3 /\ sel h3 r2 == v1 *)
```

Monadic Effects, Example

(Pre): addr_of r1 <> addr_of r2

(P1): exists h1 (v1, v2). h0 == h1 /\ v1 == sel h0 r1 /\ v2 == sel h0 r2

(P2): exists h2 (). modifies !{r1} h1 h2 /\ sel h2 r1 == v2

(P3): exists h3 (). modifies !{r2} h2 h3 /\ sel h3 r2 == v1

Goal: sel h0 r1 == sel h3 r2 ∧ sel h0 r2 == sel h3 r1

(P3): sel h3 r2 == v1 == sel h0 r1 (P1)

(P2): sel h2 r1 == v2 == sel h0 r2 (P1)

(P3) gives *modifies !{r2} h2 h3*.

By definition of modifies and (Pre), we derive sel h3 r1 == sel h2 r1 == sel h0 r2

Modifies Theory

let modifies s h0 h1 = forall r. addr_of r ∉ s ==> sel h1 r == sel h0 r

- Transitivity:

  modifies s1 h0 h1 ∧ modifies s2 h1 h2 ⇒ modifies (union s1 s2) h0 h2

- Inclusion:

  s1 ⊆ s2 ∧ modifies s1 h0 h1 ⇒ modifies s2 h0 h1

Exercises

- Stateful Sum
- Stateful Factorial

Richer Memory Models

- The stateful effect seen so far offers an OCaml-like memory model
  - A reference in scope is assumed to be live
  - References are not manually memory-managed
- We want to reason about C code:
  - Need to reason about liveness, more complex datastructures, stack vs heap, …
- **Idea:** Keep the stateful effect, but change the underlying state to provide a C-like memory model

The Low* Framework

- Low* is a shallow embedding of a subset of C into F*

```
val memcpy (dst : buffer uint64) (src : buffer uint64)
    : Stack unit
    (requires λ h → length dst == length src ∧ live h dst ∧ live h src)
    (ensures  λ h0 _ h1 → modifies (loc_buffer dst) h0 h1)
```

Low* Machine Integers

- A model of C integers, e.g., uint32
- Specified using "mathematical" integers

```
val v (n: UInt32.t) : GTot nat
```

- Arithmetic operations can lead to overflow/underflow

```
val add (n1 n2: UInt32.t) : Pure UInt32.t
    (ensures λ x → v x == (v n1 + v n2) % 2^32)
```

- Operations for signed integers require to avoid over/underflows

```
val add (n1 n2: Int32.t) : Pure Int32.t
    (requires (v n1 + v n2) < 2^31)
    (ensures λ x → v x == v n1 + v n2)
```

Low* Arrays: Specification

- At the core of the Low* memory model, named *buffers* for historical reasons
- Represented in memory as a sequence of values:

```
val buffer (a: Type) : Type

val as_seq : #a:Type -> mem -> buffer a -> GTot (seq a)
```

Low* Arrays: Core API

- Usable through an API that ensures *spatial* and *temporal memory safety*

```
val index (#a:Type) (b : buffer a) (n:UInt32.t{v n < length b})
    : Stack a
    (requires λ h → live h b)
    (ensures  λ h0 x h1 → h0 == h1 ∧ x == Seq.index (as_seq h0 b) (v n))
```

- Very similar signature for upd

Low* Arrays: Pointer Arithmetic

- Low* allows (controlled) pointer arithmetic, to access a sub-array from a live array

```
val sub (#a:Type) (b : buffer a) (start: UInt32.t) (len: ghost UInt32.t)
  : Stack (buffer a)
  (requires λ h → live h b ∧ v start + v len <= length b)
  (ensures  λ h0 x h1 → h0 == h1 ∧ sub_spec b (v start) (v len))
```

- This corresponds in C to the operation b + start
- The modifies clause is also extended to reason about possible overlaps between arrays and slices

Low* Arrays: Allocation

- Low* provides functions to create new arrays (i.e., allocate) either on the heap (malloc) or on the stack

- Newly created arrays are assumed to be live, and with a location disjoint from all previously created arrays

Modeling Stack and Heap

- Instead of a monolithic memory model, the heap is divided into a tree of regions (conceptually, mem = map region_id heap)
- A specific subset of these regions models the *C stack*
- Two different effects: ST for arbitrary stateful computations, and Stack for computations only allocating in the current stack frame
- Clients can create a stack frame in a function using push/pop_frame()

Modeling Stack and Heap, Formally

```
effect Stack (a:Type) (pre: mem -> prop) (post: mem -> a -> mem -> prop) = ST a
      (requires λ h → pre h)
      (ensures  λ h0 x h1 → post h0 x h1 ∧ equal_domains h0 h1)

let equal_domains (m0: mem) (m1: mem) =
    // The current top stack-frame is the same
    m0.tip == m1.tip ∧
    // The trees of heaps have the same shape
    Set.equal (Map.domain m0.h) (Map.domain m1.h) ∧
   // For each heap, the used addresses are the same
   (forall r. Map.contains m0.h r ==>
            Heap.equal_dom (Map.sel m0.h r) (Map.sel m1.h r))
```

A Complete (Toy) Low* Example

```
let main (): Stack Int32.t (requires λ _ -> True) (ensures λ _ _ _ -> True) =
  push_frame ();
  let b: buffer UInt32.t = alloca 0ul 8ul in
  upd b 0ul 255ul;
  pop_frame ();
  0l
```

Extraction

