Verified Low-Level Programming Embedded in F*

KARTHIKEYAN BHARGAVAN, INRIA Paris
ANTOINE DELIGNAT-LAVAUD, Microsoft Research
CÉDRIC FOURNET, Microsoft Research
CĂTĂLIN HRIŢCU, INRIA Paris
JONATHAN PROTZENKO, Microsoft Research
TAHINA RAMANANANDRO, Microsoft Research
ASEEM RASTOGI, Microsoft Research
NIKHIL SWAMY, Microsoft Research
PENG WANG, MIT CSAIL
SANTIAGO ZANELLA-BÉGUELIN, Microsoft Research
JEAN-KARIM ZINZINDOHOUÉ, Inria Paris

We present Low*, a language for low-level programming and verification, and its application to high-assurance optimized 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* does not involve any garbage collection or implicit heap allocation; instead, it has a structured memory model à la CompCert, and it provides the control required for writing efficient low-level security-critical code.

By virtue of typing, any Low* program is memory safe. In addition, 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 provide a new compiler back-end from Low* to C and, to evaluate our approach, we implement and verify various cryptographic algorithms, constructions, and tools for a total of about 28,000 lines of code, specification and proof. We show that our Low* code delivers performance competitive with existing (unverified) C cryptographic libraries, suggesting our approach may be applicable to larger-scale low-level software.

ACM Reference format:
Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Cătălin Hriţcu, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. 2017. Verified Low-Level Programming Embedded in F*. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 87 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In the pursuit of high performance, cryptographic software widely deployed throughout the internet is still often subject to dangerous attacks (Int 2017; Dou 2017; Use 2017; Afek and Sharabani 2007; AlFardan and Paterson 2013; Bhargavan et al. 2014; Bhargavan and Leurent 2016; Böck 2016; Böck et al. 2016; Dobrovitski 2003; Duong and Rizzo 2011; Heartbleed 2014; Möller et al. 2014; Pincus and Baker 2004; Rizzo and Duong 2012; Smyth and Pironti 2014; Somorovsky 2016; Stevens et al. 2016; Święcki 2016; Szekeres et al. 2013; Wagner and Schneier 1996). Recognizing a clear need, the programming language, verification, and applied cryptography communities are devoting

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

significant efforts to develop implementations proven secure by construction against broad classes of these 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. (2015) provide nqsbTLS, an implementation of TLS in OCaml, which by virtue of its type and memory safety is impervious to attacks (like Heartbleed (2014)) that exploit buffer overflows. Bhargavan et al. (2014) 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 (2017), an implementation that mixes C and OCaml, and libcrypto, a component of the widely used OpenSSL library (2017). In the worst case, linking with vulnerable C code can void all the 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. To this end, we introduce Low*, a domain-specific language for verified, efficient, low-level programming embedded within F* (Swamy et al. 2016), 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 the verification capabilities of F* to prove our code is 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 experiments indicate that compiled Low* code yields performance on par with existing C code. This code can be used on its own, or used within existing software through the C ABI. In particular, our C code may be linked to F* programs compiled to OCaml, providing large speed-ups via its foreign-function interface (FFI) without compromising safety or 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 (Leroy 2009; Leroy et al. 2012), rather than the "big array of bytes" model systems programmers sometimes 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 their computationally irrelevant parts, such as proofs and specifications, using a process similar to Coq's extraction mechanism (Letouzey 2002). In particular, our memory-model library compiles to a simple, heap-based OCaml implementation.

Importantly, Low* programs have a second, equivalent but more efficient semantics via compilation to C, with the predictable performance that comes with a native implementation of their lower-level memory model. 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:

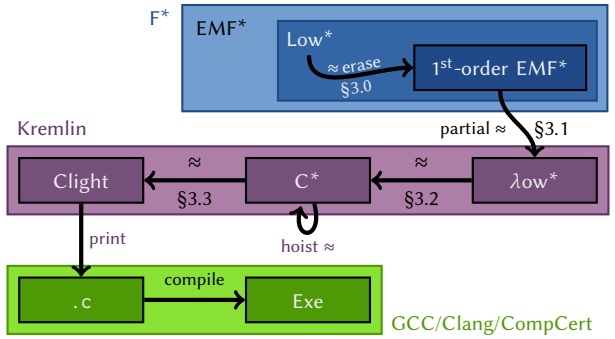

Fig. 1. Low* embedded in F*, compiled to C, with soundness and security guarantees (details in §3)

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. In this work, we instantiate the state monad of F* to use a CompCert-like structured memory model that separates the stack and the heap, supporting bulk allocation and deallocation on the stack, and allocating and freeing individual blocks on the heap. Both the heap and the stack are further divided into disjoint logical regions, which enables us to manage the separation properties necessary for modular, stateful verification. On top of this, we program a library of C-style arrays and structs passed by reference, with support for pointer arithmetic and pointers to the interior of an array or a struct. 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. As much as possible, we aim for the programmer to control even the syntactic structure of the 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 meet several restrictions, as follows: the code (1) must be first order, to prevent the need to allocate closures in C; (2) must make any heap allocation explicit; (3) must not use any recursive datatype, 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. Importantly, Low* heavily leverages F*'s capabilities for partial evaluation, hence allowing the programmer to write high-level, reusable code that is normalized via meta-programming into the Low* subset before the restrictions are enforced. 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 provide a formal model of Low*, called λow*, give a translation from λow* to Clight (Blazy and Leroy 2009) and show that it preserves trace equivalence with respect to the original F* semantics. In addition to ensuring that the functional behavior of a program is preserved, our trace equivalence also guarantees that the compiler does not accidentally introduce side-channels due to memory access patterns (as would be the case without the restrictions above) at least until it reaches Clight, a useful sanity check for cryptographic code. Our theorems cover the translation of standalone λow* programs to C, proving that execution in C preserves the original F* semantics of the λow* program.

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. After compilation, our verified standalone C libraries can be integrated within larger programs using standard means.

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

(1) We provide HACL*, a "high-assurance crypto library" implementing and proving (in ~6,000 lines of Low*) several cryptographic algorithms, including the Poly1305 MAC (Bernstein 2005), the ChaCha20 cipher (Nir and Langley 2015), and multiplication on the Curve25519 elliptic curve (Bernstein 2006). We package these algorithms to provide the popular NaCl API (Bernstein et al. 2012), yielding the first performant implementation of NaCl verified for memory safety and side-channel resistance, along with functional correctness proofs for its core components, including a verified bignum library customized for safe, fast cryptographic use (§5.1). Using this API, we build new standalone applications such as *PneuTube*, a new secure, asynchronous, file transfer application whose performance compares favorably with widely used, existing utilities like scp.

(2) Emphasizing the applicability of Low* for high-level, cryptographic security proofs on low-level code, we briefly describe its use in programming and proving (in ~14,000 lines of Low*) the Authenticated Encryption with Associated Data (AEAD) construction at the heart of the record layer of the new TLS 1.3 Internet Standard. We prove memory safety, functional correctness, and cryptographic security for its main ciphersuites, relying, where available, on verified implementations of these ciphersuites provided by HACL*. The C code extracted from our verified implementation is easily integrated within other applications, including, for example, an implementation in F* of TLS separately verified and compiled to OCaml (through OCaml's FFI).

Trusted computing base. To date, we have focused on designing and evaluating our methodology of programming and verifying low-level code shallowly embedded within a high-level programming language and proof assistant. We have yet to invest effort in minimizing the trusted computing base of our work, an effort we plan to expend now that we have evidence that our methodology is worthwhile. Currently, the trusted computing base of our verified libraries includes the implementation of the F* typechecker and the Z3 SMT solver (de Moura and Bjørner 2008). Additionally, we trust the manual proofs of the metatheory relating the semantics of λow* to CompCert Clight. The KreMLin tool is informed by this metatheory, but is currently implemented in unverified OCaml, and is also trusted. Finally, we inherit the trust assumptions of the C compiler used to compile the code extracted from Low*.

Supplementary materials. First, we provide, in the appendix, the hand proofs of the theorems described in §3. The present paper is focused on the metatheory and tools; we also authored a companion paper (Bhargavan et al. 2017) that describes the cryptographic model we used for the record layer of TLS 1.3. Finally, we have an ongoing submission of a paper focused on our HACL* library (Zinzindohoué et al. 2017), where we describe in greater detail our proof techniques for reusing the bignum formalization across different algorithms and implementations, and provide a substantial performance evaluation.

We also offer numerous software artifacts. Our tool KreMLin (Protzenko 2017) is actively developed on GitHub, and so is HACL* (Zinzindohoué et al. 2017). Most of the Low* libraries live in the F* repository, also on GitHub. The integration of HACL* within miTLS is also available on

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

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

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

GitHub. For convenience, we offer a regularly-updated Docker image of Project Everest (Microsoft Research and INRIA 2016), which bundles together F*, miTLS, HACL*, KreMLin. One may fetch it via `docker pull projecteverest/everest`. The Docker image contains a `README.md` with an overview of the proofs and the code.

2 A LOW* TUTORIAL

At the core of Low* is a library for programming with structures and arrays manually allocated on the stack or the heap (§2.2). Memory safety demands reasoning about the extents and liveness of these objects, 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 several examples from our codebase. We show the ChaCha20 stream cipher (Nir and Langley 2015), focusing on memory safety (§2.1), and the Poly1305 MAC (Bernstein 2005), focusing on functional correctness. (§2.3). Going beyond functional correctness, we explain how we prove a combination of ChaCha20 and Poly1305 cryptographically secure (§2.4). Throughout, we point out key benefits of our approach, notably our use of dependently typed metaprogramming to work at a relatively high-level of abstraction at little performance cost.

2.1 A first example: the ChaCha20 stream cipher

Figure 2 shows code snippets for the core function of ChaCha20 (Nir and Langley 2015), a modern stream cipher widely used for fast symmetric encryption. This function computes a block of pseudo-random bytes, usable for encryption, for example, by XORing them with a plaintext message. 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, contiguous state on the stack; then it calls a function to initialize the cipher block from the inputs (passing an interior pointer to the state); and finally it 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 7–10, the C code is in one-to-one correspondence with its Low* counterpart. These specifications capture the safe memory usage of `chacha20`. (Their

syntax is explained next, in §2.2.) For each argument passed by reference, and for the auxiliary state, they keep track of their liveness and size. They also capture its correctness, by describing the final state of the output buffer using a pure function.

Lines 2–3 use 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. The return type Stack unit on line 6 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 next line, the pre-condition requires that all arguments passed by reference be live. On lines 8–10, the post-condition first 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. The rest of the post-condition specifies functional correctness: the output buffer must contain a sequence of bytes equal to the first len bytes of the cipher specified by function Spec.chacha20 for the input values of key, nonce, and counter.

As usual for symmetric ciphers, RFC 7539 specifies chacha20 as imperative pseudocode, and does not further discuss its mathematical properties. We implement this pseudocode as a series of pure functions in F*, which can be extracted to OCaml and tested for conformance with the RFC test vectors. Functions such as Spec.chacha20 then serve as logical specifications for verifying our stateful implementation. In particular, the last postcondition of chacha20 ensures that its result is determined by its inputs. We describe more sophisticated functional correctness proofs for Poly1305 in §2.3.

2.2 Low*: 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 interior pointers. 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 7–8 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 diverges.[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 (Tofte and Talpin 1997) covering both stack and heap. Hyper-stacks are a generalization of hyper-heaps, a memory model proposed previously for F* (Swamy et al. 2016), 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 (McCarthy 1962) 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. (2002)'s *reaps*. However, at present, we do not 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

[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.

modular verification, as is already the case for hyper-heaps (Swamy et al. 2016), 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 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, using 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 have pushed) and yield a final memory with the same domain as the initial memory. This ensures that Low* code with Stack effect has explicitly deallocated all heap allocated references before returning, ruling out memory leaks. As such, we expect all externally callable Low* functions to have Stack effect. Other code can safely pass pointers to objects allocated in their heaps into Low* functions with Stack effect since the definition of Stack forbids the Low* code from freeing these references.

