Translating from F* to C
a progress report

P. Wang
MIT

K. Bharghavan, J-K. Zinzindohoué
INRIA

A. Anand
Cornell

C. Fournet, B. Parno, J. Protzenko, A. Rastogi, N. Swamy
Microsoft

A fashion phenomenon?

The hot new thing these days is...

A fashion phenomenon?

The hot new thing these days is...

translating to C!

Fig 1. – Two hipster C hackers with beards

Bedrock!

- deep embedding of C into Coq
- prove functional correctness (and memory safety) using manual proofs
- data structures, threads (first-class pointers)

```
Definition swap := bmodule {{
  bfunction "swap" [st ~> Ex fr : hprop,
    Ex a : nat, Ex b : nat,
    ![ st#R0 ==> a * st#R1 ==> b * ![fr] ] st
    /\ st#Rret @@ (st' ~>
      ![ st#R1 ==> a * st#R0 ==> b
         * ![fr] ] st') ] {
    R2 <- $[R0];;
    $[R0] <- $[R1];;
    $[R1] <- R2;;
    Goto Rret
  }
}}.

Theorem swapOk : moduleOk swap.
  structured; sep.
Qed.
```

Figure 2. A Bedrock function implementing pointer swapping

Cogent!

At ICFP this year.

- a DSL with linear types, polymorphism
- generates a shallow embedding into Isabelle + proof
- systems code (e.g. file systems)

```
type ExSt
type UArray a
type Opt a = <None () | Some a>
type Node = #{mbuf:Opt Buf, ptr:U32, fr:U32, to:U32}
type Acc = (ExSt, FsSt, VfsInode)
type Cnt = (UArray Node,
  (U32, Node, Acc, U32, UArray Node) -> (Node, Acc))

uarray_create: all (a :< E). (ExSt, U32)
  -> <Success (ExSt, UArray a) | Err ExSt>

ext2_free_branch: (U32, Node, Acc, U32)
  -> (Node, Acc, <Expd Cnt | Iter ()>
ext2_free_branch (depth,nd,(ex,fs,inode),mdep) =
  if depth + 1 < mdep
    then
      uarray_create[Node] (ex,nd.to-nd.fr) !nd
      | Success (ex, children) =>
        let nd_t { mbuf } = nd
        and (children, (ex, inode, _, mbuf)) =
          uarray_map_no_break #{
            arr  = children,
            f    = ext2_free_branch_entry,
            acc  = (ex, inode, node_t.fr, mbuf),
            ... } !nd_t
        and nd = nd_t { mbuf }
        in (nd, (ex, fs, inode),
          Expd (children, ext2_free_branch_cleanup))
      | Err ex -> (nd, (ex,fs,inode), Iter ())
    else ...
```

Figure 2: Cogent example

Others

- Idris has a C backend + experimental C++11 backend
- Ivory is a DSL in Haskell that generates memory-safe C code

Others

- Idris has a C backend + experimental C++11 backend
- Ivory is a DSL in Haskell that generates memory-safe C code
- F* wants to be hip. F* will generate C too.

We actually have reasons!

Everest: VERifiEd Secure Transport

> *Even before recent headline-grabbing attacks like HeartBleed, FREAK, and Logjam, entire papers were published just to summarize all of the academically "interesting" ways TLS implementations have been broken, without even getting into "boring" vulnerabilities like buffer overflows and other basic coding mistakes.*

Reminder: TLS = « the S in HTTPS »

Everest

- A collaboration with our friends at INRIA and MSR Cambridge
- Prove TLS *cryptographically* sound
- Generate shippable code

Everest

- A collaboration with our friends at INRIA and MSR Cambridge
- Prove TLS *cryptographically* sound
- Generate shippable code

Yes, this is ambitious.

