Verified low-level programming embedded in $F^*$

Jonathan Protzenko
Nikhil Swamy

Project Everest

Microsoft Research
Microsoft Research

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

\textbf{latency} is correlated with the performance of asymmetric cryptography

\textbf{throughput} is correlated with the performance of symmetric cryptography

Efficiency is essential. That’s why everyone \textit{binds} to some cryptographic library written in \texttt{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.
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 attack by corrupted issue is not considered

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

Hanno Boeck via RT rt.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

These heavily optimized C implementations have bugs.

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

David Benjamin via RT rt at openssl.org

These produce wrong results, the other three also on OpenSSL Security Advisory [10 Nov 2016]

Chacha20/Poly1305 heap-buffer-overflow (CVE-2016-7054)

Severit: High

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

Protzenko / Swamy — MSR
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.
Write C programs in F* and prove them correct!
The essence of Low* in Emacs

```
module Spec.Poly1305

let prime = pow2 130 - 5
let 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
```
Low\* Poly1305 compiled to C

Protzenko / Swamy — MSR

May. 22\textsuperscript{nd}, 2018 9 / 81
Low*: a low-level subset of F*

Disclaimer: these steps are supported by hand-written proofs.
Low*: a low-level subset of F*

F* \rightarrow \text{EMF*} \rightarrow \text{Low*} \approx \text{erase} \rightarrow 1^\text{st}-\text{order EMF}

Kremlin

\text{Clight} \rightleftharpoons \text{C*} \rightleftharpoons \text{Low*}

\text{print} \rightleftharpoons \text{hoist} \rightleftharpoons \text{compile} \rightarrow \text{Exe}

GCC/Clang/CompCert/MSVC

\textbf{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
X25519 ops/s (higher is better)

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:

```assembly
_f:         ## @f
    .cfi_startproc

    ## BB#0:
    xorl %eax, %eax
    retq
    .cfi_endproc
```

Protzenko / Swamy — MSR
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.

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

```c
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)
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
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 \texttt{Int32.t} 
  \texttt{</: nat}
- For soundness, the client must *not* be able to reason beyond the interface.

Note the shape of the post-condition of \texttt{add}: this is what guarantees proper overflow checking!
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.

```fstar
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).
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. *)

```plaintext
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 *)

```fsharp
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. *)

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

Demo!
Making this a C heap (3)

- A live address points to \texttt{Some \_}. A de-allocated or un-allocated address points to \texttt{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:

\begin{verbatim}
(* Pseudo-code. *)
val (!): #a -> r:ref a -> ... a
(requires (fun h -> live h r))
(ensures (fun h0 x h1 -> x = sel h0 r))
\end{verbatim}
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
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 \((s_i, \text{left})\) 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**.

```f
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.

```fsharp
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.
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.

```fstar
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))

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))
```

Preserves the **layout** of the stack and **doesn’t allocate** in any caller frame.
Found in `FStar.HyperStack.ST.fst`, **Stack** is the effect of well-formed client programs.

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

```fsharp
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 h r.
  (forall a.
  Map.contains (Map.sel m0.h a) r) ==> Heap.equal (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.

```fsharp
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

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!
A buffer is modeled as a reference to a sequence.

```f*
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)

```ocaml
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)

```f
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:

```fsharp
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.Buffer0ps.fsti:

```fsharp
(* 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.

```fsharp
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).

```fsharp
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.

```fsharp
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.

```fsharp
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 \ ... \  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.

```fsharp
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: \( \lambda \text{ow}^* \), a model of Low*
Removal of ghost code (1)

- Ghost is an $F^\text{*}$ effect
- Used only for proofs, i.e. specifications (“contagious”)
- Absolutely does not fit in Low$^\text{*}$
- 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*. 
Explicitly-monadic F* (POPL’17) can be translated to a primitive state semantics.

Daniel Ahman, Cătălin Hriţcu, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy. Dijkstra Monads for Free
POPL’17
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: \( \lambda \text{ow} \)

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

About \( \lambda \text{ow} \):

- a type system without progress
- in a simulation with the original F* program
- standard substitutive semantics.
The judgements of \( \lambda \text{ow}^* \)

Typing judgement:

\[ \Gamma_P; \Sigma; \Gamma \vdash e : \tau \]

where:

- \( \Gamma_P \) is the set of global program definitions
- \( \Sigma \) is the store typing
- \( \Gamma \) is the local context
The judgements of $\lambda$ow* (2)

Reduction semantics:

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

where:

- $\ell$ is the set of trace events
- $H$ is the stack of frames
The judgements of $\lambda$ow* (2)

Reduction semantics:

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

where:

- $\ell$ 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.

\[ \ell ::= \cdot \mid \text{read}(b, n, f) \mid \text{write}(b, n, f) \mid \text{brT} \mid \text{brF} \mid \ell_1, \ell_2 \]

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:

\[
\begin{align*}
n & \equiv_{\text{int}} n \\
v_1 & \equiv_{a} v_2 \\
\ldots
\end{align*}
\]

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, $\Gamma_s$, 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 $\Gamma_s$ interface, $\Gamma_s; \Sigma; \cdot \vdash \Delta P_s$, such that $P_s$ is equivalent modulo secrets.

3. A pair $(\rho_1, \rho_2)$ of well-typed (related) substitutions for $\Gamma$.

There exists $\ell, \Sigma' \supseteq \Sigma, \Gamma', H', e'$ and a pair $(\rho'_1, \rho'_2)$ of well-typed substitutions for $\Gamma'$, such that

1. $P_s, P \vdash (H, e)[\rho_1] \rightarrow^+ (H', e')[\rho'_1]$ if and only if, $P_s, P \vdash (H, e)[\rho_2] \rightarrow^+ (H', e')[\rho'_2]$, 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 $\lambda_{ow^*}$:

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

From $C^*$:

$$\hat{P} \vdash (S, V, \text{if } \hat{e} \text{ then } \vec{s}_1 \text{ else } \vec{s}_2; s) \rightsquigarrow_{brF} (S, V, \vec{s}_2; s)$$
Theorem

The C* program $\hat{P}$ terminates with trace $\ell$ and return value $v$, i.e., $\hat{P} \vdash (\langle \rangle, V, s \leftarrow \text{return } \hat{e}) \xrightarrow{\ell,*} (\langle \rangle, V', \text{return } v)$ if, and only if, so does the $\lambda$ow* 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.)