Verified Low-Level Programming Embedded in F*

Karthikeyan Bhargavan[2] Antoine Delignat-Lavaud[3] Cédric Fournet[3] Cătălin Hriţcu[2]
Jonathan Protzenko[3] Tahina Ramananandro[3] Aseem Rastogi[3] Nikhil Swamy[3] Peng Wang[1]
Santiago Zanella-Beguelin[3] Jean-Karim Zinzindohoué[2]
[1]MIT CSAIL [2]INRIA Paris [3]Microsoft Research

Abstract

We present Low*, a language for efficient low-level programming and verification, and its application to high-assurance cryptographic libraries. Low* is a shallow embedding of a small, sequential, well-behaved subset of C in F*, a dependently-typed variant of ML aimed at program verification. Departing from ML, Low* has a structured memory model à la CompCert; it does not involve garbage collection or implicit heap allocations; and it provides instead the control required for writing low-level security-critical code.

By virtue of typing, any Low* program is memory safe. On top of this, the programmer can make full use of the verification power of F* to write high-level specifications and verify the functional correctness of Low* code using a combination of SMT automation and sophisticated manual proofs. At extraction time, specifications and proofs are erased, and the remaining code enjoys a predictable translation to C. We prove that this translation preserves semantics and side-channel resistance.

We have implemented and verified cryptographic algorithms, constructions, and tools in Low*, for a total of about 20,000 lines of code. Our code delivers performance competitive with existing (unverified) C cryptographic libraries, suggesting our approach may be applicable to larger-scale, verified, low-level software.

1. Introduction

Cryptographic software widely deployed throughout the internet is still often subject to security-critical attacks [1–4, 10, 12, 27, 29, 32, 33, 38, 41, 57, 60, 61, 63–65, 67, 68, 71]. Recognizing a clear need, the programming language, verification, and applied cryptography communities are devoting significant efforts to develop implementations proven secure by construction against broad classes of attacks.

Focusing on low-level attacks caused by violations of memory safety, several researchers have used high-level, type-safe programming languages to implement standard protocols such as Transport Layer Security (TLS). For example, Kaloper-Meršinjak et al. [46] provide nqsbTLS, an implementation of TLS in OCaml, which by virtue of its type and memory safety is impervious to attacks like Heartbleed [2] that exploit buffer overflows. Bhargavan et al. [30] program miTLS in F#, also enjoying type and memory safety, but go further using a refinement type system to prove various higher-level security properties of the protocol.

While this approach is attractive for its simplicity, to get acceptable performance, both nqsbTLS and miTLS link with fast, unsafe implementations of complex cryptographic algorithms, such as those provided by nocrypto [6], an implementation that mixes C and OCaml, and libcrypto, a component of the widely used OpenSSL library [7]. Linking with vulnerable C code could, in the worst case, void all the security guarantees of the high-level code.

In this paper, we aim to bridge the gap between high-level, safe-by-construction code, optimized for clarity and ease of verification, and low-level code exerting fine control over data representations and memory layout in order to achieve better performance. Towards this end, we introduce Low*, a domain-specific language for verified, efficient low-level programming, embedded within F* [66], an ML-like language with dependent types designed for program verification. We use F* to prove functional correctness and security properties of high-level code. Where efficiency is paramount, we drop into its C-like Low* subset while still relying on F*'s verification capabilities to prove the code memory safe, functionally correct, and secure.

We have applied Low* to program and verify a range of sequential low-level programs, including libraries for multi-precision arithmetic and buffers, and various cryptographic algorithms, constructions, and protocols built on top of them. Our initial experiments indicate significant speedups when linking Low* libraries compiled to C with F* programs compiled to OCaml, without compromising security.

An embedded DSL, compiled natively Low* programs are a subset of F* programs. The programmer writes Low* code using regular F* syntax, against a library we provide that models a lower-level view of memory, akin to the structured memory layout of a well-defined C program (this is similar to the structured memory model of CompCert [51, 53], not the "big array of bytes" model systems programmers sometimes

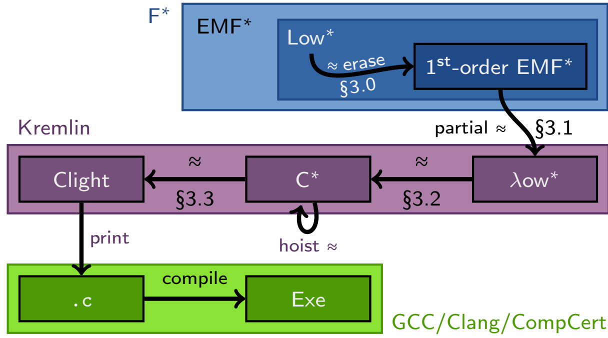

Figure 1. Low* embedded in F*, compiled to C, with soundness and security guarantees (details in §3)

use). Low* programs interoperate naturally with other F* programs, and precise specifications of Low* and F* code are intermingled when proving properties of their combination. As usual in F*, programs are type-checked and compiled to OCaml for execution, after erasing computationally irrelevant parts of a program, e.g., proofs and specifications, using a process similar to Coq's extraction mechanism [54].

Importantly, Low* programs have a second, equivalent but more efficient semantics via compilation to C, with a predictable performance model including manual memory management—this compilation is implemented by KreMLin, a new compiler from the Low* subset of F* to C. Figure 1 illustrates the high-level design of Low* and its compilation to native code. Our main *contributions* are as follows:

Libraries for low-level programming within F* (§2) At its core, F* is a purely functional language to which effects like state are added programmatically using monads. We instantiate the state monad of F* to use a CompCert-like structured memory model that separates the stack and the heap, supporting en masse allocation and deallocation on the stack, and allocating and freeing individual reference cells on the heap. The heap is organized into disjoint logical regions, which enables stating separation properties necessary for modular, stateful verification. On top of this, we program a library of buffers—C-style arrays passed by reference—with support for pointer arithmetic and referring to only part of an array. By virtue of F* typing, our libraries and all their well-typed clients are guaranteed to be memory safe, e.g., they never access out-of-bounds or deallocated memory.

Designing Low*, a subset of F* easily compiled to C We intend to give Low* programmers precise control over the performance profile of the generated C code. Inasmuch as possible, we aim for the programmer to have control even over the syntactic structure of the target C code, to facilitate its review by security experts unfamiliar with F*. As such, to a first approximation, Low* programs are F* programs well-typed in the state monad described above, which, after all their computationally irrelevant (ghost) parts have been erased, must satisfy several requirements. Specifically, the code (1) must be first order, to prevent the need to allocate closures in C; (2) must not perform any implicit allocations; (3) must not use recursive datatypes, since these would have to be compiled using additional indirections to C structs; and (4) must be monomorphic, since C does not support polymorphism directly. We emphasize that these restrictions apply only to computationally relevant code—proofs and specifications are free to use arbitrary higher-order, dependently typed F*, and very often they do.

A formal translation from Low* to CompCert Clight (§3) Justifying its dual interpretation as a subset of F* and a subset of C, we give a translation from Low* to CompCert's Clight [31] and show that it preserves trace equivalence with respect to the original F* semantics of the program. In addition to ensuring that the functional behavior of a program is preserved, our trace equivalence also guarantees that the compiler does not introduce unexpected side-channels due to memory access patterns, at least until it reaches Clight—a useful sanity check for cryptographic code. Our formal results cover the translation of standalone Low* programs to C, proving that execution in C preserves the original F* semantics of the Low* program. More pragmatically, we have built several cryptographic libraries in Low*, compiled them to C, and integrated them within larger programs compiled to OCaml, relying on OCaml's foreign function interface for interoperability and trusting it for correctness.

KreMLin, a compiler from Low* to C (§4) Our formal development guides the implementation of KreMLin, a new tool that emits C code from Low*. KreMLin is designed to emit well-formatted, idiomatic C code suitable for manual review. The resulting C programs can be compiled with CompCert for greatest assurance, and with mainstream C compilers, including GCC and Clang, for greatest performance. We have used KreMLin to extract to C the 20,000+ lines of Low* code we have written so far.

An extensive empirical evaluation (§5) We present two substantial developments of efficient, verified, interoperable cryptographic libraries programmed in Low*.

We implement in around 14,000 lines of code the Authenticated Encryption with Associated Data (AEAD) construction at the heart of the TLS-1.3 record layer—we prove memory safety, functional correctness, and cryptographic security. We also implement a robust API and side-channel protections through type abstraction. We compile our Low* AEAD code to both OCaml and C, linking it with the F* implementation of miTLS compiled to OCaml. As expected by the design of Low*, relative to the full OCaml version, miTLS linked with the C version of our libraries shows an improvement in throughput of data transfer by at least two orders of magnitude. There is still room for improvement, however—one measure shows our verified code to still be around 5× slower than OpenSSL's libcrypto. (§5.2)

We also provide HACL*, a "high-assurance crypto library" in Low*, implementing and proving several cryptographic

algorithms, including the Poly1305 MAC [22], the ChaCha20 cipher [59], and Curve25519 [23]. We package these algorithms to provide the popular box/box_open NaCl API [25], yielding the first performant implementation of NaCl that is verified for memory safety and side-channel resistance, including the first verified bigint libraries customized for safe, fast cryptographic use. Finally, on top of this API, we build *PneuTube*, a secure, asynchronous, file transfer application whose performance compares favorably with widely used, existing utilities like scp. (§5.1)

Supplementary materials Full formalization and hand proofs of all the theorems in this paper are available in the long version included with this submission. We also provide the KreMLin compiler and our verified Low* code for AEAD, HACL*, and PneuTube as supplementary materials.

2. A Low* tutorial

At the core of Low* is a library for programming with buffers manually allocated on the stack or heap (§2.2). Memory safety demands reasoning about the extents and liveness of these buffers, while functional correctness and security may require reasoning about their contents. Our library provides specifications to allow client code to be proven safe, correct and secure, while KreMLin compiles such verified client code to C. We illustrate the design of Low* using two examples from our codebase: the ChaCha20 stream cipher [59], which we prove memory safe (§2.1), and the Poly1305 MAC [22], which we prove functionally correct (§2.3). We finally explain how we prove a combination of ChaCha20 and Poly1305 cryptographically secure (§2.4).

2.1 A first example: the ChaCha20 stream cipher

Figure 2 shows code snippets for the main function of ChaCha20 [59], a modern stream cipher widely used for fast symmetric encryption. On the left is our Low* code; on the right its compilation to C. The function takes as arguments an output length and buffer, and some input key, nonce, and counter. It allocates 32 words of auxiliary state on the stack; then calls a function to initialize the cipher block from the inputs (passing an inner pointer to the state); and finally calls another function that computes a cipher block and copies its first len bytes to the output buffer.

Aside from the erased specifications at lines 4–5, the C code is in one-to-one correspondence with its Low* counterpart. These specifications capture the safe memory usage of chacha20: for each argument passed by reference, and for the auxiliary state, they keep track of their liveness and size.

Line 2 uses two type refinements to require that the len argument equals the length of the output buffer and it does not exceed the block size. (Violation of these conditions would lead to a buffer overrun in the call to chacha20_update.) Similarly, types keyBytes and nonceBytes specify pointers to fixed-sized buffers of bytes. At the beginning of line 4, Stack unit says that chacha20 returns nothing and may allocate on the stack, but not on the heap (in particular, it has no memory leak). On the same line, the pre-condition requires that all arguments passed by reference be live. On line 5, the post-condition ensures that the function modifies at most the contents of output (and, implicitly, that all buffers remain live). We further explain this specification in the next subsection.

As usual for symmetric ciphers, RFC 7539 specifies chacha20 as imperative pseudocode, and does not further discuss its mathematical properties. Our Low* code and its C extraction closely follow the RFC, and yield the same results on its test vectors. Although we could write a pure F* specification (further away from the RFC) and verify that our code implements it, it would not add much here.

2.2 An embedded DSL for low-level code

As in ML, by default F* does not provide an explicit means to reclaim memory or to allocate memory on the stack, nor does it provide support for pointing to the interior of arrays. Next, we sketch the design of a new F* library that provides a structured memory model suitable for program verification, while supporting low-level features like explicit freeing, stack allocation, and inner pointers for arrays. In subsequent sections, we describe how programs type-checked against this library can be compiled safely to C. First, however, we begin with some background on F*.

Background: F* is a dependently typed language with support for user-defined monadic effects. Its types separate computations from values, giving the former *computation types* of the form M $t_1 \ldots t_n$ where M is an effect label and $t_1 \ldots t_n$ are *value types*. For example, Stack unit (...) (...) on lines 4–5 of Figure 2 is an instance of a computation type, while types like unit are value types. There are two distinguished computation types: Tot t is the type of a total computation returning a t-typed value; Ghost t, a computationally irrelevant computation returning a t-typed value. Ghost computations are useful for specifications and proofs but are erased when extracting to OCaml or C.

To add state to F*, one defines a state monad represented (as usual) as a function from some initial memory m_0:s to a pair of a result r:a and a final memory m_1:s, for some type of memory s. Stateful computations are specified using the computation type:

```
ST (a:Type) (pre: s → Type) (post: s → a → s → Type)
```

Here, ST is a computation type constructor applied to three arguments: a result type a; a pre-condition predicate on the initial memory, pre; and a post-condition predicate relating the initial memory, result and final memory. We generally annotate the pre-condition with the keyword requires and the post-condition with ensures for better readability. A computation e of type ST a (requires pre) (ensures post), when run in an initial memory m_0:s satisfying pre m, produces a result r:a and final memory m_1:s satisfying post m_0 r m_1, unless it di-

```
1  let chacha20
2    (len: uint32{len ≤ blocklen}) (output: bytes{len = output.length})
3    (key: keyBytes) (nonce: nonceBytes{disjoint [output; key; nonce]}) (counter: uint32) :
4        Stack unit (requires (λ m0 → output ∈ m0 ∧ key ∈ m0 ∧ nonce ∈ m0))
5                       (ensures (λ m0 _m1 → modifies₁ output m0 m1)) =
6      push_frame ();
7      let state = Buffer.create 0ul 32ul in
8      let block = Buffer.sub state 16ul 16ul in
9      chacha20_init block key nonce counter;
10     chacha20_update output state len;
11     pop_frame ()
```

```
1  void chacha20 (
2    uint32_t len, uint8_t *output,
3    uint8_t *key, uint8_t *nonce, uint32_t counter)
4
5
6  {
7      uint32_t state[32] = { 0 };
8      uint32_t *block = state + 16;
9      chacha20_init(block, key, nonce, counter);
10     chacha20_update(output, state, len);
11 }
```

Figure 2. A snippet from ChaCha20 in Low* (left) and its C compilation (right)

verges.[1] F* uses an SMT solver to discharge the verification conditions it computes when type-checking a program.

Hyper-stacks: A region-based memory model for Low* For Low*, we instantiate the type s in the state monad to HyperStack.mem (which we refer to as just "hyper-stack"), a new region-based memory model [70] covering both stack and heap. Hyper-stacks are a generalization of hyper-heaps, a memory model proposed previously for F* [66], designed to provide "lightweight support for separation and framing for stateful verification". Hyper-stacks augment hyper-heaps with a shape invariant to indicate that the lifetime of a certain set of regions follow a specific stack-like discipline. We sketch the F* signature of hyper-stacks next.

A logical specification of memory Hyper-stacks partition memory into a set of regions. Each region is identified by an rid and regions are classified as either stack or heap regions, according to the predicate is_stack_region—we use the type abbreviation sid for stack regions and hid for heap regions. A distinguished stack region, root, outlives all other stack regions. The snippet below is the corresponding F* code.

```
type rid
val is_stack_region: rid → Tot bool
type sid = r:rid{is_stack_region r}
type hid = r:rid{¬ (is_stack_region r)}
val root: sid
```

Next, we show the (partial) signature of mem, our model of the entire memory, which is equipped with a select/update theory [56] for typed references ref a. Additionally, we have a function to refer to the region_of a reference, and a relation r ∈ m to indicate that a reference is live in a given memory.

```
type mem
type ref : Type → Type
val region_of: ref a → Ghost rid
val `_ ∈ _` : ref a → mem → Tot Type (* a ref is contained in a mem *)
val `_ [_]` : mem → ref a → Ghost a (* selecting a ref *)
val `_ [_] ← _` : mem → ref a → a → Ghost mem (* updating a ref *)
val rref r a = x:ref a {region_of x = r} (* abbrev. for a ref in region r *)
```

Heap regions By defining the ST monad over the mem type, we can program stateful primitives for creating new heap regions, and allocating, reading, writing and freeing references in those regions—we show some of their signatures below. Assuming an infinite amount of memory, alloc's pre-condition is trivial while its post-condition indicates that it returns a fresh reference in region r initialized appropriately. Freeing and dereferencing (!) require their argument to be present in the current memory, eliminating double-free and use-after-free bugs.

```
val alloc: r:hid → init:a → ST (rref r a)
  (ensures (λ m0 x m1 → x ∉ m0 ∧ x ∈ m1 ∧ m1 = (m0[x]← init)))
val free: r:hid → x:rref r a → ST unit
  (requires (λ m → x ∈ m))
  (ensures (λ m0 () m1 → x ∉ m1 ∧ ∀y≠x. m0[y] = m1[y]))
val (!): x:ref a → ST a
  (requires (λ m → x ∈ m))
  (ensures (λ m0 y m1 → m0 = m1 ∧ y = m1[x]))
```

Since we support freeing individual references within a region, our model of regions could seem similar to Berger et al.'s [20] *reaps*. However, at present, we do no support freeing heap objects *en masse* by deleting heap regions; indeed, this would require using a special memory allocator. Instead, for us heap regions serve only to *logically* partition the heap in support of separation and modular verification, as is already the case for hyper-heaps [66], and heap region creation is currently compiled to a no-op by KreMLin.

Stack regions, which we will henceforth call *stack frames*, serve not just as a reasoning device, but provide the efficient C stack-based memory management mechanism. KreMLin maps stack frame creation and destruction directly to the C calling convention and lexical scope. To model this, we extend the signature of mem to include a tip region representing the currently active stack frame, ghost operations to push and pop frames on the stack of an explicitly threaded memory, and their effectful analogs, push_frame and pop_frame that modify the current memory. In chacha20 in Fig. 2, the push_frame and pop_frame correspond precisely to the braces in the C program that enclose a function body's scope. We also provide a derived combinator, with_frame, which combines push_frame and pop_frame into a single, well-scoped operation. Programmers are encouraged to use the with_frame combinator, but, when more convenient for verification, may also use push_frame and pop_frame directly. KreMLin ensures that all uses of push_frame and pop_frame are well-scoped. Finally, we

[1] F* recently gained support for proving stateful computations terminating. We have begun making use of this feature to prove our code terminating, wherever appropriate, but make no further mention of this.

show the signature of salloc which allocates a reference in the current tip stack frame.

```
val tip: mem → Ghost sid
val push: mem → Ghost mem
val pop: m:mem{tip m ≠ root} → Ghost mem
val push_frame: unit → ST unit
 (ensures (λ m0 () m1 → m1 = push m0))
val pop_frame: unit → ST unit
 (requires (λ m → tip m ≠ root))
 (ensures (λ m0 () m1 → m1 = pop m0))
val salloc: init:a → ST (ref a)
  (ensures (λ m0 x m1 → x ∉ m0 ∧ x ∈ m1 ∧ region_of x = tip m1 ∧
                 tip m0 = tip m1 ∧ m1 = (m0[x] ← init)))
```

The Stack effect The specification of chacha20 claims that it uses only stack allocation and has no memory leaks, by making use of the Stack computation type. This is straightforward to define in terms of ST, as shown below.

```
effect Stack a pre post = ST a (requires pre) (ensures (λ m0 x m1 →
    post m0 x m1 ∧ tip m0 = tip m1 ∧ (∀ r. r ∈ m1 ⟺ r ∈ m0)))
```

Stack computations are ST computations that leave the stack tip unchanged (i.e., they pop all frames they may push), and which produce a final memory that has an equal domain to the initial memory. This ensures that Low* code with Stack effect has explicitly deallocated all heap allocated references before returning, effectively ruling out memory leaks. As such, we expect all Low* functions callable from F* through the OCaml FFI to have Stack effect. The F* code can safely pass pointers to objects allocated in the garbage-collected OCaml heap into Low* code with Stack effect since the definition of the Stack effect disallows the Low* code from freeing these references.

Modeling buffers Hyper-stacks separate heap and stack memory, but each region of memory still only supports abstract, ML-style references. A crucial element of low-level programming is control over the specific layout of objects, especially for arrays and structs. Leaving a proper low-level modeling of structs, including pointers inside structures, as future work, we focus on arrays and implement an abstract type of buffers in Low* using just the references provided by hyper-stacks. Relying on its abstraction, KreMLin compiles buffers to native C arrays. We sketch the library next.

The type buffer a below is a single-constructor inductive type whose arguments depend on each other. Its main content field holds a reference to a seq a, a purely functional sequence of a's, whose length is determined by the first field, max_length. A refinement type b:(buffer uint32){length b = n} is translated to a C declaration uint32_t b[n] by KreMLin and, relying on C pointer decay, further referred to via uint32_t *.

```
abstract type buffer a =
 | MkBuffer: max_length:uint32
    → content:ref (s:(seq a){Seq.length s = max_length})
    → idx:uint32
    → length:uint32{idx + length ≤ max_length} → buffer a
```

The other two fields of a buffer are there to support creating smaller sub-buffers from a larger buffer, via the Buffer.sub operation below. A call to Buffer.sub b i l returning b' is compiled to C pointer arithmetic b + i (as seen in Figure 2 line 8 in chacha20). To accurately model this, the content field is shared between b and b', but idx and length differ, to indicate that the sub-buffer b' only covers a sub-range of the original buffer b. The sub operation has computation type Tot, meaning that it does not read or modify the state. The refinement on the result b' records the length of b' and, using the includes relation, shows that b and b' are aliased.

```
val sub: b:buffer a → i:uint32{i + b.idx < pow2 32}
  → len:uint32{i + len ≤ b.length}
  → Tot (b':buffer a{b'.length = len ∧ b `includes` b'})
```

We also provide statically bounds-checked operations for indexing and updating buffers. The signature of index below requires the buffer to be live and the index location to be within bounds. Its postcondition ensures that the memory is unchanged and describes what is returned in terms of the logical model of a buffer as a sequence.

```
let get (m:mem) (b:buffer a) (i:uint32{i < b.length}) : Ghost a =
   Seq.index (m[b.content]) (b.idx + i)
val index: b:buffer a → i:uint32{i < b.length} → Stack a
  (requires (λ m → b.content ∈ m))
  (ensures (λ m0 z m1 → m1 = m0 ∧ z = get m1 b i))
```

2.3 Functional correctness and side-channel resistance

This section and the next illustrates our "high-level verification for low-level code" methodology. Although programming at a low-level, we rely on high-level features of F*, including type abstraction and dependently typed meta-programming, to prove our code functionally correct, cryptographically secure, and free of a class of side-channels.

We start with Poly1305 [22], a Message Authentication Code (MAC) algorithm.[2] Unlike chacha20, for which the main property of interest is implementation safety, Poly1305 has a mathematical definition in terms of a polynomial in the prime field $GF(2^{130} - 5)$, against which we prove our code functionally correct. Relying on correctness, we then prove injectivity lemmas on encodings of messages into field polynomials, and we finally prove cryptographic security of a one-time MAC construction for Poly1305, specifically showing unforgeability against chosen message attacks (UF1CMA). This game-based proof involves an idealization step, justified by a probabilistic proof on paper, following the methodology we explain in §2.4.

For side-channel resistance, we use type abstraction to ensure that our code's branching and memory access patterns are secret independent. This style of F* coding is advocated by Zinzindohoué et al. [73]; we place it on formal ground by showing that it is a sound way of enforcing secret independence at the source level (§3.1) and that our compilation carries such properties to the level of Clight (§3.3). To

[2] Implementation bugs in Poly1305 are a reality: in 2016 alone, the Poly1305 OpenSSL implementation experienced two correctness bugs [19, 32] and a buffer overflow [9].

carry our results further down, one may validate the output of the C compiler by relying on recent tools proving side-channel resistance at the assembly level [13, 14]. We sketch our methodology on a small snippet from our specialized arithmetic (bigint) library upon which we built Poly1305.

Representing field elements using bigints We represent elements of the field underlying Poly1305 as 130-bit integers stored in Low* buffers of machine integers called *limbs*. Spreading out bits evenly across 32-bit words yields five limbs ℓ_i, each holding 26 bits of significant data. A ghost function eval = $\sum_{i=0}^{4} \ell_i \times 2^{26 \times i}$ maps each buffer to the mathematical integer it represents. Efficient bigint arithmetic departs significantly from elementary school algorithms. Additions, for instance, can be made more efficient by leveraging the extra 6 bits of data in each limb to delay carry propagation. For Poly1305, a bigint b is in compact form in state m (compact m b) when all its limbs fit in 26 bits. Compactness does not guarantee uniqueness of representation as $2^{130} - 5$ and 0 are the same number in the field but they have two different compact representations that both fit in 130 bits—this is true for similar reasons for the range $[0, 5)$.