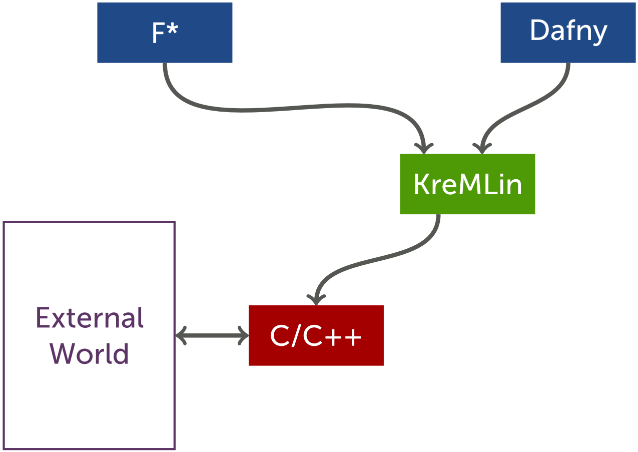
Back to C; why 😱?!

```
'FACE SCREAMING IN FEAR' (U+1F631)
```

Performance Cryptography = hand-optimized machine integers. OCaml = $n-1$ bits.

Social reasons OCaml runtime = hard sell.

The architecture

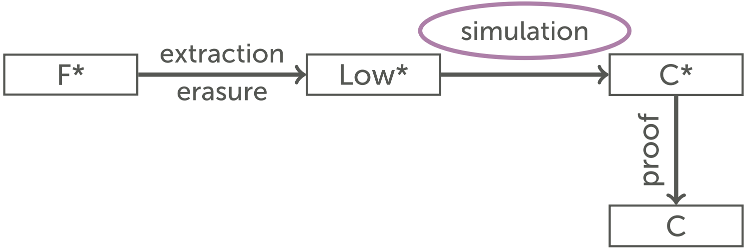


Things to cover

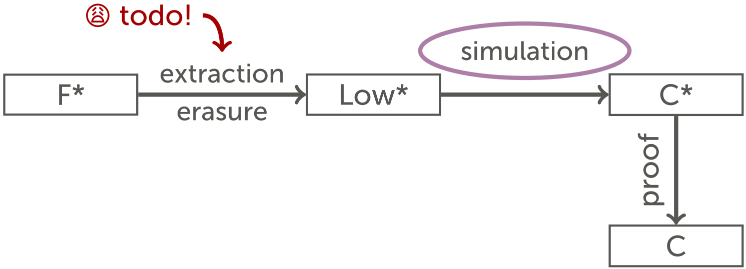
1. Theory
2. F* code & libraries
3. Overview of the tool & demo

An overview of the theory

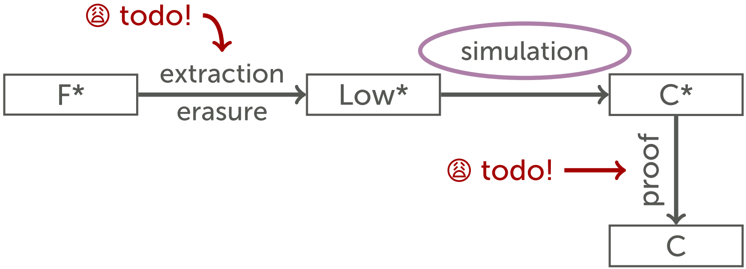
The pipeline



The pipeline



The pipeline



Low*

Low* is a low-level, first-order fragment of F*.

- Offers a limited subset of C's power: stack-allocated buffers and locally mutable variables
- Code is written against a `HyperStack` library
- Suitable pre- and post-conditions ensure memory safety
- If the code ends up in Low*, it can be translated to C.