Modeling arrays. 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. We describe first our modeling of arrays by implementing an abstract type for buffers in Low*, using just the references provided by hyper-stacks. Relying on its abstraction, KreMLin compiles our buffers to native C arrays.

The type 'buffer a' below is a single-constructor inductive type with 4 arguments. Its main content argument holds a reference to a seq a, a purely functional sequence of a's whose length is determined by the first argument max_length. The 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 last two arguments 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 13 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′ covers only 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′ indicates its length and, using the includes relation, records that b and b′ are aliased.

```
val sub: b:buffer a → i:uint32 → 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 the index function 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))
```

All lengths and indices are 32-bit machine integers, and refer to the number of elements in the buffer, not the number of bytes the buffer occupies. This currently prevents addressing very large buffers on 64-bit platforms. (To this end, we may parameterize our development over a C data model, wherein indices for buffers would reflect the underlying (proper) ptrdiff_t type.)

Similarly, memory allocation remains platform-specific. It may cause a (fatal) error as it runs out of memory. More technically, the type of create may not suffice to prevent pointer-arithmetic overflow; if the element size is greater than a byte, and if the number of elements is 2^{32}, then the argument passed to malloc will overflow on a platform where sizeof size_t == 4. To prevent such cases, KreMLin inserts defensive dynamic checks (which typically end up eliminated by the C compiler since our stack-allocated buffer lengths are compile-time constants). In the future, we may statically prevent it by mirroring the C sizeof operator at the F* level, and requiring that for each Buffer.create operation, the resulting allocation size, in bytes, is no greater than what size_t can hold.

Modeling structs. Generalizing 'buffer t' (abstractly, a reference to a finite map from natural numbers to t), we model C-style structs as an abstract reference to a 'struct key value', that is, a map from keys k:key to values whose type 'value k' depends on the key. For example, we represent the type of a colored point as follows, using a struct with two fields X and Y for coordinates and one field Color, itself a nested struct of RGB values.

```
type color_fields = R | G | B
type color = struct color_fields (λ R | G | B → uint32)
type colored_point_fields = X | Y | Color
type colored_point = struct colored_point_fields (λ X | Y → int32 | Color → color)
```

C structs are flatly allocated; the declaration above models a contiguous memory block that holds 20 bytes or more, depending on alignment constraints. As such, we cannot directly perform pointer arithmetic within that block; rather, we navigate it by referring to fields. To this end, our library of structs provides an interface to manipulate pointers to these C-like structs, including pointers that follow a path of fields throughout nested structs. The main type provided by our library is the indexed type ptr shown below, encapsulating a base reference content: ref from and a path p of fields leading to a value of type to.

```
abstract type ptr: Type → Type = Ptr: #from:Type → content: ref from → #to: Type → p: path from to → ptr to
```

When allocating a struct on the stack, the caller provides a 'struct k v' literal and obtains a 'ptr (struct k v)', a pointer to a struct literal in the current stack frame (a Ptr with an empty path).

The extend operator below supports extending the access path associated with a 'ptr (struct k v)' to obtain a pointer to one of its fields.

```
val extend: #key: eqtype → #value: (key → Tot Type) → p: ptr (struct key value) → fd: key → ST (ptr (value fd))
  (requires (λ h → live h p))
  (ensures (λ h0 p′ h1 → h0 == h1 ∧ p′ == field p fd))
```

Finally, the read and write operations allows accessing and mutating the field referred to by a ptr.

```
val read: #a:Type → p: ptr a → ST value
  (requires (λ h → live h p))
  (ensures (λ h0 v h1 → live h0 p ∧ h0 == h1 ∧ v == as_value h0 p))

val write: #a:Type → b:ptr a → z:a → ST unit
  (requires (λ h → live h b))
  (ensures (λ h0 _h1 → live h0 b ∧ live h1 b ∧ modifies_1 b h0 h1 ∧ as_value h1 b == z))
```

2.3 Using Low* for proofs of functional correctness and side-channel resistance

This section and the next illustrate our "high-level verification for low-level code" methodology. Although programming at a low-level, we rely on features like 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 (Bernstein 2005), 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. (Zinzindohoué et al. 2016); 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 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 (Almeida et al. 2016a,b). 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.

[2]Implementation bugs in Poly1305 are still a practical concern: in 2016 alone, the Poly1305 OpenSSL implementation experienced two correctness bugs (Benjamin 2016; Böck 2016) and a buffer overflow (CVE 2016).

```
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 mask to eval modulo *)
```

```
1  val poly1305_mac:
2    tag:nbytes 16ul →
3    len:u32 →
4    msg:nbytes len{disjoint tag msg} →
5    key:nbytes 32ul{disjoint msg key ∧
6                    disjoint tag key} →
7    Stack unit
8  (requires (λ m → msg ∈ m ∧ key ∈ m ∧ tag ∈ m))
9  (ensures (λ m0 _m1 →
10   let r = Spec.clamp m0[sub key 0ul 16ul] in
11   let s = m0[sub key 16ul 16ul] in
12   modifies {tag} m0 m1 ∧
13   m1[tag] ==
14   Spec.mac_1305 (encode_bytes m0[msg]) r s))
```

Fig. 3. Unique representation of a Poly1305 bigint (left) and the top-level spec of Poly1305 (right)

For Poly1305, a bigint b is in compact form in state m (i.e., 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 (Kocher 1996). 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 operation (eq_mask) 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 (Swamy et al. 2016). 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* $2^{130} - 5$. 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.

A top-level functional correctness spec. Using our bigint library, we implement poly1305_mac and prove it functionally correct. Its specification (Figure 3, right) states that the final value of the 16

byte tag (m_1[tag]) is the value of Spec.mac_1305, a polynomial of the message and the key encoded as field elements. We use this mathematical specification as a basis for the game-based cryptographic proofs of constructions built on top of Poly1305, such as the AEAD construction, described next.

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, $\varepsilon_{\text{ChaCha20}}$ and $\varepsilon_{\text{Poly1305}}$. The argument then shows that the AEAD construction also meets its own ideal functionality, except with probability, say, $\varepsilon_{\text{Chacha20}} + \varepsilon_{\text{Poly1305}}$.

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.

```
type entry = cipher * data * nonce * plain
abstract type key = { key: keyBytes; log: if Flag.aead then ref (seq entry) else unit }
let encrypt (k:key) (n:nonce) (p:plain) (a:data) =
  if Flag.aead then let c = random_bytes ℓ_c in k.log := (c, a, n, p) :: k.log; c
  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. In the code above, for instance we idealize encryption when Flag.prf is set.

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, allowing the F* normalizer to drop the then branch, and replace 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 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.

Inlining and Type Abstraction. In cryptographic constructions, we often rely on type abstraction to protect private state that depends on keys and other secrets.

A typical C application, such as OpenSSL, achieves limited type abstraction as follows. The library exposes a public C header file for its clients, relying on void * and opaque heap allocation functions for type abstraction.

```
typedef void *KEY_HANDLE;
void KEY_init(KEY_HANDLE *key);
void KEY_release(KEY_HANDLE key);
```

Opportunities for mistakes abound, since the void * casts are unchecked. Furthermore, abstraction only occurs at the public header boundary, not between internal translation units. Finally, this pattern does not allow the caller to efficiently allocate the actual key on the stack.

The Low* discipline allows the programmer to achieve type abstraction and modularity, while still supporting efficient stack allocation. As an example, for computing one-time MACs incrementally, we use an accumulator that holds the current value of a polynomial computation, which depends on a secret key. For cryptographic soundness, we must ensure that no information about such intermediate values leak to the rest of the code.

To this end, all operations on accumulators are defined in a single module of the form below—our code is similar but more complex, as it supports MAC algorithms with different field representations and key formats, and also keeps track of the functional correctness of the polynomial computation.