Abstracting integers as a side-channel mitigation Modern cryptographic implementations are expected to be protected against side-channel attacks [49]. As such, we aim to show that the branching behavior and memory accesses of our crypto code are independent of secrets. To enforce this, we use an abstract type limb to represent limbs, all of whose operations reveal no information about the contents of the limb, either through its result or through its branching behavior and memory accesses. For example, rather than providing a comparison operator, eq_leak: limb → limb → Tot bool, whose boolean result reveals information about the input limbs, we use a masking eq_mask operation to compute equality securely. Unlike OCaml, F*'s equality is not fully polymorphic, being restricted to only those types that support decidable equality, limb not being among them.

```
val v : limb → Ghost nat (∗ limbs only ghostly revealed as numbers ∗)
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})
```

In the signature above, v is a function that reveals an abstract limb as a natural number, but only in ghost code—a style referred to as translucent abstraction [66]. The signature of eq_mask claims that it returns a zero limb if the two arguments differ, although computationally relevant code cannot observe this fact. Note, the number of limbs in a Poly1305 bigint is a public constant, i.e., bigint = b:(buffer limb){b.length = 5}.

Proving normalize correct and side-channel resistant The normalize function of Figure 3 modifies a compact bigint in-place to reduce it to its canonical representation. The code is rather opaque, since it operates by strategically masking each limb in a secret independent manner. However, its specification clearly shows its intent: the new contents of the input bigint is the same as the original one, *modulo*

```
1  let normalize (b:bigint) : Stack unit
2    (requires (λ m0 → compact m0 b))
3    (ensures (λ m0 () m1 → compact m1 b ∧ modifies1 b m0 m1 ∧
4                eval m1 b = eval m0 b % (pow2 130 − 5)))
5   = let h0 = ST.get() in (∗ a logical snapshot of the initial state ∗)
6     let ones = 67108863ul in (∗ 2^26 − 1 ∗)
7     let m5 = 67108859ul in (∗ 2^26 − 5 ∗)
8     let m = (eq_mask b.(4ul) ones) & (eq_mask b.(3ul) ones)
9           & (eq_mask b.(2ul) ones) & (eq_mask b.(1ul) ones)
10          & (gte_mask b.(0ul) m5) in
11    b.(0ul) ← b.(0ul) − m5 & m;
12    b.(1ul) ← b.(1ul) − b.(1ul) & m; b.(2ul) ← b.(2ul) − b.(2ul) & m;
13    b.(3ul) ← b.(3ul) − b.(3ul) & m; b.(4ul) ← b.(4ul) − b.(4ul) & m;
14    lemma_norm h0 (ST.get()) b m (∗ relates masking to eval modulo ∗)
```

Figure 3. Unique representation of a Poly1305 bigint

$2^{130} - 5$. At line 14, we see a call to a F* lemma, which relates the masking operations to the modular arithmetic in the specification—the lemma is erased during extraction.

2.4 Cryptographic provable-security example: AEAD

Going beyond functional correctness, we sketch how we use Low* to do security proofs in the standard model of cryptography, using "authenticated encryption with associated data" (AEAD) as a sample construction. AEAD is the main protection mechanism for the TLS record layer; it secures most Internet traffic.

AEAD has a generic security proof by reduction to two core functionalities: a stream cipher (such as ChaCha20) and a one-time-MAC (such as Poly1305). The cryptographic, game-based argument supposes that these two algorithms meet their intended *ideal functionalities*, e.g., that the cipher is indistinguishable from a random function. Idealization is not perfect, but is supposed to hold against computationally limited adversaries, except with small probabilities, say $\varepsilon_{\mathrm{ChaCha20}}$ and $\varepsilon_{\mathrm{Poly1305}}$. The argument then shows that the AEAD construction also meets its own ideal functionality, except with probability say $\epsilon_{\mathrm{Chacha20}} + \epsilon_{\mathrm{Poly1305}}$.

To apply this security argument to our implementation of AEAD, we need to encode such assumptions. To this end, we supplement our real Low* code with ideal F* code. For example, ideal AEAD is programmed as follows:

- encryption generates a fresh random ciphertext, and it records it together with the encryption arguments in a log.
- decryption simply looks up an entry in the log that matches its arguments and returns the corresponding plaintext, or reports an error.

These functions capture both confidentiality (ciphertexts do not depend on plaintexts) and integrity (decryption only succeeds on ciphertexts output by encryption). Their behaviors are precisely captured by typing, using pre- and post-conditions about the ghost log shared between them, and abstract types to protect plaintexts and keys.

We show below the abstract type of keys and the encryption function for idealizing AEAD.

```
1 type entry = cipher * data * nonce * plain
2 abstract type key =
3   { key: keyBytes; log: if Flag.aead then ref (seq entry) else unit }
4 let encrypt (k:key) (n:nonce) (p:plain) (a:data) =
5   if Flag.aead
6   then let c = random_bytes ℓc in k.log := (c, a, n, p) :: k.log; c
7   else encrypt k.key n p a
```

A module Flag declares a set of abstract booleans (*idealization flags*) that precisely capture each cryptographic assumption. For every real functionality that we wish to idealize, we branch on the corresponding flag.

This style of programming heavily relies on the normalization capabilities of F*. At verification time, flags are kept abstract, so that we verify both the real and ideal versions of the code. At extraction time, we reveal these booleans to be false. The F* normalizer then e.g. drops the then branch, and replaces the log field with unit, meaning that both the high-level, list-manipulating code and corresponding type definitions are erased, leaving only low-level code from the else branch to be extracted.

Using this technique, we verify by typing e.g. that our AEAD code, when using *any* ideal cipher and one-time MAC, perfectly implements ideal AEAD. We also rely on typing to verify that our code complies with the pre-conditions of the intermediate proof steps. Finally, we also prove that our code does not reuse nonces, a common cryptographic pitfall.

3. A formal translation from Low* to Clight

Figure 1 on page 2 provides an overview of our translation from Low* to CompCert Clight, starting with EMF*, a recently proposed model of F* [11]; then λow*, a formal core of Low*; then C*, an intermediate language that switches the calling convention closer to C; and finally to Clight. In the end, our theorems establish that: (a) the safety and functional correctness properties verified at the F* level carry on to the generated Clight code (via semantics preservation), and (b) Low* programs that use the secrets parametrically enjoy the trace equivalence property, at least until the Clight level, thereby providing protection against side-channels.

Prelude: Internal transformations in* EMF We begin by briefly describing a few internal transformations on EMF*, focusing in the rest of this section on the pipeline from λow* to Clight—the formal details are in the appendix. To express computational irrelevance, we extend EMF* with a primitive Ghost effect. An erasure transformation removes ghost subterms, and we prove that this pass preserves semantics, via a logical relations argument. Next, we rely on a prior result [11] showing that EMF* programs in the ST monad can be safely reinterpreted in $\text{EMF}^\star_{\text{ST}}$, a calculus with primitive state. We obtain an instance of $\text{EMF}^\star_{\text{ST}}$ suitable for Low* by instantiating its state type with HyperStack.mem. To facilitate the remainder of the development, we transcribe $\text{EMF}^\star_{\text{ST}}$ to λow*, which is a restriction of $\text{EMF}^\star_{\text{ST}}$ to first-order terms that only use stack memory, leaving the heap out of λow*, since it is not a particularly interesting aspect of the proof.

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

Figure 4. λow* syntax

This transcription step is essentially straightforward, but is not backed by a specific proof. We plan to fill this gap as we aim to mechanize our entire proof in the future.

3.1 λow*: A formal core of Low* post-erasure

The meat of our formalization of Low* begins with λow*, a first-order, stateful language, whose state is structured as a stack of memory regions. It has a simple calling convention using a traditional, substitutive β-reduction rule. Its small-step operational semantics is instrumented to produce traces that record branching and the accessed memory addresses. As such, our traces account for side-channel vulnerabilities in programs based on the program counter model [58] augmented to track potential leaks through cache behavior [18]. We define a simple type system for λow* and prove that programs well-typed with respect to some values at an abstract type produce traces independent of those values, e.g., our bigint library when translated to λow* is well-typed with respect to an abstract type of limbs and leaks no information about them via their traces.

Syntax Figure 4 shows the syntax of λow*. A program P is a sequence of top-level function definitions, d. We omit loops but allow recursive function definitions. Values v include constants, immutable records, and buffers (b, n) passed by reference, where b is the buffer address and n is the offset in the buffer. Stack allocated buffers (readbuf, writebuf, newbuf, and subbuf) are the main feature of the expression language, along with withframe e, which pushes a new frame on the stack for the evaluation of e, after which it is popped (using pop e, an administrative form internal to the calculus). Once a frame is popped, all its local buffers become inaccessible.

Type system λow* types include the base types int and unit, record types $\{\overrightarrow{f = \tau}\}$, buffer types buf τ, and abstract types α. The typing judgment has the form, $\Gamma_P; \Sigma; \Gamma \vdash e : \tau$, where Γ_P includes the function signatures; Σ is the store typing; and Γ is the usual context of variables. We elide the rules, as it is a standard, simply-typed type system. The type system guarantees preservation, but not progress, since it does not attempt to account for bounds checks or buffer lifetime. However, memory safety (and progress) is a consequence of Low* typing and its semantics-preserving erasure to λow*.

Semantics We define evaluation contexts E for standard call-by-value, left-to-right evaluation order. The memory H is a stack of frames, where each frame maps addresses b to a sequence of values $\overrightarrow{v}$. The λow* small-step semantics judgment has the form $P \vdash (H, e) \to_\ell (H', e')$, meaning that under the program P, configuration (H, e) steps to (H', e') emitting a trace ℓ, including reads and writes to buffer references, and branching behavior, as shown below.

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

Figure 5 shows selected reduction rules from λow* at the left (in contrast with the reduction rules of C*, discussed in the next section). Rule WF pushes an empty frame on the stack, and rule POP pops the topmost frame once the expression has been evaluated. Rule LIFF is standard, except for the trace brF recorded on the transition. Rule LRD returns the value at the $(n + n_1)$ offset in the buffer at address b, and emits a $\mathsf{read}(b, n + n_1)$ event. Rule NEW initializes the new buffer, and emits write events corresponding to each offset in the buffer. Rule APP is a standard, substitutive β-reduction.

Secret independence A λow* program can be written against an interface providing secrets at an abstract type. For example, for the abstract type limb, one might augment the function signatures Γ_P of a program with an interface for the abstract type $\Gamma_{\mathsf{limb}} = \mathsf{eq_mask} : \mathsf{limb}^2 \to \mathsf{limb}$, and typecheck a source program with free limb variables (Γ = secret:limb), and empty store typing, using the judgment $\Gamma_{\mathsf{limb}}, \Gamma_p; \cdot; \Gamma \vdash e : \tau$. Given any representation τ for limb, an implementation for eq_mask whose trace is input independent, and any pair of values $v_0 : \tau, v_1 : \tau$, we prove that running $e[v_0/\mathsf{secret}]$ and $e[v_1/\mathsf{secret}]$ produces identical traces, i.e., the traces reveal no information about the secret v_i. We sketch the formal development next, leaving details to the appendix.

Given a derivation $\Gamma_s, \Gamma_P; \Sigma; \Gamma \vdash e : \tau$, let Δ map type variables in the interface Γ_s to concrete types and let P_s contain the implementations of the functions (from Γ_s) that operate on secrets. To capture the secret independence of P_s, we define a notion of an *equivalence modulo secrets*, a type-indexed relation for values ($v_1 \equiv_\tau v_2$) and memories ($\Sigma \vdash H_1 \equiv H_2$). Intuitively two values (resp. memories) are equivalent modulo secrets if they only differ in subterms that have abstract types in the domain of the Δ map—we abbreviate "equivalent modulo secrets" as "related" below. We then require that each function $f \in P_s$, when applied in related stores to related values, always returns related results, while producing *identical* traces. Practically, P_s is a (small) library written carefully to ensure secret independence.

Our secret-independence theorem is then as follows:

Theorem 1 (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 interface Δ, P_s (i.e., $\Gamma_p; \Sigma; \cdot \vdash_\Delta P_s$), such that P_s is equivalent modulo secrets.*
3. *A pair (ρ_1, ρ_2) of well-typed 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. *$P_s, P \vdash (H, e)[\rho_1] \to^*_\ell (H', e')[\rho'_1]$ if and only if, $P_s, P \vdash (H, e)[\rho_2] \to^*_\ell (H', e')[\rho'_2]$, and*
2. *$\Gamma_s, \Gamma_P; \Sigma'; \Gamma' \vdash (H', e') : \tau$*

3.2 C*: An intermediate language

We move from λow* to Clight in two steps. The C* intermediate language retains λow*'s explicit scoping structure, but switches the calling convention to maintain an explicit call-stack of continuations (separate from the stack memory regions). C* also switches to a more C-like syntax, separates side effect-free expressions from effectful statements.

$$\begin{array}{lcl} \hat{P} & ::= & \overrightarrow{\mathsf{fun}\ f\,(x:\tau):\tau\,\{\ \overrightarrow{s}\ \}} \\ \hat{e} & ::= & n \mid () \mid x \mid \hat{e} + \hat{e} \mid \{\overrightarrow{f = \hat{e}}\} \mid \hat{e}.f \mid \&\hat{e} \to f \\ s & ::= & \tau\ x = \hat{e} \mid \tau\ x = f(\hat{e}) \mid \mathsf{if}\ \hat{e}\ \mathsf{then}\ \overrightarrow{s}\ \mathsf{else}\ \overrightarrow{s} \mid \mathsf{return}\ \hat{e} \\ & & \mid \{\ \overrightarrow{s}\ \} \mid \tau\ x[n] \mid \tau\ x = *[\hat{e}] \mid *[\hat{e}] = \hat{e} \mid \mathsf{memset}\ \hat{e}\ n\ \hat{e} \end{array}$$

The syntax is unsurprising, with two notable exceptions. First, despite the closeness to C syntax, contrary to C and similarly to λow*, block scopes are not required for branches of a conditional statement, so that any local variable or local array declared in a conditional branch, if not enclosed by a further block, is still live after the conditional statement. Second, non-array local variables are immutable after initialization.

Operational semantics, in contrast to λow* A C* evaluation configuration C consists of a stack S, a variable assignment V and a statement list $\overrightarrow{s}$ to be reduced. A stack is a list of frames. A frame F includes frame memory M, local variable assignment V to be restored upon function exit, and continuation E to be restored upon function exit. Frame memory M is optional: when it is $\perp$, the frame is called a "call frame"; otherwise a "block frame", allocated whenever entering a statement block and deallocated upon exiting such block. A frame memory is just a partial map from block identifiers to value lists. Each C* statement performs at most one function call, or otherwise, at most one side effect. Thus, C* is deterministic.

The semantics of C* is shown to the right in Figure 5, also illustrating the translation from λow* to C*. There are three main differences. First, C*'s calling convention (rule CALL) shows an explicit call frame being pushed on the stack, unlike λow*'s β reduction. Additionally, C* expressions do not have side effects and do not access memory; thus, their evaluation order does not matter and their evaluation can be formalized as a big-step semantics; by themselves, expressions do not produce events. This is apparent in rules like CIFF and CREAD, where the expressions are evaluated atomically in the premises. Finally, newbuf in λow* is translated to an array declaration followed by a separate initialization. In C*, declaring an array allocates a fresh memory block in the current memory frame, and makes its memory locations

$$\frac{}{P \vdash (H, \mathsf{withframe}\ e) \to_{\cdot} (H; \{\}, \mathsf{pop}; e)}\ \textsc{WF}$$

$$\frac{}{P \vdash (H; _, \mathsf{pop}\ v) \to_{\cdot} (H, v)}\ \textsc{Pop}$$

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

$$\frac{H(b, n + n_1) = v \qquad \ell = \mathsf{read}(b, n + n_1)}{P \vdash (H, \mathsf{let}\ x = \mathsf{readbuf}\ (b, n)\ n_1\ \mathsf{in}\ e) \to_{\ell} (H, e[v/x])}\ \textsc{LRd}$$

$$\frac{\begin{array}{l} b \notin \mathsf{dom}(H; h) \quad h_1 = h[b \mapsto v^n] \\ \ell = \mathsf{write}(b, 0), \dots, \mathsf{write}(b, n-1) \quad e_1 = e[(b,0)/x] \end{array}}{P \vdash (H; h, \mathsf{let}\ x = \mathsf{newbuf}\ n\ (v : \tau)\ \mathsf{in}\ e) \to_{\ell} (H; h_1, e_1)}\ \textsc{New}$$

$$\frac{P(f) = \lambda y : \tau_1.\ e_1 : \tau_2}{P \vdash (H, \mathsf{let}\ x : \tau = f\ v\ \mathsf{in}\ e) \to (H, \mathsf{let}\ x : \tau = e_1[v/y]\ \mathsf{in}\ e)}\ \textsc{App}$$

$$\frac{}{\hat{P} \vdash (S, V, \{\overrightarrow{s_1}\}; \overrightarrow{s_2}) \rightsquigarrow (S; (\{\}, V, \square; \overrightarrow{s_2}), V, \overrightarrow{s_1})}\ \textsc{Block}$$

$$\frac{}{\hat{P} \vdash (S; (M, V', E), V, []) \rightsquigarrow (S, V', E\,[()])}\ \textsc{Empty}$$

$$\frac{\llbracket \hat{e} \rrbracket_{(V)} = 0}{\hat{P} \vdash (S, V, \mathsf{if}\ \hat{e}\ \mathsf{then}\ \overrightarrow{s_1}\ \mathsf{else}\ \overrightarrow{s_2}; \overrightarrow{s}) \rightsquigarrow_{\mathsf{brF}} (S, V, \overrightarrow{s_2}; \overrightarrow{s})}\ \textsc{CIfF}$$

$$\frac{\llbracket \hat{e} \rrbracket_{(V)} = (b, n, \overrightarrow{f}) \quad \mathsf{Get}(S, (b, n, \overrightarrow{f})) = v \quad \ell = \mathsf{read}\ (b, n, \overrightarrow{fd})}{\hat{P} \vdash (S, V, \tau\ x = *[\hat{e}]; \overrightarrow{s}) \rightsquigarrow_{\ell} (S, V[x \mapsto v], \overrightarrow{s})}\ \textsc{CRead}$$

$$\frac{S = S'; (M, V, E) \qquad b \notin S \qquad V' = V[x \mapsto (b, 0, [])]}{\hat{P} \vdash (S, V, \tau\ x[n]; \overrightarrow{s}) \rightsquigarrow (S'; (M[b \mapsto \bot^n], V, E), V', \overrightarrow{s})}\ \textsc{ArrDecl}$$

$$\frac{\hat{P}(f) = \mathsf{fun}\ (y : \tau_1) : \tau_2 \{\ \overrightarrow{s_1}\ \} \qquad \llbracket \hat{e} \rrbracket_{(V)} = v}{\hat{P} \vdash (S, V, \tau\ x = f\ \hat{e}; \overrightarrow{s}) \rightsquigarrow (S; (\bot, V, \tau\ x = \square; \overrightarrow{s}), V[y \mapsto v], \overrightarrow{s_1})}\ \textsc{Call}$$

Figure 5. Selected semantic rules from λow* (left) and C* (right)

available but uninitialized. Memory write (resp. read) produces a write (resp. read) event. memset $\hat{e}_1\ m\ \hat{e}_2$ produces m write events, and can be used only for arrays.

Correctness of the λow*-to-C* transformation We proved that execution traces are exactly preserved from λow* to C*:

Lemma 1 (λow* to C*). *Let P be a λow* program and e be a λow* entry point expression, and assume that they compile: $\Downarrow (P) = \hat{P}$ for some C* program $\hat{P}$ and $\downarrow (e) = \overrightarrow{s}; \hat{e}$ for some C* list of statements $\overrightarrow{s}$ and expression $\hat{e}$.*

Let V be a mapping of local variables containing the initial values of secrets. Then, the C program $\hat{P}$ terminates with trace ℓ and return value v, i.e., $\hat{P} \vdash ([], V, \overrightarrow{s}; \mathsf{return}\ \hat{e}) \overset{\ell,*}{\to} ([], V', \mathsf{return}\ v)$ if, and only if, so does the λow* program: $P \vdash (\{\}, e[V]) \overset{\ell,*}{\to} (H', v)$; and similarly for divergence.*

In particular, if the source λow* program is safe, then so is the target C* program. It also follows that the trace equality security property is preserved from λow* to C*. We prove this theorem by bisimulation. In fact, it is enough to prove that any λow* behavior is a C* behavior, and flip the diagram since C* is deterministic. That C* semantics use big-step semantics for C* expressions complicates the bisimulation proof a bit because λow* and C* steps may go out-of-sync at times. Within the proof we used a relaxed notion of simulation ("quasi-refinement") that allows this temporary discrepancy by some stuttering, but still implies bisimulation.

3.3 From C* to CompCert Clight and beyond

CompCert Clight is a deterministic (up to system I/O) subset of C with no side effects in expressions, and actual byte-level representation of values. Clight has a realistic formal semantics [31, 50] and tractable enough to carry out the correctness proofs of our transformations from λow* to C. More importantly, Clight is the source language of the CompCert compiler backend, which we can thus leverage to preserve at least safety and functional correctness properties of Low* programs down to assembly.[3]

We prepend each memory access and conditional statement in the Clight generated code with a *built-in call*, a no-op annotation whose only purpose is to produce an event in the trace. The main two differences between C* and Clight, which our translation deals with as described below, are local structures, and scope management for local variables.

Local structures C* handles local structures as first-class values, whereas Clight only supports non-compound data (integers, floating-points or pointers) as values.

If we naively translate local C* structures to C structures in Clight, then CompCert will allocate them in memory. This increases the number of memory accesses, which not only introduces discrepancies in the security preservation proof from C* to Clight, but also introduces significant performance overhead compared to GCC, which optimizes away structures whose addresses are never taken.

Instead, we split a structure into the sequence of all its non-compound fields, each of which is to be taken as a potentially non-stack-allocated local variable,[4] except for functions that return structures, where, as usual, we add, as an extra argument to the callee, a pointer to the memory location written to by the callee and read by the caller.

Local variable hoisting Scoping rules for C* local arrays are not exactly the same as in C, in particular for branches of

[3] As a subset of C, Clight can be compiled by any C compiler, but only CompCert provides formal guarantees.

[4] Our benchmark without this structure erasure runs 20% slower than with structure erasure, both with CompCert 2.7. Without structure erasure, code generated with CompCert is 60% slower than with `gcc -O1`. CompCert-generated code without structure erasure may even segfault, due to stack overflow, which structure erasure successfully overcomes.

conditional statements. So, it is necessary to hoist all local variables to function-scope. CompCert 2.7.1 does support such hoisting but as an unproven elaboration step. While existing formal proofs (e.g., Dockins' [39, §9.3]) only prove functional correctness, we also prove preservation of security guarantees, as shown below.

Proof techniques Contrary to the λow*-to-C* transformation, our subsequent passes modify the memory layout leading to differences in traces between C* to Clight, due to pointer values. Thus, we need to address security preservation separately from functional correctness: for each memory-changing pass, we first reinterpret the source C* program with different event traces, before carrying out the actual transformation in a trace-preserving way. Our detailed functional correctness and security preservation proofs from λow* to Clight can be found in the appendix.

Towards assembly code We claim that our reinterpretation techniques can be generalized to most passes of CompCert down to assembly. While we leave such generalization as future work, some guarantees from C to assembly can be derived using instrumented CompCert and LLVM [14, 18] turned into *certifying* (rather than certified) compilers where security guarantees are statically rechecked on the compiled code through translation validation, thus re-establishing them independently of source-level security proofs. In this case, rather than being fully preserved down to the compiled code, Low*-level proofs are still useful to *practically* reduce the risk of failures in translation validation. By contrast, applying our proof-preservation techniques to CompCert aims to avoid this further compile-time check, eliminating the risk of compilation failures.

4. KreMLin: A compiler from Low* to C

The erasure and extraction transformations described in the preceding section were already implemented as part of F*'s pre-existing OCaml extraction facility. We wrote a new tool called KreMLin that takes the extracted output from F*, then further rewrites it until it falls within the λow* subset formalized above; after which the tool performs the Low* to C* transformation; the C* to C transformation; the pretty-printing to a set of C files that can be compiled. KreMLin generates C99 code that may be compiled by GCC; Clang; Microsoft's C compiler or CompCert.

KreMLin accepts a larger input language than λow*; for the sake of programmer productivity, KreMLin handles non-recursive, parameterized data types and type abbreviations; pattern matches; tuples; nested let-bindings, assignments and conditionals. A combination of monomorphization, translation of data types to tagged unions and compilation of pattern matches reduce these constructs to λow*. Of particular interest are *stratification* and *hoisting*. Stratification goes from an expression language to a statement language of the form expected by C*, meaning that buffer allocations, assignments and conditionals are placed in statement position before going to C*. Hoisting, as discussed in §3.3, deals with the discrepancy between C99 block scope and Low* `with_frame`; a buffer allocated under a `then` branch must be hoisted to the nearest enclosing `push_frame`, otherwise its lifetime would be shortened by the resulting C99 block after translation.

KreMLin puts a strong emphasis on generating readable C, in the hope that security experts not familiar with F* can review the generated C code. Names are preserved; data types with only constant constructors generate an `enum` and are destructured by a `switch`; functions that take `unit` are compiled into functions with no parameters; functions that return `unit` are compiled into `void`-returning functions. The internal architecture relies on an abstract C AST and what we believe is a correct C pretty-printer.

KreMLin represents about 6,800 lines of OCaml, along with a minimal set of primitives implemented in a few hundred lines of C. After F* has extracted and erased the AEAD development, KreMLin takes less than a second to generate the entire set of C files. The implementation of KreMLin is optimized for readability and modularity; there was no specific performance concern in this first prototype version. KreMLin was designed to accept multiple frontends and support multiple backends. We intend to announce one new backend and one new frontend soon.