```
int32_t main(void) {
  uint32_t b[8U] = { 0U };
  b[0U] = 255U;
  return (int32_t)0;
}
```

Extracting Low* Code

- Low* code can be extracted to C (and recently, Rust) through the KaRaMeL compiler (https://github.com/FStarLang/karamel)

- KaRaMeL recognizes Low*-specific types and functions (e.g., buffer uint8 and upd), and translates them to standard C types and operations (e.g., uint8[] and assignment)

- The resulting code can be compiled with standard C compilers, and integrated in unverified projects

The KaRaMeL Compiler: Design Goals

- Produce idiomatic, readable C code
  - Despite verification, extracted code will likely be reviewed and audited by users
- Remain as simple as possible
  - Semantic preservation is proven on paper, however the implementation is trusted, the codebase needs to remain small
- Support a pragmatic subset of C
  - We control the Low* code we want to extract, we only need a reasonable subset of features for verifying real-world code, not the entire C standard

KaRaMeL: Producing Idiomatic Code

- KaRaMeL retrieves the F* AST after ghost code erasure
- Many compilation micropasses:
  - Unused Argument Elimination (often for extra unit arguments)
  - Inductives with a single constructor become structs
  - Empty structs are removed
  - Inductives with constant constructors (e.g., type t = A | B) are extracted to enums
  - …
- Also preserves original variable names, can preserve code comments, attempts to eliminate temporary variables, …

Low* Extraction Limitations

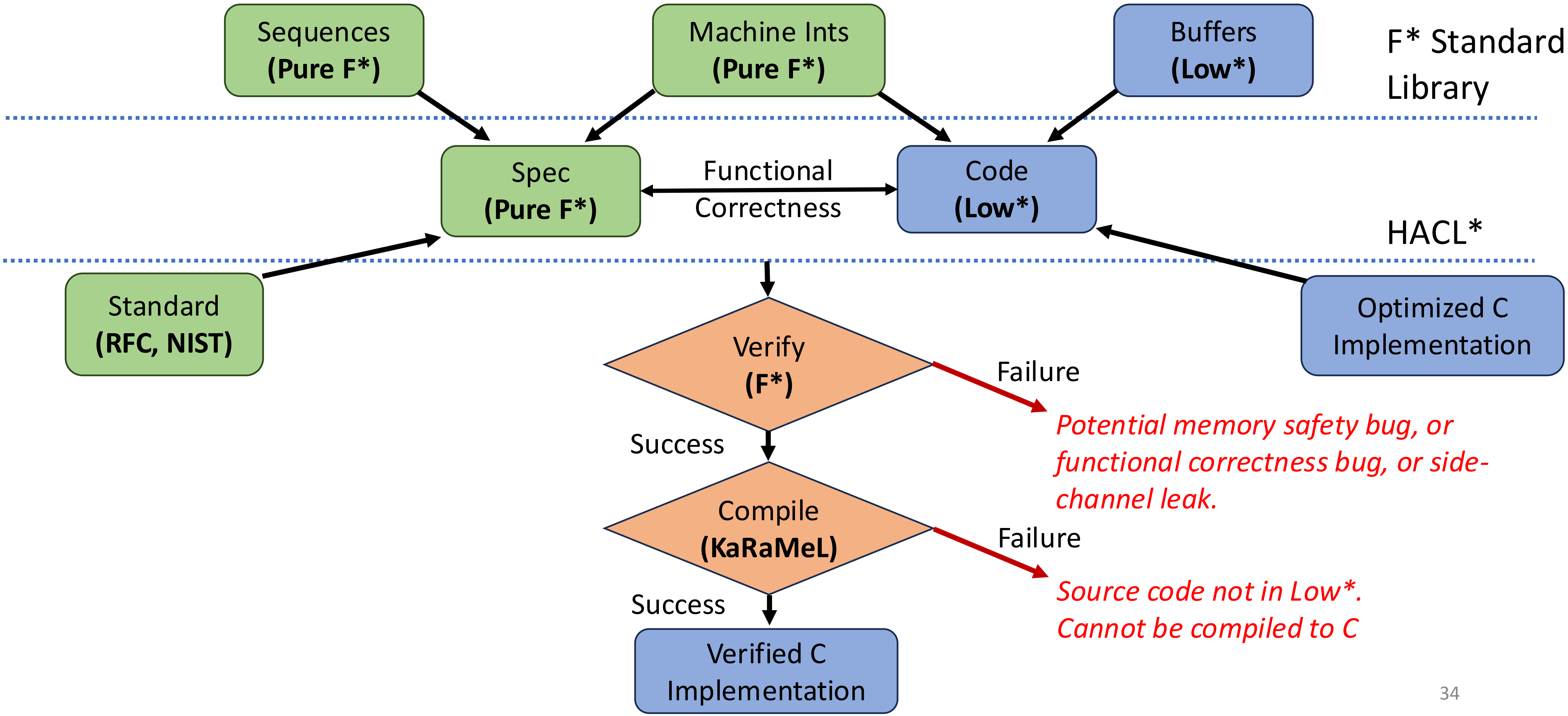
- Types with no equivalent in C (e.g., mathematical integers, sequences)
- Recursive datatypes (e.g., OCaml-style lists or trees)
- Polymorphism
  - KaRaMeL however monomorphizes code as much as possible before failing, so some *uses* of polymorphic functions and datatypes can be extracted
- Higher-order (beyond simple versions)
- Closures

All of these remain available for verification! But must be erased before reaching KaRaMeL

Using Low*: The HACL* Crypto Library

- HACL*: A verified, comprehensive cryptographic provider
- Provides guarantees about memory safety, functional correctness, resistance against side-channels
- ~150k lines of F* code compiling to ~100k lines of C (and Assembly) code
- 30+ algorithms (hashes, authenticated encryption, elliptic curves, …)
- Integrated in Linux, Firefox, Tezos, and many more

Verification Workflow



Example: Poly1305 MAC Algorithm

- Poly1305 is a message authentication code

$$poly(k, m, w_1 \dots w_n) = (m + w_1 k^1 + \ \dots + w_n k^n)\ \%\ (2^{130} - 5)$$

- It authenticates a message $w_1 \dots w_n$ by:
  - Encoding it as a polynomial in the prime-field modulo $2^{130} - 5$
  - Evaluating it at point *k* (first part of the key)
  - Masking the result with *m* (second part of the key)

Specifying Poly1305

- The specification comes from the official RFC
- The specification is our **ground truth**: it needs to be as simple and easy to review as possible
- The specification also needs to be **executable**: We can then “test” it on standard test vectors
- **Solution:** We write an inefficient but straightforward implementation in Pure F*, and benefit from extraction to OCaml

Specifying Poly1305