```
module OneTimeMAC
type elem = lbytes (v accLen) (* intermediate value (representing a a field element) *)
abstract type key (i:macID) = elem
abstract type accum (i:macID) = elem
(* newAcc allocates on the caller's frame *)
let newAcc (i:macID) : StackInline (accum i) (...) = Buffer.create 0ul accLen
let extend (i:macID) (key: macKey i) (acc:accum i) (word:elem) : Stack unit (...) = add acc word; mul acc key
```

The index i is used to separate multiple instances of MACs; for instance, it prevents calls to extend an accumulator with the wrong key. Our type-based separation between different kinds of elements is purely static. At runtime, the accumulator, and probably the key, are just bytes on the stack (or in registers), whereas the calls to add and mul are also likely to be have been inlined in the code that uses MACs.

The newAcc function creates a new buffer for a given index, initialized to 0. The function returns a pointer to the buffer it allocates. The StackInline effect indicates that the function does need to push a frame before allocation, but instead allocates in its caller's stack frame. KreMLin textually inlines the function in its caller's body at every call-site, ensuring that the allocation performed by newAcc indeed happens in the caller's stack frame. From the perspective of Low*, newAcc is a function in a separate module, and type abstraction is preserved.

3 A FORMAL TRANSLATION FROM LOW* TO CLIGHT

Figure 1 on page 3 provides an overview of our translation from Low* to CompCert Clight, starting with EMF*, a recently proposed model of F* (Ahman et al. 2017); then λow*, a formal core of Low* after all erasure of ghost code and specifications; 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.

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

Fig. 4. λow* syntax

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 (Ahman et al. 2017) 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. 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 (Molnar et al. 2006) augmented to track potential leaks through cache behavior (Barthe et al. 2014). 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 $(b, n, [])$ and mutable structures $(b, n, \overrightarrow{f})$ passed by reference, where b is the address of the buffer or structure, n is the offset in the buffer, and $\overrightarrow{f}$ designates the path to the structure field to take a reference of (this path, as a list, can be longer than 1 in the case of nested mutable structures.) Stack allocated buffers (readbuf, writebuf, newbuf, and subbuf), and their mutable structure counterparts (readstruct, writestruct, newstruct, $\rhd$), 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 and mutable structures become inaccessible.

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

$$\frac{}{P \vdash (H, \mathsf{if}\ 0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) \to_{\mathsf{brF}} (H, e_2)}\ \textsc{LIfF} \qquad \frac{P(f) = \lambda y : \tau_1.\ e_1 : \tau_2}{P \vdash (H, \mathsf{let}\ x : \tau = f\ v\ \mathsf{in}\ e) \to (H, \mathsf{let}\ x : \tau = e_1[v/y]\ \mathsf{in}\ e)}\ \textsc{App}$$

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

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

Fig. 5. Selected semantic rules from λow*

Mutable structures can be nested, and stored into buffers, in both cases without extra indirection. However, the converse is not true, as λow* currently does not allow arbitrary nesting of arrays within mutable structures without explicit indirection via separately allocated buffers. We leave such generalization as future work.

Type system. λow* types include the base types int and unit, record types $\{\overrightarrow{f = \tau}\}$, buffer types buf τ, mutable structure types struct 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/mutable structure 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 or mutable structure references, and branching behavior, as shown below.

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

Figure 5 shows selected reduction rules from λow*. 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 App is a standard, substitutive β-reduction. 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.

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 ($\Gamma = \mathsf{secret{:}limb}$), 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

$e[v_1/\text{secret}]$ 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 3.1 (Secret independence). *Given*

(1) a program well-typed against a secret interface, Γ_s, i.e, $\Gamma_s, \Gamma_P; \Sigma; \Gamma \vdash (H, e) : \tau$,
(2) a well-typed implementation of the Γ_s interface, $\Gamma_s; \Sigma; \cdot \vdash_\Delta P_s$, such that P_s is equivalent modulo secrets,
(3) a pair (ρ_1, ρ_2) of well-typed substitutions for Γ,

then either:

(1) both programs cannot reduce further, i.e. $P_s, P \vdash (H, e)[\rho_1] \not\to$ and $P_s, P \vdash (H, e)[\rho_2] \not\to$, or
(2) both programs make progress with the same trace, i.e. there exists $\Sigma' \supseteq \Sigma, \Gamma' \supseteq \Gamma, H', e'$, a pair (ρ_1', ρ_2') of well-typed substitutions for Γ', and a trace ℓ such that
  i) $P_s, P \vdash (H, e)[\rho_1] \to_\ell^+ (H', e')[\rho_1']$ and $P_s, P \vdash (H, e)[\rho_2] \to_\ell^+ (H', e')[\rho_2']$, and
  ii) $\Gamma_s, \Gamma_P; \Sigma'; \Gamma' \vdash (H', e') : \tau$

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.

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

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

$$\frac{}{\hat{P} \vdash (S, V, \{\overrightarrow{s_1}\}; \overrightarrow{s_2}) \rightsquigarrow (S; (\{\}, V, \square; \overrightarrow{s_2}), V, \overrightarrow{s_1})}\ \textsc{Block} \qquad \frac{}{\hat{P} \vdash (S; (M, V', E), V, []) \rightsquigarrow (S, V', E\,[()])}\ \textsc{Empty}$$

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

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

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

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

Fig. 6. Selected semantic rules from C*

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 6, 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 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 3.2 (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., $\hat{P} \vdash ([], V, \overrightarrow{s}; \text{return } \hat{e}) \overset{\ell,*}{\rightarrow} ([], V', \text{return } v)$ if, and only if, so does the λow* program: $P \vdash (\{\}, e[V]) \overset{\ell,*}{\rightarrow} (H', v)$; 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 (Blazy and Leroy 2009; Leroy 2016) 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]

Recall that we need to produce an event in the trace whenever a memory location is read or written, and whenever a conditional branch is taken, to account for memory accesses and statements in the semantics of the generated Clight code for the purpose of our noninterference security guarantees. However, the semantics of CompCert Clight *per se* produces no events on memory accesses; instead, CompCert provides a syntactic program annotation mechanism using no-op *built-in calls*, whose only purpose is to add extra events in the trace. Thus, we leverage this mechanism by prepending each memory access and conditional statement in the Clight generated code with one such built-in call producing the corresponding events.

The main two differences between C* and Clight, which our translation deals with as described below, are immutable local structures, and scope management for local variables.

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

If we naively translate immutable 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 an immutable 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 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' (Dockins 2012, §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 pass changing the memory layout, we split it into three passes. First, we *reinterpret* the program by replacing each pointer value in event traces with the function name and recursion depth of its function call, the name of the corresponding local variable, and the array index and

[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.

structure field name within this local variable. Then, we perform the actual transformation and prove that it exactly preserves traces in this new "abstract" trace model. Finally, we reinterpret the generated code back to concrete pointer values. We successfully used this technique to prove functional correctness and security preservation for variable hoisting.

For each pass that adds new memory accesses, we split it into two passes. First, a reinterpretation pass produces new events corresponding to the provisional memory accesses (without actually performing those memory accesses). Then, this pass is followed by the actual trace-preserving transformation that goes back to the non-reinterpreted language but adds the actual memory accesses into the program. We successfully used this technique to prove functional correctness and security preservation for structure return, where we add new events and memory accesses whenever a C* function returns a structure value.

In both cases, we mean *reinterpretation* as defining a new language with the same syntax and small-step semantic rules except that the produced traces are different, and relating executions of the same program in the two languages. There, it is easy to prove functional correctness, but for security preservation, we need to prove an invariant on two small-step executions of the same program with different secrets, to show that two equal pointer values in event traces coming from those two different executions will actually turn into two equal abstract pointer values in the reinterpreted language.

Our detailed functional correctness and security preservation proofs from λow* to Clight can be found in the appendix.

Towards verified assembly code. We conjecture 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 by instrumenting CompCert (Barthe et al. 2014) and LLVM (Almeida et al. 2016b; Zhao et al. 2012, 2013) and turning them 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.

4 KREMLIN: A COMPILER FROM LOW* TO C

4.1 From Low* to efficient, elegant C

As explained previously, λow* is the core of Low*, post erasure. Transforming Low* into λow* proceeds in several stages. First, we rely on F*'s existing normalizer and erasure and extraction facility (similar to features in Coq (Letouzey 2008)), to obtain an ML-like AST for Low* terms. Then, we use our new tool KreMLin that transforms this AST further until it falls within the λow* subset formalized above. KreMLin then performs the λow* to C* transformation, followed by the C* to C transformation and pretty-printing to a set of C files. KreMLin generates C11 code that may be compiled by GCC; Clang; Microsoft's C compiler or CompCert. We describe the main transformations performed by KreMLin, beyond those formalized in §3, next.

Structures by value. We described earlier (§2.2) our Low* struct library that grants the programmer fine-grained control over the memory layout, as well as mutability of interior fields. As an alternative, KreMLin supports immutable, by-value structs. Such structures, being pure, come with no liveness proof obligations. The performance of the generated C code, however, is less predictable: in many cases, the C compiler will optimize and pass such structs by reference, but on some ABIs (x86), the worst-case scenario may be costly.

Concretely, the F* programmer uses tuples and inductive types. Tuples are monomorphized into specialized inductive types. Then, inductive types are translated into idiomatic C code: single-branch inductive types (e.g., records) become actual C structs, inductives with only constant constructors become C enums, and other inductives becomes C tagged unions, leveraging C11 anonymous unions for syntactic elegance. Pattern matches become, respectively, switches, let-bindings, or a series of cascading if-then-elses.

Whole-program transformations. KreMLin perform a series of whole-program transformations. First, the programmer is free to use parameterized type abbreviations. KreMLin substitutes an application of a type abbreviation with its definition, since C's typedef does not support parameters. (C++11 alias templates would support this use-case.) Second, KreMLin recursively inlines all StackInline functions, as required for soundness (cf. §2.4). Third, KreMLin performs a reachability analysis. Any function that is not reachable from the main function or, in the case of a library, from a distinguished API module, is dropped. This is essential for generating palatable C code that does not contain unused helper functions used only for verification. Fourth, KreMLin supports a concept of "bundle", meaning that several F* modules may be grouped together into a single C translation unit, marking all of the functions as static, except for those reachable via the distinguished API module. This not only makes the code much more idiomatic, but also triggers a cascade of optimizations that the C compiler is unable to perform across translation units.

Going to an expression language. F* is, just like ML, an expression language. Two transformations are required to go to a statement language: *stratification* and *hoisting*. Stratification places buffer allocations, assignments and conditionals 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.

Readability. 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; we use enum and switch whenever possible; 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.

Implementation. KreMLin represents about 10,000 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 support multiple backends; we are currently implementing a WebAssembly backend to provide verified, efficient cryptographic libraries for the web.

4.2 Integrating KreMLin's output

KreMLin generates a set of C files that have no dependencies, beyond a single .h file and C11 standard headers, meaning KreMLin's output can be readily integrated into an existing source tree.

To allow code sharing and re-use, programmers may want to generate a shared library, that is, a .dll or .so file that can be distributed along with a public header (.h) file. The programmer can achieve this by writing a distinguished API module in F*, exposing only carefully-crafted function signatures. As exemplified earlier (Figure 2), the translation is predictable, meaning the programmer can precisely control, in F*, what becomes, in C, the library's public header. The bundle feature of

KreMLin then generates a single C file for the library; upon compiling it into a shared object, the only visible symbols are those exposed by the programmer in the header file.

We used this approach for our HACL* library. Our public header file exposes functions that have the exact same signature as their counterpart in the NaCL library. If an existing binary was compiled against NaCL's public header file, then one can configure the dynamic linker to use our HACL* library instead, without recompiling the original program (using the infamous "LD preload trick").

The functions exposed by the library comply with the C ABI for the chosen toolchain. This means that one may use the library from a variety of programming languages, relying on foreign-function interfaces to interoperate. One popular approach is to generate bindings for the C library *at run-time* using the ctypes and the libffi (Green 2014) libraries. This is an approach leveraged by languages such as JavaScript, Python or OCaml, and requires no recompilation.

An alternative is to write bindings by hand, which allows for better performance and control over how data is transformed at the boundary, but requires writing and recompiling potentially error-prone C code. This is the historical way of writing bindings for many languages, including OCaml. We plan to have KreMLin generate these bindings automatically. We used this approach in miTLS, effectively making it a mixed C/OCaml project. We intend to eventually lower all of miTLS into Low*.

5 BUILDING VERIFIED LOW* LIBRARIES AND APPLICATIONS

Codebase	LoC	C LoC	%annot	Verif. time
Low* standard library	8,936	N/A	N/A	8m
HACL*	6,050	11,220	28%	12h
miTLS AEAD	13,743	14,292	56.5%	1h 10m

Table 1. Evaluation of verified Low* libraries and 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 describe 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. Then, 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*: A fast and safe cryptographic library

In the wake of numerous security vulnerabilities, Bernstein et al. (2012) 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 (Bernstein 2006) for key exchange, the Salsa family of symmetric encryption algorithms (Bernstein 2008), which includes Salsa20 and ChaCha20, and Poly1305 for message authentication (Bernstein 2005). 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 (lib 2017), 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 (Bernstein et al. 2014) 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.

A Verified NaCl Library. We implement the NaCl API, including all its component algorithms, in a Low* library called HACL*, mechanically verifying that all our code is memory safe, functionally correct, 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. Our code is written and optimized for 64-bit platforms; on 32-bit machines, we rely on a stub library for performing 64x64-bit multiplications and other 128-bit operations.