5. Building verified Low* applications

Application	LoC	C LoC	%annot	Verif. time
HACL*	6,050	11,220	28%	0h 52m
AEAD	13,743	14,292	56.5%	1h 10m

Table 1. Evaluation of verified Low* applications (time reported on an Intel Core E5 1620v3 CPU)

In this section, we describe two examples (summarized in Table 1) that show how Low* can be used to build applications that balance complex verification goals with high performance. First, we implement HACL*, an efficient library of cryptographic primitives that are verified to be memory safe, side-channel resistant, and, where there exists a simple mathematical specification, functionally correct. Second, we show how to use Low* for type-based cryptographic security verification by implementing and verifying the AEAD construction in the Transport Layer Security (TLS) protocol. We show how this Low* library can be integrated within miTLS, an F* implementation of TLS that is compiled to OCaml.

5.1 HACL*: Fast and safe cryptographic library

In the wake of numerous security vulnerabilities, Bernstein et al. [25] argue that libraries like OpenSSL are inherently vulnerable to attacks because they are too large, offer too many obsolete options, and expose a complex API that programmers find hard to use securely. Instead they propose a new cryptographic API called NaCl that uses a small set of modern cryptographic primitives, such as Curve25519 [23] for key exchange, the Salsa family of symmetric encryption

algorithms [24], which includes XSalsa20 and ChaCha20, and Poly1305 for message authentication [22]. These primitives were all designed to be fast and easy to implement in a side-channel resistant coding style. Furthermore, the NaCl API does not directly expose these low-level primitives to the programmer. Instead it recommends the use of simple composite functions for symmetric key authenticated encryption (`secretbox`/`secretbox_open`) and for public key authenticated encryption (`box`/`box_open`).

The simplicity, speed, and robustness of the NaCl API has proved popular among developers. Its most popular implementation is Sodium [5], which has bindings for dozens of programming languages and is written mostly in C, with a few components in assembly. An alternative implementation called TweetNaCl [26] seeks to provide a concise implementation that is both readable and *auditable* for memory safety bugs, a useful point of comparison for our work. With Low*, we show how we can take this approach even further by placing it on formal, machine-checked ground, without compromising performance.

We implement the NACL API, including all its component algorithms, in a Low* library called HACL*, mechanically verifying that our code is memory safe and side-channel resistant. The C code generated from HACL* is ABI-compatible and can be used as a drop-in replacement for Sodium or TweetNaCl in any application, in C or any other language, that relies on these libraries.

Algorithm	HACL*	Sodium	TweetNaCl
Poly1305	2.2 cycle/B	2.15 cycle/B	27.5 cycle/B
XSalsa20	8.6 cycle/B	7.2 cycle/B	10.1 cycle/B
Box	10.8 cycle/B	9.5 cycle/B	37.5 cycle/B
Curve25519	281 µs/mul	59 µs/mul	522 µs/mul

Table 2. Performance Comparison: 64-bit HACL*, 64-bit Sodium (pure C, no assembly), and TweetNaCl, all compiled using `gcc -O3` and run on an Intel Core i7-4600U CPU

Performance Table 2 compares the performance of HACL* to TweetNaCl and Sodium by running each primitive on a 1MB input. For Curve25519, we measure the time taken for one call to scalar multiplication. Box uses Curve25519 to compute a symmetric key, which it then uses to encrypt a 1MB input. (In this case, the cost of symmetric encryption dominates over Curve25519.) We report averages over 1000 iterations expressed in cycles/byte. We observe that for Poly1305 and XSalsa20, HACL* achieves comparable performance to Sodium's optimized C code and significantly better performance than TweetNaCl's concise C implementation. Sodium also offers an assembly implementation of XSalsa20 which is 3 times faster than the C version. Our Curve25519 implementation is about 5 times slower than Sodium, and we pay this performance penalty in exchange for functional correctness, as explained below.

The cost of verification Poly1305 and Curve25519 implementations rely on bigint libraries that implement modular arithmetic over prime fields in terms of operations over bitstrings. For efficiency, and to mitigate certain side-channels, they cannot use generic bigint libraries; instead they implement custom libraries optimized for 8-bit, 32-bit, and 64-bit architectures. For Curve25519, HACL* includes a side-channel resistant, 64-bit bigint proven correct against its mathematical specification. Proving functional correctness required code restructuring and the introduction of auxiliary values in order to express and prove lemmas, and these changes account for our performance loss. Even so, our Curve25519 code is two orders of magnitude faster than the 20ms reported for a verified OCaml implementation of Curve25519 [73]. The 64-bit Poly1305 code in Table 2 is only verified for memory safety and side-channels, and consequently pays no penalty. We additionally verified for correctness a 32-bit bigint library for Poly1305 and it experiences a similar performance degradation in comparison to 32-bit Sodium, taking 16.0 cycles/byte versus 5.2 cycles/byte for the unverified code. We plan to continue to optimize our code and proofs. All the above results were obtained with GCC; to use verified assembly code via CompCert, we pay an additional cost. For example, our 64-bit XSalsa20, when compiled via CompCert incurs a 6.1x slowdown over GCC.

PneuTube: Fast encrypted file transfer Efficient security applications are typically written in C, not just for cryptography, but also for access to low-level libraries for disk and network I/O. We use Low* to provide high-assurance C libraries like HACL* that can be included within unverified C projects, but we can also write standalone applications directly in Low*.

PneuTube is a Low* program that securely transfers files from a host A to a host B across an untrusted network. Unlike classic secure channel protocols like TLS and SSH, PneuTube is *asynchronous*, meaning that if B is offline, the file may be cached at some untrusted cloud storage provider and retrieved later. PneuTube breaks the file into *blocks* and encrypts each block using the `box` API in HACL* (with an optimization that caches the result of Curve25519). It also protects file metadata, including the file name and modification time, and it hides the file size by padding the file before encryption. Hence, PneuTube provides better traffic analysis protection than SSH and TLS, which leak file length.

PneuTube's performance is determined by a combination of the crypto library, disk access (to read and write the file at each end) and network I/O. We link PneuTube to standard TCP sockets, and to a memory-mapped file I/O library written in C, and we carefully choose the block-size to balance the load across our encryption/decryption pipelines. We verify that our code is memory-safe, side-channel resistant, and that it uses the I/O libraries correctly (e.g., it only reads a file between calling open and close.)

Since PneuTube is asynchronous, a sender does not have to wait for a key exchange to be complete before it starts sending data. This design is particularly rewarding on high-latency network connections, but even when transferring a 1GB file from one TCP port to another on the same machine, PneuTube takes just 6s. In comparison, SCP (using SSH with ChaCha20-Poly1305) takes 8 seconds.

5.2 Fast cryptographically secure AEAD for miTLS

We use our cryptographically secure AEAD library (§2.4) within miTLS [28], an existing implementation of TLS in F*. In a previous verification effort, AEAD encryption was idealized as a cryptographic assumption (concretely realized using bindings to OpenSSL) to show that miTLS implements a secure authenticated channel. However, given vulnerabilities such as CVE-2016-7054, this AEAD idealization is a leap of faith that can undermine security when the real implementation diverges from its ideal behavior.

We integrated our verified AEAD construction within miTLS at two levels. First, we replace the previous AEAD idealization with a module that implements a similar ideal interface but translates the state and buffers to Low* representations. This approach is perfectly secure, in as much as the security of TLS is now reduced to the PRF and MAC idealizations in AEAD when extracted to OCaml. For performance, we also integrate AEAD at the C level by substituting the OpenSSL bindings with bindings to the C-extracted version of AEAD. This introduces a slight security gap, as a small adapter that translates miTLS bytes to Low* buffers and calls into AEAD in C is not verified.

We confirm that miTLS with our verified AEAD interoperates with mainstream implementations of TLS 1.2 and TLS 1.3 on ChaCha20-Poly1305 ciphersuites. The OCaml-extracted version only achieves a 74 KB/s throughput on a 1GB file transfer. Thus, while it is useful as a reference for verification, it is not usable in practice. In C, we achieve a 69 MB/s throughput, which is about 20% of the 354 MB/s obtained with OpenSSL. There is a noticeable cost stemming from the AEAD proof infrastructure, such as indirections needed for algorithmic agility. Better specialization support in KreMLin may help reduce this overhead.

6. Related work

Many approaches have been proposed for verifying the functional correctness and security of efficient low-level code. A first approach, is to build verification frameworks for C using verification condition generators and SMT solvers [36, 44, 47], While this approach has the advantage of being able to verify existing C code, this is very challenging: one needs to deal with the complexity of C and with any possible optimization trick in the book. Moreover, one needs an expressive specification language and escape hatches for doing manual proofs in case SMT automation fails. So others have deeply embedded C, or C-like languages, into proof assistants such as Coq [17, 21, 34] and Isabelle [62, 72] and built program logics and verification infrastructure starting from that. This has the advantage of using the full expressive power of the proof assistant for specifying and verifying properties of low-level programs. This remains a very labor-intensive task though, because C programs are very low-level and working with a deep embedding is often cumbersome. Acknowledging that uninteresting low-level reasoning was a determining factor in the size of the seL4 verification effort [48], Greenawayet al. [42, 43] have recently proposed sophisticated tools for automatically abstracting the low-level C semantics into higher-level monadic specifications to ease reasoning. We take a different approach: we give up on verifying existing C code and embrace the idea of writing low-level code in a subset of C shallowly embedded in F*. This shallow embedding has significant advantages in terms of reducing verification effort and thus scaling up verification to larger programs. This also allows us to port to C only the parts of an F* program that are a performance bottleneck, and still be able to verify the complete program.

Verifying the correctness of low-level cryptographic code is receiving increasing attention [17, 21, 40]. The verified cryptographic applications we have written in Low* and use for evaluation in this paper are an order of magnitude larger than most previous work. Moreover, for AEAD we target not only functional correctness, but also cryptographic security.

In order to prevent the most devastating low-level attacks, several researchers have advocated dialects of C equipped with type systems for memory safety [37, 45, 69]. Others have designed new languages with type systems aimed at low-level programming, including for instance linear types as a way to deal with memory management [15, 55]. One drawback is the expressiveness limitations of such type systems: once memory safety relies on more complex invariants than these type systems can express, compromises need to be made, in terms of verification or efficiency. Low* can perform arbitrarily sophisticated reasoning to establish memory safety, but does not enjoy the benefits of efficient decision procedures [8] and currently cannot deal with concurrency.

We are not the first to propose writing efficient and verified C code in a high-level language. LMS-Verify [16] recently extended the LMS meta-programming framework for Scala with support for lightweight verification. Verification happens at the generated C level, which has the advantage of taking the code generation machinery out of the TCB, but has the disadvantage that the verification happens very far away from the original source code.

Bedrock [35] is a generative meta-programming tool for verified low-level programming in Coq. The idea is to start from assembly and build up structured code generators that are associated verification condition generators. The main advantage of this "macro assembly language" view of low-level verification is that no performance is sacrificed while obtaining some amount of abstraction. One disadvantage is that the verified code is not portable.

A concurrent submission to Oakland, “Implementing and Proving the TLS 1.3 Record Layer”, is included as an anonymous supplementary material. It describes a cryptographic model and proof of security for AEAD using a combination of F* verification and meta-level cryptographic idealization arguments. To make the point that verified code need not be slow, the Oakland submission mentions that the AEAD implementation can be “extracted to C using an experimental backend for F*”, but makes no further claims about this backend. The current work introduces the design, formalization, implementation, and experimental evaluation of this C backend for F*.

7. Conclusion

We have acquired significant experience writing large-scale developments in Low*, and can confidently say that the approach scales up, both in terms of verification effort and performance of the generated code. Some algorithms suffer from a performance penalty, due to intermediary values and indirections required for proving functional correctness. As we improve our toolchain, we hope to get rid of these inefficiencies.

Low* was driven by the demands of cryptographic code and libraries, and having a concrete use-case was highly beneficial to help drive the design. However, our approach is general and relevant beyond cryptographic algorithms, as exemplified by PneuTube, a systems application that integrates with C-style network APIs. We plan to push further on the cryptographic side, by writing the rest of miTLS in the Low* fragment of F*, but also to expand our reach and write more systems-oriented code, such as parsers and servers.

These new applications will require new features from our toolchain; we plan to add support for polymorphic code via monomorphization; an automatically-managed heap region to ease the transition from existing code, using a conservative GC or reference-counting; support for arenas and mass deallocation in the manually-managed heap. We are looking closely at also targetting C++11, to leverage closures and in-library shared (reference counted) pointers.

Several Low*-to-Low* passes are not formalized; these include the briefly mentioned StackInline effect, and the various hoistings required to go to the statement form expected by the λow*-to-C* translation. In the short run, we plan to formalize these extra passes on paper. In the long run, we would like to rewrite KreMLin in F* and construct a machine checked proof of semantics preservation.

References

[1] Common weakness enumeration (CWE-415: Double free). URL http://cwe.mitre.org/data/definitions/415.html.

[2] The Heartbleed bug. http://heartbleed.com/.

[3] Common weakness enumeration (CWE-190: Integer overflow or wraparound). URL https://cwe.mitre.org/data/definitions/190.html.

[4] Common weakness enumeration (CWE-416: Use after free). URL http://cwe.mitre.org/data/definitions/416.html.

[5] The sodium crypto library (libsodium). URL https://www.gitbook.com/book/jedisct1/libsodium/details.

[6] nocrypto: OCaml cryptographic library. URL https://github.com/mirleft/ocaml-nocrypto.

[7] OpenSSL: Cryptography and SSL/TLS toolkit. URL https://www.openssl.org/.

[8] The Rust programming language. URL https://www.rust-lang.org.

[9] CVE-2016-7054: ChaCha20/Poly1305 heap-buffer-overflow, Nov. 2016. URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-7054.

[10] J. Afek and A. Sharabani. Dangling pointer – smashing the pointer for fun and profit. BlackHat USA, July 2007.

[11] D. Ahman, C. Hriţcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy. Dijkstra monads for free. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Jan. 2017. URL https://www.fstar-lang.org/papers/dm4free/. To appear.

[12] N. J. AlFardan and K. G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540, 2013.

[13] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir. Verifiable side-channel security of cryptographic implementations: Constant-time MEE-CBC. In *Fast Software Encryption - 23rd International Conference, FSE 2016*, pages 163–184, 2016.

[14] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium, USENIX Security 16*, pages 53–70, 2016.

[15] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, et al. COGENT: Verifying high-assurance file system implementations. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 175–188. ACM, 2016.

[16] N. Amin and T. Rompf. LMS-Verify: Abstraction without regret for verified systems programming. To appear in 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'17), 2017. URL https://www.cs.purdue.edu/homes/rompf/papers/amin-draft2016b.pdf.

[17] A. W. Appel. Verification of a cryptographic primitive: SHA-256. *ACM Trans. Program. Lang. Syst.*, 37(2):7, 2015.

[18] G. Barthe, G. Betarte, J. D. Campo, C. D. Luna, and D. Pichardie. System-level non-interference for constant-time cryptography. In *2014 ACM SIGSAC Conference on Computer and Communications Security, CCS 2014*, pages 1267–1279, 2014.

[19] D. Benjamin. poly1305-x86.pl produces incorrect output. https://mta.openssl.org/pipermail/openssl-dev/2016-March/006161, 2016.

[20] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA 2002*, pages 1–12. ACM, 2002.

[21] L. Beringer, A. Petcher, Q. Y. Katherine, and A. W. Appel. Verified correctness and security of OpenSSL HMAC. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 207–221, 2015.

[22] D. J. Bernstein. The Poly1305-AES message-authentication code. In *International Workshop on Fast Software Encryption*, pages 32–49. Springer, 2005.

[23] D. J. Bernstein. Curve25519: new Diffie-Hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

[24] D. J. Bernstein. The Salsa20 family of stream ciphers. In *New stream cipher designs*, pages 84–97. Springer, 2008.

[25] D. J. Bernstein, T. Lange, and P. Schwabe. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America, LATINCRYPT 2012*, pages 159–176. Springer, 2012.

[26] D. J. Bernstein, B. Van Gastel, W. Janssen, T. Lange, P. Schwabe, and S. Smetsers. TweetNaCl: A crypto library in 100 tweets. In *International Conference on Cryptology and Information Security in Latin America, LATINCRYPT 2014*, pages 64–83, 2014.

[27] K. Bhargavan and G. Leurent. On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. Cryptology ePrint Archive, Report 2016/798, 2016. `http://eprint.iacr.org/2016/798`.

[28] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with verified cryptographic security. In *IEEE Symposium on Security and Privacy*, pages 445–459, 2013.

[29] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, , A. Pironti, and P.-Y. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *2014 IEEE Symposium on Security and Privacy*, pages 98–113, 2014.

[30] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Zanella-Béguelin. Proving the TLS handshake secure (as it is). In *Advances in Cryptology–CRYPTO 2014*, pages 235–255. Springer, 2014.

[31] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43 (3):263–288, 2009.

[32] H. Böck. Wrong results with Poly1305 functions. `https://mta.openssl.org/pipermail/openssl-dev/2016-March/006413`, 2016.

[33] H. Böck, A. Zauner, S. Devlin, J. Somorovsky, and P. Jovanovic. Nonce-disrespecting adversaries: Practical forgery attacks on GCM in TLS. Cryptology ePrint Archive, Report 2016/475, 2016. `http://eprint.iacr.org/2016/475`.

[34] H. Chen, X. N. Wu, Z. Shao, J. Lockerman, and R. Gu. Toward compositional verification of interruptible OS kernels and device drivers. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016*, pages 431–447, 2016.

[35] A. Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *ACM SIGPLAN Notices*, volume 48, pages 391–402. ACM, 2013.

[36] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *International Conference on Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

[37] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *European Symposium on Programming*, pages 520–535. Springer, 2007.

[38] I. Dobrovitski. Exploit for CVS double free() for Linux pserver, Feb. 2003. URL `http://archives.neohapsis.com/archives/fulldisclosure/2003-q1/0545.html`.

[39] R. W. Dockins. *Operational Refinement for Compiler Correctness*. PhD thesis, Princeton University, 2012.

[40] J. Dodds. Part one: Verifying s2n HMAC with SAW. Galois Blog, Sept. 2016. URL `https://galois.com/blog/2016/09/specifying-hmac-in-cryptol/`.

[41] T. Duong and J. Rizzo. Here come the ⊕ ninjas. Available at `http://nerdoholic.org/uploads/dergln/beast_part2/ssl_jun21.pdf`, May 2011.

[42] D. Greenaway, J. Andronick, and G. Klein. Bridging the gap: Automatic verified abstraction of C. In *3rd International Conference on Interactive Theorem Proving, ITP 2012*, volume 7406 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2012.

[43] D. Greenaway, J. Lim, J. Andronick, and G. Klein. Don't sweat the small stuff: formal verification of C code without the pain. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014*, pages 429–439. ACM, 2014.

[44] B. Jacobs, J. Smans, and F. Piessens. The VeriFast program verifier: A tutorial. iMinds-DistriNet, Department of Computer Science, KU Leuven - University of Leuven, Belgium, 2014. URL `https://people.cs.kuleuven.be/~bart.jacobs/verifast/tutorial.pdf`.

[45] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[46] D. Kaloper-Meršinjak, H. Mehnert, A. Madhavapeddy, and P. Sewell. Not-quite-so-broken TLS: Lessons in re-engineering a security protocol specification and implementation. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 223–238, 2015.

[47] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.

[48] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the Symposium on Operating Systems Principles*, pages 207–220. ACM, 2009.

[49] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology – CRYPTO 1996*, pages 104–113. Springer, 1996.

[50] X. Leroy. The CompCert C verified compiler. `http://compcert.inria.fr/`, 2004–2016.

[51] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[52] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[53] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert memory model, version 2. Research report RR-7987, INRIA, June 2012. URL `http://hal.inria.fr/hal-00703441`.

[54] P. Letouzey. A new extraction for Coq. In *Types for proofs and programs*, pages 200–219. Springer, 2002.

[55] N. D. Matsakis and F. S. Klock II. The Rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[56] J. McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

[57] B. Möller, T. Duong, and K. Kotowicz. This POODLE Bites: Exploiting The SSL 3.0 Fallback. Available at `https://www.openssl.org/~bodo/ssl-poodle.pdf`, 2014.

[58] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *8th International Conference on Information Security and Cryptology, ICISC 2005*, pages 156–168. Springer, 2006.

[59] Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF protocols. IETF RFC 7539, 2015.

[60] J. D. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security & Privacy*, 2(4):20–27, 2004.

[61] J. Rizzo and T. Duong. The CRIME Attack, September 2012.

[62] N. Schirmer. *Verification of sequential imperative programs in Isabelle-HOL*. PhD thesis, Technical University Munich, 2006.

[63] B. Smyth and A. Pironti. Truncating TLS connections to violate beliefs in web applications. Technical Report hal-01102013, Inria, Oct. 2014. URL `https://hal.inria.fr/hal-01102013`.

[64] J. Somorovsky. Systematic fuzzing and testing of TLS libraries. In *23rd ACM Conference on Computer and Communications Security, CCS 2016*, 2016.

[65] M. Stevens, P. Karpman, and T. Peyrin. Freestart collision for full SHA-1. In *Advances in Cryptology – EUROCRYPT 2016*, pages 459–483. Springer, 2016.

[66] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, 2016. URL `https://www.fstar-lang.org/papers/mumon/`.

$$\frac{s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-2}} s_{n-1} \xrightarrow{t_{n-1}} s_n \qquad s_0 \text{ initial} \qquad s_n \text{ final with return value } r \qquad t = t_0; t_1; \ldots; t_{n-2}; t_{n-1}}{\mathsf{Terminates}(t, r)}$$

$$\frac{s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots \qquad s_0 \text{ initial} \qquad T = t_0; t_1; \ldots}{\mathsf{Diverges}(T)}$$

$$\frac{s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-2}} s_{n-1} \xrightarrow{t_{n-1}} s_n \qquad s_0 \text{ initial} \qquad s_n \text{ not final} \qquad t = t_0; t_1; \ldots; t_{n-2}; t_{n-1}}{\mathsf{GoesWrong}(t)}$$

[67] R. Święcki. ChaCha20/Poly1305 heap-buffer-overflow. CVE-2016-7054, 2016.

[68] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, pages 48–62. IEEE Computer Society, 2013.

[69] D. Tarditi. Extending C with bounds safety. Checked C Technical Report, Version 0.6, Nov. 2016. URL `https://github.com/Microsoft/checkedc`.

[70] M. Tofte and J.-P. Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, Feb. 1997.

[71] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *2nd USENIX Workshop on Electronic Commerce, WOEC 1996*, pages 29–40, 1996.

[72] S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock, and M. Norrish. Mind the gap. In *22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 500–515. Springer, 2009.

[73] J. K. Zinzindohoué, E.-I. Bartzia, and K. Bhargavan. A verified extensible library of elliptic curves. In *IEEE Computer Security Foundations Symposium (CSF)*, 2016.

A. Big-stepping a small-step semantics

Actual observable behaviors will not account for the detailed sequence of small-step transitions taken. Given an execution first represented as the sequence of its transition steps from the initial state, we follow CompCert to derive an observable behavior by only characterizing termination or divergence and collecting the event traces, thus erasing all remaining information about the execution (number of transition steps, sequence of configurations, etc.) by *big-stepping* the small-step semantics as shown in Figure A.

B. C* and λow* Definition

Notations used in the document are summarized in Figure 6. Function name f and variable name x are of different syntax classes. A term is closed if it does not contain unbound

variables (but can contain function names). The grammar of C* and λow* are listed in Figure 7 and 9 respectively. C* syntax is defined in such a way that C* expressions do not have side effects (but can fail to evaluate because of illegal memory read). Locations, which only appear during reduction, consist of a block id, an offset and a list of field names (a "field path"). The "getting field address" syntax $\&e \to fd$ is for constructing a pointer to a field of a struct pointed to by pointer e.

In λow* syntax, buffer allocation, buffer write and function application are distinctive syntax constructs (not special cases of let-binding). In this way we force effectful operations to be in let-normal-form, to be aligned with C* (C* does not allow effectfull expressions because of C's nondeterministic expression evaluation order). Let-binding and anonymous let-binding are also distinctive syntax constructs, because they need to be translated into different C* constructs. Locations and pop le only appear during reduction.

The operational semantics of C* is listed in Figure 11 and 12. Because C expressions do not have a deterministic evaluation order, in C* we use a mixed big-step/small-step operational semantics, where C* expressions are evaluated with big-step semantics defined by the evaluation function (interpreter) $[\![e]\!]_{(p,V)}$, while C* statements are evaluated with small-step semantics. Definitions used in C* semantics are summarized in Figure 8. A C* evaluation configuration C consist of a stack S, a variable assignment V and a statement list ss to be reduced. A stack is a list of frames. A frame F includes frame memory M, variable assignment V to be restored upon function exit, and continuation E to be restored upon function exit. Frame memory M is optional: when it is none, the frame is called a "call frame"; otherwise a "block frame". A frame memory is just a partial map from block ids to value lists.

Both C* and λow* reductions generate traces that include memory read/write with the address, and branching to true/false. Reduction steps that don't have these effects are silent.