```
let prime = pow2 130 – 5

let felem = x: nat{x < prime}
let fadd (x: felem) (y: felem) : felem = (x + y) % prime
let fmul (x: felem) (y: felem) : felem = (x * y) % prime

type key = s:seq uint8{length s == 32}
…
```

Implementing Poly1305

Several steps:

1. Create a bignum library for representing field elements
2. Optimize prime-specific field arithmetic
3. Implement Poly1305, and expose the corresponding API

Bignum Library for $\mathbb{Z}/(2^{130}-5)\mathbb{Z}$

- The numbers are too large to fit machine integers
- We use an unsaturated 44-44-42 representation
- feval allows to retrieve the corresponding mathematical number

```
type felem = b:buffer uint64{length b = 3}

let feval (h: mem) (f: felem) : GTot Spec.felem =
   let s = as_seq h f in
   (v s.[0] + v s.[1] * pow2 44 + v s.[2] * pow2 88) % Spec.prime
```

Implementing Field Arithmetic

```
val fadd (a b : felem) : Stack unit
    (requires λ h → live h a ∧ live h b ∧
                     disjoint a b ∧
                     no_overflow h a b)
    (ensures  λ h0 _ h1 → modifies (loc a) h0 h1 ∧
                          feval h1 a == Spec.fadd (feval h0 a) (feval h0 b)
     )
```

This specification guarantees:

- Memory Safety
- Functional Correctness
- Side-Channel Resistance (omitted here, see last week)

Optimizing Field Arithmetic

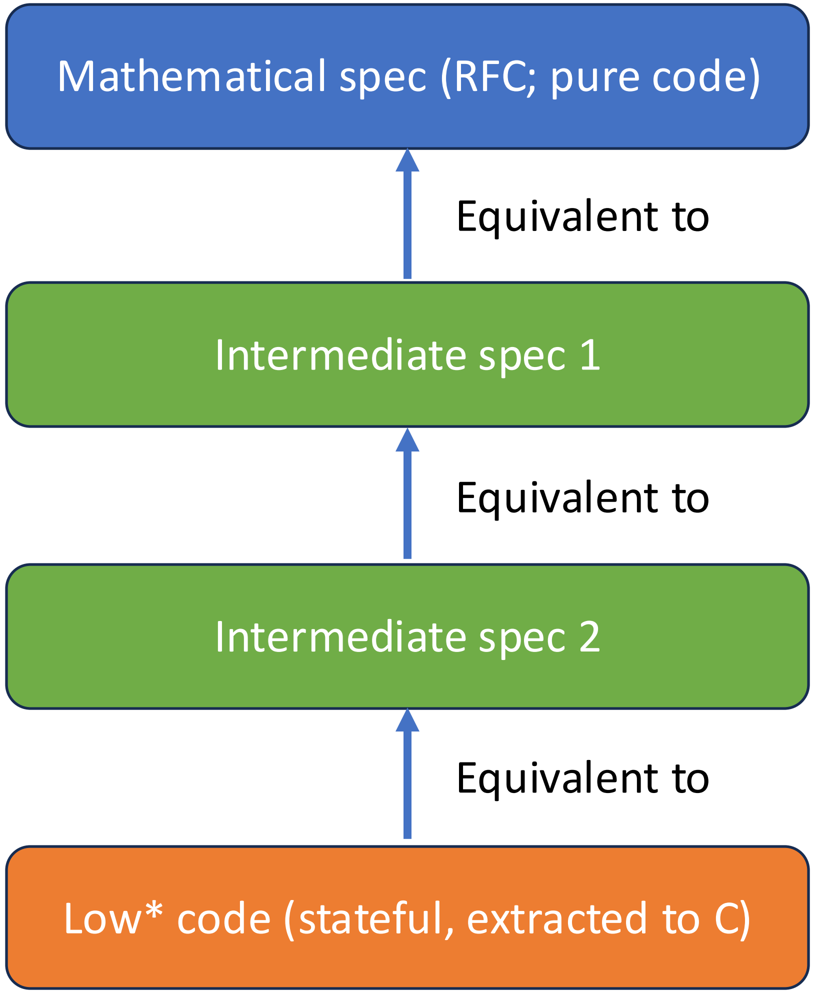
- Many possible optimizations are purely algorithmic:
  - Replace a modular reduction by a Barrett reduction
  - Replace a modular multiplication by a Montgomery multiplication

- These are orthogonal from memory optimizations (e.g., unsaturated memory representation of bignums)

- Ideally, we want to reason about them in isolation to simplify verification

Proof by Refinement

- We write intermediate specifications
- Each layer is proven semantically equivalent to the layer above
- We can reason independently about different elements in each layer (algorithmic optimization, memory layout, aliasing, …)



Example: Modular Exponentiation