A word about Low*

- An expression language
- Semantics by substitution
- Frame = buffer id $\rightarrow$ list of values

if le then le else le	conditional
let $x : t = f\ le$ in le	application
let $x : t =$ readbuf $le\ le$ in le	read buffer
let $\_ =$ writebuf $le\ le\ le$ in	write buffer
let $x =$ newbuf $n\ (le : t)$ in le	new buffer
subbuf $le\ le$	sub-buffer
withframe le	with-frame

$$\frac{lp(f) = \lambda y : t_1.\ le_1 : t_2}{lp \vdash (H, \text{let } x : t = f\ v \text{ in } le) \rightarrow (H, \text{let } x : t = [v/y]le_1 \text{ in } le)}\ \textsc{App}$$

A word about C*

- A statement language
- Semantics with continuation contexts (telescope)
- Frame = location to values + immutable values
- Pointer arithmetic for buffers

$s ::=$		statements
	$t\ x = e$	immutable variable declaration
	$t\ x[n] = \{e\}$	array declaration (w. initial value)
$e ::=$		expressions
	n	integer constant
	$()$	unit value
	x	variable

$$\frac{p(f) = \mathsf{fun}\ (y : t_1) : t_2\ \{\ ss_1\ \} \qquad [\![e]\!]_{(p,V)} = v}{p \vdash (S, V, t\ x = f\ e; ss) \rightsquigarrow (S; (\bot, V, t\ x = \Box; ss), V[y \mapsto v], ss_1)}\ \textsc{Call}$$

Low* to C*

For any Low* expr. e and C* statements $s = \mathrm{trans}(e)$:

safety: if e is safe, then s is safe.

refinement:

$$\begin{array}{ccc} e & \rightsquigarrow^n & \exists\, e' \\ \wr_R & & \wr_R \\ s & \rightsquigarrow & s' \end{array}$$

Low* to C*

For any Low* expr. e and C* statements $s = \text{trans}(e)$:

safety: if e is safe, then s is safe.

refinement:

$$
\begin{array}{ccc}
e & \rightsquigarrow^{n} & \exists\, e' \\
\wr_R & & \wr_R \\
s & \rightsquigarrow & s'
\end{array}
$$

Any reduction step of the C* program corresponds to an admissible sequence of reduction steps for the Low* program.

Low* to C*

For any Low* expr. e and C* statements $s = \text{trans}(e)$:

safety: if e is safe, then s is safe.

refinement:

$$\begin{array}{ccc} e & \rightsquigarrow^n & \exists\, e' \\ \wr_R & & \wr_R \\ s & \rightsquigarrow & s' \end{array}$$

Any reduction step of the C* program corresponds to an admissible sequence of reduction steps for the Low* program.

The C* program only does “things” allowed by the original semantics of Low*.

Low* to C*

But, this is hard ($n = 0$; stuttering). Instead, we use the CompCert style:

$$\begin{array}{ccc} e & \rightsquigarrow & e' \\ \wr_R & & \wr_R \\ s & \rightsquigarrow^n & \exists\, s' \end{array}$$

Works only if C* is deterministic (yes) and Low* is safe (yes).

Side channels

Right now: safety and observational equivalence of traces

Next: side-channel resistance using parametricity

$$
\begin{array}{lcc}
\alpha, x : \alpha, l \vdash e & & [v_1/x][\tau/\alpha]e \\
\alpha, x : \alpha, l \vdash v_1 : \tau & \Rightarrow & \wr\wr \\
\alpha, x : \alpha, l \vdash v_2 : \tau & & [v_2/x][\tau/\alpha]e
\end{array}
$$

What is a side channel?

- Memory access
- Branching

Problem: how do we carry these guarantess all the way to the assembly? In particular:

- the compiler may introduce a branch on a secret, or
- may introduce some spilling that depends on a secret.