$\overrightarrow{a}$	list	$\widetilde{a}$	option a
$\perp$	None	$\lfloor a \rfloor$	Some a
n	integer	x	variable name
f	function name	fd	field name
$a \rightharpoonup b$	partial map	$\{\}$	empty map
$\{x \mapsto a\}$	singleton map	$m[x \mapsto a]$	map update
$[]$	empty list	$a; b$	list concat or cons
$[a/x]b$	substitute a for x in b		

Figure 6. Notations

p ::=		program
	$\overrightarrow{d}$	series of declarations
d ::=		declaration
	fun $f\,(x : t) : t\,\{\,ss\,\}$	top-level function
	$t\ x = v$	top-level value
ss ::=		statement lists
	$\overrightarrow{s}$	
s ::=		statements
	$t\ x = e$	immutable variable declaration
	$t\ x[n]$	array declaration
	memset $e\ n\ e$	memory set
	$t\ x = f(e)$	application
	$t\ x = *[e]$	read
	$*[e] = e$	write
	if e then ss else ss	conditional
	{ss}	block
	e	expression
	return e	return
e ::=		expressions
	n	integer constant
	$()$	unit value
	x	variable
	$e_1 + e_2$	pointer add (e_1 is a pointer and e_2 is an int)
	$\{\overrightarrow{fd = e}\}$	struct
	$e.fd$	struct field projection
	$\&e \to fd$	struct field address (e is a pointer)
	loc	location
loc ::=		locations
	$(b, n, \overrightarrow{fd})$	

Figure 7. C* Syntax

C. λow* to C* Compilation

The compilation procedure is defined in Figure 14 as inference rules, which should be read as a function defined by pattern-matching, with earlier rules shadowing later rules. The compilation is a partial function, encoding syntactic constraints on λow* programs that can be compiled. For example, compilable λow* top-level functions must be wrapped in a withframe construct.

D. Bisimulation Proof

The main results are Theorem 2 and 3, in terms of some notions defined before them in this section. The two theorems are proved by using the crucial Lemma 4 to "flip the diagram", i.e., proving C* refines λow* by proving λow* refines C*. The flipping relies on the fact that C* is deterministic modulo renaming of block identifiers. An alternative way of determinization to renaming of block identifiers is to have the stream of random coins for choosing block identifiers as part of the configuration (state).

v ::=		values
	n	constant
	$()$	unit value
	$\{\overrightarrow{fd = v}\}$	constant struct
	$(b, n, \overrightarrow{fd})$	location
E ::=		evaluation ctx (plug expr to get stmts)
	$\square; ss$	discard returned value
	$t\ x = \square; ss$	receive returned value
F ::=		frames
	$(\bot, V, E)$	call frame
	$(\lfloor M \rfloor, V, E)$	block frame
M ::=		memory
	$b \rightharpoonup \overrightarrow{\overrightarrow{v}}$	map from block id to list of optional values
V ::=		variable assignments
	$x \rightharpoonup \overrightarrow{v}$	map from variable to value
S ::=		stack
	$\overrightarrow{F}$	list of frames
C ::=		configuration
	(S, V, ss)	
l ::=		label
	read loc	read
	write loc	write
	brT	branch true
	brF	branch false

Figure 8. C* Semantics Definitions

lp ::=		program
	$\overrightarrow{ld}$	series of declarations
ld ::=		declaration
	let $x = \lambda y : t.\ le : t$	top-level function
	let $x : t = v$	top-level value
le ::=		expressions
	n	constant
	$()$	unit value
	x	variable
	$\{\overrightarrow{fd = le}\}$	struct
	$le.fd$	struct field projection
	let $x : t = le$ in le	let-binding
	let _ = le in le	anonymous let-binding
	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
	pop le	pop frame
	$lloc$	location
$lloc$ ::=		locations
	(b, n)	

Figure 9. λow* Syntax

Definition 1 ((Labelled) Transition System)**.** *A transition system is a 5-tuple $(\Sigma, L, \rightsquigarrow, s_0, F)$, where Σ is a set of states, L is a monoid which is the set of labels, $\rightsquigarrow \subseteq \Sigma \times \mathsf{option}\ L \times \Sigma$ is the step relation, s_0 is the initial state, and F is a set of designated final state.*

In the following text, we use ϵ to denote an empty label (the unit of the monoid L), l to range over non-empty (non-ϵ) labels and o to range over possibly empty labels. When the label is empty, we can omit it. The label of multiple steps is the combined label of each steps, using the addition operator of monoid L. We define $a \Downarrow_o a' \stackrel{\mathsf{def}}{=} a \rightsquigarrow_o^* a'$ and $a' \in F$.

Definition 2 (Safety)**.** *A transition system A is safe iff for all s so that $s_0 \rightsquigarrow^* s$, s is unstuck, where $\mathsf{unstuck}(s)$ is defined as either $s \in F$ or there exists s' so that $s \rightsquigarrow s'$.*

Definition 3 (Refinement)**.** *A transition system A refines a transition system B (with the same label set) by R (or R is a refinement for transition system A of transition system B) iff*

1. *there exists a well-founded measure $|-|_b$ (indexed by a B-state b) defined on the set of A-states $\{a : \Sigma_A \mid a\ R\ b\}$;*
2. *$a_0\ R\ b_0$, that is, the two initial states are in relation R;*
3. *for all $a : \Sigma_A$ and $b : \Sigma_B$ such that $a\ R\ b$,*
   - *(a) if $a \rightsquigarrow_o a'$ for some $a' : \Sigma_A$, then there exists $b' : \Sigma_B$ and n such that $b \rightsquigarrow_o^n b'$ and $a'\ R\ b'$ and that $n = 0$ implies that $|a|_b > |a'|_b$;*
   - *(b) if $a \in F_A$, then there exists $b' : \Sigma_B$ such that $b \Downarrow_\epsilon b'$ and $a\ R\ b'$.*

A refines B iff there exists R so that A refines B by R.

Definition 4 (Bisimulation)**.** *A transition system A bisimulates a transition system B iff A refines B and B refines A.*

Definition 5 (Transition System of C* and λow*)**.**

$$\mathsf{sys}_{C^*}(p, V, ss) \stackrel{\mathsf{def}}{=} (C, \{\overrightarrow{l}\}, p \vdash \rightsquigarrow, ([], V, ss), \{([], V', \mathit{return}\ e)\})$$
$$\mathsf{sys}_{\lambda ow^*}(lp, le) \stackrel{\mathsf{def}}{=} (\{(H, le)\}, \{\overrightarrow{ll}\}, lp \vdash \rightsquigarrow, ([], le), \{([], lv)\})$$

In the following text, we treat label read/write (b, n) and read/write $(b, n, [])$ as equal, and if we have a λow* value or substitution, we freely use it as a C* one because coercion from λow* value to C* value is straight-forward.

values

$$lv \quad ::= \quad n \mid () \mid \{\overrightarrow{fd = lv}\} \mid (b, n)$$

evaluation contexts

$$\begin{array}{rcl} LE & ::= & \square \mid LE.fd \mid \{\overrightarrow{fd = lv}; fd = LE; \overrightarrow{fd = le}\} \\ & \mid & \mathsf{subbuf}\ LE\ le\ \mid \mathsf{subbuf}\ lv\ LE\ \mid \mathsf{readbuf}\ LE\ le \\ & \mid & \mathsf{readbuf}\ lv\ LE \mid \mathsf{let}\ x : t = f\ LE\ \mathsf{in}\ le \\ & \mid & \mathsf{let}\ x : t = LE\ \mathsf{in}\ le \mid \mathsf{let}\ _ = LE\ \mathsf{in}\ le \\ & \mid & \mathsf{let}\ x = \mathsf{newbuf}\ n\ (LE : t)\ \mathsf{in}\ le \\ & \mid & \mathsf{pop}\ LE \mid \mathsf{let}\ _ = \mathsf{writebuf}\ LE\ le\ le\ \mathsf{in} \\ & \mid & \mathsf{let}\ _ = \mathsf{writebuf}\ lv\ LE\ le\ \mathsf{in} \\ & \mid & \mathsf{let}\ _ = \mathsf{writebuf}\ lv\ lv\ LE\ \mathsf{in}\ \mid \mathsf{if}\ LE\ \mathsf{then}\ le\ \mathsf{else}\ le \end{array}$$

stack

$$H \quad ::= \quad \overrightarrow{h}$$

stack frame

$$h \quad ::= \quad b \rightharpoonup \overrightarrow{v}$$

label

$$\begin{array}{rcl} ll & ::= & \mathsf{read}\ lloc \\ & \mid & \mathsf{write}\ lloc \\ & \mid & \mathsf{brT} \\ & \mid & \mathsf{brF} \end{array}$$

Figure 10. λow* Semantics Definitions

Theorem 2 (Safety)**.** *For all λow^* program lp, closed expression le and closing substitution V, if $\Downarrow lp = p$, $\downarrow le = ss$ and $\mathsf{sys}_{\lambda ow^*}(lp, V(le))$ is safe, then $\mathsf{sys}_{C^*}(p, V, ss)$ is safe.*

Proof. Appeal to Lemma 4, Lemma 6 and Lemma 5. □

Theorem 3 (Bisimulation)**.** *For all λow^* program lp, closed expression le and closing substitution V, if $\Downarrow lp = p$ and $\downarrow le = ss$, then $\mathsf{sys}_{C^*}(p, V, ss)$ bisimulates $\mathsf{sys}_{\lambda ow^*}(lp, V(le))$.*

Proof. Appeal to Corollary 1, Lemma 6 and Lemma 5. □

Definition 6 (Determinism)**.** *A transition system A is deterministic iff for all s so that $s_0 \rightsquigarrow^* s$, $s \in F$ implies that s cannot take any step, and $s \rightsquigarrow_{o_1} s_1$ and $s \rightsquigarrow_{o_2} s_2$ implies that $o_1 = o_2$ and $s_1 = s_2$.*

Definition 7 (Quasi-Refinement)**.** *A transition system A quasi-refines a transition system B (with the same label set) by R (or R is a quasi-refinement for transition system A of transition system B) iff*

1. *there exists a well-founded measure $|-|_b$ (indexed by a B-state b) defined on the set of A-states $\{a : \Sigma_A \mid a\ R\ b\}$;*
2. *$a_0\ R\ b_0$, that is, the two initial states are in relation R;*
3. *for all $a : \Sigma_A$ and $b : \Sigma_B$ such that $a\ R\ b$,*
   - *(a) if $a \rightsquigarrow_o a'$ for some $a' : \Sigma_A$, then there exists $a'' : \Sigma_A$, $b' : \Sigma_B$ and n such that $a' \rightsquigarrow_\epsilon^* a''$ and $b \rightsquigarrow_o^n b'$ and $a''\ R\ b'$ and that $n = 0$ implies that $|a|_b > |a'|_b$;*
   - *(b) if $a \in F_A$, then there exists $b' : \Sigma_B$ such that $b \Downarrow_\epsilon b'$ and $a\ R\ b'$.*

A quasi-refines B iff there exists R so that A quasi-refines B by R.

Lemma 2 (Quasi-refine-Refine)**.** *If transition system A is deterministic, then A quasi-refines transition system B implies that A refines B.*

Proof. Let R be the quasi-refinement for A of B.
Define R' to be: $a\ R'\ b$ iff $\exists n.\ (\exists a'.\ a \rightsquigarrow_\epsilon^n a' \wedge a'\ R\ b)$.
We are to show that A refines B by R'. Unfold Definition 3.
For Condition 1, define $|a|_b$ to be the minimal of the number n in the definition of R', which uniquely exists.
For Condition 2, we are to show $a_0\ R'\ b_0$. We know that $a_0\ R\ b_0$, so it's obviously true.
For Condition 3(a), we have $a\ R'\ b$ and $a \rightsquigarrow_o a'$.
We are to exhibit b' and n so that $b \rightsquigarrow_o^n b'$ and $a'\ R'\ b'$ and that $n = 0$ implies $|a|_b > |a'|_b$.
From $a\ R'\ b$, we have $a \rightsquigarrow_\epsilon^m a''$ and $a''\ R\ b$.
If $m = 0$, we know $a\ R\ b$. Because A quasi-refines B by R, we have $a' \rightsquigarrow_\epsilon^* a_2$ and $b \rightsquigarrow_o^n b'$ and $a_2\ R\ b'$ and that $n = 0$ implies $|a|_b > |a'|_b$.
Pick b' to be b' and n to be n. It suffices to show $a'\ R'\ b'$, which is true because $a' \rightsquigarrow_\epsilon^* a_2$ and $a_2\ R\ b'$.
If $m > 0$, pick b' to be b and n to be 0. Because A is deterministic, we know $a'\ R'\ b$ with $m - 1$ and $|a|_b = m$ and $|a'|_b = m - 1$.
For Condition 3(b), we have $a\ R'\ b$ and $a \in F_A$.
We are to exhibit b' such that $b \Downarrow_\epsilon b'$ and $a\ R'\ b'$.
Because A is deterministic and $a \in F_A$, we have $m = 0$ and $a\ R\ b$.
Because A quasi-refines B with R, we have $b \Downarrow_\epsilon b'$ and $a\ R\ b'$.
Pick b' to be b'. It suffices to show $a\ R'\ b'$, which is trivially true.

□

Lemma 3 (Refine-Safety)**.** *If transition system A refines transition system B by R and for any a and b we have $a\ R\ b$ implies $\mathsf{unstuck}(a)$, then A is safe.*

Proof. From Definition 3 we know $(\exists b.\ a\ R\ b)$ is an invariant of A. Hence $\mathsf{unstuck}(a)$ is also an invariant of A. □

Lemma 4 (Deterministic Reverse)**.** *If transition system A is deterministic and transition system B is safe, then B refines A implies that A refines B and A is safe.*

Proof. Appealing to Lemma 3, we will exhibit the refinement for A of B and show that it implies unstuckness.
Let R be the refinement for B of A. Define R' to be:
$a\ R'\ b$ iff

$b_0 \rightsquigarrow^* b \wedge ((\exists o\, b'.\ b \rightsquigarrow_o b' \wedge \exists n_1\ a_2\ a_3\ a_4.\ (a \rightsquigarrow_\epsilon^* a_2 \vee a_2 \rightsquigarrow_\epsilon^* a) \wedge b\ R\ a_2 \wedge a_2 \rightsquigarrow_o^* a_3 \wedge a \rightsquigarrow_o^{n_1} a_3 \wedge a_3 \rightsquigarrow_\epsilon^* a_4 \wedge b'\ R\ a_4) \vee (b \in F_B \wedge \exists n_2\ a_2\ a_3.\ (a \rightsquigarrow_\epsilon^* a_2 \vee a_2 \rightsquigarrow_\epsilon^* a) \wedge b\ R\ a_2 \wedge a_2 \rightsquigarrow_\epsilon^* a_3 \wedge a \rightsquigarrow_\epsilon^{n_2} a_3 \wedge a_3 \in F_A \wedge b\ R\ a_3))$.
Let's first prove the fact (Fact 1) that if $b_0 \rightsquigarrow^* b$ and $a \rightsquigarrow_\epsilon^* a'$ and $b\ R\ a$, then $a\ R'\ b$.
Because B is safe, we know that either $b \rightsquigarrow_o b'$ or $b \in F_B$.
In the first case, because B refines A by R, we have $a' \rightsquigarrow_o^n a''$ and $b\ R\ a''$.
It's easy to show $a\ R'\ b$ by choosing the first disjunct and picking o, b', a_2, a_4 to be o, b', a', a''. n_1 and a_3 exist in this case.
In the second case, because B refines A by R, we have $a' \Downarrow_\epsilon a''$ and $b\ R\ a''$.
It's easy to show $a\ R'\ b$ by choosing the second disjunt and picking a_2, a_3 to be a', a''. n_2 obviously exists.

Now we are to show A refines B by R'. Unfold Definition 3.
For Condition 1, define $|a|_b$ to be lexicographic order of two numbers.
The first number is the minimal of the number n_2 in the definition of R' if b is a value, which uniquely exists; or 0 otherwise.
The second number is the minimal of the number n_1 in the definition of R' if b can take a step, which uniquely exists; or 0 otherwise.
For Condition 2, we are to show $a_0\ R'\ b_0$, which is true because of $b_0\ R\ a_0$ and Fact 1.
For Condition 3(a), we have $a\ R'\ b$ and $a \rightsquigarrow_o a'$.
We are to exhibit b' and n such that $b \rightsquigarrow_o^n b'$ and $a'\ R\ b'$ and that $n = 0$ implies $|a|_b > |a'|_b$.
Unfold $a\ R'\ b$, we have the two disjuncts.
In case $o = l$, only the first disjunt is possible, and we have $b \rightsquigarrow_l b'$ and $a' \rightsquigarrow_\epsilon^* a_4$ and $b'\ R\ a_4$.
Pick b', n to be $b', 1$. From Fact 1, we know $a'\ R'\ b'$.
In case $o = \epsilon$, both disjuncts of $a\ R'\ b$ are possible.
If $a\ R'\ b$ because of the first conjunct, we have $b \rightsquigarrow_{o'} b' \wedge (a \rightsquigarrow_\epsilon^* a_2 \vee a_2 \rightsquigarrow_\epsilon^* a) \wedge b\ R\ a_2 \wedge a_2 \rightsquigarrow_{o'}^* a_3 \wedge a \rightsquigarrow_{o'}^{n_1} a_3 \wedge a_3 \rightsquigarrow_\epsilon^* a_4 \wedge b'\ R\ a_4$.
We case-analyse on whether o' is ϵ.

If $o' = l$, let the second component of $|a|_b$ (denoted by $|a|_b.2$) be m. We case-analyse on whether $m > 1$.
If $m > 1$, pick b', n to be $b, 0$ (i.e. do not move on the B side). We need to show $a'\ R'\ b$ and $|a|_b > |a'|_b$.
$a'\ R'\ b$ because according to $m > 1$ and $a \rightsquigarrow_\epsilon a'$, we know that a' is still before the l-label step.
$|a|_b > |a'|_b$ is true because according to $m > 1$, it must be the case that $|a'|_b.2 = |a|_b.2 - 1$; and as for $|a'|_b.1$, which represents the minimal number of steps to terminate (or 0 otherwise), taking one step will not increase it.
If $m \leq 1$, we know that $a \rightsquigarrow_l a''$ for some a''. But we also have $a \rightsquigarrow_\epsilon a'$, so this case is impossible because of A's determinism.

If $o' = \epsilon$, because B is safe and B refines A by R, we can step on the B side for finite steps to reach b_2 such that $b' \rightsquigarrow_\epsilon^* b_2$ and $b_2\ R\ a$ and either $b_2 \rightsquigarrow_{o''} b_3 \wedge a \rightsquigarrow_{o''}^+ a'' \wedge b_3\ R\ a''$ or $b_2 \in F_B \wedge a \rightsquigarrow_\epsilon^* a'' \wedge a'' \in F_A \wedge b_2\ R\ a''$.
In the first case, because A is deterministic, we have $a \rightsquigarrow_\epsilon a' \rightsquigarrow_{o''}^* a''$.
If $o'' = \epsilon$, pick b' to be b_3. Because of $a' \rightsquigarrow_\epsilon^* a''$ and $b_3\ R\ a''$ and Fact 1, we get $a'\ R'\ b_3$.
If $o'' = l$, pick b' to be b_2. Since $b \rightsquigarrow_\epsilon b' \rightsquigarrow_\epsilon^* b_2$, we just need to show that $a'\ R'\ b_2$, which is easy to show by choosing the first disjunct for R' and picking b', a_2, a_4 to be b_3, a, a''.
In the second case ($b_2 \in F_B$), it must be case that $a \rightsquigarrow_\epsilon a' \rightsquigarrow_\epsilon^* a'' \in F_A$. Pick b' to be b_2, we need to show $a'\ R'\ b_2$, which is true because $a' \rightsquigarrow_\epsilon^* a''$ and $a''\ R'\ b_2$.

If $a\ R'\ b$ because of the second conjunct, we have $b \in F_B \wedge (a \rightsquigarrow_\epsilon^* a_2 \vee a_2 \rightsquigarrow_\epsilon^* a) \wedge b\ R\ a_2 \wedge a_2 \rightsquigarrow_\epsilon^* a_3 \wedge a \rightsquigarrow_\epsilon^{n_2} a_3 \wedge a_3 \in F_A \wedge b\ R\ a_3$.
Because A is deterministic, it must be the case that $a \rightsquigarrow_\epsilon a' \rightsquigarrow_\epsilon^* a_3$.
Pick b', n to be $b, 0$. We need to show $a'\ R'\ b$ and $|a|_b > |a'|_b$.
$a'\ R'\ b$ because $a' \rightsquigarrow_\epsilon^* a_3$ and $a_3\ R'\ b$. $|a|_b > |a'|_b$ because $|a|_b.1 > |a'|_b.1$, which is true because a' is one step closer to terminate.

For Condition 3(b), we have $a\ R'\ b$ and $a \in F_A$.
We are to exhibit b' such that $b \Downarrow_\epsilon b'$ and $a\ R'\ b'$.
If $a\ R'\ b$ because of the second disjunct, we have $b \in F_B \wedge a \rightsquigarrow_\epsilon^* a_3 \wedge b\ R\ a_3$. Because $a \in F_A$ and A is deterministic, we know that $a_3 = a$
Pick b' to be b. $a\ R'\ b$ is true because $b\ R\ a$ and Fact 1.
If $a\ R'\ b$ because of the first disjunct, we have $b \rightsquigarrow_o b_2 \wedge a \rightsquigarrow_o^* a_4 \wedge b_2\ R\ a_4$.
Because $a \in F_A$ and A is deterministic, we know that $a_4 = a$ and $o = \epsilon$.
If $b_2 \in F_B$, pick b' to be b_2. $a\ R'\ b_2$ is true with the same reasoning as before.
Otherwise, because B is safe and B refines A by R, we can step b_2 for finite steps (because a cannot step and $|b|_a > |b_2|_a$) to have $b_2 \rightsquigarrow_\epsilon^* b_3 \wedge b_3\ R\ a$.
Pick b' to be b_3. $a\ R'\ b_3$ is true with the same reasoning as before.

Now we prove that $a\ R'\ b$ implies $\mathsf{unstuck}(a)$ for any a and b.
Unfolding $a\ R'\ b$, in both disjuncts we have $a \rightsquigarrow^n a'$ and $b'\ R\ a'$ for some b'.
If $n > 0$, $\mathsf{unstuck}(a)$ is obviously true.
If $n = 0$, we have $b'\ R\ a$. Because B is safe, we know that either $b' \rightsquigarrow b''$ or $b' \in F_B$.
In case $b' \in F_B$, we know $a \Downarrow_\epsilon a_2$. Because A is deterministic, $\mathsf{unstuck}(a)$ is true.

In case $b' \rightsquigarrow b''$, because B refines A by R, we know $a \rightsquigarrow^k a_2$ and $b'' \; R \; a_2$ and that $k = 0$ implies $|b'|_a > |b''|_a$.
Thus b' can step finite number of $k = 0$ steps before hitting the $b' \in F_B$ case or the $k > 0$ case, in both of which we have $\mathsf{unstuck}(a)$. □

Corollary 1 (Deterministic Reverse)**.** *If transition system A is deterministic and transition system B is safe, then B refines A implies that A bisimulates B and A is safe.*

Definition 8 (Relation R)**.** *For any p and lp, define relation $R_{p,lp}$ as: $(H, le) \; R_{p,lp} \; (S, V, ss)$ iff there exists a minimal n such that $(H, le) \rightsquigarrow^n_{lp} (H, le')$ and $(H, le') =\!\!\updownarrow (S, V, ss)$, where $\updownarrow (S, V, ss) \stackrel{\mathsf{def}}{=} (\mathit{mem}(S), \mathit{unravel}(S, V(\uparrow (\Downarrow_{(p,V)} ss))))$ and $\mathit{mem}(S)$ is all the memory parts of S collected together (and requiring that there is no $\bot$ in S's memory parts).*
$\mathit{unravel}(S, le) \stackrel{\mathsf{def}}{=} \mathit{foldl} \; \mathit{unravel_frame} \; le \; S$
$\mathit{unravel_frame}((M, V, E), le) \stackrel{\mathsf{def}}{=}$
$$\begin{cases} V((\Uparrow E) \, [le]) & \text{if } M = \bot \\ V((\Uparrow E) \, [\mathit{pop} \; le]) & \text{if } M = \lfloor _ \rfloor \end{cases}.$$

Lemma 5 (λow* Refines C*)**.** *For all λow^* program lp, closed expression le and closing substitution V, if $\Downarrow lp = p$ and $\downarrow le = ss$, then $\mathsf{sys}_{\lambda ow^*}(lp, V(le))$ refines $\mathsf{sys}_{C^*}(p, V, ss)$.*

Proof. We apply Lemma 2 and 7, and prove that $\mathsf{sys}_{\lambda ow^*}(lp, le)$ quasi-refines $\mathsf{sys}_{C^*}(p, ss)$. We pick the relation $R_{p,lp}$ in Definition 8 to be the simulation relation and prove $R_{p,lp}$ is a quasi-refinement for $\mathsf{sys}_{\lambda ow^*}(lp, le)$ of $\mathsf{sys}_{C^*}(p, ss)$.
Unfold Definition 7.
For Condition 1, define the well-founded measure $|(H, le)|_{(S,V,ss)}$ (where $(H, le) \; R \; (S, V, ss)$) to be the minimal of the number n in R's definition.
For condition 2, appeal to Lemma 8.
Now prove Condition 3(a). Let (H, le) be the λow* configuration and $C = (S, V, ss)$ be the C* configuration.
We are to exhibit (H'', le'') and C' and n such that $(H', le') \rightsquigarrow^* (H'', le'')$ and $C \rightsquigarrow^n C'$ and $(H'', le'') \; R \; C'$ and that $n = 0$ implies $|(H, le)|_C > |(H', le')|_C$.
For all the cases except *Case Pop*, we pick (H'', le'') to be (H', le') (i.e. do not use the extra flexibility offered by Quasi-Refinement).
Induction on $(H, le) \rightsquigarrow (H', le')$.