```
let rec exp (g: int) (n: nat) = if n = 0 then 1 else g * exp g (n-1)      // Simple spec


let rec exp_opt (g:int) (n:nat) =
   if n = 0 then 1
   else if n % 2 = 0 then exp_opt (g*g) (n/2)
   else g * exp_opt (g*g) (n-1/2)

let equiv_proof (g:int) (n:nat) : Lemma (exp g n == exp_opt g n)
```

- *exp* and *exp_opt* are in the pure fragment of F*. Proving equivalence only requires reasoning about mathematical facts, not about memory

Example: Modular Exponentiation

```
val exp_opt (g:int) (n:nat) : int

type felem_spec = s: seq uint64{length s == 3}
let feval_spec (f: felem_spec) : int = (v s.[0] + v s.[1] * pow2 44 + v s.[2] * pow2 88)

let fexp_spec (f: felem_spec) (n:uint64) : felem_spec = …

val equiv_proof (f: felem_spec) (n: uint64) : Lemma (
        feval_spec (fexp_spec f n) == exp_opt (feval_spec f) (v n))
```

- Introduce low-level representation of integers using *mathematical sequences*

Example: Modular Exponentiation

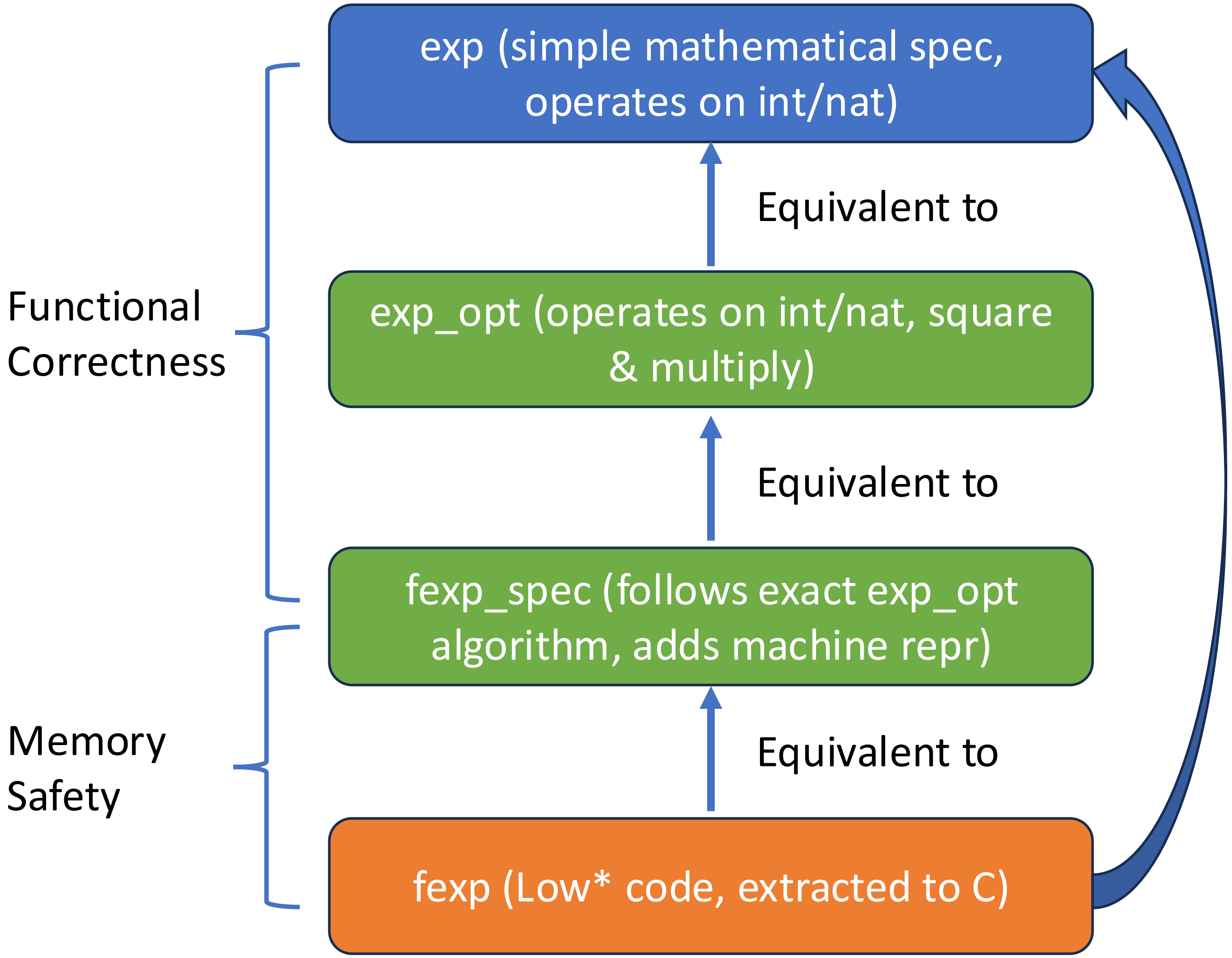
```
type felem = b:buffer uint64{length b = 3}

val fexp (a : felem) (n: uint64) : Stack unit
    (requires λ h → live h a ∧ live h b ∧ disjoint a b ∧
                          no_overflow h a b)
    (ensures  λ h0 _ h1 → modifies (loc a) h0 h1 ∧
                                as_seq h1 a == fexp_spec (as_seq h0 a) n
    )
```

- Introduce memory reasoning (aliasing, disjointness)
- The implementation of fexp closely follows the structure of fexp_spec

Modular Exponentiation: Summary

- Equivalence proofs at each layer compose to provide an end-to-end proof
- Using different layers allows to separate proofs into independent parts
- How to separate and how many layers to create is up to the proof engineer



Implementing Poly1305