We implement and verify four cryptographic primitives: ChaCha20, Salsa20, Poly1305, and Curve25519, and then use them to build three cryptographic constructions: AEAD, `secretbox` and `box`. For all our primitives, we prove that our stateful optimized code matches a high-level functional specification written in F*. These are new verified implementations. Previously, Tomb (2016) used SAW and Cryptol to verify C and Java implementations of Chacha20, Salsa20, Poly1305, AES, and ECDSA. Using a different methodology, Bond et al. (2017) verifies an assembly version of Poly1305. Curve25519 has been verified before: Chen et al. (2014) verified an optimized low-level assembly implementation using an SMT solver; Zinzindohoué et al. (2016) wrote and verified a high-level library of three curves, including Curve25519, in F* and generated an OCaml implementation from it. Our verified Curve25519 code explores a third direction by targeting reference C code that is both fast and readable.

A companion paper currently under review (Zinzindohoué et al. 2017) is entirely devoted to the HACL* library, and contains an in-depth evaluation of the proof methodology, several new algorithms that were verified since the present paper was written, along with a more comprehensive performance analysis.

Algorithm	HACL*	Sodium	TweetNaCl	OpenSSL	eBACS Fastest
ChaCha20	6.17 cy/B	6.97 cy/B	-	8.04 cy/B	1.23 cy/B
Salsa20	6.34 cy/B	8.44 cy/B	15.14 cy/B	-	1.39 cy/B
Poly1305	2.07 cy/B	2.48 cy/B	32.32 cy/B	2.16 cy/B	0.68 cy/B
Curve25519	157k cy/mul	162k cy/mul	1663k cy/mul	359k cy/mul	145k cy/mul
AEAD-ChaCha20-Poly1305	8.37 cy/B	9.60 cy/B	-	8.53 cy/B	
SecretBox	8.43 cy/B	11.03 cy/B	50.56 cy/B	-	
Box	18.10 cy/B	20.97 cy/B	149.22 cy/B	-	

Table 2. Performance in CPU cycles: 64-bit HACL*, 64-bit Sodium (pure C, no assembly), 32-bit TweetNaCl, 64-bit OpenSSL (pure C, no assembly), and the fastest assembly implementation included in eBACS SUPERCOP. All code was compiled using `gcc -O3` optimized and run on a 64-bit Intel Xeon CPU E5-1630. Results are averaged over 1000 measurements, each processing a random block of 2^{14} bytes; Curve25519 was averaged over 1000 random key-pairs.

Algorithm	HACL*	OpenSSL	CNG
Curve25519	17700 mul/s (σ = 246)	8033 mul/s (σ = 120)	7490 mul/s (σ = 114)

Table 3. Performance in operations per second: 64-bit HACL*, 64-bit OpenSSL (assembly disabled) and Microsoft's "Crypto New Generation" (CNG) library on a 64-bit Windows 10 machine. These results were obtained by writing an OpenSSL engine that calls back to either HACL*, CNG, or OpenSSL itself (so as to include the overhead of going through a pluggable engine). The speed ecdhx25519 command runs multiplications for 10s, then counts the number of multiplications performed. We show the average over 10 runs of this command. The machine is a desktop machine with a 64-bit Intel Xeon CPU E5-1620 v2 nominally clocked at 3.70Ghz.

Performance. Table 2 compares the performance of HACL* to Sodium, TweetNaCl, and OpenSSL by running each primitive on a 16KB input; we chose this size since it corresponds to the maximum record size in TLS and represents a good balance between small network messages and large files. We report averages over 1000 iterations expressed in cycles/byte. For Curve25519, we measure the time taken for one call to scalar multiplication. For comparison with state-of-the-art assembly implementations, for each primitive, we also include the best performance for any implementation (assembly or C) included in the eBACS SUPERCOP benchmarking framework.[5] These fastest implementation are typically in architecture-specific assembly.

We performed these tests on a variety of 64-bit Intel CPUs (the most popular desktop configuration) and these performance numbers were similar across machines. To confirm these measurements, we also ran the full eBACS SUPERCOP benchmarks on our code, as well as the OpenSSL speed benchmarks, and the results closely mirrored Table 2. However, we warn the performance numbers could be quite different on (say) 32-bit ARM platforms.

We observe that for ChaCha20, Salsa20, and Poly1305, HACL* achieves comparable performance to the optimized C code in OpenSSL and Sodium and significantly better performance than TweetNaCl's concise C implementation. Assembly implementations of these primitives are about 3-4 times faster; they typically rely on CPU-specific vector instructions and careful hand-optimizations.

Our Curve25519 implementation is about the same speed as Sodium's 64-bit C implementation (donna_c64) and an order of magnitude faster than TweetNaCl's 32-bit code. It is also significantly faster than OpenSSL because even 64-bit OpenSSL uses a Curve25519 implementation that was optimized for 32-bit integers, whereas the implementations in Sodium and HACL* take advantage of the 64x64-bit multiplier available on Intel's 64-bit platforms. The previous F* implementation of Curve25519 (Zinzindohoué et al. 2016) running in OCaml was not optimized for performance; it is more than 100x slower than HACL*. The fastest assembly code for Curve25519 on eBACS is the one verified in (Chen et al. 2014). This implementation is only 1.08x faster than our C code, at least on the platform on which we tested, which supported vector instructions up to 256 bits. We anticipate that the assembly code may be significantly faster on platforms that support larger 512-bit vector instructions.

AEAD and secretbox essentially amount to a ChaCha20/Salsa20 cipher sequentially followed by Poly1305, and their performance reflects the sum of the two primitives. Box uses Curve25519 to compute a symmetric key, which it then uses to encrypt a 16KB input. Here, the cost of symmetric encryption dominates over Curve25519.

In summary, our measurements show that HACL* is as fast as (or faster than) state-of-the-art C crypto libraries and within a small factor of hand-optimized assembly code. This finding is not entirely unexpected, since we wrote our Low* code by effectively porting the fastest C

[5] https://bench.cr.yp.to/supercop.html

implementations to F*, and any algorithmic optimization that is implemented in C can, in principle, be written (and verified) in Low*. What is perhaps surprising is that we get good performance even though our Low* code, and consequently the generated C, heavily relies on functional programming patterns such as tail-recursion, and even though we try to write generic compact code wherever possible, rather than trying to mimic the verbose inlined style of assembly code. We find that modern compilers like GCC and CLANG are able to optimize our code quite well, and we are able to benefit from their advancements, without having to change our coding style. Where needed, KreMLin helps the C compiler by inserting attributes like `const`, `static` and `inline` that act as optimization hints.

Balancing Trust and Performance. All the above performance numbers were obtained with GCC-6 with most architecture-specific optimizations turned on (`-march=native`). Consequently, any bug in GCC or its plugins could break the correctness and security guarantees we proved in F* for our source code. For example, GCC has an auto-vectorizer that significantly improves the performance of our ChaCha20 and Salsa20 code in certain use cases, but does so by substantially changing its structure to take advantage of the parallelism provided by SIMD vector instructions. To avoid trusting this powerful but unverified mechanism, and for more consistent results across platforms, we turned off auto-vectorization (`-fno-tree-vectorize`) for the numbers in Table 2. For similar reasons, we turned off link-time optimization (`-fno-lto`) since it relies on an external linker plugin, and can change the semantics of our library every time it is linked with a new application.

Ideally, we would completely remove the burden of trust on the C compiler by moving to CompCert, but at significant performance cost. Our Salsa20 and ChaCha20 code incurs a relatively modest 3x slowdown when compiled with CompCert 3.0 (with `-O3`). However, our Poly1305 and Curve25519 code incurs a 30-60x slowdown, which makes the use of CompCert impractical for our library. We anticipate that this penalty will reduce as CompCert improves, and as we learn how to generate C code that would be easier for CompCert to optimize. For now, we continue to use GCC and CLANG and comprehensively test the generated code using third-party tools. For example, we test our code against other implementations, and run all the tests packaged with OpenSSL. We also test our compiled code for side-channel leaks using tools like DUDECT.[6]

PneuTube: Fast encrypted file transfer. Using HACL*, we can build a variety of high-assurance security 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 to a user-defined size. 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 or writes a file or a socket between calling open and close).

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. Its aynchronous 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.

[6] https://github.com/oreparaz/dudect

5.2 Cryptographically secure AEAD for miTLS

We use our cryptographically secure AEAD library (§2.4) within miTLS (Bhargavan et al. 2013), 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 (Bhargavan et al. 2017). 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 reduces the security of TLS to the PRF and MAC idealizations in AEAD. We 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.

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 (Cohen et al. 2009; Jacobs et al. 2014; Kirchner et al. 2015). 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 (Appel 2015; Beringer et al. 2015; Chen et al. 2016) and Isabelle (Schirmer 2006; Winwood et al. 2009) 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 (Klein et al. 2009), Greenaway et al. (2012, 2014) 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 (Appel 2015; Beringer et al. 2015; Dodds 2016). 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 (Condit et al. 2007; Jim et al. 2002; Tarditi 2016). 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 (Amani et al. 2016; Matsakis and Klock II 2014). 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 (rus 2017) 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 (Amin and Rompf 2017) 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 of being far away from the original source code.

Bedrock (Chlipala 2013) 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.

Our companion paper "Implementing and Proving the TLS 1.3 Record Layer" (Bhargavan et al. 2017) is available online. 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 paper 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

This paper advocates a new methodology for carrying out high-level proofs on low-level code. By embedding a low-level language and memory model within F*, the programmer not only enjoys sophisticated proofs but also gets to write their low-level code in a more modular style, using features functional programmers take for granted, including recursion and type abstraction. Our toolchain, relying on partial evaluation and the latest advances in C compilers, shows that we can write code in a style suitable for verification *and* enjoy the same performance as hand-written C code.

We are currently making progress in three different directions. First, continuing our integration of AEAD within miTLS, we aim to port the miTLS protocol layer to Low*, in order to get an entire verified, TLS library in C. Second, parts of our toolchain are unverified. We plan to formalize and verify using F* parts of the KreMLin tool, notably the λow* to C* transformation. Third, we are working on embedding assembly instructions within Low*, allowing us to selectively optimize our code further towards closing the performance gap that still remains relative to architecture-specific, hand-written assembly routines.

REFERENCES

2008–2017. The Sodium crypto library (libsodium). (2008–2017). https://www.gitbook.com/book/jedisct1/libsodium/details

2010–2017. The Rust Programming Language. (2010–2017). https://www.rust-lang.org

2016. CVE-2016-7054: ChaCha20/Poly1305 heap-buffer-overflow. (Nov. 2016). http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-7054

2017. Common Weakness Enumeration (CWE-190: Integer Overflow or Wraparound). (2017). https://cwe.mitre.org/data/definitions/190.html

2017. Common Weakness Enumeration (CWE-415: Double Free). (2017). http://cwe.mitre.org/data/definitions/415.html

2017. Common Weakness Enumeration (CWE-416: Use After Free). (2017). http://cwe.mitre.org/data/definitions/416.html

J. Afek and A. Sharabani. 2007. Dangling Pointer – Smashing The Pointer For Fun And Profit. BlackHat USA. (July 2007).

Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. 2017. Dijkstra Monads for Free. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages*

(POPL). ACM, 515–529. DOI: http://dx.doi.org/10.1145/3009837.3009878

Nadhem J. AlFardan and Kenneth G. Paterson. 2013. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *2013 IEEE Symposium on Security and Privacy*. 526–540.

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2016a. Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC. In *Fast Software Encryption - 23rd International Conference, FSE 2016*. 163–184.

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016b. Verifying Constant-Time Implementations. In *25th USENIX Security Symposium, USENIX Security 16*. 53–70.

Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, and others. 2016. COGENT: Verifying High-Assurance File System Implementations. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 175–188.

Nada Amin and Tiark Rompf. 2017. 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). https://www.cs.purdue.edu/homes/rompf/papers/amin-draft2016b.pdf

Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.* 37, 2 (2015), 7.

Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. 2014. System-level Non-interference for Constant-time Cryptography. In *2014 ACM SIGSAC Conference on Computer and Communications Security, CCS 2014*. 1267–1279.

David Benjamin. 2016. poly1305-x86.pl produces incorrect output. https://mta.openssl.org/pipermail/openssl-dev/2016-March/006161. (2016).

Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. 2002. Reconsidering Custom Memory Allocation. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA 2002*. ACM, 1–12.

Lennart Beringer, Adam Petcher, Q Ye Katherine, and Andrew W Appel. 2015. Verified correctness and security of OpenSSL HMAC. In *24th USENIX Security Symposium (USENIX Security 15)*. 207–221.

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

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

Daniel J Bernstein. 2008. The Salsa20 family of stream ciphers. In *New stream cipher designs*. Springer, 84–97.

Daniel J Bernstein, Tanja Lange, and Peter Schwabe. 2012. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America, LATINCRYPT 2012*. Springer, 159–176.

Daniel J Bernstein, Bernard Van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. 2014. TweetNaCl: A crypto library in 100 tweets. In *International Conference on Cryptology and Information Security in Latin America, LATINCRYPT 2014*. 64–83.

Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, , Alfredo Pironti, and Pierre-Yves Strub. 2014. Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS. In *2014 IEEE Symposium on Security and Privacy*. 98–113.

Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jianyang Pan, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella Béguelin, and Jean Karim Zinzindohoue. 2017. Implementing and Proving the TLS 1.3 Record Layer. *IEEE Security & Privacy* (2017).

Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and P Strub. 2013. Implementing TLS with verified cryptographic security. In *IEEE Symposium on Security and Privacy*. 445–459.

Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Santiago Zanella-Béguelin. 2014. Proving the TLS handshake secure (as it is). In *Advances in Cryptology–CRYPTO 2014*. Springer, 235–255.

Karthikeyan Bhargavan and Gaëtan Leurent. 2016. 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.

Sandrine Blazy and Xavier Leroy. 2009. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning* 43, 3 (2009), 263–288.

Hanno Böck. 2016. Wrong results with Poly1305 functions. https://mta.openssl.org/pipermail/openssl-dev/2016-March/006413. (2016).

Hanno Böck, Aaron Zauner, Sean Devlin, Juraj Somorovsky, and Philipp Jovanovic. 2016. Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS. Cryptology ePrint Archive, Report 2016/475. (2016). http://eprint.iacr.org/2016/475.

Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *Proceedings of the USENIX Security Symposium*.

Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2016. Toward compositional verification of interruptible OS kernels and device drivers. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016*. 431–447.

Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. 2014. Verifying curve25519 software. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 299–309.

Adam Chlipala. 2013. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 391–402.

Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A practical system for verifying concurrent C. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 23–42.

Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C Necula. 2007. Dependent types for low-level programming. In *European Symposium on Programming*. Springer, 520–535.

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS (Lecture Notes in Computer Science)*, Vol. 4963. Springer, 337–340. DOI: http://dx.doi.org/10.1007/978-3-540-78800-3_24

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