Case Let: on case $(H, LE \; [\mathsf{let} \; x : t = lv \; \mathsf{in} \; le]) \rightsquigarrow (H, LE \; [[lv/x]le])$.
We are to exhibit C' such that $(S, V, ss) \rightsquigarrow^+ C'$ and $(H, LE \; [[lv/x]le]) \; R \; C'$.
Apply Lemma 10.
In the first case, we have $le = V(\uparrow ss')$ and $lv =\!\!\wr \; v$ and $LE = \mathsf{unravel}(S, \Box)$, where $v \stackrel{\mathsf{def}}{=} [\![e]\!]_{(p,V)}$.
The C* side runs with nonzero steps to $(S, V[x \mapsto v], ss')$.
Pick C' to be this configuration.
It suffices to show that $(H, LE \; [[lv/x]le]) \; R \; (S, V[x \mapsto v], ss')$.
Appealing to Lemma 9, it suffices to show that $LE \; [[lv/x]le] = \mathsf{unravel}(S, V[x \mapsto v](\uparrow ss'))$, which is true.
In the second case, the C* side runs with nonzero steps to $(S', V'[x \mapsto v], ss')$. The proof is the same as the first case.
End of case.

Case ALet: on case $(H, LE \; [\mathsf{let} _ = lv \; \mathsf{in} \; le]) \rightsquigarrow (H, LE \; [le])$.
Appealing to Lemma 11, the proof is similar to the previous case.
End of case.

Case Newbuf: on case $(H; h, \mathsf{let} \; x = \mathsf{newbuf} \; n \; (lv : t) \; \mathsf{in} \; le) \rightsquigarrow (H; h[b \mapsto lv^n], [(b, 0)/x]le)$ and $b \notin H; h$.
We have $(H; h, \mathsf{let} \; n = (lv : t) \; \mathsf{in} \; le) \; R \; (S, V, ss)$.
We are to exhibit C' so that
$(H; h[b \mapsto lv^n], [(b, 0)/x]le) \; R \; C'$ and $(S, V, ss) \rightsquigarrow^+ C'$.
Appealing to Lemma 12, we have $ss = (t \; x[n]; \mathsf{memset} \; x \; n \; v; ss')$ and $le = V(\uparrow ss')$ and $lv =\!\!\wr \; v$ and $LE = \mathsf{unravel}(S, \Box)$ and $S = S'; (M, V', E)$.
Pick C' to be $(S'; (M[b \mapsto v^n], V', E), V[x \mapsto (b, 0, [])], ss')$.
It suffices to show that $(H; h[b \mapsto lv^n], [(b, 0)/x]le) \; R \; (S'; (M[b \mapsto v^n], V', E), V[x \mapsto (b, 0, [])], ss')$, which is true.
End of case.

Case App: on case $(H, LE \; [\mathsf{let} \; x : t = f \; lv \; \mathsf{in} \; le]) \rightsquigarrow (H, LE \; [\mathsf{let} \; x : t = [lv/y]le_1 \; \mathsf{in} \;]le)$ and $lp(f) = \lambda y : t_1. \, le_1 : t_2$.
Appealing to Lemma 15, we have $ss = (t \; x = f(v); ss')$ and $le = V(\uparrow ss')$ and $lv =\!\!\wr \; v$ and $LE = \mathsf{unravel}(S, \Box)$.
Because $\Downarrow lp = p$, we know $le_1 = \mathsf{withframe} \; le_2$ and $\downarrow le_2 = ss_2; e$ and $p(f) = \mathsf{fun} \; (y : t_1) : t_2 \, \{ \; ss_1 \; \}$ and $ss_1 = (ss_2; \mathsf{return} \; e)$.
Appealing to Lemma 19, we know $\uparrow ss_1 = le_2$ hence $\uparrow \{ss_1\} = le_1$.
Pick C' to be $(S; (\bot, V, t \; x = \Box; ss'), \{y \mapsto v\}, \{ss_1\})$.
It suffices to show that
$(H, LE \; [\mathsf{let} \; x : t = [lv/v]le_1 \; \mathsf{in} \; le]) \; R \; (S; (\bot, V, t \; x = \Box; ss'), \{y \mapsto v\}, \{ss_1\})$, which is true.
End of case.

Case Withframe:
on case $(H, LE \; [\mathsf{withframe} \; le]) \rightsquigarrow (H; \{\}, LE \; [\mathsf{pop} \; le])$.
Appealing to Lemma 16, we have $ss = \{ss_1\}; ss_2$ and $le = V(\uparrow ss_1)$ and $LE = \mathsf{unravel}(S, V(\Uparrow (\Box; ss_2)))$.
Pack C' to be $(S; (\{\}, V, \Box; ss_2), V, ss_1)$.
It suffices to show that
$(H; \{\}, LE \; [\mathsf{pop} \; le]) \; R \; (S; (\{\}, V, \Box; ss_2), V, ss_1)$, which is true.
End of case.

Case Readbuf: on case
$(H, LE \; [\mathsf{let} \; x : t = \mathsf{readbuf} \; (b, n) \; n' \; \mathsf{in} \; le]) \rightsquigarrow_{\mathsf{read} \; (b, n+n')}$

$(H, LE\ [[lv/x]le])$ and $H(b, n+n') = lv$.
Appealing to Lemma 13, we have $ss = (t\ x = (b,n,[])[n']; ss')$ and $le = V(\uparrow ss')$ and $LE = \mathsf{unravel}(S, \square)$.
Pick C' to be $(S, V[x \mapsto v], ss')$ where $v =\wr\ lv$.
We know $C \rightsquigarrow^{+}_{\mathsf{read}\ (b,n+n',[])} C'$.
It suffices to show that $(H, LE\ [[lv/x]le])\ R\ (S, V[x \mapsto v], ss')$, which is true because $[lv/x]le = V[x \mapsto v](\uparrow ss')$.
End of case.

Case Writebuf: on case
$(H, LE\ [\mathsf{let}\ _ = \mathsf{writebuf}\ (b,n)\ n'\ lv\ \mathsf{in}\ le]) \rightsquigarrow_{\mathsf{write}\ (b,n+n')} (H[(b,n+n') \mapsto lv], LE\ [le])$ and $(b, n+n') \in H$.
Appealing to Lemma 14, we have $ss = ((b,n,[])[n'] = v; ss')$ and $le = V(\uparrow ss')$ and $lv =\wr\ v$ and $LE = \mathsf{unravel}(S, \square)$.
Pick C' to be (S', V, ss') where $\mathsf{Set}(S, (b,n,[]), v) = S'$.
We know $C \rightsquigarrow^{+}_{\mathsf{write}\ (b,n+n',[])} C'$.
It suffices to show that $(H[(b,n+n') \mapsto lv], LE\ [le])\ R\ (S', V, ss')$, which is true.
End of case.

Case Subbuf: on case $(H, LE\ [\mathsf{subbuf}\ (b,n)\ n'\ ]) \rightsquigarrow (H, LE\ [(b, n+n')])$.
Pick C' to be (S, V, ss).
Because $(H, LE\ [\mathsf{subbuf}\ (b,n)\ n'\ ])\ R\ C'$ with some m, it must be the case that $m \geq 1$ and $(H, LE\ [(b,n+n')])\ R\ C'$ with $m-1$.
End of case.

Case IfTrue: on case $(H, LE\ [\mathsf{if}\ n\ \mathsf{then}\ le_1\ \mathsf{else}\ le_2]) \rightsquigarrow_{\mathsf{brT}} (H, LE\ [le_1])$ and $n \neq 0$.
Appealing to Lemma 17, we have $ss = \mathsf{if}\ e\ \mathsf{then}\ ss_1\ \mathsf{else}\ ss_2; ss'$ and $[\![e]\!]_{(p,V)} = n$ and $le_i = V(\uparrow ss_i)$ $(i = 1,2)$ and $LE = \mathsf{unravel}(S, V(\Uparrow (\square; ss')))$.
Pick C' to be $(S, V, ss_1; ss')$.
We know $C \rightsquigarrow^{+}_{\mathsf{brT}} C'$.
It suffices to show that $(H, LE\ [le_1])\ R\ (S, V, ss_1; ss')$, which is true.
End of case.

Case IfFalse: on case $(H, LE\ [\mathsf{if}\ n\ \mathsf{then}\ le_1\ \mathsf{else}\ le_2]) \rightsquigarrow_{\mathsf{brF}} (H, LE\ [le_2])$ and $n = 0$.
Similar to previous case.
End of case.

Case Proj: on case $(H, LE\ [\{\overrightarrow{fd = lv}\}.fd']) \rightsquigarrow (H, LE\ [lv'])$ and $\{\overrightarrow{fd = lv}\}(fd') = lv'$.
Pick C' to be (S, V, ss).
Because $(H, LE\ [\{\overrightarrow{fd = lv}\}.fd'])\ R\ C'$ with some m, it must be the case that $m \geq 1$ and $(H, LE\ [lv'])\ R\ C'$ with $m-1$.
End of case.

Case Pop: on case $(H; h, LE\ [\mathsf{pop}\ lv]) \rightsquigarrow (H, LE\ [lv])$.
Apply Lemma 18.

In the first case, from $LE = \mathsf{unravel}(S', V'(\Uparrow \square; ss'))$ we know $LE = (\mathsf{let}\ _ = \square\ \mathsf{in}\ le)$ and $le = \mathsf{unravel}(S', V'(ss'))$.
pick C' to be (S', V', ss') and (H'', le'') to be (H, le).
Obviously $(H, LE\ [lv]) \rightsquigarrow^{*} (H, le)$. It suffices to show that $(H, le)\ R\ (S', V', ss')$, which is true.
In the second case, pick C' to be $(S', V', E\ [v])$ and (H'', le'') to be $(H, LE\ [lv])$.
To suffices to show $(H, LE\ [lv])\ R\ (S', V', E\ [v])$, which follows from $LE = \mathsf{unravel}(S', V'(\Uparrow E))$.

For Condition 3(b), because λow* and C*'s values are almost the same (except that C* locations have a field-path component), every λow* value has an obvious corresponding C* value, so condition 3(b) is trivially true. ☐

Lemma 6 (C* Deterministic)**.** *For all p and ss, transition system $\mathsf{sys}_{C^*}(p, ss)$ is deterministic, modulo renaming of block identifiers.*

Lemma 7 (λow* Deterministic)**.** *For all lp and le, transition system $\mathsf{sys}_{\lambda ow^*}(lp, le)$ is deterministic, modulo renaming of block identifiers.*

Lemma 8 (Init)**.** *For all λow* program lp, closed expression le and closing substitution V, if $\Downarrow lp = p$ and $\downarrow le = ss$, then $([], V(le))\ R_{p,lp}\ ([], V, ss)$.*

Proof. Unfold R's definition, it suffices to show:
$([], V(le)) \rightsquigarrow^{*} ([], V(\uparrow (\Downarrow_{(p,\{\})} \downarrow le)))$. ☐

Lemma 9 (Equal-Normalize)**.** *If $H = mem(S)$ and $le = unravel(S, V(\uparrow ss))$ and $\Downarrow_{(p,V)} ss = ss'$, then $(H, le) \rightsquigarrow^{*} (H, unravel(S, V(\uparrow ss')))$.*

Lemma 10 (Invert Let)**.** *If $(H, LE\ [let\ x : t = lv\ in\ le])\ R\ (S, V, ss)$, then either $ss = (t\ x = e; ss')$ and $le = V(\uparrow ss')$ and $lv =\wr\ v$ and $LE = unravel(S, \square)$, where $v \stackrel{\mathsf{def}}{=} [\![e]\!]_{(p,V)}$ or $S = S'; (\bot, V', t\ x = \square; ss')$ and $ss = return\ v$ and $le = V'(\uparrow ss')$ and $lv =\wr\ v$ and $LE = unravel(S', \square)$.*

Lemma 11 (Invert ALet)**.** *If $(H, LE\ [let\ _ = lv\ in\ le])\ R\ (S, V, ss)$, then either $ss = (e; ss')$ and $le = V(\uparrow ss')$ and $lv =\wr\ v$ and $LE = unravel(S, \square)$, where $v \stackrel{\mathsf{def}}{=} [\![e]\!]_{(p,V)}$ or $S = S'; (\bot, V', \square; ss')$ and $ss = return\ v$ and $le = V'(\uparrow ss')$ and $lv =\wr\ v$ and $LE = unravel(S', \square)$.*

Lemma 12 (Invert Newbuf)**.** *If $(H; h, let\ x = newbuf\ n\ (lv : t)\ in\ le)\ R\ (S, V, ss)$, then $ss = (t\ x[n]\ memset\ x\ n\ v; ss')$ and $le = V(\uparrow ss')$ and $lv =\wr\ v$ and $LE = unravel(S, \square)$ and $S = S'; (M, V', E)$.*

Lemma 13 (Invert Readbuf)**.** *If $(H; h, let\ x : t = readbuf\ (b,n)\ n'\ in\ le)\ R\ (S, V, ss)$, then $ss = (t\ x = (b,n,[])[n']; ss')$ and $le = V(\uparrow ss')$ and $LE = unravel(S, \square)$.*

Lemma 14 (Invert Writebuf)**.** *If $(H; h, let\ _ = writebuf\ (b,n)\ n'\ lv\ in\ le)\ R\ (S, V, ss)$, then $ss = ((b,n,[])[n'] = v; ss')$ and $le = V(\uparrow ss')$ and $lv =\wr\ v$ and $LE = unravel(S, \square)$.*

Lemma 15 (Invert App)**.** *If* $(H, LE\ [\mathit{let}\, x : t = f\ lv\ \mathit{in}\, le])\ R\ (S, V, ss)$*, then* $ss = (t\ x = f(v); ss')$ *and* $le = V(\uparrow ss')$ *and* $lv =\wr\ v$ *and* $LE = \mathit{unravel}(S, \Box)$.

Lemma 16 (Invert Withframe)**.** *If* $(H, LE\ [\mathit{withframe}\, le])\ R\ (S, V, ss)$*, then* $ss = \{ss_1\}; ss_2$ *and* $le = V(\uparrow ss_1)$ *and* $LE = \mathit{unravel}(S, V(\Uparrow (\Box; ss_2)))$.

Lemma 17 (Invert If)**.** *If* $(H, LE\ [\mathit{if}\, n\ \mathit{then}\, le_1\ \mathit{else}\, le_2])\ R\ (S, V, ss)$*, then* $ss = \mathit{if}\ e\ \mathit{then}\ ss_1\ \mathit{else}\ ss_2; ss'$ *and* $[\![e]\!]_{(p,V)} = n$ *and* $le_i = V(\uparrow ss_i)$ $(i = 1, 2)$ *and* $LE = \mathit{unravel}(S, V(\Uparrow (\Box; ss')))$.

Lemma 18 (Invert Pop)**.** *If* $(H; h, LE\ [\mathit{pop}\, lv])\ R\ (S, V, ss)$*, then either* $ss = e$ *and* $[\![e]\!]_{(p,V)} = v$ *and* $\wr\ v = lv$ *and* $S = S'; (M, V', \Box; ss')$ *and* $LE = \mathit{unravel}(S', V'(\Uparrow \Box; ss'))$*, or* $ss = \mathit{return}\ e$ *and* $[\![e]\!]_{(p,V)} = v$ *and* $\wr\ v = lv$ *and* $S = S'; (M, V', E)$ *and* $LE = \mathit{unravel}(S', V'(\Uparrow E))$.

Lemma 19 (Low2C-C2Low)**.** *If* $\downarrow\ le = ss; e$*, then* $\uparrow (ss; \mathit{return}\ e) = le$.

E. From C* to CompCert C and beyond

To further back our claim that KreMLin offers a practical yet trustworthy way to preserve security properties of F* programs down to the executable code, we have to demonstrate that security guarantees can be propagated from C* down to assembly.

Our idea here is to use the CompCert verified C compiler [50, 51]. CompCert formally proves the preservation of functional correctness guarantees from C down to assembly code (for x86, PowerPC and ARM platforms.)

E.1 Reminder: CompCert Clight

CompCert Clight [31] is a subset of C with no side effects in expressions, and actual byte-level representation of values. Syntax in Figure 17. Semantics definitions in Figure 18. Evaluation of expressions in Figure 19. Small-step semantics in Figure 20.

The semantics of a Clight program is given by the return value of its main function called with no arguments.[5] Thus, given a Clight program p, the initial configuration of a CompCert Clight transition from p is $(\{\}, [], [], [], \mathsf{int}\ r = \mathsf{main}())$, and a configuration is final with return value i if, and only if, it is of the form $(\{\}, _, _V, _, [])$ with $_V(r) = i$.

Just like C, there are two ways to evaluate Clight expressions: in lvalue position or in rvalue position. Roughly speaking, in an expression assignment $e_l =_t e_r$, expression e_l is said to be at lvalue position and thus must evaluate into a memory location, whereas e_r is said to be at rvalue position and evaluates into a value (integer or pointer to memory location). The operation $\&e$ takes an lvalue e and transforms it into a rvalue, namely the pointer to the memory location e designates as an lvalue. Conversely, $*e$ takes an rvalue e, which must evaluate to a pointer, and turns it into the corresponding memory location as an lvalue.

Memory accesses in the trace To account for memory accesses in the trace, we make each statement perform at most one memory access in our generated Clight code. Then, we prepend each such memory access statement with a *built-in call*, a no-op annotation $\mathsf{annot}(ev, t, e)$ whose semantics is merely to produce the corresponding memory access event $ev(b, n)$ in the trace, where e evaluates to the pointer to offset n within block e.

The CompCert memory model The semantics of CompCert Clight statements depends on the CompCert memory model [52]. Here we need three operations: Get, Set and Alloc, whose description follows.[6]

$\mathsf{Get}(M, (b, n), n')$ reads n' bytes from memory M at offset n within block b, and decodes the obtained byte fragments into a value. It fails if not all locations from (b, n) to $(b, n' - 1)$ are defined. It returns unkn if all locations are defined but the decoding fails.

$\mathsf{Set}(M, (b, n), n', v)$ writes n' bytes from memory M at offset n within block b corresponding to encoding value v into n' value fragments. It fails if not all locations from (b, n) to $(b, n' - 1)$ are defined.

$\mathsf{Alloc}(M, n)$ returns a pair (b, M') where $b \notin M$ is a fresh block identifier and all locations from $(b, 0)$ to $(b, n - 1)$ in M' contain unkn.

E.2 Issues

Local structures The main difference between C* and Clight is that C* allows structures as values. Although converting a C* value into a Clight value is no problem in terms of memory representation (since the layout of Clight structures[7] is already formalized in CompCert with basic proofs such as the fact that two distinct fields of a structure designate disjoint sets of memory locations), local structures cause issues in terms of managing memory accesses (due to our desire for noninterference in terms of memory accesses), as we describe in Section E.6.

Stack-allocated local variables Another difference between C* and Clight is that, whereas C* allows the user to stack-allocate local variables on the fly, Clight mandates all local variables of a function to be hoisted to the beginning of the function (in fact, the list of all stack-allocated variables

[5] CompCert does not support semantics preservation with system arguments.

[6] In the current version of CompCert and its memory model [53], each memory location is equipped with a *permission* to more faithfully model the fact that the memory locations of a local variable cannot be read or coincidentally reused in a Clight program after exiting the scope of the variable. To this end, thanks to this permission model, the CompCert memory model also defines a free operation which invalidates memory accesses while preventing from reusing the memory block for further allocations; although we do not describe it here, CompCert Clight actually uses this operator to free all local variables upon function exit. Thus, it is also necessary to amend the semantics of C* in a similar way.

[7] which can be chosen when configuring CompCert with a suitable platform

$$\boxed{[\![e]\!]_{(p,V)} = v}$$

$$\frac{V(x) = v}{[\![x]\!]_{(p,V)} = v}\ \textsc{Var} \qquad \frac{[\![e_1]\!]_{(p,V)} = (b, n, \overrightarrow{fd}) \qquad [\![e_2]\!]_{(p,V)} = n'}{[\![e_1 + e_2]\!]_{(p,V)} = (b, n + n', \overrightarrow{fd})}\ \textsc{PtrAdd}$$

$$\frac{[\![e]\!]_{(p,V)} = (b, n, \overrightarrow{fd})}{[\![\&e \to fd]\!]_{(p,V)} = (b, n + n', \overrightarrow{fd}; fd)}\ \textsc{PtrFd} \qquad \frac{x \notin V \qquad p(x) = v}{[\![x]\!]_{(p,V)} = v}\ \textsc{GVar} \qquad \frac{}{[\![\{\overrightarrow{fd = e}\}]\!]_{(p,V)} = \{\overrightarrow{fd = [\![e]\!]_{(p,V)}}\}}\ \textsc{Struct}$$

$$\frac{[\![e]\!]_{(p,V)} = \{\overrightarrow{fd = v}\} \qquad \{\overrightarrow{fd = v}\}(fd') = v'}{[\![e.fd']\!]_{(p,V)} = v'}\ \textsc{Proj}$$

Figure 11. C* Expression Evaluation

$$\boxed{p \vdash C \rightsquigarrow_o C'}$$

$$\frac{[\![e]\!]_{(p,V)} = v}{p \vdash (S, V, t\ x = e; ss) \rightsquigarrow (S, V[x \mapsto v], ss)}\ \textsc{VarDecl} \qquad \frac{S = S'; (M, V, E) \qquad b \notin S}{p \vdash (S, V, t\ x[n]; ss) \rightsquigarrow (S'; (M[b \mapsto \bot^n], V, E), V[x \mapsto (b, 0, [])], ss)}\ \textsc{ArrDecl}$$

$$\frac{[\![e_1]\!]_{(p,V)} = (b, n, []) \qquad [\![e_2]\!]_{(p,V)} = v \qquad \mathsf{Set}(S, (b, n, []), v^m) = S'}{p \vdash (S, V, \mathsf{memset}\ e_1\ m\ e_2; ss) \rightsquigarrow_{\mathsf{write}\ (b,n,[]),\ldots,\mathsf{write}\ (b,n+m-1,[])} (S', V, ss)}\ \textsc{Memset} \qquad \frac{[\![e]\!]_{(p,V)} = (b, n, \overrightarrow{fd}) \qquad \mathsf{Get}(S, (b, n, \overrightarrow{fd})) = v}{p \vdash (S, V, t\ x = *[e]; ss) \rightsquigarrow_{\mathsf{read}\ (b,n,\overrightarrow{fd})} (S, V[x \mapsto v], ss)}\ \textsc{Read}$$

$$\frac{[\![e_1]\!]_{(p,V)} = (b, n, \overrightarrow{fd}) \qquad [\![e_2]\!]_{(p,V)} = v \qquad \mathsf{Set}(S, (b, n, \overrightarrow{fd}), v) = S'}{p \vdash (S, V, *e_1 = e_2; ss) \rightsquigarrow_{\mathsf{write}\ (b,n,\overrightarrow{fd})} (S', V, ss)}\ \textsc{Write} \qquad \frac{[\![e]\!]_{(p,V)} = v}{p \vdash (S; (\bot, V', E), V, \mathsf{return}\ \ e; ss) \rightsquigarrow (S, V', E\ [v])}\ \textsc{Ret}$$

$$\frac{p(f) = \mathsf{fun}\ (y : t_1) : t_2\ \{\ ss_1\ \} \qquad [\![e]\!]_{(p,V)} = v}{p \vdash (S, V, t\ x = f\ e; ss) \rightsquigarrow (S; (\bot, V, t\ x = \Box; ss), V[y \mapsto v], ss_1)}\ \textsc{Call} \qquad \frac{[\![e]\!]_{(p,V)} = v}{p \vdash (S; (M, V', E), V, \mathsf{return}\ \ e; ss) \rightsquigarrow (S, \{\}, \mathsf{return}\ \ v)}\ \textsc{RetBlk}$$

$$\frac{[\![e]\!]_{(p,V)} = v}{p \vdash (S, V, e; ss) \rightsquigarrow (S, V, ss)}\ \textsc{Expr} \qquad \frac{}{p \vdash (S; (M, V', E), V, []) \rightsquigarrow (S, V', E\ [()])}\ \textsc{Empty} \qquad \frac{}{p \vdash (S, V, \{ss_1\}; ss_2) \rightsquigarrow (S; (\{\}, V, \Box; ss_2), V, ss_1)}$$

$$\frac{n \neq 0}{p \vdash (S, V, \mathsf{if}\ n\ \mathsf{then}\ ss_1\ \mathsf{else}\ ss_2; ss) \rightsquigarrow_{\mathsf{brT}} (S, V, ss_1; ss)}\ \textsc{IfT} \qquad \frac{n = 0}{lp \vdash (S, V, \mathsf{if}\ n\ \mathsf{then}\ ss_1\ \mathsf{else}\ ss_2; ss) \rightsquigarrow_{\mathsf{brF}} (S, V, ss_2; ss)}\ \textsc{IfF}$$

Figure 12. C* Configuration Reduction

of a function is actually part of the function definition), and so they are allocated all at once when entering the function.

Hoisting local variables is not supported in the verified part of CompCert.