```
[@"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)
```

memory safety

math spec

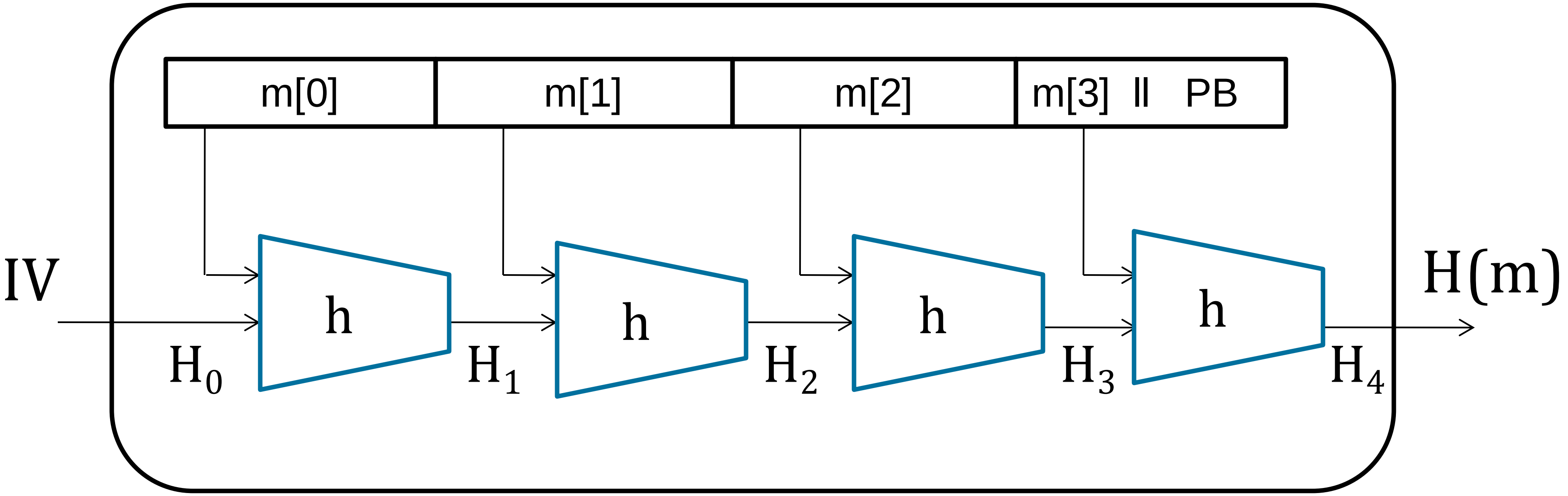
code

proof

```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
```

The Need for Generic Implementations

- Families of cryptographic algorithms often share the same structure



MD5, SHA-1, SHA2-224, SHA2-256, SHA2-384, SHA2-512

The Need for Generic Implementations

- Families of cryptographic algorithms often share the same structure
- Implementing many high-quality variants is tedious and time-consuming
- Verification makes it even more costly
- Can we reason about implementations generically?

Generic SHA2 Implementations

```
let state (a : sha2_alg) = match a with
  | SHA2_224 | SHA2_256 -> buffer uint32
  | SHA2_384 | SHA2_512 -> buffer uint64

let bitwise_and #a (x y: state a) : state a =
  if alg == SHA2_224 || alg == SHA2_256 then UInt32.bitwise_and x y
  else UInt64.bitwise_and x y

let shuffle (a: sha2_alg) … = … bitwise_and #a x y …
```

- We can write a generic implementation of each basic block
- We then write SHA2 functions generically

Generic SHA2 Implementations: Issues

- Naive generality leads to poor performance

```
sha2_state bitwise_and (alg:sha2_alg, x:sha2_state, y:sha2_state) {
  if (alg == SHA2_224 | alg == SHA2_256) {
       return UInt32.bitwise_and(x, y);
  } else {
       return UInt64.bitwise_and(x, y);
  }
}
```

- At each arithmetic operation, we now have a branching
- For performance-critical code, this is unacceptable
- We want genericity, but it must not impact performance

Reminder: The F* Normalizer

- Dependently typed proof assistants include a *normalizer* which reduces computations.