Robert W. Dockins. 2012. *Operational Refinement for Compiler Correctness*. Ph.D. Dissertation. Princeton University.

Joey Dodds. 2016. Part one: Verifying s2n HMAC with SAW. Galois Blog. (Sept. 2016). https://galois.com/blog/2016/09/specifying-hmac-in-cryptol/

Thai Duong and Juliano Rizzo. 2011. Here Come The ⊕ Ninjas. Available at http://nerdoholic.org/uploads/dergln/beast_part2/ssl_jun21.pdf. (May 2011).

Anthony Green. 2014. The libffi home page. (2014). http://sourceware.org/libffi

David Greenaway, June Andronick, and Gerwin Klein. 2012. Bridging the Gap: Automatic Verified Abstraction of C. In *3rd International Conference on Interactive Theorem Proving, ITP 2012 (Lecture Notes in Computer Science)*, Vol. 7406. Springer, 99–115.

David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. 2014. 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*. ACM, 429–439.

Heartbleed. 2014. The Heartbleed Bug. http://heartbleed.com/. (2014).

Bart Jacobs, Jan Smans, and Frank Piessens. 2014. The VeriFast Program Verifier: A Tutorial. iMinds-DistriNet, Department of Computer Science, KU Leuven - University of Leuven, Belgium. (2014). https://people.cs.kuleuven.be/~bart.jacobs/verifast/tutorial.pdf

Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C.. In *USENIX Annual Technical Conference, General Track*. 275–288.

David Kaloper-Meršinjak, Hannes Mehnert, Anil Madhavapeddy, and Peter Sewell. 2015. Not-quite-so-broken TLS: Lessons in re-engineering a security protocol specification and implementation. In *24th USENIX Security Symposium (USENIX Security 15)*. 223–238.

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Asp. Comput.* 27, 3 (2015), 573–609.

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. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the Symposium on Operating Systems Principles*. ACM, 207–220.

Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology – CRYPTO 1996*. Springer, 104–113.

Xavier Leroy. 2004–2016. The CompCert C verified compiler. http://compcert.inria.fr/. (2004–2016).

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2*. Research report RR-7987. INRIA. http://hal.inria.fr/hal-00703441

Xavier Leroy and Sandrine Blazy. 2008. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning* 41, 1 (2008), 1–31.

Pierre Letouzey. 2002. A new extraction for Coq. In *Types for proofs and programs*. Springer, 200–219.

Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *4th Conference on Computability in Europe (Lecture Notes in Computer Science)*, Vol. 5028. Springer, 359–369. DOI: http://dx.doi.org/10.1007/978-3-540-69407-6_39

Nicholas D Matsakis and Felix S Klock II. 2014. The Rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.

John McCarthy. 1962. Towards a Mathematical Science of Computation. In *IFIP Congress*. 21–28.

Microsoft Research and INRIA. 2016. Everest: VERifiEd Secure Transport. https://project-everest.github.io/. (2016).

Bodo Möller, Thai Duong, and Krzysztof Kotowicz. 2014. This POODLE Bites: Exploiting The SSL 3.0 Fallback. Available at https://www.openssl.org/~bodo/ssl-poodle.pdf. (2014).

David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2006. 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*. Springer, 156–168.

Yoav Nir and Adam Langley. 2015. ChaCha20 and Poly1305 for IETF Protocols. IETF RFC 7539. (2015).

nocrypto. 2014–2017. nocrypto: OCaml cryptographic library. (2014–2017). https://github.com/mirleft/ocaml-nocrypto

OpenSSL library. 1998–2017. OpenSSL: Cryptography and SSL/TLS Toolkit. (1998–2017). https://www.openssl.org/

Jonathan D. Pincus and Brandon Baker. 2004. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns. *IEEE Security & Privacy* 2, 4 (2004), 20–27.

Jonathan Protzenko. 2017. The KreMLin compiler. (2017). https://www.github.com/FStarLang/kremlin

Julian Rizzo and Thai Duong. 2012. The CRIME Attack. (September 2012).

Norbert Schirmer. 2006. *Verification of sequential imperative programs in Isabelle-HOL*. Ph.D. Dissertation. Technical University Munich.

Ben Smyth and Alfredo Pironti. 2014. *Truncating TLS Connections to Violate Beliefs in Web Applications*. Technical Report hal-01102013. Inria. https://hal.inria.fr/hal-01102013

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

Marc Stevens, Pierre Karpman, and Thomas Peyrin. 2016. Freestart Collision for Full SHA-1. In *Advances in Cryptology – EUROCRYPT 2016*. Springer, 459–483.

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. https://www.fstar-lang.org/papers/mumon/

Robert Święcki. 2016. ChaCha20/Poly1305 heap-buffer-overflow. CVE-2016-7054. (2016).

Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 48–62.

David Tarditi. 2016. Extending C with bounds safety. Checked C Technical Report, Version 0.6. (Nov. 2016). https://github.com/Microsoft/checkedc

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Inf. Comput.* 132, 2 (Feb. 1997), 109–176.

A. Tomb. 2016. Automated Verification of Real-World Cryptographic Implementations. *IEEE Security Privacy* 14, 6 (2016), 26–33.

David Wagner and Bruce Schneier. 1996. Analysis of the SSL 3.0 Protocol. In *2nd USENIX Workshop on Electronic Commerce, WOEC 1996*. 29–40.

Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. 2009. Mind the Gap. In *22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009 (Lecture Notes in Computer Science)*, Vol. 5674. Springer, 500–515.

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 427–440.

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2013. Formal verification of SSA-based optimizations for LLVM. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 175–186.

Jean Karim Zinzindohoué, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. 2016. A Verified Extensible Library of Elliptic Curves. In *IEEE Computer Security Foundations Symposium (CSF)*.

Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. (2017). http://eprint.iacr.org/2017/536

Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. (2017). https://www.github.com/mitls/hacl-star

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 7.

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

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

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

Fig. 7. Big-stepping a small-step semantics

B C* AND λOW* DEFINITION

Notations used in the document are summarized in Figure 8. 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 9 and 13 respectively. C* syntax is defined in such a way that C* expressions do not have side effects (but can fail to evaluate because of e.g. referring to a nonexistent variable). 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 (as well as mutable struct allocation, mutable struct write) 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 10. 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		

Fig. 8. Notations

B.1 C* Definition

The following are the definitions of the syntax and operational semantics of C*.

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
	$\&e \rightarrow fd$	struct field address (e is a pointer)
	loc	location
loc ::=		locations
	$(b, n, \overrightarrow{fd})$	

Fig. 9. C* Syntax

$v ::=$		values
	n	constant
	$()$	unit value
	$\{\overrightarrow{fd = v}\}$	constant struct
	loc	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
	$b \rightharpoonup \overrightarrow{\widetilde{v}}$	map from block id to list of optional values
$V ::=$		variable assignments
	$x \rightharpoonup v$	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

Fig. 10. C* Semantics Definitions

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

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

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

$$\frac{}{\left[\!\left[\{\overrightarrow{fd = e}\}\right]\!\right]_{(p,V)} = \{\overrightarrow{fd = [\![e]\!]_{(p,V)}}\}}\ \textsc{NonMutStruct}$$

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

Fig. 11. C* Expression Evaluation

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

$$\frac{[\![e]\!]_{(p,V)} = v}{p \vdash (S, V, t\ x = e; ss) \rightsquigarrow (S, V[x \mapsto v], ss)}\ \textsc{VarDecl}$$

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

$$\frac{[\![e_1]\!]_{(p,V)} = (b, n, []) \qquad [\![e_2]\!]_{(p,V)} = v \qquad \mathsf{Set}(S, (b, n, []), v^m) = S'}{p \vdash (S, V, \mathsf{memset}\ e_1\ m\ e_2; ss) \rightsquigarrow_{\mathsf{write}\ (b,n,[]),\ldots,\mathsf{write}\ (b,n+m-1,[])} (S', V, ss)}\ \textsc{Memset}$$

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

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

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

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

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

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

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

$$\frac{[\![e]\!]_{(p,V)} = n \qquad n \neq 0}{p \vdash (S, V, \mathsf{if}\ e\ \mathsf{then}\ ss_1\ \mathsf{else}\ ss_2; ss) \rightsquigarrow_{\mathsf{brT}} (S, V, ss_1; ss)}\ \textsc{IfT}$$

$$\frac{[\![e]\!]_{(p,V)} = n \qquad n = 0}{lp \vdash (S, V, \mathsf{if}\ e\ \mathsf{then}\ ss_1\ \mathsf{else}\ ss_2; ss) \rightsquigarrow_{\mathsf{brF}} (S, V, ss_2; ss)}\ \textsc{IfF}$$

Fig. 12. C* Configuration Reduction

B.2 λow* Definition

The following are the definitions of the syntax and operational semantics of λow*.

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 as value
	$le.fd$	immutable struct field projection
	$le \triangleright fd$	sub-structure: mutable structure field projection
	subbuf le le	sub-buffer
	if le then le else le	conditional
	let $x : t = le$ in le	let-binding
	let _ = le in le	anonymous let-binding
	let $x : t = f$ le in le	application
	let x = newbuf n $(le : t)$ in le	new buffer
	let $x : t$ = readbuf le le in le	read from buffer
	let _ = writebuf le le le in le	write to buffer
	let x = newstruct $(le : t)$ in le	new mutable structure
	let $x : t$ = readstruct le in le	read from mutable structure
	let _ = writestruct le in le	write to mutable structure
	withframe le	with-frame
	pop le	pop frame
	loc	location

Fig. 13. λow* Syntax

values

$$lv \quad ::= \quad n \mid () \mid \{\overrightarrow{f\,d = lv}\} \mid loc$$

evaluation contexts

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

stack

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

stack frame

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

Fig. 14. λow^* Semantics Definitions

$\boxed{lp \vdash (H, le) \twoheadrightarrow_l (H', le')}$ and $\boxed{lp \vdash (H, le) \rightsquigarrow_l (H', le')}$

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

$$\frac{H(b, n, \overrightarrow{fd}) = lv}{lp \vdash (H, \text{let } x = \text{readstruct } (b, n, \overrightarrow{fd}) \text{ in } le) \twoheadrightarrow_{\text{read } (b, n, \overrightarrow{fd})} (H, [lv/x]le)} \textsc{ReadStruct}$$

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

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

$$\frac{(b, n, \overrightarrow{fd}) \in H}{lp \vdash (H, \text{let } \_ = \text{writestruct } (b, n, \overrightarrow{fd}) \; lv \text{ in } le) \twoheadrightarrow_{\text{write } (b, n, \overrightarrow{fd})} (H[(b, n, \overrightarrow{fd}) \mapsto lv], le)} \textsc{WriteStruct}$$

$$\frac{}{lp \vdash (H, \text{subbuf } (b, n, []) \; n') \twoheadrightarrow (H, (b, n+n', []))} \textsc{Subbuf}$$

$$\frac{}{lp \vdash (H, (b, n, \overrightarrow{fd}) \rhd fd') \twoheadrightarrow (H, (b, n, (\overrightarrow{fd}; fd')))} \textsc{StructField}$$

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

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

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

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

$$\frac{b \notin H}{lp \vdash (H, \text{let } x = \text{newstruct } (lv : t) \text{ in } le) \twoheadrightarrow_{\text{write } (b, 0, [])} (H[b \mapsto lv], [(b, 0, [])/x]le)} \textsc{NewStruct}$$

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

$$\frac{lp \vdash (H, le) \twoheadrightarrow_l (H', le')}{lp \vdash (H, LE \; [le]) \rightsquigarrow_l (H', LE \; [le'])} \textsc{Step}$$

Fig. 15. λow^* Atomic Reduction and Reduction

C λOW* TO C* COMPILATION

The compilation procedure is defined in Figure 16 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.

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

$$\frac{}{\wr\ n = n} \qquad \frac{}{\wr\ (b, n, []) = (b, n, [])} \qquad \frac{}{\wr\ \{\overrightarrow{fd = le}\} = \{\overrightarrow{fd =\wr\ le}\}} \qquad \frac{}{\wr\ x = x}$$

$$\frac{}{\wr\ \text{subbuf}\ le_1\ le_2\ =\wr\ le_1 +\wr\ le_2}$$

$$\frac{}{\wr\ le \triangleright fd = \&\wr\ le \rightarrow fd}$$

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

$$\frac{\wr\ le_i = e_i\ (i = 1) \qquad \downarrow le = ss}{\downarrow \text{let}\ x : t = \text{readstruct}\ le_1\ \text{in}\ le = (t\ x = *[e_1]; ss)}$$

$$\frac{\wr\ le_i = e_i\ (i = 1, 2, 3) \qquad \downarrow le = ss}{\downarrow (\text{let}\ \_ = \text{writebuf}\ le_1\ le_2\ le_3\ \text{in}\ le) = (*[e_1 + e_2] = e_3; ss)}$$

$$\frac{\wr\ le_i = e_i\ (i = 1, 2) \qquad \downarrow le = ss}{\downarrow (\text{let}\ \_ = \text{writestruct}\ le_1\ le_2\ \text{in}\ le) = (*[e_1] = e_2; ss)}$$

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

$$\frac{\wr\ le = e \qquad \downarrow le_1 = ss}{\downarrow (\text{let}\ x = \text{newstruct}\ (le : t)\ \text{in}\ le_1) = (t\ x[1]; \text{memset}\ x\ 1\ e; ss)}$$

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

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

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

Fig. 16. λow* to C* compilation

D BISIMULATION PROOF

The main results are Theorem D.6 and D.7, in terms of some notions defined before them in this section. The two theorems are proved by using the crucial Lemma D.12 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).

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.

The specific relation between a λow* configuration and its C* counterpart is defined in Definition D.14 as Relation R. It is defined in terms of a C*-to-λow* back-translation, listed in Fig 18. We need a back-translation here instead of the λow*-to-C* forward-translation as defined before, because a C* configuration has a clear call-stack with each frame containing its own variable environment and continuation, while a λow* configuration contains one giant λow* expression which makes it impossible to recover the call-stack. Hence for relating two configurations in the middle of reduction, only the C*-to-λow* direction is possible.

Definition D.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, $\rightsquigarrow \subseteq \Sigma \times \text{option } L \times \Sigma$ 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 $a \Downarrow_o a' \stackrel{\text{def}}{=} a \rightsquigarrow_o^* a'$ and $a' \in F$.

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

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

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

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

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.

Theorem D.6 (Safety). *For all λow^* program lp, closed expression le and closing substitution V, if $\Downarrow lp = p$, $\downarrow le = ss$ and $sys_{\lambda ow^*}(lp, V(le))$ is safe, then $sys_{C^*}(p, V, ss)$ is safe.*

Proof. Appeal to Lemma D.12, Lemma D.16 and Lemma D.15. □

Theorem D.7 (Bisimulation). *For all λow^* program lp, closed expression le and closing substitution V, if $\Downarrow lp = p$ and $\downarrow le = ss$, then $sys_{C^*}(p, V, ss)$ bisimulates $sys_{\lambda ow^*}(lp, V(le))$.*

Proof. Appeal to Corollary D.13, Lemma D.16 and Lemma D.15. □

Definition D.8 (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 D.9 (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 D.10 (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 D.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 D.11 (Refine-Safety). *If transition system A refines transition system B by R and for any a and b we have a R b implies unstuck(a), then A is safe.*

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

Lemma D.12 (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 D.11, 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 D.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 D.13 (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 D.14 (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 $(H, le) \rightsquigarrow_{lp}^n (H, le')$ and $(H, le') = \uparrow (S, V, ss)$, where $\uparrow (S, V, ss) \stackrel{\text{def}}{=} (\mathsf{mem}(S), \mathsf{unravel}(S, V(\uparrow (\Downarrow_{(p,V)} ss))))$ and $\mathsf{mem}(S)$ is all the memory parts of S collected together (and requiring that there is no $\perp$ in S's memory parts).

$$\mathsf{unravel}(S, le) \stackrel{\text{def}}{=} \mathsf{foldl}\ \mathsf{unravel\_frame}\ le\ S$$
$$\mathsf{unravel\_frame}((M, V, E), le) \stackrel{\text{def}}{=} \begin{cases} V((\Uparrow E)\ [le]) & \text{if } M = \bot \\ V((\Uparrow E)\ [\mathsf{pop}\ le]) & \text{if } M = \lfloor \_ \rfloor \end{cases}.$$

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

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

$$\frac{}{\Downarrow_{(p,V)}\ (t\ x[n]; ss) = (t\ x[n]; ss)}$$

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

$$\frac{[\![e_i]\!]_{(p,V)} = v_i\ (i = 1, 2, 3)}{\Downarrow_{(p,V)}\ (*[e_1 + e_2] = e_3; ss) = (*[v_1 + v_2] = v_3; ss)}$$

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

Fig. 17. Normalize C* head expression

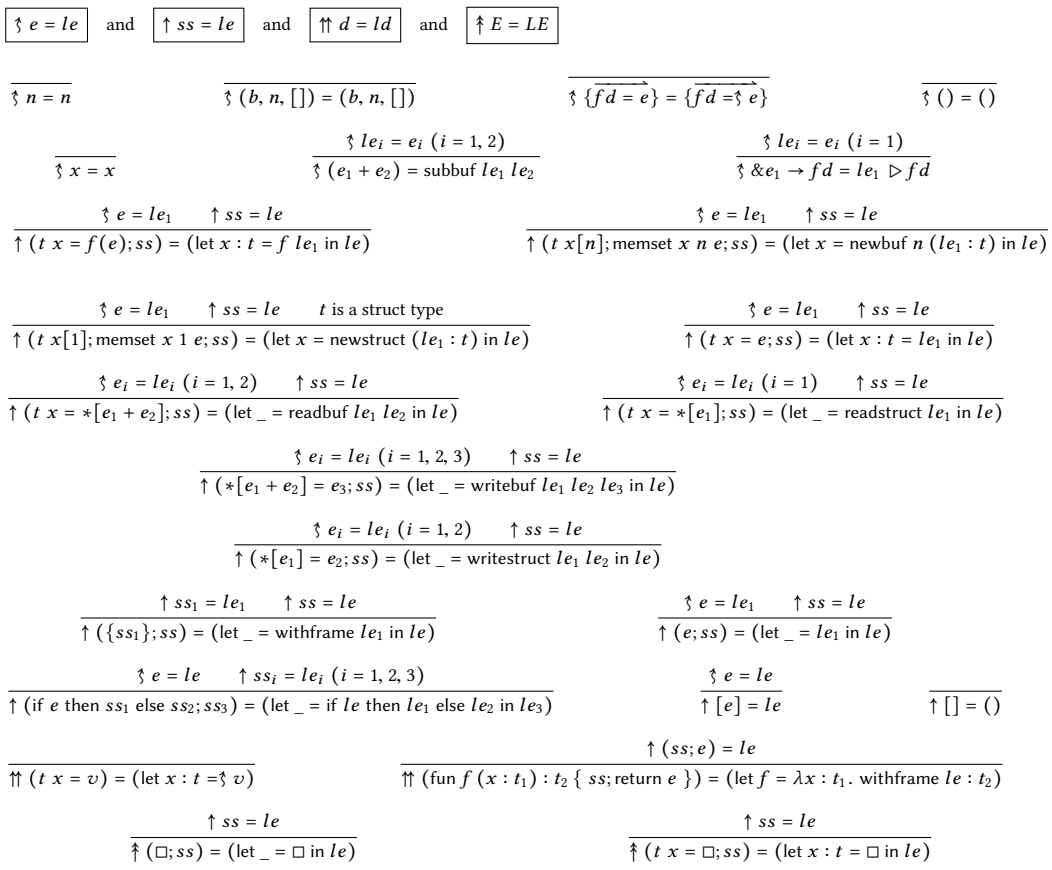

Fig. 18. C* to λow* back-translation

Lemma D.15 (λow* Refines C*). *For all λow^* program lp, closed expression le and closing substitution V, if $\Downarrow lp = p$ and $\downarrow le = ss$, then $sys_{\lambda ow^*}(lp, V(le))$ refines $sys_{C^*}(p, V, ss)$.*