Consider the following C* example, for a given conditional expression e:

$$\mathsf{if}\ e\ \mathsf{then}\ \mathsf{int}\ x[1]18;\ \mathsf{else}\ \{\mathsf{word}\ x[1]42; *[x+0] = 1729\}$$

After hoisting, following the same strategy as the corresponding unverified pass of CompCert, C* code will look like this:

$$\mathsf{int}\ x_1[1]; \mathsf{word}\ x_2[1];$$

$$\mathsf{if}\ e\ \mathsf{then}\ *[x_1{+}0] = 18;\ \mathsf{else}\ \{*[x_2{+}0] = 42; *[x_2{+}0] = 1729\}$$

This example shows the following issue: when producing the trace, the memory blocks corresponding to the accessed memory location will differ between the non-hoisted and the hoisted C* code. Indeed, in the non-hoisted C* code, only one variable x is allocated in the stack, whereas in the hoisted C* code, two variables x_1 and x_2 corresponding to both branches will be allocated anyway, regardless of the fact that only one branch is executed. Thus, in the C* code before hoisting, allocating x will create, say, block 1, whereas after hoisting, two variables will be allocated, x_1 at block 1, and x_2 at block 2. Thus, the statement $*[x+0] = 1729$ in the C* code before hoisting will produce read $(1, 0, [])$ on the trace, whereas its corresponding translation after hoisting, $*[x_2 + 0] = 1729$, will produce read $(2, 0, [])$.

$\boxed{lp \vdash (H, le) \twoheadrightarrow_{lo} (H', le')}$ and $\boxed{lp \vdash (H, le) \rightsquigarrow_{lo} (H', le')}$

$$\frac{H(b, n+n') = lv}{lp \vdash (H, \mathsf{let}\ x = \mathsf{readbuf}\ (b,n)\ n'\ \mathsf{in}\ le) \twoheadrightarrow_{\mathsf{read}\ (b, n+n')} (H, [lv/x]le)}\ \textsc{Read}$$

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

$$\frac{(b, n+n') \in H}{lp \vdash (H, \mathsf{let}\ _ = \mathsf{writebuf}\ (b,n)\ n'\ lv\ \mathsf{in}\ le) \twoheadrightarrow_{\mathsf{write}\ (b, n+n')} (H[(b, n+n') \mapsto lv], le)}\ \textsc{Write}$$

$$\frac{}{lp \vdash (H, \mathsf{subbuf}\ (b,n)\ n'\ ) \twoheadrightarrow (H, (b, n+n'))}\ \textsc{SubB}$$

$$\frac{}{lp \vdash (H, \mathsf{let}\ x : t = v\ \mathsf{in}\ le) \twoheadrightarrow (H, [v/x]le)}\ \textsc{Let} \qquad \frac{}{lp \vdash (H, \mathsf{let}\ _ = v\ \mathsf{in}\ le) \twoheadrightarrow (H, le)}\ \textsc{ALet}$$

$$\frac{\{\overrightarrow{fd = lv}\}(fd') = lv'}{lp \vdash (H, \{\overrightarrow{fd = lv}\}.fd') \twoheadrightarrow (H, lv')}\ \textsc{Proj}$$

$$\frac{n \neq 0}{lp \vdash (H, \mathsf{if}\ n\ \mathsf{then}\ le_1\ \mathsf{else}\ le_2) \twoheadrightarrow_{\mathsf{brT}} (H, le_1)}\ \textsc{IfT} \qquad \frac{n = 0}{lp \vdash (H, \mathsf{if}\ n\ \mathsf{then}\ le_1\ \mathsf{else}\ le_2) \twoheadrightarrow_{\mathsf{brF}} (H, le_2)}\ \textsc{IfF}$$

$$\frac{b \notin H}{lp \vdash (H, \mathsf{let}\ x = \mathsf{newbuf}\ n\ (lv : t)\ \mathsf{in}\ le) \twoheadrightarrow_{\mathsf{write}\ (b,0), \ldots, \mathsf{write}\ (b, n-1)} (H[b \mapsto lv^n], [(b,0)/x]le)}\ \textsc{NewBuf}$$

$$\frac{}{lp \vdash (H, \mathsf{withframe}\ le) \twoheadrightarrow (H; \{\}, \mathsf{pop}\ le)}\ \textsc{WF} \qquad \frac{}{lp \vdash (H; h, \mathsf{pop}\ lv) \twoheadrightarrow (H, lv)}\ \textsc{Pop}$$

$$\frac{lp \vdash (H, le) \twoheadrightarrow_{lo} (H', le')}{lp \vdash (H, LE\ [le]) \rightsquigarrow_{lo} (H', LE\ [le'])}\ \textsc{Step}$$

Figure 13. λow* Atomic Reduction and Reduction

$\boxed{\wr\ le = e}$ and $\boxed{\downarrow le = ss}$ and $\boxed{\Downarrow ld = d}$

$$\frac{}{\wr\ n = n} \qquad \frac{}{\wr\ (b,n) = (b, n, [])} \qquad \frac{}{\wr\ \{\overrightarrow{fd = le}\} = \{\overrightarrow{fd = \wr\ le}\}} \qquad \frac{}{\wr\ x = x} \qquad \frac{}{\wr\ \mathsf{subbuf}\ le_1\ le_2\ = \wr\ le_1 + \wr\ le_2}$$

$$\frac{\wr\ le = e \qquad \downarrow le_1 = ss}{\downarrow (\mathsf{let}\ x : t = f\ le\ \mathsf{in}\ le_1) = (t\ x = f(e); ss)} \qquad \frac{\wr\ le_i = e_i\ (i = 1,2) \qquad \downarrow le = ss}{\downarrow \mathsf{let}\ x : t = \mathsf{readbuf}\ le_1\ le_2\ \mathsf{in}\ le = (t\ x = e_1[e_2]; ss)}$$

$$\frac{\wr\ le_i = e_i\ (i = 1,2,3) \qquad \downarrow le = ss}{\downarrow (\mathsf{let}\ _ = \mathsf{writebuf}\ le_1\ le_2\ le_3\ \mathsf{in}\ le) = (e_1[e_2}$$

$$\frac{\wr\ le = e \qquad \downarrow le_1 = ss}{\downarrow (\mathsf{let}\ x = \mathsf{newbuf}\ n\ (le : t)\ \mathsf{in}\ le_1) = (t\ x[n]; \mathsf{memset}\ x\ n\ e; ss)} \qquad \frac{}{\downarrow (\mathsf{withframe}\ le) = \{\downarrow le\}}$$

$$\frac{\wr\ le = e \qquad \downarrow le_i = ss_i\ (i = 1,2)}{\downarrow (\mathsf{if}\ le\ \mathsf{then}\ le_1\ \mathsf{else}\ le_2) = (\mathsf{if}\ e\ \mathsf{then}\ ss_1\ \mathsf{else}\ ss_2}$$

$$\frac{\wr\ le = e \qquad \downarrow le_1 = ss}{\downarrow (\mathsf{let}\ x : t = le\ \mathsf{in}\ le_1) = (t\ x = e; ss)} \qquad \frac{\wr\ le = e \qquad \downarrow le_1 = ss}{\downarrow (\mathsf{let}\ _ = le\ \mathsf{in}\ le_1) = (e; ss)} \qquad \frac{}{\downarrow le = \wr\ le}$$

$$\frac{}{\Downarrow (\mathsf{let}\ x : t = lv) = (t\ x = \wr\ lv)} \qquad \frac{\downarrow le = ss; e}{\Downarrow (\mathsf{let}\ f = \lambda x : t_1.\ \mathsf{withframe}\ le : t_2) = \mathsf{fun}\ f\ (x : t_1) : t_2\ \{\ ss; \mathsf{return}\ e\ \}}$$

Figure 14. λow* to C* compilation

One quick solution to ensure that those event traces are exactly preserved, is to replace the actual pointer (b, n, fd) on read and write events with (f, i, x, n, fd) where f is the name of the function, i is the recursion depth of the function (which would be maintained as a global variable, increased whenever entering the function, and decreased whenever exiting) and x is the local variable being accessed. (In concurrent contexts, one could add a parameter θ recording the identifier of the current thread within which f is run, and so the global variable maintaining recursion depth would become an array indexed by thread identifiers.)

Finally, we adopted this solution as we describe in Section E.4, and we heavily use it to prove the correctness of hoisting within C* in Section E.5.

$$\boxed{\Downarrow_{(p,V)}\ ss = ss}$$

$$\frac{\llbracket e \rrbracket_{(p,V)} = v}{\Downarrow_{(p,V)}\ (t\ x = e; ss) = (t\ x = v; ss)} \qquad \frac{\llbracket e \rrbracket_{(p,V)} = v}{\Downarrow_{(p,V)}\ (t\ x = f(e); ss) = (t\ x = f(v); ss)} \qquad \frac{}{\Downarrow_{(p,V)}\ (t\ x[n]; ss) = (t\ x[n]; ss)}$$

$$\frac{\llbracket e \rrbracket_{(p,V)} = v}{\Downarrow_{(p,V)}\ (\mathsf{return}\ e; ss) = (\mathsf{return}\ v; ss)} \qquad \frac{\llbracket e_i \rrbracket_{(p,V)} = v_i\ (i = 1,2)}{\Downarrow_{(p,V)}\ (t\ x = e_1[e_2]; ss) = (t\ x = v_1[v_2]; ss)} \qquad \frac{\llbracket e_i \rrbracket_{(p,V)} = v_i\ (i = 1,2,3)}{\Downarrow_{(p,V)}\ (e_1[e_2] = e_3; ss) = (v_1[v_2] = v_3; ss)}$$

$$\frac{\llbracket e \rrbracket_{(p,V)} = v}{\Downarrow_{(p,V)}\ (e; ss) = (v; ss)} \qquad \frac{\llbracket e_i \rrbracket_{(p,V)} = v_i\ (i = 1,2)}{\Downarrow_{(p,V)}\ (\mathsf{memset}\ e_1\ n\ e_2; ss) = (\mathsf{memset}\ v_1\ n\ v_2; ss)}$$

Figure 15. Normalize C* head expression

$$\boxed{\upsquigarrow e = le} \text{ and } \boxed{\uparrow ss = le} \text{ and } \boxed{\upuparrows d = ld} \text{ and } \boxed{\twoheaduparrow E = LE}$$

$$\frac{}{\upsquigarrow n = n} \qquad \frac{}{\upsquigarrow (b, n, []) = (b, n)} \qquad \frac{}{\upsquigarrow \{\overrightarrow{fd = e}\} = \{\overrightarrow{fd = \upsquigarrow e}\}} \qquad \frac{\upsquigarrow le_i = e_i\ (i = 1,2)}{\upsquigarrow e_1[e_2] = \mathsf{readbuf}\ le_1\ le_2} \qquad \frac{\upsquigarrow le_i = e_i\ (i = 1,2)}{\upsquigarrow (e_1 + e_2) = \mathsf{subbuf}\ le_1\ le_2}$$

$$\frac{\upsquigarrow e = le_1 \qquad \uparrow ss = le}{\uparrow (t\ x = f(e); ss) = (\mathsf{let}\ x : t = f\ le_1\ \mathsf{in}\ le)} \qquad \frac{\upsquigarrow e = le_1 \qquad \uparrow ss = le}{\uparrow (t\ x[n]; \mathsf{memset}\ x\ n\ e; ss) = (\mathsf{let}\ x = \mathsf{newbuf}\ n\ (le_1 : t)\ \mathsf{in}\ le)} \qquad \frac{\upsquigarrow e = le_1 \qquad \uparrow ss = le}{\uparrow (t\ x = e; ss) = (\mathsf{let}\ x : t = le_1\ \mathsf{in}}$$

$$\frac{\upsquigarrow e_i = le_i\ (i = 1,2) \qquad \uparrow ss = le}{\uparrow (t\ x = e_1[e_2]; ss) = (\mathsf{let}\ _ = \mathsf{readbuf}\ le_1\ le_2\ \mathsf{in}\ le)} \qquad \frac{\upsquigarrow e_i = le_i\ (i = 1,2,3) \qquad \uparrow ss = le}{\uparrow (e_1[e_2] = e_3; ss) = (\mathsf{let}\ _ = \mathsf{writebuf}\ le_1\ le_2\ le_3\ \mathsf{in}\ le)}$$

$$\frac{\uparrow ss_1 = le_1 \qquad \uparrow ss = le}{\uparrow (\{ss_1\}; ss) = (\mathsf{let}\ _ = \mathsf{withframe}\ le_1\ \mathsf{in}\ le)} \qquad \frac{\upsquigarrow e = le_1 \qquad \uparrow ss = le}{\uparrow (e; ss) = (\mathsf{let}\ _ = le_1\ \mathsf{in}\ le)}$$

$$\frac{\upsquigarrow e = le \qquad \uparrow ss_i = le_i\ (i = 1,2,3)}{\uparrow (\mathsf{if}\ e\ \mathsf{then}\ ss_1\ \mathsf{else}\ ss_2; ss_3) = (\mathsf{let}\ _ = \mathsf{if}\ le\ \mathsf{then}\ le_1\ \mathsf{else}\ le_2\ \mathsf{in}\ le_3)} \qquad \frac{\upsquigarrow e = le}{\uparrow [e] = le} \qquad \frac{}{\uparrow [] = ()}$$

$$\frac{}{\upuparrows (t\ x = v) = (\mathsf{let}\ x : t = \upsquigarrow v)} \qquad \frac{\uparrow (ss; e) = le}{\upuparrows (\mathsf{fun}\ f\ (x : t_1) : t_2\ \{\ ss; \mathsf{return}\ e\ \}) = (\mathsf{let}\ f = \lambda x : t_1.\ \mathsf{withframe}\ le : t_2)}$$

$$\frac{\uparrow ss = le}{\twoheaduparrow (\square; ss) = (\mathsf{let}\ _ = \square\ \mathsf{in}\ le)} \qquad \frac{\uparrow ss = le}{\twoheaduparrow (t\ x = \square; ss) = (\mathsf{let}\ x : t = \square\ \mathsf{in}\ le)}$$

Figure 16. C* to λow* back-translation

E.3 Summary: from C* to Clight

Given a C* program p and an entrypoint ss, we are going to transform it into a CompCert Clight program in such a way that both functional correctness and noninterference are preserved.

This will not necessarily mean that traces are exactly preserved between C* and Clight, due to the memory representation discrepancy described before. Instead, by functional correctness, we mean that a safe C* program is turned into a safe Clight program, and for such safe programs, termination, I/O events and return value are preserved; and by noninterference, we mean that if two executions with different secrets produce identical (whole) traces in C*, then they will also produce identical traces in CompCert Clight (although the trace may have changed between C* and Clight.)

1. In section E.4.1, we transform a C* programs into a program with unambiguous variable names, and we take advantage of such a syntactic property by enriching the configuration of C* with more information regarding variable names, thus yielding the C* 2 language.
2. In section E.4.2, we execute the obtained C* 2 program with a different, more abstract, trace model, as proposed above. This is not a program transformation, but only a

reinterpretation of the same C* program with a different operational semantics, which we call C* 3.

3. In section E.5, we transform the C* 3 program into a C* 3 program where all local arrays are hoisted from block-scope to function-scope. The abstract trace model critically helps in the success of this proof where memory state representations need to change.
4. In section E.6.1, we transform the obtained C* 3 program into a C* 3 program where functions returning structures are replaced with functions taking a pointer to the return location as additional argument. Thus, we need to account for an additional memory access, which we do through the C* 4 intermediate semantics, another reinterpretation of the source C* 3 program producing new events at functions returning structures. Then, our reinterpreted C* 4 program is translated back to C* 3 with those additional memory accesses made explicit.
5. In section E.6.2, we reinterpret our obtained C* 3 program with a different event model where memory access events of structure type are replaced by the sequences of memory access events of all their non-structure fields. We call this new language C* 5.
6. In section E.6.3, we transform our C* 5 program back into C* 3 by erasing all local structures that are not local arrays, replacing them with their individual non-structure fields. Thus, the more "elementary" memory accesses introduced in the C* 3 to C* 5 reinterpretation are made concrete.
7. We then reinterpret the obtained C* 3 program back into C* 2 as described in E.4.2, reverting to the traces with concrete memory locations at events, thus accounting for all memory accesses.
8. Finally, in section E.7, we compile the obtained C* 2 program, now in the desired form, into CompCert Clight.

E.4 Normalized event traces in C*

As described above, traces where memory locations explicitly appear are notoriously hard to reason about in terms of semantics preservation for verified compilation. Thus, it becomes desirable to find a common representation of traces that can be preserved between different memory layouts across different intermediate languages.

In particular here, we would like to replace concrete pointers into abstract pointers representing the local variable being modified in a given nested function call.

E.4.1 Disambiguation of variable names

To this end, we first need to disambiguate the names of the local variables of a C* function:

Definition 9 (Unambiguous local variables)**.** *We say that a list of C* instructions has* unambiguous local variables *if, and only if, it contains no two distinct array declarations with the same variable name, and does not contain both an array declaration and a non-array declaration with the same variable name.*

We say that a C program has unambiguous local variables if, and only if, for each of its functions, its body has unambiguous local variables.*

We say that a C transition system* $\mathsf{sys}(p, V, ss)$ *has unambiguous local variables if, and only if,* p *has unambiguous local variables,* ss *has unambiguous local variables, and* V *does not define any variable with the same name as an array declared in* ss*.*

Lemma 20 (Disambiguation)**.** *There exists a transformation* T *on lists of instructions (extended to programs by morphism) such that, for any C* program* p*, and for any list of C* instructions* ss*, for any variable mapping* V' *such that* $\mathsf{sys}(T(p), V', T(ss))$ *has unambiguous local variables, there exists a variable mapping* V *such that* $\mathsf{sys}(p, V, ss)$ *and* $\mathsf{sys}(T(p), V', T(ss))$ *have the same execution traces.*

Proof. α-renaming. □

The noninterference property can be proven to be stable by such α-renaming:

Lemma 21. *Let* p *be a C* program and* ss *be a list of instructions. Assume that for any* V_1, V_2 *such that* $\mathsf{sys}(p, V_1, ss)$ *and* $\mathsf{sys}(p, V_2, ss)$ *are both safe, then they have the same execution traces.*

Then, for any V_1', V_2' *such that* $\mathsf{sys}(T(p), V_1', T(ss))$ *and* $\mathsf{sys}(T(p), V_2', T(ss))$ *are both safe, they have the same execution traces.*

Proof. By Lemma 20, there is some V_1 such that $\mathsf{sys}(p, V_1, ss)$ and $\mathsf{sys}(T(p), V_1', T(ss))$ have the same execution traces, thus in particular, $\mathsf{sys}(p, V_1, ss)$ is safe. Same for some V_2. By hypothesis, $\mathsf{sys}(p, V_1, ss)$ and $\mathsf{sys}(p, V_2, ss)$ have the same execution traces, thus the result follows by transitivity of equality. □

Thus, we can now restrict our study to C* programs whose functions have no two distinct array declarations with the same variable names.

Let us first enrich the configuration (S, V, ss) of C* small-step semantics with additional information recording the current function f being executed (or maybe $\bot$) and the set A of the variable names of local arrays currently declared in the scope. Thus, a C* stack frame $(\bot, V, E)$ becomes $(\bot, V, E, f, A)$ where f is the caller, a block frame (M, V, E) becomes (M, V, E, f, A) where f is the enclosing function of the block, and the configuration (S, V, ss) becomes (S, V, ss, f, A) where f is the current function (with the frames of S changed accordingly.) Let us then change some rules accordingly as described in Fig. 21, leaving other rules unchanged except with the corresponding f and A components preserved.

Let us call C* 2 the obtained language where the initial state of the transition system $\mathsf{sys}(p, V, ss)$ shall now be $([], V, ss, \bot, [])$.

Then, it is easy to prove the following:

Lemma 22 (C* to C* 2)**.** *If $\mathsf{sys}(p, V, ss)$ has unambiguous local variables and is safe in C*, then it has the same execution traces in C* as in C* 2 (and in particular, it is also safe in C* 2.)*

Thus, both functional correctness and noninterference are preserved from C to C* 2.*

Proof. Lock-step bisimulation where the common parts of the configurations (besides the C* 2-specific f, A parts) are equal between C* and C* 2. □

Then, we can prove an invariant over the small-step execution of a C* 2 program:

Lemma 23 (C* 2 invariant)**.** *Let p be a C* 2 program and V a variable environment such that $\mathsf{sys}(p, V, ss)$ has unambiguous local variables.*

Let $n \in \mathbb{N}$. Then, for any C 2 configuration (S, V', ss', f, A) obtained after n C* 2 steps from $(\{\}, V, ss, \bot, \{\})$, the following invariants hold:*

1. *for any variable or array declaration x in ss', it does not appear in A_1*
2. *any variable name in A or in an array declaration of ss' is in an array declaration of ss (if $f = \bot$) or the body of f (otherwise.)*
3. *for each frame of S of the form $(_, _, E, f'', A'')$, then any variable name or array declaration in E does not appear in A'', and any variable name in A'' or in an array declaration of E is in an array declaration in ss (if $f'' = \bot$) or the body of f'' (otherwise.)*
4. *if $S = S'; (M, _, _, f', A')$ with $M \neq \bot$, then $f' = f$, $A' \subseteq A$ and for all block identifiers b defined in M, there exists a unique variable $x \in A$ such that $V'(x) = b$*
5. *for any two consecutive frames $(M, _, _, f_1, A_1)$ just below $(_, V_2, _, f_2, A_2)$ with $M \neq \bot$, then $f_1 = f_2$ and $A_1 \subseteq A_2$ and $V_1(x) = V_2(x)$ for all variables $x \in A_1$, and for all block identifiers b defined in M, there exists a unique variable $x \in A_2$ such that $V_2(x) = b$*
6. *for any two different frames of S of the form $(M_1, _, _, _, _)$ and $(M_2, _, _, _, _)$ with $M_1 \neq \bot$ and $M_2 \neq \bot$, the sets of block identifiers of M_1 and M_2 are disjoint*

Proof. By induction on n and case analysis on $\rightsquigarrow$. □

Then, we can prove a strong invariant between two executions of the same C* 2 program with different secrets. This strong invariant will serve as a preparation towards changing the event traces of C* 2.

Lemma 24 (C* 2 noninterference invariant)**.** *Let p be a C* 2 program, and V_1, V_2 be two variable environments such that $\mathsf{sys}(p, V_1, ss)$ and $\mathsf{sys}(p, V_2, ss)$ have unambiguous local variables, are both safe and produce the same traces in C* 2.*

Let $n \in \mathbb{N}$. Then, for any two C 2 configurations $(S_1, V_1', ss_1, f_1, A_1)$ and $(S_2, V_2', ss_2, f_2, A_2)$ obtained after n C* 2 execution steps, the following invariants hold:*

- *$ss_1 = ss_2$*
- *S_1, S_2 have the same length*
- *$f_1 = f_2$*
- *$A_1 = A_2$*
- *$V_1(x) = V_2(x)$ for each $x \in A_1$*
- *for each i, if the i-th frames of S_1, S_2 are $(M_1, V_1'', E_1, f_1'', A_1'')$ and $(M_2, V_2'', E_2, f_2'', A_2'')$, then $E_1 = E_2$, $f_1'' = f_2''$ and $A_1'' = A_2''$ and $V_1''(x) = V_2''(x)$ for any $x \in A_1''$. Moreover, $M_1 = \bot$ if and only if $M_2 = \bot$, and if $M_1 \neq \bot$, then M_1 and M_2 have the same block domain.*

Thus, the $n+1$-th step in both executions applies the same C 2 rule.*

Proof. By induction on n and case analysis on $\rightsquigarrow$, also using the invariant of Lemma 23. In particular the equality of codes is a consequence of the fact that there are no function pointers in C*.[8] Then, both executions apply the same C* 2 rules since C* 2 small-step rules are actually syntax-directed. □

E.4.2 Normalized traces

Now, consider an execution of C* 2 from some initial state. In fact, for any block identifier b defined in S, it is easy to prove that it actually corresponds to some variable defined in the scope. The corresponding VarOfBlock algorithm is shown in Figure 22.

Then, let C* 3 be the C* 2 language where the Read and Write rules are changed according to Figure 23, with event traces where the actual pointer is replaced into an abstract pointer obtained using the VarOfBlock algorithm above.

Lemma 25 (C* 2 to C* 3 functional correctness)**.** *If $\mathsf{sys}(p, V, ss)$ has no unambiguous variables, then $\mathsf{sys}(p, V, ss)$, has the same behaviors in C* 2 with event traces with $\mathsf{read}, \mathsf{write}$ removed, as in C* 3 with event traces with $\mathsf{read}, \mathsf{write}$ removed.*

Proof. Lock-step bisimulation with equal configurations. Steps Read and Write need the invariant of Lemma 23 on C* 2 to prove that C* 3 does not get stuck (ability to apply VarOfBlock.) □

Lemma 26 (VarOfBlock inversion)**.** *Let C_1, C_2 two C* 2 configurations such that invariants of Lemma 23 and Lemma 24 hold. Then, for any block identifiers b_1, b_2 such*

[8] If we were to allow function pointers in C*, then we would have to add function call/return events into the C* trace beforehand, and assume that traces with those events are equal before renaming = prove that they are equal on the Low* program as well. This might have consequences in the proof of function inlining in the F*-to-C* translation.

that $\text{VarOfBlock}(C_1, b_1)$ *and* $\text{VarOfBlock}(C_2, b_2)$ *are both defined and equal, then* $b_1 = b_2$*.*

Proof. Assume $\text{VarOfBlock}(C_1, b_1) = \text{VarOfBlock}(C_2, b_2) = (f, n, x)$. When applying VarOfBlock, consider the frames F_1, F_2 holding the memory states defining b_1, b_2. Consider the variable mapping V_2' in the frame just above F_2 (or in C_2 if such frame is missing.) Then, it is such that $V_2'(x) = b_2$.