```
assert_norm (length [1; 2; 3; 4; 5; 6; 7; 8; 9; 10] == 10)

match [1; 2; 3; 4; 5; 6; 7; 8; 9; 10] with | [] -> 0 | hd :: tl -> 1 + length tl     == 10     ↝
1 + match [2; 3; 4; 5; 6; 7; 8; 9; 10] with | [] -> 0 | hd :: tl -> 1 + length tl   == 10     ↝
…
10 == 10 ↝
True
```

- Requires concrete terms, cannot reduce symbolic terms

Partial Evaluation and Inlining

- We rely on two mechanisms: *compile-time inlining*, and *partial evaluation* (which uses the normalizer under the hood)

```
inline_for_extraction noextract
let bitwise_and #a (x y: state a) : state a =
  if alg == SHA2_224 || alg == SHA2_256 then UInt32.bitwise_and x y
  else UInt64.bitwise_and x y

let shuffle (a: sha2_alg) … = … bitwise_and #a x y …
```

Partial Evaluation and Inlining

- We rely on two mechanisms: *compile-time inlining*, and *partial evaluation*

```
let shuffle (a: sha2_alg) … =
        …
        if alg == SHA2_224 || alg == SHA2_256 then UInt32.bitwise_and x y
        else UInt64.bitwise_and x y
        …
```

Partial Evaluation and Inlining

- We rely on two mechanisms: *compile-time inlining*, and *partial evaluation*

```
inline_for_extraction noextract
let shuffle (a: sha2_alg) … =
        …
        if alg == SHA2_224 || alg == SHA2_256 then UInt32.bitwise_and x y
        else UInt64.bitwise_and x y
        …

let shuffle_224 = shuffle SHA2_224
```

Partial Evaluation and Inlining

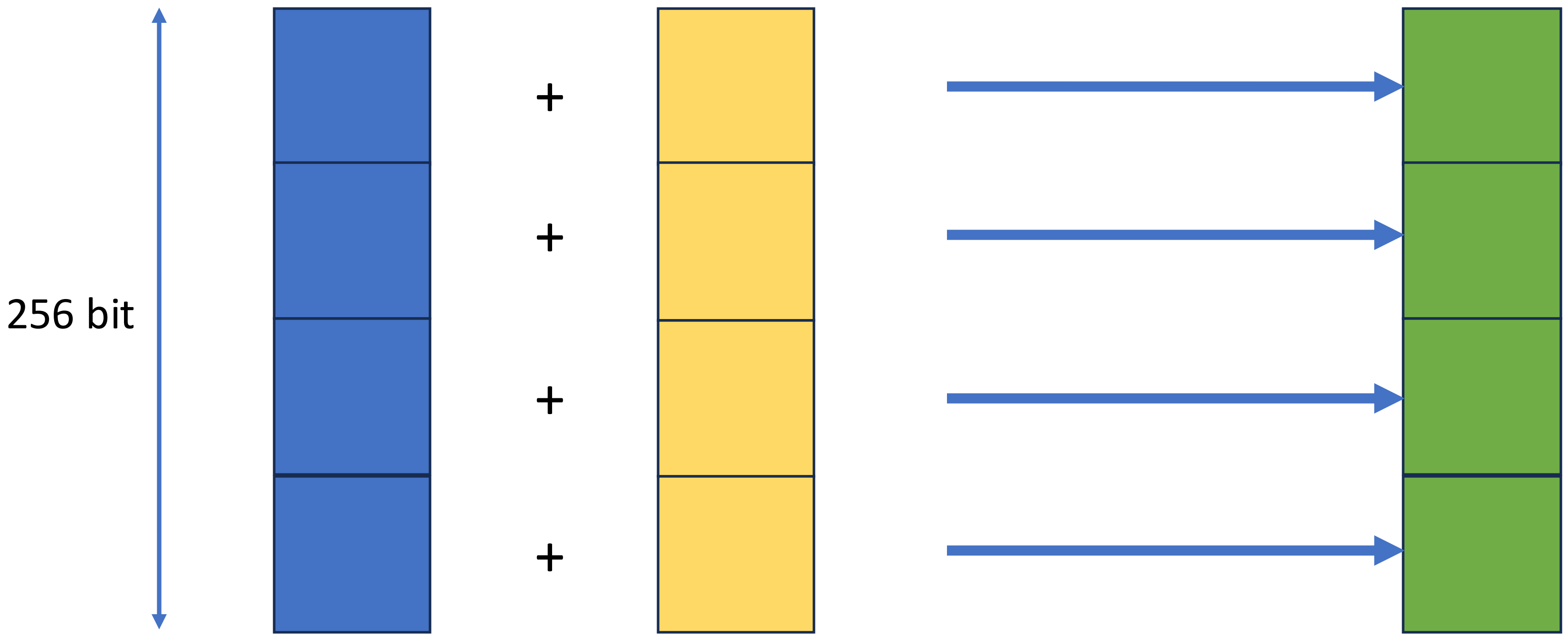
- We rely on two mechanisms: *compile-time inlining*, and *partial evaluation*

```
let shuffle_224 … =
        …
        if SHA2_224 == SHA2_224 || SHA2_224 == SHA2_256 then UInt32.bitwise_and x y
        else UInt64.bitwise_and x y
        …

let shuffle_224 … =
        …
        UInt32.bitwise_and x y …
```

SIMD Optimizations

- Modern CPUs offer SIMD (Single-instruction Multiple-Data) instructions for lightweight parallelism



SIMD Optimizations

- Crypto is highly amenable to SIMD-based optimizations
  - Process several blocks in parallel
  - Parallelize the inner block cipher in ChaCha20 (intended by designers)
    - 10-20x speedup on ChaCha20 using AVX512 SIMD parallelism
- However, SIMD instructions are platform-specific

master ▾ openssl / crypto / chacha / asm / chacha-x86_64.pl