Proof. We apply Lemma D.10 and D.17, and prove that $\mathrm{sys}_{\lambda ow^*}(lp, le)$ quasi-refines $\mathrm{sys}_{C^*}(p, ss)$.
We pick the relation $R_{p,lp}$ in Definition D.14 to be the simulation relation and prove $R_{p,lp}$ is a quasi-refinement for $\mathrm{sys}_{\lambda ow^*}(lp, le)$ of $\mathrm{sys}_{C^*}(p, ss)$.
Unfold Definition D.9.
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 D.18.
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 D.20.
In the first case, we have $le = V(\uparrow ss')$ and $lv = \updownarrow v$ and $LE = \mathsf{unravel}(S, \square)$, where $v \stackrel{\text{def}}{=} [\![e]\!]_{(p,V)}$.
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 D.19, 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 D.21, the proof is similar to the previous case.
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 D.28, we have $ss = (t\ x = f(v); ss')$ and $le = V(\uparrow ss')$ and $lv = \updownarrow v$ and $LE = \mathsf{unravel}(S, \square)$.
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 D.32, we know $\uparrow ss_1 = le_2$ hence $\uparrow \{ss_1\} = le_1$.
Pick C' to be $(S; (\bot, V, t\ x = \square; ss'), \{y \mapsto v\}, \{ss_1\})$.
It suffices to show that
$(H, LE\ [\mathsf{let}\ x : t = [lv/v]le_1\ \mathsf{in}\ le])\ R\ (S; (\bot, V, t\ x = \square; ss'), \{y \mapsto v\}, \{ss_1\})$, which is true.
End of case.

Case Withframe:
on case $(H, LE\ [\mathsf{withframe}\ le]) \rightsquigarrow (H; \{\}, LE\ [\mathsf{pop}\ le])$.
Appealing to Lemma D.29, 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 Newbuf: on case $(H; h, LE\ [\mathsf{let}\ x = \mathsf{newbuf}\ n\ (lv : t)\ \mathsf{in}\ le]) \rightsquigarrow_{\mathsf{write}\ (b,0,[]),\ldots,\mathsf{write}\ (b,n-1,[])} (H; h[b \mapsto lv^n], LE\ [[(b, 0, [])/x]le])$ and $b \notin H; h$.
We have $(H; h, LE\ [\mathsf{let}\ x = \mathsf{newbuf}\ n\ (lv : t)\ \mathsf{in}\ le])\ R\ (S, V, ss)$.
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 D.22, 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 $(H; h[b \mapsto lv^n], LE\ [[(b, 0, [])/x]le])\ R\ (S'; (M[b \mapsto v^n], V', E), V[x \mapsto (b, 0, [])], ss')$, which is true.
End of case.

Case Newstruct: on case $(H; h, LE\ [\mathsf{let}\ x = \mathsf{newstruct}\ (lv : t)\ \mathsf{in}\ le]) \rightsquigarrow_{\mathsf{write}\ (b,0,[])} (H; h[b \mapsto lv], [(b, 0, [])/x]le)$ and $b \notin H; h$.
We have $(H; h, \mathsf{let}\ x = \mathsf{newstruct}\ (lv : t)\ \mathsf{in}\ le)\ R\ (S, V, ss)$.
We are to exhibit C' so that
$(H; h[b \mapsto lv], LE\ [[(b, 0, [])/x]le])\ R\ C'$ and $(S, V, ss) \rightsquigarrow^+ C'$.
Appealing to Lemma D.23, we have $ss = (t\ x[1]; \mathsf{memset}\ x\ 1\ v; ss')$ 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 $(S'; (M[b \mapsto v], V', E), V[x \mapsto (b, 0, [])], ss')$.
It suffices to show that $(H; h[b \mapsto lv], LE\ [[(b, 0, [])/x]le])\ R\ (S'; (M[b \mapsto v], V', E), V[x \mapsto (b, 0, [])], ss')$, 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 D.24, we have $ss = (t\ x = (b, n, [])[n']; ss')$ and $le = V(\uparrow ss')$ and $LE = \mathsf{unravel}(S, \Box)$.
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 Readstruct: on case
$(H, LE\ [\mathsf{let}\ x : t = \mathsf{readstruct}\ (b, n, \overrightarrow{fd})\ \mathsf{in}\ le]) \rightsquigarrow_{\mathsf{read}\ (b,n,\overrightarrow{fd})} (H, LE\ [[lv/x]le])$ and $H(b, n, \overrightarrow{fd}) = lv$.

Appealing to Lemma D.25, we have $ss = (t\ x = *[(b, n, \overrightarrow{fd})]; ss')$ 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 $C \rightsquigarrow^{+}_{\mathsf{read}\ (b,n,\overrightarrow{fd})} C'$.
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 D.26, 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 Writestruct: on case
$(H, LE\ [\mathsf{let}\ \_ = \mathsf{writestruct}\ (b, n, \overrightarrow{fd})\ lv\ \mathsf{in}\ le]) \rightsquigarrow_{\mathsf{write}\ (b,n,\overrightarrow{fd})} (H[(b, n, \overrightarrow{fd}) \mapsto lv], LE\ [le])$ and $(b, n, \overrightarrow{fd}) \in H$.
Appealing to Lemma D.27, we have $ss = (*[(b, n, \overrightarrow{fd})] = v; ss')$ and $le = V(\uparrow ss')$ and $lv =\wr v$ and $LE = \mathsf{unravel}(S, \square)$.
Pick C' to be (S', V, ss') where $\mathsf{Set}(S, (b, n, \overrightarrow{fd}), v) = S'$.
We know $C \rightsquigarrow^{+}_{\mathsf{write}\ (b,n,\overrightarrow{fd})} C'$.
It suffices to show that $(H[(b, n, \overrightarrow{fd}) \mapsto lv], LE\ [le])\ R\ (S', V, ss')$, 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 Structfield: on case $(H, LE\ [(b, n, \overrightarrow{fd}) \triangleright fd'']) \rightsquigarrow (H, LE\ [(b, n, (\overrightarrow{fd}; fd''))])$.
Pick C' to be (S, V, ss).
Because $(H, LE\ [(b, n, \overrightarrow{fd}) \triangleright fd''])\ R\ C'$ with some m, it must be the case that $m \geq 1$ and $(H, LE\ [(b, n, (\overrightarrow{fd}; fd''))])\ R\ C'$ 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 D.30, 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 $(H, LE\ [\text{if } n \text{ then } le_1 \text{ else } le_2]) \rightsquigarrow_{\mathsf{brF}} (H, LE\ [le_2])$ 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 $(H; h, LE\ [\text{pop } lv]) \rightsquigarrow (H, LE\ [lv])$.
Apply Lemma D.31.
In the first case, from $LE = \mathsf{unravel}(S', V'(\uparrow \square; ss'))$ we know $LE = (\text{let } \_ = \square \text{ in } le)$ 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 D.16 (C* Deterministic). *For all p and ss, transition system $sys_{C^*}(p, ss)$ is deterministic, modulo renaming of block identifiers.*

Lemma D.17 (λow* Deterministic). *For all lp and le, transition system $sys_{\lambda ow^*}(lp, le)$ is deterministic, modulo renaming of block identifiers.*

Lemma D.18 (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 D.19 (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 D.20 (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 $lv =\!\!\upharpoonright v$ and $LE = unravel(S, \square)$, where $v \stackrel{\text{def}}{=} [\![e]\!]_{(p,V)}$ or $S = S'; (\bot, V', t\ x = \square; ss')$ and $ss = return\ v$ and $le = V'(\uparrow ss')$ and $lv =\!\!\upharpoonright v$ and $LE = unravel(S', \square)$.*

Lemma D.21 (Invert ALet). *If $(H, LE\ [let\ \_ = lv\ in\ le])\ R\ (S, V, ss)$, then either $ss = (e; ss')$ and $le = V(\uparrow ss')$ and $lv =\!\!\upharpoonright v$ and $LE = unravel(S, \square)$, where $v \stackrel{\text{def}}{=} [\![e]\!]_{(p,V)}$ or $S = S'; (\bot, V', \square; ss')$ and $ss = return\ v$ and $le = V'(\uparrow ss')$ and $lv =\!\!\upharpoonright v$ and $LE = unravel(S', \square)$.*

Lemma D.22 (Invert Newbuf). *If* $(H; h, LE\ [let\ x = newbuf\ n\ (lv : t)\ in\ le])\ R\ (S, V, ss)$, *then* $ss = (t\ x[n]; memset\ x\ n\ v; ss')$ *and* $le = V(\uparrow ss')$ *and* $lv = \natural\ v$ *and* $LE = unravel(S, \square)$ *and* $S = S'; (M, V', E)$.

Lemma D.23 (Invert Newstruct). *If* $(H; h, LE\ [let\ x = newstruct\ (lv : t)\ in\ le])\ R\ (S, V, ss)$, *then* $ss = (t\ x[1]; memset\ x\ 1\ v; ss')$ *and* $le = V(\uparrow ss')$ *and* $lv = \natural\ v$ *and* $LE = unravel(S, \square)$ *and* $S = S'; (M, V', E)$.

Lemma D.24 (Invert Readbuf). *If*
$(H; h, LE\ [let\ x : t = readbuf\ (b, n, [])\ n'\ in\ le])\ R\ (S, V, ss)$, *then* $ss = (t\ x = (b, n, [])[n']; ss')$ *and* $le = V(\uparrow ss')$ *and* $LE = unravel(S, \square)$.

Lemma D.25 (Invert Readstruct). *If*
$(H; h, LE\ [let\ x : t = readstruct\ (b, n, \overrightarrow{fd})\ in\ le])\ R\ (S, V, ss)$, *then* $ss = (t\ x = *[(b, n, \overrightarrow{fd})]; ss')$ *and* $le = V(\uparrow ss')$ *and* $LE = unravel(S, \square)$.

Lemma D.26 (Invert Writebuf). *If*
$(H; h, LE\ [let\ \_ = writebuf\ (b, n, [])\ n'\ lv\ in\ le])\ R\ (S, V, ss)$, *then* $ss = ((b, n, [])[n'] = v; ss')$ *and* $le = V(\uparrow ss')$ *and* $lv = \natural\ v$ *and* $LE = unravel(S, \square)$.

Lemma D.27 (Invert Writestruct). *If*
$(H; h, LE\ [let\ \_ = writestruct\ (b, n, \overrightarrow{fd})\ lv\ in\ le])\ R\ (S, V, ss)$, *then* $ss = (*[(b, n, \overrightarrow{fd})] = v; ss')$ *and* $le = V(\uparrow ss')$ *and* $lv = \natural\ v$ *and* $LE = unravel(S, \square)$.

Lemma D.28 (Invert App). *If* $(H, LE\ [let\ x : t = f\ lv\ in\ le])\ R\ (S, V, ss)$, *then* $ss = (t\ x = f(v); ss')$ *and* $le = V(\uparrow ss')$ *and* $lv = \natural\ v$ *and* $LE = unravel(S, \square)$.

Lemma D.29 (Invert Withframe). *If* $(H, LE\ [withframe\ le])\ R\ (S, V, ss)$, *then* $ss = \{ss_1\}; ss_2$ *and* $le = V(\uparrow ss_1)$ *and* $LE = unravel(S, V(\Uparrow (\square; ss_2)))$.

Lemma D.30 (Invert If). *If* $(H, LE\ [if\ n\ then\ le_1\ else\ le_2])\ R\ (S, V, ss)$, *then* $ss = if\ e\ then\ ss_1\ else\ ss_2; ss'$ *and* $[\![e]\!]_{(p,V)} = n$ *and* $le_i = V(\uparrow ss_i)$ $(i = 1, 2)$ *and* $LE = unravel(S, V(\Uparrow (\square; ss')))$.

Lemma D.31 (Invert Pop). *If* $(H; h, LE\ [pop\ lv])\ R\ (S, V, ss)$, *then either* $ss = e$ *and* $[\![e]\!]_{(p,V)} = v$ *and* $\natural\ v = lv$ *and* $S = S'; (M, V', \square; ss')$ *and* $LE = unravel(S', V'(\Uparrow \square; ss'))$, *or* $ss = return\ e$ *and* $[\![e]\!]_{(p,V)} = v$ *and* $\natural\ v = lv$ *and* $S = S'; (M, V', E)$ *and* $LE = unravel(S', V'(\Uparrow E))$.

Lemma D.32 (Low2C-C2Low). *If* $\downarrow le = ss; e$, *then* $\uparrow (ss; return\ e) = le$.

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 (Leroy 2016, 2009). 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 (Blazy and Leroy 2009) is a subset of C with no side effects in expressions, and actual byte-level representation of values. Syntax in Figure 19. Semantics definitions in Figure 20. Evaluation of expressions in Figure 21. Small-step semantics in Figure 22.

The semantics of a Clight program is given by the return value of its main function called with no arguments.[7] 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 $\_V(r) = i$.

p ::=		program
	$\overrightarrow{d}$	series of declarations
d ::=		declaration
	$\mathsf{fun}\ f\ (x : t) : t\ \{\ \overrightarrow{ad}, ss\ \}$	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
	$\mathsf{annot}(\mathsf{read}, t, e)$	annotation to produce read event
	$\mathsf{annot}(\mathsf{write}, t, e)$	annotation to produce write event
	$\mathsf{if}\ e\ \mathsf{then}\ ss\ \mathsf{else}\ ss$	conditional
	$\{ss\}$	block
	$\mathsf{return}\ e$	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 (e lvalue)
	$\&e$	address of a lvalue (rvalue, e lvalue)
	$^*e$	pointer dereference (lvalue, e rvalue)

Fig. 19. Clight Syntax

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.

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

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.

$v ::=$		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
	$(S, V, \_V, M, ss)$	

Fig. 20. Clight Semantics Definitions

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

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

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

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

$$\frac{\mathsf{lv}(e, (p, V, \_V)) = (b, n)}{\mathsf{rv}(\&e, (p, V, \_V)) = (b, n)}\ \textsc{AddrOf}$$

Fig. 21. Clight Expression Evaluation

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

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

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

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

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

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

$$\frac{}{\mathsf{valloc}([], V, M) = (V, M)}\ \textsc{AllocNil}$$

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

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

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

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

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

Fig. 22. Clight Configuration Reduction

The CompCert memory model. The semantics of CompCert Clight statements depends on the CompCert memory model (Leroy and Blazy 2008). Here we need three operations: Get, Set and Alloc, whose description follows.[8]

$\mathsf{Get}(M, (b, n), n')$ 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.

[8]In the current version of CompCert and its memory model (Leroy et al. 2012), 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.

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[9] 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 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 $(2, 0, [])$.

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.

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,

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

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 E.1 (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.

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

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

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

Fig. 23. C* 2 Amended Configuration Reduction

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 E.2 (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 E.3. *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 E.2, 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. 23, 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 E.4 (C* to C* 2). *If $sys(p, V, ss)$ 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 E.5 (C* 2 invariant). *Let p be a C* 2 program and V a variable environment such that $sys(p, V, ss)$ 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 E.6 (C* 2 noninterference invariant). *Let p be a C* 2 program, and V_1, V_2 be two variable environments such that $sys(p, V_1, ss)$ and $sys(p, V_2, ss)$ 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 E.5. In particular the equality of codes is a consequence of the fact that there are no function pointers

Algorithm: VarOfBlock
Inputs:

- C* 2 configuration $(S, V, \_, \_, A)$ such that the invariants of Lemma E.5 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 $\bot$.)

Let n be the number of frames in S_1 of the form $(\bot, \_, \_, f', \_)$ 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)

Fig. 24. C* 2: retrieving the local variable corresponding to a memory block

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

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

Fig. 25. C* 3 Amended Configuration Reduction

in C*.[10] 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 24.

Then, let C* 3 be the C* 2 language where the Read and Write rules are changed according to Figure 25, with event traces where the actual pointer is replaced into an abstract pointer obtained using the VarOfBlock algorithm above.

Lemma E.7 (C* 2 to C* 3 functional correctness). *If $sys(p, V, ss)$ has no unambiguous variables, then $sys(p, V, ss)$, has the same behaviors in C* 2 with event traces with read, write removed, as in C* 3 with event traces with read, write removed.*

Proof. Lock-step bisimulation with equal configurations. Steps Read and Write need the invariant of Lemma E.5 on C* 2 to prove that C* 3 does not get stuck (ability to apply VarOfBlock.) □

[10] 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.

Lemma E.8 (VarOfBlock inversion). *Let C_1, C_2 two C* 2 configurations such that invariants of Lemma E.5 and Lemma E.6 hold. Then, for any block identifiers b_1, b_2 such that $\textsc{VarOfBlock}(C_1, b_1)$ and $\textsc{VarOfBlock}(C_2, b_2)$ are both defined and equal, then $b_1 = b_2$.*

Proof. Assume $\textsc{VarOfBlock}(C_1, b_1) = \textsc{VarOfBlock}(C_2, b_2) = (f, n, x)$. 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 E.5, 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 E.6, we have $V_1'(x) = V_2'(x)$, thus $b_1 = b_2$.

□

Lemma E.9 (C* 2 to C* 3 noninterference). *Assume that $sys(p, V_1, ss)$ and $sys(p, V_2, ss)$ 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 E.5 and Lemma E.6. 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 E.8. □

Lemma E.10 (C* 3 invariants). *The invariants of Lemma E.5 and Lemma E.6 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.[11] 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 E.11. *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. □

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

Hoisting.

Definition E.12 (Hoisting). For any list of statements ss with unambiguous local variables, the *hoisting* operation $\text{hoist}(ss) = (ads, ss')$ 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 $\text{hoist}(ss) = (ads, ss')$; and then, hoisting the local variables in a program p, $\text{hoist}(p)$, is defined as hoisting the local variables in each of its functions.

Definition E.13 (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 $\text{VarOfBlock}(C_1, b_1)$ is undefined, or $\text{VarOfBlock}(C_1, b_1)$ is defined and equal to $\text{VarOfBlock}(C_2, b_2)$.

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 E.14 (Correctness of hoisting). *If $sys(p, V, ss)$ is safe in C* 3 with a CompCert-style allocator, then $sys(p, V, ss)$ and $sys(hoist(p), V, hoist(ss))$ have the same execution traces (and in particular, the latter is also safe) in C* 3 using the same CompCert-style allocator.*

[11] 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 $\text{alloc}(D, \Sigma) = (b, \Sigma')$, 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*).

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 E.5 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 $\mathrm{Get}(S, b, n, fds)$, if defined, corresponds to $\mathrm{Get}(S', b', n, fd)$ 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 $(\bot, V_1', E, f_1, A_1')$; $(M', V_2', \square$ 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 26, extracted to C as

