Secure compilation from F* to C using the KreMLin compiler

K. Bharghavan, C. Hritcu, J-K. Zinzindohoué
INRIA

P. Wang
MIT

A. Delignat-Lavaud, C. Fournet, J. Protzenko,
T. Ramananandro, A. Rastogi, N. Swamy, S. Zanella-Beguelin
Microsoft
Everest:
Deploying Verified-Secure Implementations in the HTTPS Ecosystem
An overview of our approach
Verification in F*

miTLS

HACL*

interop

Application

interop

compiles to

C library

Software integration
High-level whole-stack proofs

We want a very expressive language for:

- side-channel resistance proofs,
- memory safety,
- cryptographic security,
- functional correctness.

We use F*.
High-level whole-stack proofs

We want a very expressive language for:
- side-channel resistance proofs,
- memory safety,
- cryptographic security,
- functional correctness.

We use F*.
Low-level software

We want to go to C.

- Go where the software is.
- **Piecewise** release of software components (e.g. HACL*).
- **Progressive** replacement, not wholesale switch (incremental).
High-level verification for low-level code

This our **motto**: we shallowly embed C in F*.

Low-level **memory model + low-level libraries** = compilation scheme to C.

The code is **low-level**, but the **verification is not**.

We call the low-level subset of F* is **Low***. 
Typically, F* extracts to OCaml.

With the C backend, we have explicit control over performance:

- no implicit allocations,
- manual, stack-based memory management,
- buffers, loops, structures.
Our **low-level, stack-based memory model**.

```plaintext
let equal_domains (m0:mem) (m1:mem) =
  m0.tip = m1.tip
\Set.equal (Map.domain m0.h) (Map.domain m1.h)
\Set (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 \Set (forall a h1.
    (pre h \Set post h a h1 \Set equal_domains h h1) =>> p a h1))

Preserves the **layout** of the stack and **doesn’t allocate** in any frame.
```
Our **low-level, sequence-based** buffer model.

```
noeq private type _buffer (a:Type) =
  | MkBuffer: max_length:UInt32.t
  -> content:reference (s:seq a{Seq.length s == v max_length})
  -> idx:UInt32.t
  -> length:UInt32.t{v idx + v length <= v max_length}
  -> _buffer a

val index: #a:Type -> b:buffer a -> n:UInt32.t{v n < length b} ->
  Stack a
  (requires (fun h -> live h b))
  (ensures (fun h0 z h1 -> live h0 b \ h1 == h0
            \ z == Seq.index (as_seq h0 b) (v n)))
let index #a b n =
  let s = !b.content in
  Seq.index s (v b.idx + v n)
```

We swap this F* model with a low-level implementation. 
**buffer int** becomes **int* and index b i** becomes **b[i]**.
Our toolchain

Disclaimer: these steps are currently hand-written proofs.
A function in HACL*, original F* code

[@ "c_inline"]
val chacha20_block:
    log: log_t ->
    stream_block: uint8_p {length stream_block = 64} ->
    st: state {disjoint st stream_block} ->
    ctr: Uint32.t ->
Stack log_t
    (requires (fun h -> live h stream_block /
                invariant log h st))
    (ensures (fun h0 updated_log h1 -> live h1 stream_block /
                invariant log h0 st
                /
                invariant updated_log h1 st
                /
                modifies_2 stream_block st h0 h1
                /
                (let block = reveal_sbytes (as_seq h1 stream_block) in
                match Ghost.reveal log, Ghost.reveal updated_log with
                | MkLog k n, MkLog k' n' ->
                block == chacha20_block k n (U32.v ctr) /
                k == k' /
                n == n')))
[@ "c_inline"]
let chacha20_block log stream_block st ctrl =
    push_frame();
    let h_0 = ST.get() in
    let st' = Buffer.create (uint32_to_sint32 0ul) 16ul in
    let log' = chacha20_core log st' st ctrl in
    uint32s_to_le_bytes stream_block st' 16ul;
    let h = ST.get() in
    cut (reveal_sbytes (as_seq h stream_block) == chacha20_block (Ghost.reveal log').k (Ghost.reveal log').n (cut (modifies_3_2 stream_block st h_0 h));
    pop_frame();
    Ghost.elift1 (fun l -> match l with | MkLog k n -> MkLog k n) log'}
A function in HACL*, extracted C code

```c
inline static void
Hacl_Impl_Chacha20_chacha20_block(uint8_t *stream_block,
        uint32_t *st,
        uint32_t ctr)
{
    uint32_t st_[16] = { 0 };
    Hacl_Impl_Chacha20_chacha20_core(st_, st, ctr);
    Hacl_Lib_LoadStore32_uint32s_to_le_bytes(stream_block,
            st_,
            (uint32_t )16);
    return;
}
```
Applications of our methodology

• **HACL\(^*\)**, our high-assurance crypto library
  • as a standalone library
  • within an OpenSSL engine (speed / benchmark)
  • as a NaCl alternative (**LD\_PRELOAD**)

• **miTLS**, our TLS library in F\(^*\) (WIP)
  • as an alternate SSL backend for **curl**
  • a fork of **Nginx**

• low-level **parsers** (e.g. **ASN.1**) (WIP)
Applications of our methodology

- **HACL**, our high-assurance crypto library
  - as a standalone library
  - within an OpenSSL engine (speed / benchmark)
  - as a NaCl alternative (**LD_PRELOAD**)
- **miTLS**, our TLS library in F* (WIP)
  - as an alternate SSL backend for curl
  - a fork of Nginx
- low-level **parsers** (e.g. ASN.1) (WIP)
Preservation of correctness and side-channel resistance
Removal of ghost code (1)

- Ghost is an F* effect
- Used only for proofs, i.e. specifications ("contagious")
- 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.
Explicitly-monadic F* (POPL’17) can be translated to a primitive state semantics.

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: \( \lambda \text{ow}^* \)

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

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

- a type system without progress
- in a simulation with the original F* program
- standard substitutive semantics.
The judgements of $\lambda$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

Let’s talk about traces!
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!
Traces in λow

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

We record:

- memory accesses, reads and writes
- branching
Traces in \texttt{low*}: the secret library

We assume a trusted module that operates on secrets, whose traces are secret-independent.

\begin{verbatim}
(* 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})
\end{verbatim}

We trust that the \texttt{eq_mask} function has been written properly.
Secret-independence: an intuition

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

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

**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.
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. $P_s, P \vdash (H, e)[\rho_1] \rightarrow^+_{\ell} (H', e')[\rho'_1]$ 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 $\lambda\text{ow}^*$ programs to $C^*$ programs via a simulation.
A glimpse of the reduction rules

From \lambda ow^*:

\[
P \vdash (H, \text{if } 0 \text{ then } e_1 \text{ else } e_2) \rightarrow_{brF} (H, e_2) \quad \text{LlfF}
\]

From C^*:

\[
\hat{P} \vdash (S, V, \text{if } \hat{e} \text{ then } s_1 \text{ else } \hat{s}_2; \hat{s}) \sim_{brF} (S, V, \hat{s}_2; \hat{s}) \quad \text{CffF}
\]
Theorem

The C* program \( \hat{P} \) terminates with trace \( \ell \) and return value \( v \), i.e., \( \hat{P} \vdash (\emptyset, V, s; \text{return } \hat{e}) \xrightarrow{\ell,*} (\emptyset, V', \text{return } v) \) if, and only if, so does the \( \lambda \text{ow}^* \) program: \( P \vdash (\emptyset, e[V]) \xrightarrow{\ell,*} (H', v) \); 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.)
Thanks. Questions?