- If F_1 and F_2 are at the same level in their respective stacks, then the variable mapping V_1' in the frame directly above F_1 (or in C_1 if such frame is missing) is such that $V_1'(x) = b_1$, and also $V_1'(x) = V_2'(x)$ by the invariant, so $b_1 = b_2$.
- Otherwise, without loss of generality (by symmetry), assume that F_2 is strictly above F_1 (i.e. F_2 is strictly closer to the top of its own stack than F_1 is in its own stack.) Thus, in the stack of C_1, all frames in between F_1 and the frame F_1'' corresponding to F_2 are of the form $(M', V', _, f', _)$ with $f' = f$ and $M' \neq \bot$ (otherwise the functions and/or recursion depths would be different.) By invariant 5 of Lemma 23, it is easy to prove that $V'(x) = b_1$, and thus also for the variable mapping V_1' in the frame just above F_1'' (or in C_1 directly if there is no such frame.) By invariants of Lemma 24, we have $V_1'(x) = V_2'(x)$, thus $b_1 = b_2$.

□

Lemma 27 (C* 2 to C* 3 noninterference)**.** *Assume that* $\mathsf{sys}(p, V_1, ss)$ *and* $\mathsf{sys}(p, V_2, ss)$ *have no unambiguous variables. Then, they are both safe in C* 2 and produce the same traces in C* 2, if and only if they are both safe in C* 3 and produce the same traces in C* 3.*

Proof. Use the invariants of Lemma 23 and Lemma 24. In fact, the configurations and steps are the same in C* 2 as in C* 3, only the traces differ between C* 2 and C* 3. Read and Write steps match between C* 2 and C* 3 thanks to Lemma 26. □

Lemma 28 (C* 3 invariants)**.** *The invariants of Lemma 23 and Lemma 24 also hold in C* 3.*

E.5 Local variable hoisting

On C* 3, hoisting can be performed, which will modify the structure of the memory (namely the number of memory blocks allocated), which is fine thanks to the fact that event traces carry abstract pointer representations instead of concrete pointer values.

Memory allocator and dangling pointers However, we have to cope with dangling pointers whose address should not be reused. Consider the following C* code:

```
int x[1]18;
int * p[1]x;
{
 int y[1]42;
  * [p] = y;
}
{
 int z[1]1729;
 int * q = *[p];
 f(q)
}
```

With a careless memory allocator which would reuse the space of y for z, the above program would call f not with a dangling pointer to y, but instead with a valid pointer to z, which might not be expected by the programmer. Then, if f uses its argument to access memory, what should the VarOfBlock algorithm compute? I claim that such a C* 3 program generated from a safe F* program should never try to access memory through dangling pointers.

As far as I understood, a Low* program obtained from a well-typed F* program should be safe *with any memory allocator*, including with a memory allocator which never reuses previously allocated block identifiers, as in CompCert.[9] In particular, a Low* program safe with such a CompCert-style allocator will actually never try to access memory through a dangling pointer to a local variable no longer in scope.

Then, traces with concrete pointer values are preserved from Low* to C* 2 *with the allocator fixed* in advance in all of Low*, C* and C* 2; and functional correctness and noninterference are also propagated down to C* 3 using the same memory allocator.

There should be a way to prove the following:

Lemma 29. *If a C* 3 program is safe with a CompCert-style memory allocator, then it is safe with any memory allocator and the traces (with abstract pointer representations) are preserved by change of memory allocator.*

Proof. Lock-step simulation where the configurations have the same structure but a (functional but not necessarily injective) renaming of block identifiers from a CompCert-style allocator to any allocator is maintained and augmented throughout the execution. In particular, we have to prove that VarOfBlock is stable under such renaming. □

[9] Formally, a Low* (or C*) configuration should be augmented with a state Σ so that the Low* NewBuf rule (or the C* ArrDecl rule), instead of picking a block identifier b not in the domain of the memory, call an allocator alloc with two parameters, the domain D of the memory and the state Σ, and returning the fresh block $b \notin D$ and a new state Σ' for future allocations. Then, a CompCert-style allocator would, for instance, use $\mathbb{N}$ as the type of block identifiers, as well as for the type of Σ, so that if $\mathsf{alloc}(D, \Sigma) = (b, \Sigma')$, then it is ensured that $b \notin D$, $\Sigma \leq b$ and $b < \Sigma'$. In that case, the domain of the memory being always within Σ, could then be easily proven as an invariant of Low* (or C*).

If so, then for the remainder of this paper, we can consider a CompCert-style allocator.

Hoisting

Definition 10 (Hoisting)**.** *For any list of statements ss with unambiguous local variables, the* hoisting *operation $\mathsf{hoist}(ss) = (ads, ss')$ is so that ads is the list of all array declarations in ss (regardless of their enclosing code blocks) and ss' is the list of statements ss with all array declarations replaced with $()$.*

Then, hoisting the local variables in the body ss of a function is defined as replacing ss with the code block $\{ads; ss'\}$ where $\mathsf{hoist}(ss) = (ads, ss')$; and then, hoisting the local variables in a program p, $\mathsf{hoist}(p)$, is defined as hoisting the local variables in each of its functions.

Definition 11 (Renaming of block identifiers)**.** *Let C_1, C_2 be two C* 3 configurations. Block identifier b_1 is said to* correspond *to block identifier b_2 from C_1 to C_2 if, and only if, either* $\textsc{VarOfBlock}(C_1, b_1)$ *is undefined, or* $\textsc{VarOfBlock}(C_1, b_1)$ *is defined and equal to* $\textsc{VarOfBlock}(C_2, b_2)$.

Then, value v_1 corresponds to v_2 from C_1 to C_2 if, and only if, either they are equal integers, or they are pointers (b_1, n_1, fd_1), (b_2, n_2, fd_2) such that $fd_1 = fd_2$, $n_1 = n_2$ and b_1 corresponds to b_2 from C_1 to C_2, or they are structures with the same field identifiers and, for each field f, the value of the field f in v_1 corresponds to the value of the field f in v_2 from C to C'.

Theorem 4 (Correctness of hoisting)**.** *If $\mathsf{sys}(p, V, ss)$ is safe in C* 3 with a CompCert-style allocator, then $\mathsf{sys}(p, V, ss)$ and $\mathsf{sys}(\mathsf{hoist}(p), V, \mathsf{hoist}(ss))$ have the same execution traces (and in particular, the latter is also safe) in C* 3 using the same CompCert-style allocator.*

Proof. Forward downward simulation from C* 3 before to C* 3 after hoisting, where one step before corresponds to one step after, except at function entry where at least two steps are required in the compiled program (function entry, followed by entering the enclosing block that was added at function translation, then allocating all local variables if any), and at function exit, where two steps are required in the compiled program (exiting the added block before exiting the function.)

Then, since C* 3 is deterministic, the forward downward simulation is flipped into an upward simulation in the flavor of CompCert; thus preservation of traces.

For the simulation diagram, we combine the invariants of Lemma 23 with the following invariant between configurations $C = (S, V, ss, f, A)$ before hoisting and $C' = (S', V', ss', f', A')$ after hoisting:

- for all variables x defined in V, $V(x)$, if defined, corresponds to $V'(x)$ from C to C'
- ss' is obtained from ss by replacing all array declarations with $()$
- $f' = f$
- $A \subseteq A'$
- the set of variables declared in ss is included in A'
- if a block identifier b corresponds to b' from C to C', then the value $\mathsf{Get}(S, b, n, fds)$, if defined, corresponds to $\mathsf{Get}(S', b', n, fd)$ from C to C'

Each frame of the form $(\bot, V_1, E, f_1, A_1)$ in S is replaced with two frames in S', namely $(\bot, V_1', E, f_1, A_1'); (M', V_2', \Box, f_2, A_2')$ where:

- for all variables x defined in V_1, $V_1(x)$ corresponds to $V_1'(x)$ from C to C'
- all array declarations of E are present in A_1'
- E' is obtained from E by replacing all array declarations with $()$
- $f_1' = f_1$
- $A_1 \subseteq A_1'$
- A_2' contains all variable names of arrays declared in f_2, and is the block domain of M'
- V_2' is defined for all variable names in A_2' as a block identifier valid in M'
- all memory locations of arrays declared in f_2 are valid in M'

Each frame of the form (M, V_1, E, f_1, A_1) in S with $M \neq \bot$ is replaced with one frame in S', namely $(\{\}, V_1', E', f_1', A_1')$ where:

- all blocks of M are defined in A_1'
- for all variables x defined in V_1, $V_1(x)$, if defined, corresponds to $V_1'(x)$ from C to C'
- all array declarations of E are present in A_1'
- E' is obtained from E by replacing all array declarations with $()$
- $f_1' = f_1$
- $A_1 \subseteq A_1'$

The fact that we are using a CompCert-style memory allocator is crucial here to ensure that, once a source block identifier b starts corresponding to a target one, it remains so forever, in particular after its block has been freed (i.e. after its corresponding variable has fallen out of scope), since in the latter case, it corresponds to any block identifier and nothing has to be proven then (since accessing memory through it will fail in the source, per the fact that the CompCert-style memory allocator will never reuse b.) □

E.6 Local structures

C* has structures as values, unlike CompCert C and Clight, which both need all structures to be allocated in memory. With a naive C*-to-C compilation phase, where C* structures are compiled as C structures and passed by value to functions, we experienced more than 60% slowdown with CompCert compared to GCC -O1, using the λow* benchmark in Figure 24, extracted to C as Figure 25. This is because, unlike GCC, CompCert cannot detect that a structure is never taken

its address, which is mostly the case for local structures in code generated from C*. This is due to the fact that, even at the level of the semantics of C structures in CompCert, a field access is tantamount to reading in memory through a constant offset. In other words, CompCert has no view of C structures other than as memory regions. To solve this issue, we replace local structures with their individual non-compound fields, dubbed as *structure erasure*. Our benchmark after structure erasure is shown in Figure 26.

In our noninterference proofs where we prove that memory accesses are the same between two runs with different secrets, treating all local structures as memory accesses would become a problem, especially whenever a field of a local structure is read as an expression (in addition to the performance decrease using CompCert.) This is another reason why, in this paper (although a departure from our current KreMLin implementation), we propose an easier proof based on the fact that C* local structures should not be considered as memory regions in the generated C code.

In addition to buffers (stack-allocated arrays), C* uses local structures in three ways: as local expressions, passed as an argument to a function by value, and returned by a function. Here we claim that it is always possible to not take them as memory accesses, except for structures returned by a function: in the latter case, it is necessary for the caller to allocate some space on its own stack and pass a pointer to it to the callee, which will use this pointer to store its result; then, the caller will read the result back from this memory area. Thus, we claim that, at the level of CompCert Clight, the only additional memory accesses due to local structures are structures returned by value.

So we extend C* 3 with the ability for functions to have several arguments, all of which shall be passed at each call site (there shall be no partial applications.)

E.6.1 Structure return

To handle structure return, we also have to account for their memory accesses by adding corresponding events in the trace. Instead of directly adding the memory accesses and trying to prove both program transformation and trace transformation at the same time, we will first add new read and write events at function return, without those events corresponding to actual memory accesses yet; then, in a second pass, we will actually introduce the corresponding new stack-allocated variables.

We assume given a function FunResVar such that for any list of statements ss and any variable x, $\mathsf{FunResVar}(ss, x)$ is a local variable that does not appear in ss and is distinct from $\mathsf{FunResVar}(ss, x')$ for any $x' \neq x$.

Let p be a program p and ss be an entrypoint list of statements, so we define $\mathsf{FunResVar}(f, x)$ = FunResVar(ss', x) if $f(_)\{ss'\}$ is a function defined in p, and $\mathsf{FunResVar}(\bot, x)$ = $\mathsf{FunResVar}(ss, x)$.

Then, we define C* 4 as the language C* 3 where the Ret_2 function return rule is replaced with two rules following Figure 27, adding the fake read and write. We do not produce any such memory access event if the result is discarded by the caller; thus, we also need to check in the callee whether the caller actually needs the result. To prepare for the second pass where this check will be done by testing whether the return value pointer argument is null, we need to account for this test in the event trace in C* 4 as well.

Theorem 5 (C* 3 to C* 4 functional correctness)**.** *If $\mathsf{sys}(p, V, ss)$ is safe in C* 3 and has unambiguous local variables, then it has the same behavior and trace as in C* 4 with brT, brF, read and write events removed.*

Proof. With all such events removed, C* 3 and C* 4 are actually the same language. □

Lemma 30 (C* 4 invariants)**.** *The C* 3 invariants of Lemma 23 and 24 also hold on C* 4.*

Proof. This is true because the invariants of Lemma 23 actually do not depend on the traces produced; and it is obvious to prove that, if two executions have the same traces in C* 4, then they have the same traces in C* 3 (because in C* 3, some events are just removed.) □

Theorem 6 (C* 3 to C* 4 noninterference)**.** *If $\mathsf{sys}(p, V_1, ss)$ and $\mathsf{sys}(p, V_2, ss)$ are safe in C* 3, have unambiguous local variables, and produce the same traces in C* 3, then they also produce the same traces in C* 4.*

Proof. Two such executions actually make the same C* 4 steps. □

Then, we define the StructRet structure return transformation from C* 4 to C* 3 in Figure 28, thus removing all structure returns from C* 3 programs.

Then, the transformation back to C* 3 exactly preserves the traces of C* 4 programs, so that we obtain both functional correctness and noninterference at once:

Theorem 7 (StructRet correctness)**.** *If $\mathsf{sys}(p, V, ss)$ is safe in C* 4 and has unambiguous local variables, then it has the same behavior and trace as $\mathsf{sys}(\mathsf{StructRet}(p), V, \mathsf{StructRet}(ss, \bot))$.*

Proof. Forward downward simulation, where the compilation invariant also involves block identifier renaming from Definition 11 due to the new local arrays introduced by the transformed program.

Each C* 4 step is actually matched by the same C* 3 step, except for function return and return from block: for the RETBLOCK rule, the simulation diagram has to stutter as many times as the level of block nesting in the source program before the actual application of a RET_4 rule. Then, when the $\mathrm{RETSOME}_4$ rule applies, the trace events are produced by the transformed program, the brT and the write events from within the callee, then the callee blocks are exited, and finally the read event is produced from within the caller.

Then, the diagram is turned into bisimulation since C* 3 is deterministic. □

After a further hoisting pass, we can now restrict our study to those C* 3 programs with unambiguous local variables, functions with multiple arguments, function-scoped local arrays, and no functions returning structures.

E.6.2 Events for accessing structure buffers

Now, we transform an access to one structure into the sequence of accesses to all of its individual atomic (non-structure) fields.

Consider the following transformation for C* 3 read (and similarly for write) events:

$$
\begin{array}{ll}
\multicolumn{2}{l}{[\![\mathsf{read}\ (f,i,x,j,\overrightarrow{fd},t)]\!]} \\
= & [\![\mathsf{read}\ (f,i,x,j,\overrightarrow{fd};fd_1,t_1)]\!] \\
; & \ldots \\
; & [\![\mathsf{read}\ (f,i,x,j,\overrightarrow{fd};fd_n,t_n)]\!] \\
\text{if} & t = \mathsf{struct}\{fd_1 : t_1, \ldots, fd_n : t_n\}
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{[\![\mathsf{read}\ (f,i,x,j,\overrightarrow{fd},t)]\!]} \\
= & \mathsf{read}\ (f,i,x,j,\overrightarrow{fd},t) \\
\multicolumn{2}{l}{\text{otherwise}}
\end{array}
$$

Then, let C* 5 be the C* 3 language obtained by replacing each read, write event with its translation. Then, it is easy to show the following:

Lemma 31 (C* 3 to C* 5 correctness)**.** *Let p be a C* 3 program. Then, p has a trace t in C* 5 if, and only if, there exists a trace t' such that p has trace t' in C* 3 and $[\![t']\!] = t$.*

Thus, this trace transformation preserves functional correctness; and, although this trace transformation is not necessarily injective (since it is not possible to disambiguate between an access to a 1-field structure and an access to its unique field), noninterference is also preserved.

After such transformation, all read and write events now are restricted to atomic (non-structure) types.

E.6.3 Local structures

Now, we are removing all local structures, in such a way that the only remaining structures are those of local arrays, and all structures are accessed only through their atomic fields. In particular, we are replacing every local (non-array) variable x of type struct with the sequence of variable names $x_\overrightarrow{fds}$ for all field name sequences $\overrightarrow{fds}$ valid from x such that $x.\overrightarrow{fds}$ is of non-struct type. (We omit the details as to how to construct names of the form $x_\overrightarrow{fds}$ so that they do not clash with other variables; at worst, we could also rename other variables to avoid clashes as needed.)

Using our benchmark in Figure 24, with C code after structure erasure in Figure 26, on a 4-core Intel Core i7 1.7 GHz laptop with 8 Gb RAM, structure erasure saves 20% time with CompCert 2.7.

If we assume that we know about the type of a C* expression, then it can be first statically reduced to a normal form as in Figure 29.

Lemma 32 (C* structure erasure in expressions: correctness)**.** *For any value v of type t, if $[\![e]\!]_{(p,V)} = v$ and $\Gamma \vdash e \downarrow^t e'$ and V' is such that:*

- *for any $(x' : t') \in \Gamma$, $V(x')$ exists, is of type t' and is equal to $V(x)$*
- *for any $(x' : t') \in \Gamma$ that is a struct and for any $\overrightarrow{fds}$ such that $x.\overrightarrow{fds}$ is not a struct, then $V'(x_\overrightarrow{fds}) = V(x)(\overrightarrow{fds})$*

Then, $[\![e']\!]_{(p,V')} = v$

Proof. By structural induction on $\downarrow$. □

Definition 12 (C* expression without structures)**.** *A C* expression e is said to be of type t without structures if, and only if, one of the following is true:*

- *e contains neither a structure field projection nor a structure expression*
- *e is of the form $x.\overrightarrow{fds}$ where x is a variable such that $x.\overrightarrow{fds}$ is of struct type*
- *t is of the form $\{\overrightarrow{fd_i : t_i}\}$ and e is of the form $\{\overrightarrow{fd_i = e_i}\}$ where for each i, e_i is of type t_i without structures*

Lemma 33 (C* expression reduction: shape)**.** *If $\Gamma \vdash e \downarrow^t e'$, then e' is of type t without structures.*

Proof. By structural induction on $\downarrow$. □

Then, once structure expressions are reduced within an expression computing a non-structure value, we can show that evaluating such a reduced expression no longer depends on any local structures:

Lemma 34. *If $[\![e]\!]_{(p,V)} = v$ for some value v, and e is of type t without structures, and t is not a struct type, then, for any variable mapping V' such that $V'(x) = V(x)$ for all variables x of non-struct types, $[\![e]\!]_{(p,V')} = v$.*

Proof. By structural induction on $[\![e]\!]_{(p,V)}$. □

Now, we take advantage of this transformation to transform C* 5 statements into C* 3 statements without structure assignments. This $\downarrow$ translation is detailed in Figure 30

In particular, each function parameter of structure type passed by value is replaced with its recursive list of all non-structure fields.[10]

[10] Our solution, although semantics-preserving as we show further down, yet causes ABI compliance issues. Indeed, in the System V x86 ABI, structures passed by value must be replaced not with their fields, but with their sequence of bytes, some of which may correspond to padding related to no field of the original structure. CompCert does support this feature but as an **unverified** elaboration pass over source C code. So, we should investigate whether we really need to expose functions taking structures passed by value at the interface level.

Theorem 8 (C* 5 to C* 3 structure erasure: shape)**.** *If $p \downarrow p'$ following Figure 30, then p' no longer has any variables of local structure type, and no longer has any structure or field projection expressions.*

Proof. By structural induction over $\downarrow$, also using Lemma 33. □

Theorem 9 (C* 5 to C* 3 structure erasure: correctness)**.** *If p is a C* 5 program (that is, syntactically, a C* program with unambiguous local variables, no block-scoped local arrays other than function-scoped, and no functions returning structures) such that p is safe in C* 5 and $p \downarrow p'$ following Figure 30, then p and p' have the same execution traces.*

Proof. Forward downward simulation where one C* 5 step triggers one or several C* 3 steps. Then, determinism of C* 3 turns this forward downward simulation into bisimulation.

The compilation invariant is as follows: the code fragments are translated using $\downarrow$, and variable maps V in source C* 5 vs. their compiled C* 3 counterparts V' follow the conditions of Lemma 32, also using Lemma 34. Memory states M are exactly preserved, as well as the structure of the stack. □

Then, after a further α-renaming pass, we obtain a C* 3 program that no longer has any local (non-stack-allocated) structures at all, and where all memory accesses are of non-structure type. The shape of this program is now suitable for translation to a CompCert Clight program in a straightforward way, which we describe in the next subsection.

E.7 Generation of CompCert Clight code

Recall that going from C* 3 (with abstract pointer events) back to C* 2 (with concrete pointer events) is possible thanks to the fact that Lemma 25 and Lemma 27 are actually equivalences.

Recall that a C* n transition system is of the form $\mathsf{sys}(p, V, ss)$ where p is a list of functions[11], ss is a list of C* statements with undeclared local variables, the values of which shall be taken from the map V. ss is actually taken as the entrypoint of the program, and V is deemed to store the initial values of secrets, ensuring that p and ss are syntactically secret-independent.

In CompCert Clight, it is not nominally possible to start with a set of undeclared variables and a map to define them. So, when translating the C* entrypoint into Clight, we have to introduce a *secret-independent* way of representing V and how they are read in the entrypoint. Fortunately, CompCert introduces the notion of *built-in functions*, which are special constructs whose semantics can be customized and that are guaranteed to be preserved by compilation down to the assembly.

Thus, we can populate the values of local non-stack-allocated variables of a C* entrypoint by uniformly calling builtins in Clight, and only the semantics of those builtins will depend on secrets, so that the actually generated Clight code will be syntactically secret-independent.

Translating C* 2 expressions with no structures or structure field projections into CompCert Clight is straightforward, as shown in Figure 31. For any C* 2 expression e of type t, assuming that A is a set of local variables to be considered as local arrays, we define $\mathbb{C}^t_A(e)$ to be the compiled Clight expression corresponding to e.

Lemma 35. *Let V be a C* 2 local variable map, and A be a set of local variables to be considered as local arrays. Assume that, for all $x \in A$, there exists a block identifier b such that $V(x) = (b, 0)$, and define $V'(x)$ be such block identifier b. Then, define $_V'(_x) = V(x)$ for all $x \notin A$.*

Then, for any expression e with no structures or structure projections, $\mathit{rv}(\mathbb{C}^t_A(e), (p, V', _V')) = [\![e]\!]_{(p,V)}$.

Proof. By structural induction on e. □

Translating C* 2 statements with no local array declarations, no read or write of structure type and no functions returning structures into Clight is straightforward as well, as shown in Figure 32.

Let p be a C* 2 program, and ss be a C* 2 entrypoint sequence of statements. Assume that p and ss have unambiguous local variables, no functions returning structures, no local arrays other than function-scoped arrays, and no local structures other than local arrays. Further assume that p has no function called main, and no function with the same name as a built-in function. Then, we can define the compiled CompCert Clight program $\mathbb{C}(p, ss)$ as in Figure 33.

Theorem 10 (C* 2 to Clight: correctness)**.** *If $\mathsf{sys}(p, V, ss)$ is safe in C* 2, then it has the same execution trace as $\mathbb{C}(p, ss)$ in Clight, when the semantics of the built-in functions* get_x *are given by V.*

Proof. Forward downward simulation where one C* 2 step corresponds to one or several Clight steps. Then, since Clight is deterministic, the forward downward simulation diagram is turned into a bisimulation.

The structure of the Clight stack is the same as in the C* 2 stack, and the values of variables are the same, as well as the memory block identifiers. The only change is in the Clight representation of C* 2 structure values . A C* 2 memory state M is said to correspond to a Clight memory state M' if, for any block identifier b, for any array index i, and for any field sequence fds leading to a non-structure value, $M'(b, n + \mathsf{offsetof}(fds)) = M(b, n)(fds)$. □

Thus, since both C* 2 and Clight records all memory accesses in their traces, this theorem entails both functional correctness and preservation of noninterference.