```
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 *)
```

Fig. 26. Low* benchmarking for structure erasure

Figure 27. 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 28.

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

```
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 }
```

Fig. 27. Extracted C code, before structure erasure

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 $\mathsf{FunResVar}(f, x) = \mathsf{FunResVar}(\text{ss', x})$ if $f(\_)\{ss'\}$ is a function defined in p, and $\mathsf{FunResVar}(\bot, x) = \mathsf{FunResVar}(ss, x)$.

```
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 }
```

Fig. 28. Extracted C code, after structure erasure

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

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

Fig. 29. C* 4 Amended Configuration Reduction

Then, we define C* 4 as the language C* 3 where the Ret_2 function return rule is replaced with two rules following Figure 29, 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 E.15 (C* 3 to C* 4 functional correctness). *If $sys(p, V, ss)$ 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 E.16 (C* 4 invariants). *The C* 3 invariants of Lemma E.5 and E.6 also hold on C* 4.*

Proof. This is true because the invariants of Lemma E.5 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.) □

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

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

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

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

Fig. 30. C* 4 to C* 3 structure return transformation

Theorem E.17 (C* 3 to C* 4 noninterference). *If $sys(p, V_1, ss)$ and $sys(p, V_2, ss)$ 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 30, 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 E.18 (StructRet correctness). *If $sys(p, V, ss)$ is safe in C* 4 and has unambiguous local variables, then it has the same behavior and trace as $sys(StructRet(p), V, StructRet(ss, \bot))$.*

Proof. Forward downward simulation, where the compilation invariant also involves block identifier renaming from Definition E.13 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 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:

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

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

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 E.19 (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 26, with C code after structure erasure in Figure 28, 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 31.

Lemma E.20 (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 E.21 (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

$$\frac{}{\Gamma \vdash n \downarrow^{\text{Int}} n}\ \textsc{Int} \qquad \frac{(x:t) \in \Gamma}{\Gamma \vdash x \downarrow^{t} x}\ \textsc{Var} \qquad \frac{\Gamma \vdash e_1 \downarrow^{t*} e_1' \qquad \Gamma \vdash e_2 \downarrow^{\text{int}} e_2'}{\Gamma \vdash e_1 + e_2 \downarrow^{t*} e_1' + e_2'}\ \textsc{PtrAdd}$$

$$\frac{\Gamma \vdash e \downarrow^{\text{struct}\{fd:t;\dots\}*} e'}{\Gamma \vdash \&e \to fd \downarrow^{t*} \&e' \to fd}\ \textsc{PtrFd} \qquad \frac{\Gamma \vdash e \downarrow^{\text{struct}\{fd:t;\dots\}} x.\overrightarrow{fds} \qquad t \text{ is a struct}}{\Gamma \vdash e.fd \downarrow^{t} x.\overrightarrow{fds}.fd}\ \textsc{StructFieldName}$$

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

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

Fig. 31. C* Structure Erasure: Expressions

- 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 E.22 (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 E.23. *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 32

In particular, each function parameter of structure type passed by value is replaced with its recursive list of all non-structure fields.[12]

Theorem E.24 (C* 5 to C* 3 structure erasure: shape). *If $p \downarrow p'$ following Figure 32, 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 E.22. □

Theorem E.25 (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 32, then p and p' have the same execution traces.*

[12]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.

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

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

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

$$\frac{t \text{ not a struct} \qquad \Gamma \vdash e \downarrow^{t} e'}{p, \Gamma \vdash \text{return } e \downarrow \text{return } e'}\ \textsc{Ret} \qquad \frac{p, \Gamma \vdash ss \downarrow ss'}{p, \Gamma \vdash \{ss\} \downarrow \{ss'\}}\ \textsc{Block}$$

$$\frac{p(f) = \text{fun}(\overrightarrow{\_ : t_i}) : t\{\_\} \qquad t \text{ not a struct} \qquad p, \Gamma \vdash (\overrightarrow{t_i}, el) \downarrow el' \qquad \Gamma[x] \leftarrow t}{p, \Gamma \vdash t\ x = f\ (el) \downarrow t\ x = f(el')}\ \textsc{Call}$$

$$\frac{}{p, \Gamma \vdash ([], []) \downarrow []}\ \textsc{ArgNil} \qquad \frac{p, \Gamma \vdash (t, e) \downarrow el_1 \qquad p, \Gamma \vdash (tl, el) \downarrow el_2}{p, \Gamma \vdash (t; tl, e; el) \downarrow el_1; el_2}\ \textsc{ArgCons}$$

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

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

$$\frac{t \text{ not a struct}}{(x : t) \downarrow (x : t)}\ \textsc{ScalarParam} \qquad \frac{t = \text{struct}\{\overrightarrow{fd_i : t_i}\} \qquad \overrightarrow{(x\_fd_i : t_i) \downarrow vt_i}}{(x : t) \downarrow \overrightarrow{vt_i}}\ \textsc{StructParam}$$

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

Fig. 32. C* 5 to C* 3 Structure Erasure: Statements

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 E.20, also using Lemma E.23. 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.

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

Fig. 33. C* 2 to Clight: Expressions

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 E.7 and Lemma E.9 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[13], 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 33. 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 E.26. *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, $rv(\mathbb{C}^t_A(e), (p, V', \_V')) = [\![e]\!]_{(p,V)}$.

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 34.

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

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

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

Fig. 34. C* 2 to Clight: Statements

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

Fig. 35. C* 2 to Clight: Program and Entrypoint

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 35.

Theorem E.27 (C* 2 to Clight: correctness). *If $sys(p, V, ss)$ 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.

F PROOF OF THE SECRET INDEPENDENCE THEOREM

