Verified low-level programming *embedded in F**

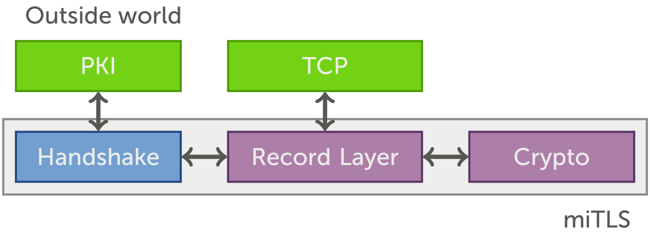
Jonathan Protzenko — Microsoft Research
Nikhil Swamy — Microsoft Research

Project Everest — INRIA Paris
MSR Redmond, Cambridge, Bangalore
CMU

Part 1: Motivation

Some motivation: the TLS protocol

TLS stands for *transport layer security* and now powers more than half of the internet traffic.



TLS is a cryptographic protocol:

- "protocol": handshake, negotiation, key derivation
- "cryptographic": well, a lot of math that needs to be fast

Need for speed

latency is correlated with the performance of asymmetric cryptography

throughput is correlated with the performance of symmetric cryptography

Efficiency is *essential*. That's why everyone binds to some cryptographic library written in C.

Need for speed

latency is correlated with the performance of asymmetric cryptography

throughput is correlated with the performance of symmetric cryptography

Efficiency is *essential*. That's why everyone binds to some cryptographic library written in C.

Sadly, most of the time, this means, OpenSSL.

You can guess what happens next

These heavily optimized C implementations have bugs.

You can guess what happens next

```
OpenSSL Security Advisory [10 Nov 2016]
========================================

ChaCha20/Poly1305 heap-buffer-overflow (CVE-2016-7054)
======================================================

Severity: High

TLS connections using *-CHACHA20-POLY1305 ciphersuites are susceptible to a DoS
attack by corrupting larger payloads. This can result in an OpenSSL crash. This
issue is not considered to be exploitable beyond a DoS.
```

nave bugs.

You can guess what happens next

ave bugs.

```
OpenSSL Security Advisory [10 Nov 2016]
========================================

ChaCha20/Poly1305 heap-buffer-overflow (CVE-2016-7054)
======================================================

Severity: High

TLS connections us
attack by corrupti
issue is not consi
```

[openssl-dev] [openssl.org #4482] Wrong results with Poly1305 functions

Hanno Boeck via RT rt at openssl.org
Fri Mar 25 12:10:32 UTC 2016

- Previous message: [openssl-dev] [openssl.org #4480] PATCH: Ubuntu 14 (x86_64): Compile errors and warnings when using "no-asm -ansi"
- Next message: [openssl-dev] [openssl.org #4483] Re: [openssl.org #4482] Wrong results with Poly1305 functions
- **Messages sorted by:** [date] [thread] [subject] [author]

```
Attached is a sample code that will test various inputs for the
Poly1305 functions of openssl.

These produce wrong results. The first example does so only on 32 bit,
the other three also on 64 bit.
```

You can guess what happens next

…have bugs.

```
OpenSSL Security Advisory [10 Nov 2016]
========================================

ChaCha20/Poly1305 heap-buffer-overflow (CVE-2016-7054)
======================================================

Severity: High

TLS connections us
attack by corrupti
issue is not consi
```

[openssl-dev] [openssl.org #4482] Wrong results with Poly1305 functions

Hanno Boeck via RT rt at
Fri Mar 25 12:10:32 UTC

- Previous message: [o… when using "no-asm
- Next message: [opens
- **Messages sorted by:**

```
Attached is a sample code
Poly1305 functions of ope

These produce wrong resul
the other three also on 6
```

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

David Benjamin via RT rt at openssl.org
Thu Mar 17 21:22:26 UTC 2016

- Previous message: [openssl-dev] [openssl-users] Removing some systems
- Next message: [openssl-dev] [openssl.org #4439] poly1305-x86.pl produces incorrect output
- **Messages sorted by:** [date] [thread] [subject] [author]

```
Hi folks,

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

$ OPENSSL_ia32cap=0 ./poly1305_test2
PASS
$ ./poly1305_test2
Poly1305 test failed.
got:      2637408fe03086ea73f971e3425e2820
expected: 2637408fe13086ea73f971e3425e2820

I believe this affects both the SSE2 and AVX2 code. It does seem to be
dependent on this input pattern.

This was found because a run of our SSL tests happened to find a
problematic input. I've trimmed it down to the first block where they
disagree.

I'm probably going to write something to generate random inputs and stress
all your other poly1305 codepaths against a reference
recommend doing the same in your own test harness, to
```

Why the sorry situation?

In essence: cryptography is hard.

- Need to understand complex math
- Need to optimize complex math
- Need to be familiar with compilers
- Need to reason about side-channel resistance

No wonder it's easy to get wrong.