[11] and global variables, although the semantics of C* says nothing about how to actually initially allocate them in memory

p ::=		program
	$\overrightarrow{d}$	series of declarations
d ::=		declaration
	fun $f\,(x:t) : t\,\{\,\overrightarrow{ad}, ss\,\}$	top-level function with stack-allocated local variables $\overrightarrow{ad}$
	ad	top-level value
ad ::=		array declaration
	$t\;x[n]$	uninitialized global variable
ss ::=		statement lists
	$\overrightarrow{s}$	
s ::=		statements
	$_x = e$	assign rvalue to a non-stack-allocated local variable
	$_x = f(e)$	application
	$_x =_t [e]$	memory read from lvalue
	$e =_t e$	memory write rvalue to lvalue
	annot(read, t, e)	annotation to produce read event
	annot(write, t, e)	annotation to produce write event
	if e then ss else ss	conditional
	$\{ss\}$	block
	return e	return
e ::=		expressions
	n	integer constant (rvalue)
	x	stack-allocated variable (lvalue)
	$_x$	non-stack-allocated variable (rvalue)
	$e_1 +_t e_2$	pointer add (rvalue, e_1 is a rvalue pointer to a value of type t and e_2 is a rvalue int)
	$e._t fd$	struct field projection (lvalue, e lvalue)
	&e	address of a lvalue (rvalue, e lvalue)
	*e	pointer dereference (lvalue, e rvalue)

Figure 17. Clight Syntax

v ::=		values
	n	integer constant
	(b, n)	memory location
	unkn	defined but unknown value
vf ::=		value fragments
	n	byte constant
	$((b, n), n')$	n'-th byte of pointer value (b, n)
	unkn	defined but unknown value fragment
V ::=		stack-allocated variable assignments
	$x \rightharpoonup b$	map from variable to memory block identifier
$_V$ ::=		non-stack-allocated variable assignments
	$_x \rightharpoonup v$	map from variable to value
E ::=		evaluation ctx (plug expr to get stmts)
	$\square; ss$	discard returned value
	$_x = \square; ss$	receive returned value
F ::=		frames
	$(V, _V, E)$	stack frame
S ::=		stack
	$\overrightarrow{F}$	list of frames
M ::=		memory
	$(b, n) \rightharpoonup vf$	map from block id and offset to value fragment
C ::=		configuration
	$(S, V, _V, M, ss)$	

Figure 18. Clight Semantics Definitions

$\boxed{\mathsf{lv}(e,(p,V,_V)) = (b,n)}$ and $\boxed{\mathsf{rv}(e,(p,V,_V)) = v}$

$$\frac{V(x) = b}{\mathsf{lv}(x,(p,V,_V)) = (b,0)}\ \textsc{Var} \qquad \frac{x \notin V \qquad p(x) = b}{\mathsf{lv}(x,(p,V,_V)) = (b,0)}\ \textsc{GVar}$$

$$\frac{\mathsf{lv}(e,(p,V,_V)) = (b,n)}{\mathsf{lv}(e._t fd,(p,V,_V)) = (b, n + \mathsf{offsetof}(t, fd))}\ \textsc{PtrFd} \qquad \frac{\mathsf{rv}(e,(p,V,_V)) = (b,n)}{\mathsf{lv}(*e,(p,V,_V)) = (b,n)}\ \textsc{PtrDeref}$$

$$\frac{_V(_x) = v}{\mathsf{rv}(_x,(p,V,_V)) = v}\ \textsc{RVar} \quad \frac{\mathsf{rv}(e_1,(p,V,_V)) = (b,n) \qquad \mathsf{rv}(e_2,(p,V,_V)) = n'}{\mathsf{rv}(e_1 + e_2,(p,V)) = (b, n+n')}\ \textsc{PtrAdd} \quad \frac{\mathsf{lv}(e,(p,V,_V)) = (b,n)}{\mathsf{rv}(\&e,(p,V,_V)) = (b,n)}\ \textsc{AddrOf}$$

Figure 19. Clight Expression Evaluation

$\boxed{p \vdash C \rightsquigarrow_o C'}$

$$\frac{\mathsf{lv}(e,(p,V,_V)) = (b,n) \qquad \mathsf{Get}(M,(b,n),\mathsf{sizeof}(t)) = v}{p \vdash (S,V,_V,M,_x =_t [e]; ss) \rightsquigarrow (S,V,_V[_x \mapsto v],M,ss)}\ \textsc{Read}$$

$$\frac{\mathsf{lv}(e_1,(p,V,_V)) = (b,n) \qquad \mathsf{rv}(e_2,(p,V,_V)) = v \qquad \mathsf{Set}(M,(b,n),\mathsf{sizeof}(t),v) = S'}{p \vdash (S,V,_V,M,e_1 =_t e_2; ss) \rightsquigarrow (S',V,_V,M,ss)}\ \textsc{Write}$$

$$\frac{\mathsf{lv}(e,(p,V,_V)) = (b,n)}{p \vdash (S,V,_V,M,\mathsf{annot} ev, e; ss) \rightsquigarrow_{ev\ (b,n)} (S,V,_V,M,SS)}\ \textsc{Annot}$$

$$\frac{\mathsf{rv}(e,(p,V,_V)) = v}{p \vdash (S;(V',_V',E),V,M,\mathsf{return}\ \ e; ss) \rightsquigarrow (S,V',_V',M,E\ [v])}\ \textsc{Ret}$$

$$\frac{p(f) = \mathsf{fun}\ (_y : t_1) : t_2\ \{\ ads; ss_1\ \} \qquad \mathsf{rv}(e,(p,V,_V)) = v \qquad \mathsf{valloc}(ads,\bot,M) = (V',M')}{p \vdash (S,V,_V,M,_x = f\ e; ss) \rightsquigarrow (S;(V,_V,_x = \Box; ss),V',_V[_y \mapsto v],M',ss_1)}\ \textsc{Call}$$

$$\frac{}{\mathsf{valloc}([],V,M) = (V,M)}\ \textsc{AllocNil} \quad \frac{\mathsf{Alloc}(M, n \times \mathsf{sizeof}(t)) = (b,M_1) \qquad \mathsf{valloc}(ads, V[x \mapsto (b,0)], M_1) = (V',M')}{\mathsf{valloc}((t\ x[n]; ads),V,M) = (V',M')}\ \textsc{AllocCons}$$

$$\frac{\mathsf{rv}(e,(p,V,_V)) = v}{p \vdash (S,V,_V,M,e; ss) \rightsquigarrow (S,V,_V,M,ss)}\ \textsc{Expr} \qquad \frac{}{p \vdash (S,V,_V,M,[]) \rightsquigarrow (S,V,_V,M,\mathsf{return}\ \ \mathsf{unkn})}\ \textsc{Empty}$$

$$\frac{}{p \vdash (S,V,_V,M,\{ss_1\}; ss_2) \rightsquigarrow (S;V,_V,M,ss_1;ss_2)}\ \textsc{Block}$$

$$\frac{\mathsf{rv}(e,(p,V,_V)) = v \qquad v \neq 0 \qquad v \neq \mathsf{unkn}}{p \vdash (S,V,_V,M,\mathsf{if}\ e\ \mathsf{then}\ ss_1\ \mathsf{else}\ ss_2; ss) \rightsquigarrow_{\mathsf{brT}} (S,V,ss_1;ss)}\ \textsc{IfT} \qquad \frac{\mathsf{rv}(e,(p,V,_V)) = 0}{p \vdash (S,V,_V,M,\mathsf{if}\ e\ \mathsf{then}\ ss_1\ \mathsf{else}\ ss_2; ss) \rightsquigarrow_{\mathsf{brF}} (S,V,_V,M,ss_2;ss)}\ \textsc{IfF}$$

Figure 20. Clight Configuration Reduction

$$\frac{[\![e]\!]_{(p,V)} = v \qquad S = S'; (M, V, E, f', A') \qquad b \notin S}{p \vdash (S, V, t\ x[n] = e; ss, f, A) \rightsquigarrow (S'; (M[b \mapsto v^n], V, E, f', A'), V[x \mapsto (b, 0, [])], ss, f, A \cup \{x\})} \text{ ARRDECL}_2$$

$$\frac{[\![e]\!]_{(p,V)} = v}{p \vdash (S; (\bot, V', E, f', A'), V, \mathsf{return}\ \ e; ss, f, A) \rightsquigarrow (S, V', E\ [v], f', A')} \text{ RET}_2$$

$$\frac{p(f) = \mathsf{fun}\ (y : t_1) : t_2\ \{\ ss_1\ \} \qquad [\![e]\!]_{(p,V)} = v}{p \vdash (S, V, t\ x = f\ e; ss, f', A') \rightsquigarrow (S; (\bot, V, t\ x = \square; ss, f', A'), V[y \mapsto v], ss_1, f, \{\})} \text{ CALL}_2$$

Figure 21. C* 2 Amended Configuration Reduction

Algorithm: VarOfBlock
Inputs:

- C* 2 configuration $(S, V, _, _, A)$ such that the invariants of Lemma 23 hold
- Memory block b defined in S

Output: function, recursion depth and local variable corresponding to the memory block
Let $S = S_1; (M, _, _, f, _); S_2$ such that b defined in M. (Such a decomposition exists and is unique because of Invariant 6. f may be $\perp$.)
Let n be the number of frames in S_1 of the form $(\perp, _, _, f', _)$ with $f' = f$.
Let V' and A' such that $S_2 = (_, V', _, _, A); _$, or $V' = V$ and $A' = A$ if $S_2 = \{\}$.
Let x such that $V'(x) = (b, 0)$ (exists and is unique because of Invariants 4 and 5.)
Result: (f, n, x)

Figure 22. C* 2: retrieving the local variable corresponding to a memory block

$$\frac{C = (S, V, t\ x = *[e]; ss, f, A) \qquad [\![e]\!]_{(p,V)} = (b, n, \overrightarrow{fd}) \qquad \mathsf{Get}(S, (b, n, \overrightarrow{fd})) = v \qquad \ell = \textsc{VarOfBlock}(C, b)}{p \vdash C \leadsto_{\mathsf{read}\ (\ell, n, \overrightarrow{fd})} (S, V[x \mapsto v], ss, f, A)} \textsc{Read}_3$$

$$\frac{C = (S, V, *e_1 = e_2; ss, f, A) \qquad [\![e_1]\!]_{(p,V)} = (b, n, \overrightarrow{fd}) \qquad [\![e_2]\!]_{(p,V)} = v \qquad \mathsf{Set}(S, (b, n, \overrightarrow{fd}), v) = S' \qquad \ell = \textsc{VarOfBlock}(C, b)}{p \vdash C \leadsto_{\mathsf{write}\ (\ell, n, \overrightarrow{fd})} (S', V, ss, f, A)} \textsc{Write}_3$$

Figure 23. C* 3 Amended Configuration Reduction

```
1  module StructErase
2  open FStar.Int32
3  open FStar.ST
4
5  type u = { left: Int32.t; right: Int32.t }
6
7  let rec f (r: u) (n: Int32.t): Stack unit (λ _→ true) (λ ___→ true) =
8    push_frame();
9    (
10     if lt n 1l
11     then ()
12     else
13       let r' : u = { left = sub r.right 1l ; right = add r.left 1l } in
14       f r' (sub n 1l)
15   );
16   pop_frame()
17
18 let test () =
19   let r : u = { left = 18l ; right = 42l } in
20   let z2 = mul 2l 2l in
21   let z4 = mul z2 z2 in
22   let z8 = mul z4 z4 in
23   let z16 = mul z8 z8 in
24   let z24 = mul z8 z16 in
25   let z = mul z24 2l in
26   f r z (* without structure erasure, CompCert segfaults
27                 if replaced with 2*z *)
```

Figure 24. Low* benchmarking for structure erasure

```
1  typedef struct {
2    int32_t left;
3    int32_t right;
4  } StructErase_u;
5
6  void StructErase_f(StructErase_u r, int32_t n) {
7    if (n < (int32_t )1) { } else {
8      StructErase_u r_ = {
9        .left = r.right − (int32_t )1,
10       .right = r.left + (int32_t )1
11     };
12     StructErase_f(r_, n − (int32_t )1);
13   }
14 }
15
16 void StructErase_test() {
17   StructErase_u r = {
18     .left = (int32_t )18,
19     .right = (int32_t )42
20   };
21   int32_t z2 = (int32_t )4;
22   int32_t z4 = z2 ∗ z2;
23   int32_t z8 = z4 ∗ z4;
24   int32_t z16 = z8 ∗ z8;
25   int32_t z24 = z8 ∗ z16;
26   int32_t z = z24 ∗ z2;
27   StructErase_f(r, z);
28   return;
29 }
```

Figure 25. Extracted C code, before structure erasure

```
 1 void StructErase_f(int32_t r_left, int32_t r_right, int32_t n) {
 2   if (n < (int32_t )1) { } else {
 3     int32_t r__left = r_right − (int32_t )1
 4     int32_t r__right = r_left + (int32_t )1;
 5     StructErase_f(r__left, r__right, n − (int32_t )1);
 6   }
 7 }
 8
 9 void StructErase_test() {
10   int32_t r_left = (int32_t )18;
11   int32_t r_right = (int32_t )42;
12   int32_t z2 = (int32_t )4;
13   int32_t z4 = z2 ∗ z2;
14   int32_t z8 = z4 ∗ z4;
15   int32_t z16 = z8 ∗ z8;
16   int32_t z24 = z8 ∗ z16;
17   int32_t z = z24 ∗ z2;
18   StructErase_f(r_left, r_right, z);
19   return;
20 }
```

Figure 26. Extracted C code, after structure erasure

$$\frac{[\![e]\!]_{(p,V)} = v \qquad \mathsf{FunResVar}(f', x) = x'}{p \vdash (S; (\bot, V', t\ x = \Box; ss', f', A'), V, \mathsf{return}\ \ e; ss, f, A) \rightsquigarrow_{\mathsf{brT};\mathsf{write}\ (x',0,[]);\mathsf{read}\ (x',0,[])} (S, V', t\ x = v; ss', f', A')}\ \textsc{Ret}_4\textsc{Some}$$

$$\frac{[\![e]\!]_{(p,V)} = v}{p \vdash (S; (\bot, V', \Box; ss', f', A'), V, \mathsf{return}\ \ e; ss, f, A) \rightsquigarrow_{\mathsf{brF}} (S, V', ss', f', A')}\ \textsc{Ret}_4\textsc{None}$$

Figure 27. C* 4 Amended Configuration Reduction

$$\mathsf{StructRet}(_, \mathsf{return}\ e, x) = \begin{cases} \mathsf{if}\ x\ \mathsf{then}\ *[x] = e\ \mathsf{else}\ (); \mathsf{return}\ () & \text{if } x \neq \bot \\ \mathsf{return}\ e & \text{otherwise} \end{cases}$$

$$\mathsf{StructRet}(f', t\ x = f(e), _) = \begin{cases} t\ x'[1]; f(x', e); t\ x = *[x'] & \text{if } t \text{ is a struct} \\ & \text{and } x' = \mathsf{FunRetVar}(f', x) \\ t\ x = f(e) & \text{otherwise} \end{cases}$$

$$\mathsf{StructRet}(_, f(e), _) = \begin{cases} f(0, e) & \text{if the return type of } f \text{ is a struct} \\ f(e) & \text{otherwise} \end{cases}$$

$$\mathsf{StructRet}(\mathsf{fun}\ f\ (x : t) : t'\ \{\ ss\ \}) = \begin{cases} \mathsf{fun}\ f\ (r : t'*, x : t) : \mathsf{unit}\ \{\ \{\mathsf{StructRet}(f, ss, r)\}\} & \text{if } t' \text{ is a struct} \\ & (r \text{ fresh}) \\ \mathsf{fun}\ f\ (x : t) : t'\ \{\ \{\mathsf{StructRet}(f, ss, \bot)\}\ \} & \text{otherwise} \end{cases}$$

Figure 28. C* 4 to C* 3 structure return transformation

$$\frac{}{\Gamma \vdash n \downarrow^{\mathsf{Int}} n}\ \textsc{Int} \qquad \frac{(x : t) \in \Gamma}{\Gamma \vdash x \downarrow^{t} x}\ \textsc{Var} \qquad \frac{\Gamma \vdash e_1 \downarrow^{t*} e_1' \qquad \Gamma \vdash e_2 \downarrow^{\mathsf{int}} e_2'}{\Gamma \vdash e_1 + e_2 \downarrow^{t*} e_1' + e_2'}\ \textsc{PtrAdd} \qquad \frac{\Gamma \vdash e \downarrow^{\mathsf{struct}\{fd:t;\dots\}*} e'}{\Gamma \vdash \&e \to fd \downarrow^{t*} \&e' \to fd}\ \textsc{PtrFd}$$

$$\frac{\Gamma \vdash e \downarrow^{\mathsf{struct}\{fd:t;\dots\}} x.\overrightarrow{fds} \qquad t \text{ is a struct}}{\Gamma \vdash e.fd \downarrow^{t} x.\overrightarrow{fds}.fd}\ \textsc{StructFieldName} \qquad \frac{\Gamma \vdash e \downarrow^{\mathsf{struct}\{fd:t;\dots\}} x.\overrightarrow{fds} \qquad t \text{ is not a struct}}{\Gamma \vdash e.fd \downarrow^{t} x_\overrightarrow{fds}_fd}\ \textsc{ScalarFieldName}$$

$$\frac{\Gamma \vdash e \downarrow^{\mathsf{struct}\{fd:t;\dots\}} \{f = e'; \dots\}}{\Gamma \vdash e.fd \downarrow^{t} e'}\ \textsc{FieldProj} \qquad \frac{\overrightarrow{\Gamma \vdash e_i \downarrow^{t_i} e_i'}}{\Gamma \vdash \{\overrightarrow{fd_i = e_i}\} \downarrow^{\mathsf{struct}\{\overrightarrow{fd_i:t_i}\}} \{\overrightarrow{fd_i = e_i'}\}}\ \textsc{Struct}$$

Figure 29. C* Structure Erasure: Expressions

$$\frac{t \text{ not a struct} \qquad \Gamma \vdash e \downarrow^{t*} e'}{p, \Gamma \vdash t\ x = *[e] \downarrow t\ x = *[e'] \qquad \Gamma[x] \leftarrow t} \text{ READSCALAR} \qquad \frac{t \text{ not a struct} \qquad \Gamma \vdash e_1 \downarrow^{t*} e_1' \qquad \Gamma \vdash e_2 \downarrow^{t} e_2'}{p, \Gamma \vdash *[e_1] = e_2 \downarrow *[e_1'] = e_2'} \text{ WRITESCALAR}$$

$$\frac{t = \text{struct}\{\overrightarrow{fd_i : t_i}\} \qquad \Gamma \vdash e \downarrow^{t*} e' \qquad (x, _) \notin \Gamma \qquad (x', _) \notin \Gamma \qquad \overrightarrow{p, \Gamma \cup (x' : t*) \cup (x_fd_i : t_i) \vdash x_fd_i = *[\&x' \to fd_i] \downarrow ss_i}}{p, \Gamma \vdash t\ x = *[e] \downarrow t*\ \ x' = e'; \overrightarrow{ss_i} \qquad \Gamma[x] \leftarrow t} \text{ READSTRUCT}$$

$$\frac{t = \text{struct}\{\overrightarrow{fd_i : t_i}\} \qquad \Gamma \vdash e_1 \downarrow^{t*} e_1' \qquad \Gamma \vdash e_2 \downarrow^{t} e_2' \qquad (x', _) \notin \Gamma \qquad \overrightarrow{p, \Gamma \cup (x' : t*) \vdash *[\&x' \to fd_i] = e_2'.fd_i \downarrow ss_i}}{p, \Gamma \vdash *[e_1] = e_2 \downarrow t*\ \ x' = e_1'; \overrightarrow{ss_i}} \text{ WRITESTRUCT}$$

$$\frac{t \text{ not a struct} \qquad \Gamma \vdash e \downarrow^{t} e'}{p, \Gamma \vdash \text{return } e \downarrow \text{return } e'} \text{ RET} \qquad \frac{p, \Gamma \vdash ss \downarrow ss'}{p, \Gamma \vdash \{ss\} \downarrow \{ss'\}} \text{ BLOCK} \qquad \frac{p(f) = \text{fun}(\overrightarrow{_ : t_i}) : t\{_\} \qquad t \text{ not a struct} \qquad p, \Gamma \vdash (\overrightarrow{t_i}, el) \downarrow el'}{p, \Gamma \vdash t\ x = f\ (el) \downarrow t\ x = f(el') \qquad \Gamma[x] \leftarrow t} \text{ CALL}$$

$$\frac{}{p, \Gamma \vdash ([], []) \downarrow []} \text{ ARGNIL} \qquad \frac{p, \Gamma \vdash (t, e) \downarrow el_1 \qquad p, \Gamma \vdash (tl, el) \downarrow el_2}{p, \Gamma \vdash (t; tl, e; el) \downarrow el_1; el_2} \text{ ARGCONS}$$

$$\frac{t \text{ not a struct} \qquad \Gamma \vdash e \downarrow^{t} e'}{p, \Gamma \vdash (t, e) \downarrow [e']} \text{ SCALARARG} \qquad \frac{t = \text{struct}\{\overrightarrow{fd_i : t_i}\} \qquad \Gamma \vdash e \downarrow^{t} e' \qquad \overrightarrow{p, \Gamma \vdash (t_i, e'.fd_i) \downarrow el_i}}{p, \Gamma \vdash (t, e) \downarrow \overrightarrow{el_i}} \text{ STRUCTARG}$$

$$\frac{t \text{ not a struct} \qquad \Gamma \vdash e \downarrow^{t} e' \qquad \overrightarrow{p, \Gamma \vdash ss_i \downarrow ss_i'}}{p, \Gamma \vdash \text{if } e \text{ then } ss_1 \text{ else } ss_2 \downarrow \text{if } e' \text{ then } ss_1' \text{ else } ss_2'} \text{ IF}$$

$$\frac{t \text{ not a struct}}{(x : t) \downarrow (x : t)} \text{ SCALARPARAM} \qquad \frac{t = \text{struct}\{\overrightarrow{fd_i : t_i}\} \qquad \overrightarrow{(x_fd_i : t_i) \downarrow vt_i}}{(x : t) \downarrow \overrightarrow{vt_i}} \text{ STRUCTPARAM}$$

$$\frac{vt \downarrow vt' \qquad p, (vt \cup \overrightarrow{x_i : t_i*}) \vdash ss \downarrow ss'}{\text{fun}(vt) : t\{\overrightarrow{t_i\ x_i[_] = \{\}}; ss\} \downarrow \text{fun}(vt') : t\{\overrightarrow{t_i\ x_i[_] = \{\}}; ss'\}} \text{ FUN}$$

Figure 30. C* 5 to C* 3 Structure Erasure: Statements

$$
\begin{array}{lcl}
\mathbb{C}^{\mathsf{int}}_A(n) & = & n \\
\mathbb{C}^{\mathsf{unit}}_A(()) & = & 0 \\
\mathbb{C}^{t}_A(x) & = & \&x \\
 & \text{if} & x \in A \\
\mathbb{C}^{t}_A(x) & = & _x \\
 & \text{if} & x \notin A \\
\mathbb{C}^{t*}_A(e_1 + e_2) & = & \mathbb{C}^{t*}_A(e_1) +_t \mathbb{C}^{\mathsf{int}}_A(e_2) \\
\mathbb{C}^{t*}_A(\&e \to fd) & = & \&(*\mathbb{C}^{t'*}_A(e)._{t'}fd) \\
 & \text{if} & t' = \mathsf{struct}\{fd : t, \dots\}
\end{array}
$$

Figure 31. C* 2 to Clight: Expressions

$$
\begin{array}{ll}
\mathbb{C}_A(t\ x = e) & = t\ x = C^t_A(e) \\
\mathbb{C}_A(t\ x = f(\overrightarrow{e_i})) & = t\ x = f(\overrightarrow{C^{t_i}_A(e_i)}) \\
 & \quad \text{if } f \text{ is } \mathsf{fun}(\overrightarrow{(_ : t_i)}) : _\{_\} \\
\mathbb{C}_A(t\ x = *[e]) & = \mathsf{annot}(\mathsf{read}, t, e); t\ x = [*\mathbb{C}^{t*}_A(e)] \\
\mathbb{C}_A(*[e_1] = e_2) & = \mathsf{annot}(\mathsf{write}, t, e); *\mathbb{C}^{t*}_A(e_1) = \mathbb{C}^{t}_A(e_2) \\
\mathbb{C}_A(\mathsf{if}\ e\ \mathsf{then}\ ss_1\ \mathsf{else}\ ss_2) & = \mathsf{if}\ \mathbb{C}^t_A(e)\ \mathsf{then}\ \mathbb{C}_A(ss_1)\ \mathsf{else}\ \mathbb{C}_A(ss_2) \\
 & \quad \text{for some } t \text{ not struct} \\
\mathbb{C}_A(\{ss\}) & = \{\mathbb{C}_A(ss)\} \\
\mathbb{C}_A(\mathsf{return}\ e) & = \mathsf{return}\ \mathbb{C}^t_A(e)
\end{array}
$$

Figure 32. C* 2 to Clight: Statements

$$
\begin{array}{lcl}
\mathbb{C}(p, ss)(f) & = & \mathsf{fun}\ (\overrightarrow{x : t}) : t'\{\overrightarrow{t_i\ x_i[n_i]}; \mathbb{C}_{\overrightarrow{x_i}}(ss')\} \\
 & \text{if} & p(f) = \mathsf{fun}\ (\overrightarrow{x : t}) : t'\{\overrightarrow{t_i\ x_i[n_i]}; ss'\} \\
\mathbb{C}(p, ss)(\mathsf{main}) & = & \mathsf{fun}() : \mathsf{int}\{ \\
 & & \overrightarrow{t_i\ x_i[n_i]}; \\
 & & \overrightarrow{_y_i = \mathsf{get}_y_i()}; \\
 & & \mathbb{C}_{\overrightarrow{x_i}}(ss') \\
 & & \} \\
 & \text{if} & ss = \{\overrightarrow{t_i\ x_i[n_i]}; ss'\} \\
 & & \text{with free variables } \overrightarrow{y_i}
\end{array}
$$

Figure 33. C* 2 to Clight: Program and Entrypoint