Some ideas

- reuse Barthe's fork of CompCert (2014)
- annotate all IRs of CompCert:
  - add metadata (secret / non-secret), then
  - introduce fake memory accesses to compensate for spills in secret-controlled branches
  - tell stack/offset accesses and heap accesses apart

Some issues

- C* has block-scoped variables; CompCert has function-scope variables
- Transformation in C*; need uninitialized variables
- Issues with hoisting one variable into two; changes memory accesses
- Pointer values not abstract enough

A look at some code

The memory model

- A list of stack frames
- The tip is the current stack frame
- Each stack frame maps locations to values
- Special well-parenthesized **push_frame** and **pop_frame**

```
let test1 (_: unit): Stack unit (fun _ -> true) (fun _ _ _ -> true) =
  push_frame ();
  let b = Buffer.create 21l 2ul in
  print_int32 (index b 0ul +%^ index b 1ul);
  pop_frame ()
```

The Stack effect

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

A trickier example

A function in **`Stack`** requires **`push_region`** and **`pop_region`** to allocate. What about code re-use?

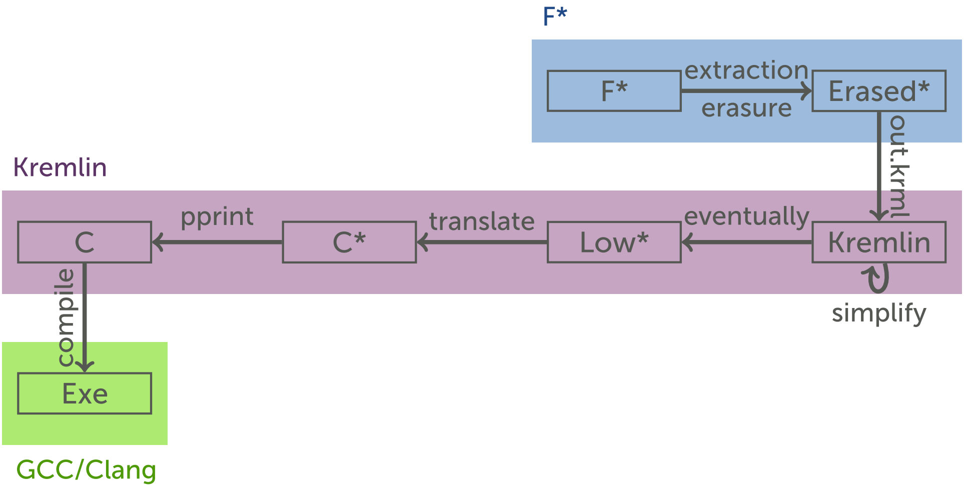
```
let test2 (_: unit):
  StackInline (Buffer.buffer Int32.t)
  (requires (fun h0 -> is_stack_region h0.tip))
  (ensures (fun h0 b h1 -> live h1 b /\ Buffer.length b = 2))
=
  let b = Buffer.create 0l 2ul in
  upd b 0ul (C.rand ());
  upd b 1ul (C.rand ());
  b
```

The StackInline effect

```
let inline_stack_inv h h' : GTot Type0 =
  (* The frame invariant is enforced *)
  h.tip = h'.tip
  (* The heap structure is unchanged *)
  /\ Map.domain h.h == Map.domain h'.h
  (* Any region that is not the tip has not seen any allocations *)
  /\ (∀ (r:HH.rid). (r <> h.tip /\ Map.contains h.h r)
       ==> Heap.domain (Map.sel h.h r) == Heap.domain (Map.sel h'.h r))

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

The tool: KreMLin



Things KreMLin does

- Monomorphization of parameterized types
- Data types to tagged unions
- Decompilation of pattern matches
- Going from an expression language to a statement language (including hoisting)
- (Hopefully) correct name-disambiguation according to C's block-scoping rules
- Inlining of in-scope closures (soon)
- Inlining of the StackInline effect

Demo time!

Conclusion

Our approach:	a shallow embedding of C into F* with a curated set of primitives
Our flagship code:	12,000 lines of F* code (bignum, curve, Chacha20, Poly1305, AEAD)
Our tool:	KreMLin (open-source! go and use it for Coq too?)
Soon:	HACL* (High Assurance Crypto Libraries)
Hopefully soon:	extract more code, including miTLS