Enter: Low*

$\boxed{\text{Write C programs in F}^* \text{ and prove them correct!}}$

The essence of Low* in Emacs

Spec.Poly1305.fst

```
module Spec.Poly1305

let prime = pow2 130 - 5
type elem = e:ℤ{e ≥ 0 ∧ e < prime}
let fadd (e1:elem) (e2:elem) = (e1 + e2) % prime
let fmul (e1:elem) (e2:elem) = (e1 × e2) % prime

(* Specification code *)
let encode (w:word) =
  (pow2 (8 × length w)) `fadd` (little_endian w)

let rec poly (txt:text) (r:e:elem) : Tot elem (decreases (length txt)) =
  if length txt = 0 then zero
  else
    let a = poly (Seq.tail txt) r in
    let n = encode (Seq.head txt) in
    (n `fadd` a) `fmul` r
```

```
-:**-  Spec.Poly1305.fst   All (2,0)     Git-master  (F★ company)
Auto-saving...done
```

Hacl.Impl.Poly1305_64.fst

File Edit Options Buffers Tools F✪ Help

memory safety

math spec

```
[@"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)))
```

code

```
[@"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'
```

proof

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

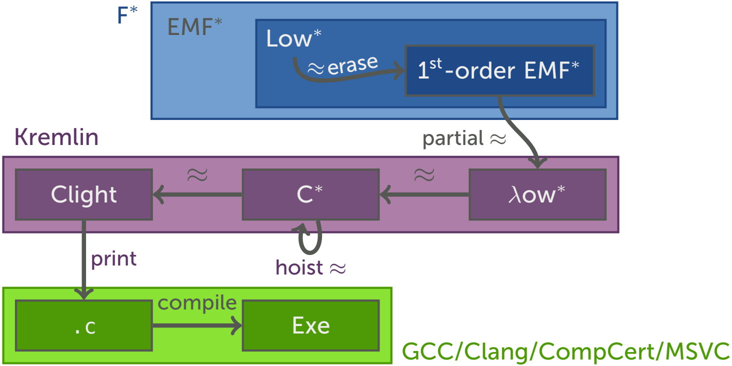
Poly1305_64.c

File Edit Options Buffers Tools C Help

```
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;
}
```

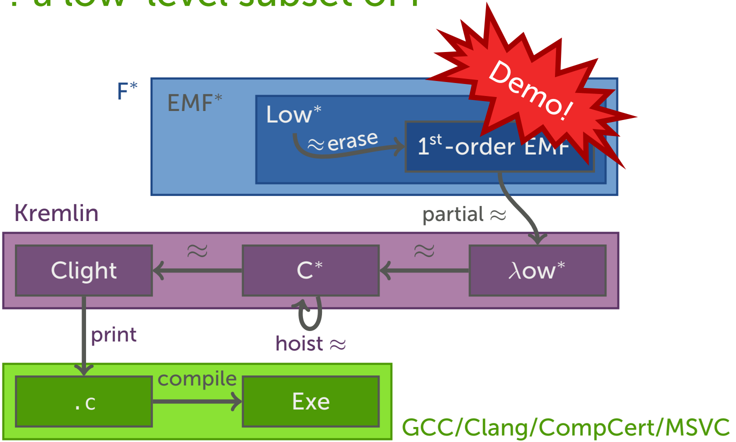
-:**- **Poly1305_64.c**   49% L272   Git-master   (C/l company A

Low*: a low-level subset of F*



Disclaimer: these steps are supported by hand-written proofs.

Low*: a low-level subset of F*



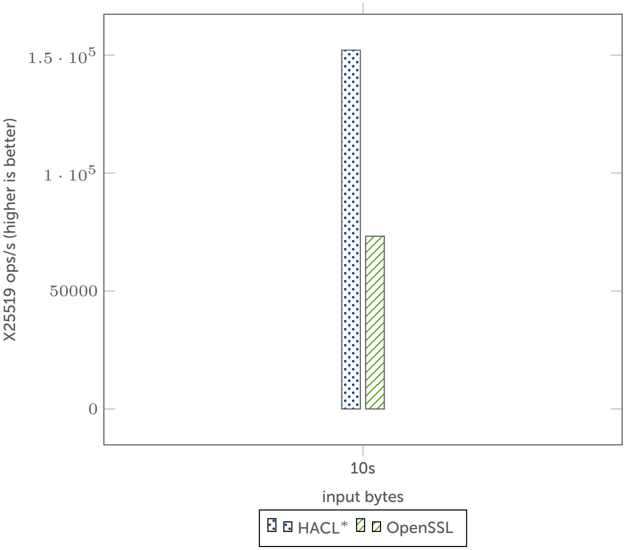
Disclaimer: these steps are supported by hand-written proofs.

It works: HACL*

Our crypto algorithms library. Available standalone, as an OpenSSL engine, or via the NaCl API.

- Implements Chacha20, Salsa20, Curve25519, X25519, Poly1305, SHA-2, HMAC
- 7000 lines of C code
- 23,000 lines of F* code
- Performance is comparable to existing C code (not ASM)
- Some bits are in the Firefox web browser!

Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche
HACL*: A Verified Modern Cryptographic Library
CCS'17



$1.5 \cdot 10^5$
$1 \cdot 10^5$
50000
0
X25519 ops/s (higher is better)
10s
input bytes
HACL*
OpenSSL

Part 2: a curated subset of C

We don't need (want) all the power of C

C: hard to reason about. Instead: a *curated* subset.

Let's start with machine integers.

Modeling fixed-width integers

The size of `int`, `long`, `long long` types is up to the compiler.
Two prevalent 64-bit data models:

- LLP64, for `long long` and `t*` = 64bits (Windows)
- LP64, for `long`, `long long` and `t*` = 64bits (mostly everyone else)

That's without looking at 32-bits.

Some sanity: C99 defines fixed-width integers like `uint32_t`, etc. in `<inttypes.h>`.

We use this revolutionary concept and expose fixed-width integer types in Low*. (Think: curated.)

A public service announcement about undefined behavior

```c
void f(long *x) {
  long y = x = 1;
  if (y < x) {
    // Integer overflow, abort
    return -1;
  }
  // ...
  return 0;
}
```

Now: **`gcc -O3 -fomit-frame-pointer -S`**:

```
_f:                                     ## @f
        .cfi_startproc
## BB#0:
        xorl    %eax, %eax
        retq
        .cfi_endproc
```

A challenge of appropriately modeling C

Even when *curating* the set of features, we need to make sure the model is restrictive enough.

From the C11 standard, §4.3.4:

> *EXAMPLE An example of undefined behavior is the behavior on integer overflow.*

In Low*, this means no wraparound addition for signed integer types.

For more context: “Undefined Behavior in 2017”
`https://blog.regehr.org/archives/1520`

Modeling allocations

We want explicit control over allocations. This means:

temporal safety an allocation is either in a stack frame, or on the heap (well-parenthesized vs. manually-managed)

spatial safety in-bounds array accesses, pointer arithmetic, no runtime length

And of course, no GC.

Another public service announcement about undefined behavior

```c
int f(int *x) {
  // C89-style
  int foo, bar = 0;
  int *buf = x + 16;

  if (x == NULL) {
    return -1;
  }
  // Proceed assuming x is non-null
  ...
}
```

Any decent C compiler will remove the `NULL` check. Ask the Linux developers.

From the C standard, §6.2.4:

> *If an object is referred to outside of its lifetime, the behavior is undefined.*

Modeling C data layout

We offer a predictible compilation scheme for inductives and structures.

```
type point = {
  x: Int32.t;
  y: Int32.t;
  z: Int32.t
}
```

```
typedef {
  int32_t x;
  int32_t y;
  int32_t z;
} point;
```

- Pass by value, pay the runtime price
- Pass by address, pay the verification price

Things we don't want from C

- Addresses everywhere: a value is a value, no unfettered **&** operator
- Unstructured control flow (Duff's device, goto)
- Preprocessor (use meta-programming instead)
- Concurrency (maybe one day)

Things we have but I won't talk about

Many more features in support of programmer productivity.

- compilation of inductives to tagged unions
- compilation of pattern-matches to if-then-else
- Whole-program monomorphization
- A loop combinator library

Part 3: defining Low*

An overview of Low* in three parts

1. A word on C99 integers
2. A C memory model and effects for Low*
3. A theory of C arrays

① A word on C99 integers

Modeling a C concept, concretely

```
module FStar.Int

(* Generic bounded integer model, based on int *)
let n = 32
let max_int (n:pos) : Tot int =
  pow2 (n-1) - 1
let min_int (n:pos) : Tot int =
  - (pow2 (n-1))
let fits (x:int) (n:pos): Tot bool =
  min_int n <= x && x <= max_int n
let size (x:int) (n:pos): Tot Type0 =
  b2t(fits x n)

(* The type of integers on n bits *)
type int_t (n:pos) = x:int{size x n}
```

Modeling a C concept, concretely (2)

```
(* Looking at the .fst file *)
module FStar.Int32

(* Instantiate the model *)
let n = 32
type t = | Mk: v:int_t n -> t

(* For proofs only *)
let v (x:t): GTot (int_t n) = x.v

(* Expose operations *)
let add (a:t) (b:t) : Pure t
  (requires (size (v a + v b) n))
  (ensures (fun c -> v a + v b = v c))
  = Mk (add (v a) (v b))
```

Modeling a C concept, concretely (3)

```
(* Looking at the .fsti file *)
module FStar.Int32

(* Abstract type *)
type t

val v (x:t) -> GTot (int_t n)

(* Lemmas re. v bijection *)

(* Expose operations *)
val add: (a:t) -> (b:t) -> Pure t
  (requires (size (v a + v b) n))
  (ensures (fun c -> v a + v b = v c))
```

Modeling a C concept, concretely (4)

Note that the type is *abstract*. This is essential.

- At compilation-time, we swap the model for native C integers, so we need **`Int32.t </: nat`**
- For soundness, the client must not be able to reason beyond the interface.

Note the shape of the post-condition of **`add`**: this is what guarantees proper overflow checking!