```
sub XOP_lane_ROUND {
my ($a0,$b0,$c0,$d0)=@_;
sub AVX2_lane_ROUND {
my ($a0,$b0,$c0,$d0)=@_;
sub AVX512ROUND {       # critical path is 14 "SIMD ticks" per round
        &vpaddd ($a,$a,$b);
        &vpxord ($d,$d,$a);
```

- Maintaining optimized implementations for all platforms is hard

Verified Generic SIMD Crypto

- Similar technique as for SHA2, except, we abstract over vectorization level

```
val vec_t:  w:width → Type
val (+|) :   #w:width → vec_t w → vec_t w → vec_t w
```

- We then verify a generic implementation

```
let chacha20_init  (w:width) (state:vec_t w) … = …
  state.(a) <- state.(a) +|  state.(b)
  …
```

- And finally specialize many times

```
let chacha20_init_avx = chacha20_init 4
let chacha20_init_avx2 = chacha20_init 8
let chacha20_init_avx512 = chacha20_init 16
```

Higher-Order Combinators

- So far, this pattern applied to a parameter used for pattern-matching

- Cryptographic **constructions** frequently **combine** core operations.

Example:

- The **Merkle-Damgård construction** only requires an (abstract) compression function
- The construction consists of folding the core compression function over multiple blocks of data

Higher-Order Combinators

```
// Write once; this is not Low*
noextract inline_for_extraction
let mk_compress_blocks (a: hash_alg)
  (compress: compress_st a)
  (s: state a)
  (input: blocks)
  (n: u32 { length input = block_size a * n })
=
  C.Loops.for 0ul n (fun i ->
    compress s (Buffer.sub input (i * block_size a) (block_size a)))


// Specialize many times; now this is Low*
let compress_blocks_224 = mk_compress_blocks SHA2_224 compress_224
...
let compress_md5 = mk_compress_blocks MD5 compress_md5
...
```

Higher-Order and Partial Evaluation

- The methodology so far relies on partial evaluation and inlining

```
C.Loops.for 0ul n (fun i ->
  compress s (Buffer.sub input (i * block_size a) (block_size a)))
```

- This would inline the entire compress function inside the loop

```
noextract inline_for_extraction
let mk_hash (a: hash_alg)
  (init: init_st a)
  (compress_blocks: compress_blocks_st a)
  (compress_last: compress_last_st a)
  (extract: extract_st a)
```

- Worse as combinator complexity grows. We need another methodology for generic code

Encoding Functors: Associative List Example

OCaml:

```
module type Map = sig
  type k
  val find: k -> (k * 'a) list -> 'a option
end

module type EqType = sig
  type t
  val eq: t -> t -> bool end

module MkMap (E : EqType) :
  Map with type k = E.t = struct
  type k = E.t
  let find x ls =
    let b = ref true in
    let lsp = ref ls in
    while !b do
      match !lsp with
      | [] -> b := false
      | (x', _) :: tl ->
        if E.eq x x' then b := false
        else lsp := tl done;
    match !lsp with
    | [] -> None
    | (_, y) :: _ -> Some y
end
```

Doesn't compile to C (same with typeclasses)

Type constraint

We want a loop in the generated code

F*:

Replace with a linked list

```
type map (a : Type) = {
  k: Type;
  find: k -> list (k * a) -> ST (option a) ... }

type eq_type = {
  t: Type;
  eq: t -> t -> bool; }

let mk_map (e : eq_type) (a : Type) :
  m:map a{m.k == e.t} = {
  k = e.t;
  find = (fun x ls ->
    let b = alloc true in
    let lsp = alloc ls in
    while (fun () -> !* b)
      (fun () ->
       let ls = !* lsp in
       match ls with
       | [] -> upd b false
       | (x', _) :: tl ->
         if e.eq x x' then upd b false
         else upd lsp tl);
    match !* lsp with
    | [] -> None
    | (_, y) :: _ -> Some y) }
```

Dictionary has runtime cost

Refinement

Proofs and annotations omitted

Extraction to C?

⇒ Specialization and partial evaluation?

Zero-Cost Functors: First Attempt (i)

Generic code (F*):

```
type map (a : Type) = {
  k: Type;
  find: k -> list (k * a) -> ST (option a) ... }

type eq_type = {
  t: Type;
  eq: t -> t -> bool; }

let mk_map (e : eq_type) (a : Type) :
  m:map a{m.k == e.t} = {
  k = e.t;
  find = (fun x ls ->
    let b = alloc true in
    let lsp = alloc ls in
    while (fun () -> !* b)
      (fun () ->
       let ls = !* lsp in
       match ls with
       | [] -> upd b false
       | (x', _) :: tl ->
         if e.eq x x' then upd b false
         else upd lsp tl);
    match !* lsp with
    | [] -> None | (_, y) :: _ -> Some y) }
```

Specialization:

```
let str_eqty : eq_type = { t = string; eq = String.eq; }
let ifind = (mk_map str_eqty int).find
```

After partial evaluation: **Types are specialized**

```
let ifind (x: string) (ls: list (string * int)): option int =
  let b = alloc true in let lsp = alloc ls in
  while (fun () -> !* b)
   (fun () ->
    let ls = !* lsp in
    match ls with
    | [] -> upd b false
    | (x', _) :: tl ->
     if String.eq x x'
     then upd b false
     else upd lsp tl);
  match !* lsp with
  | [] -> None | (_, y) :: _ -> Some y
```

e.eq is inlined

What happens if the code has several layers?

Zero-Cost Functors: First Attempt (ii)

Peer device for a secure channel protocol:

```
(* "Module signature" *)
type dv = {
  pid : Type;
  send : pid -> list (pid * ckey) -> bytes -> option bytes;
  recv : pid -> list (pid * ckey) -> bytes -> option bytes; }
```

```
(* "Module implementation" *)
type cipher = {
  enc : ckey -> bytes -> bytes;
  dec : ckey -> bytes -> option bytes; }

let mk_dv (m : map ckey) (c : cipher) : d:dv{d.pid == m.k} = {

  pid = m.k;

  send = (fun id dv plain ->
    match m.find id dv with
    | None -> None
    | Some sk -> Some (c.enc sk plain));

  recv = (fun id dv secret ->
    match m.find id dv with
    | None -> None
    | Some sk -> c.dec sk secret)
}
```

find gets inlined and duplicated

Zero-Cost Functors: Encoding

Parameterize with eq

```
inline_for_extraction noextract
let mk_find (k v : Type) (eq: k -> k -> bool) (x: k) (ls: list (k * v)) : option v =
  let b = alloc true in let lsp = alloc ls in
  while (fun () -> !* b)
    (fun () -> let ls = !* lsp in
    match ls with | [] -> upd b false
    | (x', _) :: tl -> if eq x x' then upd b false else upd lsp tl);
    match !* lsp with | [] -> None | (_, y) :: _ -> Some y)

(* Don't inline ifind *)
let ifind = mk_find string ckey String.eq
```

Cumbersome to write and maintain

```
inline_for_extraction noextract
let mk_send (pid : Type) (find : pid -> list (pid * ckey) -> option ckey) (enc : ckey -> bytes -> bytes)
  (id : pid) (dv : list (pid * ckey)) (plain : bytes) : option bytes =
  match find id dv with
  | None -> None
  | Some sk -> Some (enc sk plain)

(* Don't inline isend *)
let isend = mk_send string ifind aes_enc

... (* mk_recv and irec *)
```

Zero-Cost Functors: Call-graph Rewriting

What we want to write:

```
type mindex = { k : Type; v : Type }

[@ Specialize]
assume val eq (i : mindex): i.k -> i.k -> bool

[@ Eliminate]
let while_cond (b: pointer bool) (_:unit) = !*b

[@ Eliminate]
let while_body (i: mindex) (b: pointer bool)
  (lsp: list (i.k * i.v)) (x:i.k) (_:unit) =
  let ls = !* lsp in
  match ls with
  | [] -> upd b false
  | (x', _) :: tl ->
    if eq x x' then upd b false
    else upd lsp tl

[@ Specialize]
let find (i : mindex) (x : i.k)
  (ls : list (i.k * i.v)) : option i.v =
  let b = alloc true in
  let lsp = alloc ls in
  while (while_cond b) (while_body i b lsp x);
  match !* lsp with
  | [] -> None | (_, y) :: _ -> Some y
```

What we want to get:

```
type mindex = { k : Type; v : Type }

let mk_find (i: mindex) (eq: i.k-> i.k -> bool)
  (x: i.k) (ls: list (i.k * i.v)): option i.v =
  let b = alloc true in let lsp = alloc ls in
  while (fun () -> !* b)
   (fun () -> let ls = !* lsp in
    match ls with
    | [] -> upd b false
    | (x', _) :: tl ->
     if eq x x' then upd b false
     else upd lsp tl);
  match !* lsp with
  | [] -> None
  | (_, y) :: _ -> Some y
```

The code is re-checked

```
%splice [ mk_find ] (specialize (`mindex) [`find ])
```

Call-graph rewriting by means of meta-programming

The HPKE Example

Hybrid Public Key Encryption
draft-irtf-cfrg-hpke-06

Abstract

This document describes a scheme for hybrid public-key encryption (HPKE). This scheme provides authenticated public key encryption of arbitrary-sized plaintexts for a recipient public key. HPKE works

- Generic in three classes of algorithms
  - Authenticated Encryption with Additional Data (AEAD)
  - Key Encapsulation Mechanism (KEM)
  - Key Derivation Function (KDF)
- 24 possible ciphersuites, many more implementations

A Generic, Verified HPKE Implementation

- Abstract over algorithms to verify a generic implementation (800 lines)

```
val hpke_encrypt: cs:ciphersuite -> aead_encrypt cs -> …
```

- Instantiate and extract each desired version/implementation (10 lines)

```
let hpke_encrypt_avx_aes = hpke_encrypt (AESGCM, …) aes_encrypt_avx
let hpke_encrypt_avx2_aes = hpke_encrypt (AESGCM, …) aes_encrypt_avx2
```

- Call-graph rewriting yields a specialized, idiomatic implementation. Calls to encrypt call directly into the corresponding AES-GCM library

Genericity: A Summary

- No performance hit due to genericity (”zero-cost abstraction”)
- Reduces maintenance of verified code (only one generic implementation to maintain)
- Lowers development cost of new variants
  - Adding a new SIMD architecture only requires providing a model for basic operations (add, mul, …) and extending a few datatypes
  - Adding a new HPKE ciphersuite only requires 10 lines of code (assuming the underlying primitives are implemented)

HACL* and Low*: Summary

- **Low***: A subset of F*, modeling a well-behaved subset of C
- **HACL***: A comprehensive, verified cryptographic library written in Low*, yielding human-readable, high-performance C code
- Specifications are executable and directly translated from RFCs
- Proof methodology relies on successive refinements to separate verification conditions
- Engineering methodology relies on generic implementations, which are specialized at extraction-time through partial evaluation and metaprogramming