Fixed-width integers modules in Low*

Fixed-width machine integers are modeled in **`FStar.{U,}Int{8,16,32,64,128}.fst`**.

F* has integer literal syntax, in the style of C, e.g. **`16ul`** or **`32UL`**. Operators are suffixed with ^ to distinguish them from the operations on **`int`**.

```
module U32 = FStar.UInt32

let x = U32.( 0ul +%^ 1ul )

(* No unary minus! *)
let y = U32.( 0ul -%^ 1ul )
```

We do not have overloading (in the works).

② C memory model

Building on yesterday's discussion

We saw a heap memory model: a map from addresses to values. Reflects and specifies stateful operations in terms of pure counterparts.

```
(* Pseudo-code. *)
val (!): #a -> r:ref a -> ... a
  (requires (fun h -> True))
  (ensures (fun h0 x h1 -> x = sel h0 r))
```

- The heap appears in for pre- and post-conditions
- At each program point, the state of the memory is reflected by a map from addresses to values
- The state evolves with program execution and so does the map that reflects it

Making this a C heap

We need to restrict yesterday's model.

- **yesterday** automatic memory management (GC)
- **today** manual memory management (life becomes harder)

We want to enforce temporal safety: no uninitialized memory accesses; no use-after-free; no double free.

Making this a C heap (2)

```
(* Reserving 0 for the NULL pointer, cf. later *)
type addr = x:nat { x > 0 }
type heap = {
  next_addr: addr;
  memory: addr -> Tot (option (a: Type0 & ... & a))
}
type ref (a:Type0): Type0 = { addr: addr; init: a; ...  }

let free #a ... h r = {
  h with memory =
  fun r' -> if r' = r.addr then None else h.memory r'
}

(* Should be named: live *)
let contains #a h ... r =
  Some? (h.memory r.addr) /\
  ...
```

Making this a C heap (3)

- A live address points to **`Some _`**. A de-allocated or un-allocated address points to **`None`**.
- We do not allow the client to observe address reuse (important).

This heap model propagates all the way to the stateful operator, now requiring:

```
(* Pseudo-code. *)
val (!): #a -> r:ref a -> ... a
  (requires (fun h -> live h r))
  (ensures (fun h0 x h1 -> x = sel h0 r))
```

Making this a C heap (3)

Demo!

- A live address points to **`Some _`**. A de-allocated or un-allocated address points to **`None`**.
- We do not allow the client to observe address reuse (important).

This heap model propagates all the way to the stateful operator, now requiring:

```
(* Pseudo-code. *)
val (!): #a -> r:ref a -> ... a
  (requires (fun h -> live h r))
  (ensures (fun h0 x h1 -> x = sel h0 r))
```

Making this a C stack and heap

We need to go beyond a simple heap. We need structure.

About the stack.

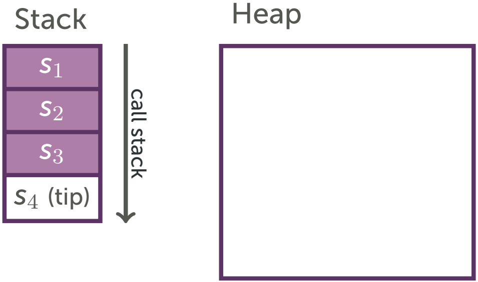
- Stack frames conceptually form a linked list
- Stack frames are pushed and popped on the call stack as execution progresses
- A function can only allocate within its own stack frame and should leave its calling stack intact
- A stack reference is only live as long as its stack frame is live

Making this a C stack and heap (2)

Stack references behave differently. Their lifetime is tied to their enclosing stack frame. They are allocated and de-allocated in bulk.

We are going to use *regions* to model the C stack frames, a classic framework from the 90s / 2000s.

A C memory model with regions: HyperStack



- The memory is partitioned into stack (s_i, left) and heap (the rest).
- Stack regions nest linearly. The topmost stack frame is the tip.
- Allocation is only possible in white regions.

Modeling the stack with regions (1)

The stack frames s_i are distinguished in the model. They are transient and may disappear. Conversely, the root and its sub-regions are eternal.

```
module HS = FStar.HyperStack

let root: HS.rid = HS.root

let _ =
  assert (ST.is_eternal_region root /\
  ~ (Monotonic.HyperStack.is_stack_region root))
```

Modeling the stack with regions (2)

Remember that we're still crafting the model.

The mechanics of the call stack are handled by combinators **push_frame** and **pop_frame**.

```
let push_frame (_:unit): Internal unit
  (requires (fun m -> True))
  (ensures (fun (m0:mem) _ (m1:mem) -> fresh_frame m0 m1))
  = ...

let fresh_frame (m0:mem) (m1:mem) =
  not (Map.contains m0.h m1.tip) /\
  parent m1.tip = m0.tip         /\
  m1.h == Map.upd m0.h m1.tip Heap.emp
```

The programmer does not have access to **Internal**.

Modeling the stack with regions (3)

```
let alloca_post (#a:Type) ...
  (init:a) (m0:mem)
  (s:mreference a ...{is_stack_region (frameOf s)}) (m1:mem)
  = is_stack_region m0.tip                     /\
    Map.domain m0.h == Map.domain m1.h /\
    m0.tip = m1.tip                            /\
    frameOf s = m1.tip                         /\
    HS.fresh_ref s m0 m1                       /\
    m1 == HyperStack.upd m0 s init
```

The liveness predicate is refined to capture manual lifetime in the heap, or region-based lifetime on the stack.

Modeling the stack with regions (4)

Found in **FStar.HyperStack.ST.fst**, **Stack** is the effect of well-formed client programs.

```
effect Stack (a:Type) (pre:st_pre) (post: (mem -> Tot (st_post a))) =
  STATE a (fun (p:st_post a) (h:mem) ->
    pre h /\ (forall a h1.
      (pre h /\ post h a h1 /\ equal_domains h h1) ==> p a h1))

let equal_domains (m0:mem) (m1:mem) =
  m0.tip = m1.tip
  /\ Set.equal (Map.domain m0.h) (Map.domain m1.h)
  /\ (forall r. Map.contains m0.h r ==>
    Heap.equal_dom (Map.sel m0.h r) (Map.sel m1.h r))
```

Preserves the layout of the stack and doesn't allocate in any caller frame.

Modeling the stack with regions (4)

Found in **FStar.HyperStack.ST.fst**, **Stack** is the effect of well-formed client programs.

preservation of the stack structure

```
effect Stack (a:Ty                                        post a))) =
  STATE a (fun (p
    pre h /\ (forall a h1.
      (pre h /\ post h a h1 /\ equal_domains h h1) ==> p a h1))

let equal_domains (m0:mem) (m1:mem) =
  m0.tip = m1.tip
  /\ Set.equal (Map.domain m0.h) (Map.domain m1.h)
  /\ (forall r. Map.contains m0.h r ==>
    Heap.equal_dom (Map.sel m0.h r) (Map.sel m1.h r))
```

Preserves the layout of the stack and doesn't allocate in any caller frame.

Modeling the stack with regions (4)

Found in **`FStar.HyperStack.ST.fst`**, **`Stack`** is the effect of well-formed client programs.

```
effect Stack (a:Type) (pre:st_pre) (post: (mem -> Tot (st_post a))) =
  STATE a (fun (p:st_post a) (h:mem) ->
    pre h /\ (forall a h1.
      (pre h /\ post h a h1 /\ equal_domains h h1) ==> p a h1))

let equal_domains (m0:mem) (m1:mem) =
  m0.tip = m1.tip
  /\ Set.equal (Map.domain m0.h) (Map.domain m1.h)
  /\ (for                                   =>
    Heap.                                 Map.sel m1.h r))
```

the tip remains the same

Preserves the layout of the stack and doesn't allocate in any caller frame.

Modeling the stack with regions (4)

Found in **FStar.HyperStack.ST.fst**, **St** effect of well-formed client programs.

```
effect Stack (a:Type) (pre:st_pre) (post: (mem -> Tot (st_post a))) =
  STATE a (fun (p:st_post a) (h:mem) ->
    pre h /\ (forall a h1.
      (pre h /\ post h a h1 /\ equal_domains h h1) ==> p a h1))

let equal_domains (m0:mem) (m1:mem) =
  m0.tip = m1.tip
  /\ Set.equal (Map.domain m0.h) (Map.domain m1.h)
  /\ (forall r. Map.contains m0.h r ==>
    Heap.equal_dom (Map.sel m0.h r) (Map.sel m1.h r))
```

Preserves the layout of the stack and doesn't allocate in any caller frame.

Recap

- Talking about the C memory model in F*.
- Reflect the semantics of stateful operations in the memory model
- `push_frame` and `pop_frame` control the call stack
- Well-formed stateful operations are in the `Stack` effect

③ A theory of C arrays

What do we need to capture…

…in order to be a faithful model of C arrays?

lifetime of variables, either tied to a stack frame, or manually-managed

spatial safety, guaranteeing access within bounds

functional model C arrays as a sequence

The `Buffer` library

Two flavors of buffers:

- stack-allocated buffers (think **`alloca(3)`**), lifetime tied to stack frame
- heap-allocated buffers (think **`malloc`**), must be **`free`**'d

Needs to be a faithful model of C:

- no pointer arithmetic if pointer is not live
- no zero-length allocations on the stack (use **`NULL`** instead)
- length cannot be computed at run-time
- only pointer one-past allowed

Beware of undefined behaviors!

The `Buffer` library

Demo!

Two flavors of buffers:

- stack-allocated buffers (think **`alloca(3)`**), lifetime tied to stack frame
- heap-allocated buffers (think **`malloc`**), must be **`free`**'d

Needs to be a faithful model of C:

- no pointer arithmetic if pointer is not live
- no zero-length allocations on the stack (use **`NULL`** instead)
- length cannot be computed at run-time
- only pointer one-past allowed

Beware of undefined behaviors!

The Buffer library (1)

A buffer is modeled as a reference to a sequence.

```
noeq
type buffer (a: Type0) : Type0 =
| Null
| Buffer:
  (max_length: U32.t { U32.v max_length > 0 } ) ->
  (content: HST.reference (vec a (U32.v max_length))) ->
  (idx: U32.t) ->
  (length: U32.t { U32.v idx + U32.v length <= U32.v max_length
  buffer' a

(* In a given heap *)
let as_seq #a h b =
  match b with
  | Null -> Seq.createEmpty
  | Buffer max_len content idx len ->
    Seq.slice ...
```

The Buffer library (2)

```
let alloc_post_common
  (#a: Type) (r: HS.rid) (len: nat) (b: buffer a)
  (h0 h1: HS.mem): GTot Type0
= b ‘unused_in‘ h0 /\ live h1 b /\
  (not (g_is_null b)) /\ frameOf b == r /\
  Map.domain h1.HS.h ‘Set.equal‘ Map.domain h0.HS.h /\
  h1.HS.tip == h0.HS.tip /\ length b == len /\
  modifies_0 h0 h1

val alloca (#a: Type) (init: a) (len: U32.t):
HST.StackInline (buffer a)
  (requires (fun h -> U32.v len > 0))
  (ensures (fun h b h’ ->
    alloc_post_common h.HS.tip (U32.v len) b h h’ /\
    as_seq h’ b == Seq.create (U32.v len) init
  ))
```

The Buffer library (3)

```
val malloc (#a: Type) (r: HS.rid) (init: a)
  (len: U32.t): HST.ST (buffer a)
  (requires (fun h -> HST.is_eternal_region r /\ U32.v len > 0))
  (ensures (fun h b h’ ->
    alloc_post_common r (U32.v len) b h h’ /\
    as_seq h’ b == Seq.create (U32.v len) init /\
    freeable b
  ))

val free (#a: Type) (b: buffer a): HST.ST unit
  (requires (fun h0 -> live h0 b /\ freeable b))
  (ensures (fun h0 _ h1 ->
    (not (g_is_null b)) /\
    Map.domain h1.HS.h ‘Set.equal‘ Map.domain h0.HS.h /\
    h1.HS.tip == h0.HS.tip /\
    modifies_1 b h0 h1 /\
    HS.live_region h1 (frameOf b)
  ))
```

The `Buffer` library (4)

Other bits from **`LowStar.Buffer.fsti`**:

```
val is_null (#a: Type) (b: buffer a):
  HST.Stack bool ...
val offset (#a: Type) (b: buffer a) (i: U32.t):
  HST.Stack (buffer a) ...
val index (#a: Type) (b: buffer a) (i: U32.t):
  HST.Stack a ...
val upd (#a: Type) (b: buffer a) (i: U32.t) (v: a):
  HST.Stack unit ...
```

The `Buffer` library (4)

Other bits from `LowStar.BufferOps.fsti`:

```
(* e1.(e2) *)
let op_Array_Access = ...

(* e1.(e2) <- e3 *)
let op_Array_Assignment = ...
```

You want to **open** this module in your code.

Reasoning with abstract modifies clauses

The heap model precisely defines what happens for allocations, de-allocations, modifications, etc.

However, this is too precise and we oftentimes want more abstract reasoning.

The modifies clauses library provides a set of abstract, composable predicates to talk about heap modification.

The `Modifies` library (1)

`LowStar.Modifies.fsti` defines an abstract type of memory locations which form a monoid.

```
val loc: Type0
val loc_none: loc
val loc_union (s1 s2: loc) : GTot loc
```

Buffers can be injected into locations (and so can regions and individual references).

```
val loc_buffer (#t: Type) (b: B.buffer t): GTot loc
```

The `Modifies` library (2)

`LowStar.Modifies.fsti` also defines an inclusion relation. The programmer reasons at the granularity of individual buffers or regions.

```
val loc_includes_region_buffer
  (#t: Type)
  (s: Set.set HS.rid)
  (b: B.buffer t)
: Lemma
  (requires (Set.mem (B.frameOf b) s))
  (ensures (loc_includes (loc_regions s) (loc_buffer b)))
  [SMTPat (loc_includes (loc_regions s) (loc_buffer b))]
```

The Modifies library (3)

Equipped with locations and the **includes** relation, the module defines modifies clauses.

```
val modifies (s: loc) (h1 h2: HS.mem): GTot Type0

val modifies_buffer_elim (#t1: Type) (b: B.buffer t1)
  (p: loc) (h h': HS.mem): Lemma
    (requires (
      loc_disjoint (loc_buffer b) p /\
      B.live h b /\ ... /\
      modifies p h h'
    ))
    (ensures (
      B.live h' b /\ (
      B.as_seq h b == B.as_seq h' b
    )))
    [ SMTPat ... ]
```

The **Modifies** library (4)

Combinators from **LowStar.Buffer** are tagged with appropriate **modifies** clauses.

```
val upd (#a: Type) (b: buffer a) (i: U32.t) (v: a):
  HST.Stack unit
    (requires (fun h -> live h b /\ U32.v i < length b))
    (ensures (fun h _ h' ->
      (not (g_is_null b)) /\
      modifies_1 b h h' /\
      live h' b /\
      as_seq h' b == Seq.upd (as_seq h b) (U32.v i) v
    ))
```

This equational theory allows us to reason efficiently about stateful programs.

High-level verification for low-level code

For code, the programmer:

- opts in the Low* effect to model the C stack and heap;
- uses low-level libraries for arrays and structs;
- leverages combinator libraries to get C loops;
- meta-programs first-order code;
- relies on data types sparingly.

For proofs and specs, the programmer:

- can use all of F*,
- prove memory safety, correctness, crypto games, relying on
- erasure to yield a first-order program.

Motto: the code is low-level but the verification is not.

Part 4: KreMLin

A compiler from F* to *readable* C

The KreMLin facts:

- about 14,000 lines of OCaml
- carefully engineered to generate readable C code
- essential for integration into existing software.

Design:

- relies on the same Letouzey-style erasure from F*
- one internal AST with several compilation passes
- abstract C grammar + pretty-printer
- small amounts of hand-written C code (host functions)

So far, about 120k lines of C generated.

Tooling support: killing abstraction

Abstraction	= good for verification
No Abstraction	= good for compilation

- At the module level (**`-bundle`**)
- At the function level (**`inline_for_extraction`**)

This triggers enough compiler optimizations to fulfill the original promise.

Tooling support: data types

Or: "programmer productivity".

- Tuples, inductives (tagged unions) are supported
- Four (!) different compilation schemes
- Use at your own risk (MSVC! CompCert! x86 ABI!)
- Requires:
  - monomorphization
  - implementation in KreMLin of recursive equality predicates
  - mutual recursion; forward declarations

Tooling support: misc

- Type abbreviations
- C loops (syntactic closures for bodies)
- Removal of `uu___`
- Optimal visibility
- Removal of unused function and data types arguments
- Passing structures by reference

Learning KreMLin

Our work-in-progress is online:

`https://fstarlang.github.io/lowstar/html/`

Coming up: 90-minute lab session on Low* programming and KreMLin.

Consider **`opam install`**'ing KreMLin.

Part 5: λow*, a model of Low*

Removal of ghost code (1)

- `Ghost` is an F*effect
- Used only for proofs, i.e. specifications ("contagious")
- $\boxed{\text{ABSOLUTELY}}$ does not fit in Low*
- Removed via a logical relations argument

Removal of erased (2)

- `erased a` is a computationally-irrelevant value
- unlike `Ghost`, can be used within code
- used for the `log` of operations, say, in Chacha
- the value can be used in specifications via:
  `val reveal: #a -> erased a -> GTot a`

All erased values, being irrelevant, can be compiled to `()` (ML).
We remove them via a whole-program analysis.

From monadic to effectful semantics (3)

Explicitly-monadic F* (POPL'17) can be translated to a primitive state semantics.

Danel Ahman, Cătălin Hriţcu, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy.
Dijkstra Monads for Free
POPL'17

From the user-facing Low* to λow* (4)

A series of (unproven) transformations for programmer convenience.

- going from an expression language to a statement language
- compilation of pattern-matching
- structures by value
- etc.

These are performed by the KreMLin tool.

The core lambda-calculus: λow^*

$$
\begin{aligned}
\tau &::= \mathsf{int} \mid \mathsf{unit} \mid \{\overrightarrow{f = \tau}\} \mid \mathsf{buf}\ \tau \mid \alpha \\
v &::= x \mid n \mid () \mid \{\overrightarrow{f = v}\} \mid (b, n, \overrightarrow{f}) \\
e &::= \mathsf{let}\ x : \tau = \mathsf{readbuf}\ e_1\ e_2\ \mathsf{in}\ e \mid \mathsf{let}\ \_ = \mathsf{writebuf}\ e_1\ e_2\ e_3\ \mathsf{in}\ e \\
&\mid\ \ \mathsf{let}\ x = \mathsf{newbuf}\ n\ (e_1 : \tau)\ \mathsf{in}\ e_2 \mid \mathsf{subbuf}\ e_1\ e_2 \\
&\mid\ \ \mathsf{let}\ x : \tau = \mathsf{readstruct}\ e_1\ \mathsf{in}\ e \mid \mathsf{let}\ \_ = \mathsf{writestruct}\ e_1\ e_2\ \mathsf{in}\ e \\
&\mid\ \ \mathsf{let}\ x = \mathsf{newstruct}\ (e_1 : \tau)\ \mathsf{in}\ e_2 \mid e_1 \triangleright f \\
&\mid\ \ \mathsf{withframe}\ e \mid \mathsf{pop}\ e \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\
&\mid\ \ \mathsf{let}\ x : \tau = d\ e_1\ \mathsf{in}\ e_2 \mid \mathsf{let}\ x : \tau = e_1\ \mathsf{in}\ e_2 \mid \{\overrightarrow{f = e}\} \mid e.f \mid v \\
P &::= \cdot \mid \mathsf{let}\ d = \lambda y : \tau_1.\ e : \tau_2, P
\end{aligned}
$$

About λow^*:

- a type system without progress
- in a simulation with the original F* program
- standard substitutive semantics.

The judgements of λow*

Typing judgement:

$$\Gamma_P; \Sigma; \Gamma \vdash e : \tau$$

where:

- Γ_P is the set of global program definitions
- Σ is the store typing
- Γ is the local context

The judgements of λow* (2)

Reduction semantics:

$$P \vdash (H, e) \xrightarrow{\ell} (H', e')$$

where:

- ℓ is the set of trace events
- H is the stack of frames

The judgements of λow* (2)

Reduction semantics:

$$P \vdash (H, e) \xrightarrow{\ell} (H', e')$$

where:

- ℓ is the set of trace events
- H is the stack of frames

Let's talk about traces!

What are we protecting against

- We want to guard against some memory and timing side-channels
- Our secret data is at an abstract type
- By using abstraction, we can control what operations we allow on secret data

Abstraction to the rescue

Our module for secret integers exposes a handful of audited, carefully-crafted functions that we trust have secret-independent traces.

```
(* limbs only ghostly revealed as numbers *)
val v : limb -> Ghost nat

val eq_mask: x:limb -> y:limb ->
  Tot (z:limb{if v x <> v y then v z = 0 else v z = pow2 26 - 1})
```

By construction, the programmer cannot use a `limb` for branching or array accesses.

What we show

We model trace events as part of our reduction.

$$\ell ::= \cdot \mid \mathsf{read}(b, n, \overrightarrow{f}) \mid \mathsf{write}(b, n, \overrightarrow{f}) \mid \mathsf{brT} \mid \mathsf{brF} \mid \ell_1, \ell_2$$

Note: this does not rule out ALL side channels!

Secret-independence: an intuition

A type-indexed relation $v_1 \equiv_\tau v_2$ over values:

$$
\begin{aligned}
&n \equiv_{\mathsf{int}} n \\
&v_1 \equiv_a v_2 \\
&\ldots
\end{aligned}
$$

Intuition: terms are related if they only differ on sub-terms at secret types.

Main theorem: functions, when applied to related values in related stores, have related reductions and emit the same traces.

Note: this only goes up to CompCert Clight

Theorem (Secret independence)

Given

1. *A program well-typed against a secret interface,* Γ_s*, i.e,* $\Gamma_s, \Gamma_P; \Sigma; \Gamma \vdash (H, e) : \tau$*, where e is not a value.*
2. *A well-typed implementation of the* Γ_s *interface,* $\Gamma_s; \Sigma; \cdot \vdash_\Delta P_s$*, such that* P_s *is equivalent modulo secrets.*
3. *A pair* (ρ_1, ρ_2) *of well-typed (related) substitutions for* Γ*.*

There exists $\ell, \Sigma' \supseteq \Sigma, \Gamma', H', e'$ *and a pair* (ρ_1', ρ_2') *of well-typed substitutions for* Γ'*, such that*

1. $P_s, P \vdash (H, e)[\rho_1] \rightarrow_\ell^+ (H', e')[\rho_1']$ *if and only if,* $P_s, P \vdash (H, e)[\rho_2] \rightarrow_\ell^+ (H', e')[\rho_2']$*, and*
2. $\Gamma_s, \Gamma_P; \Sigma'; \Gamma' \vdash (H', e') : \tau$

Next step: C^*, an imperative language

This our next intermediary language.

- statement language
- not substitutive semantics (stack of contexts with holes)
- expressions are pure
- deterministic

We relate λow* programs to C^* programs via a simulation.

A glimpse of the reduction rules

From λow*:

$$\frac{}{P \vdash (H, \text{if } 0 \text{ then } e_1 \text{ else } e_2) \rightarrow_{\mathsf{brF}} (H, e_2)} \text{ LIfF}$$

From C*:

$$\frac{[\![\hat{e}]\!]_{(V)} = 0}{\hat{P} \vdash (S, V, \text{if } \hat{e} \text{ then } \overrightarrow{s_1} \text{ else } \overrightarrow{s_2}; \overrightarrow{s}) \rightsquigarrow_{\mathsf{brF}} (S, V, \overrightarrow{s_2}; \overrightarrow{s})} \text{ CIfF}$$

Theorem

The C^* program $\hat{P}$ terminates with trace ℓ and return value v, i.e., $\hat{P} \vdash ([], V, \overrightarrow{s}; \textit{return } \hat{e}) \xrightarrow{\ell,*} ([], V', \textit{return } v)$ if, and only if, so does the λow^* program: $P \vdash (\{\}, e[V]) \xrightarrow{\ell,*} (H', v)$; and similarly for divergence.

Next step: C* to CompCert Clight

We encode the trace preservation using builtins that generate trace events.

Two relevant bits:

- hoisting, which changes the memory layout (abstract traces)
- struct passing, which changes the memory accesses (two passes)

Final step: Clight to assembly

Some possible approaches:

- instrument CompCert (Barthe *et al.*)
- Use Vellvm (Zdancewicz *et al.*)