```
---------
Notations
---------

t ::= int | unit | { f_i : t_i } | buf t | a

v ::= x | n | () | { f_i = v_i } | (b, n) | stuck

P ::= Top level functions    (same as lp)

G ::= x_i:t_i    (type enviroment)

S ::= (b_i, n_i):t_i    (store typing)

G_s = a_i, f_j:t_j -> t'_j    (type variables and signature of functions that
                                manipulate secrets)

G_f = x_i:a_i    (free variables of secret types)

q ::= a_i |-> t_i    (instantiations of type variables)

p ::= x_i |-> v_i    (substitutions of secret variables)


-----------
Judgments:
-----------

Remark: In the following judgments, we use the program P itself in the typing context,
rather than just the signatures, as we do in the main paper (G_p). This is equivalent
since the rule [T-App] below, which looks up the function signature in P, does not
rely on the function body.

P; S; G_s, G |- e : t    (expression typing)

P; S; G_s |- P'    (program typing)

G_s |- t    (type well-formedness)

G_s |- q    (well-formedness of type variables instantiations)

S; G_s, G |-_q p    (well-typed substitutions)

P; S; G_s, G |- H    (heap typing)



** Expression typing    P; S; G_s, G |- e : t
```

```
 G(x) = t    G_s |- t
--------------------------- [T-Var]
 P; S; G_s, G |- x : t


------------------------- [T-Int]
 P; S; G_s, G |- n : int


---------------------------- [T-Unit]
 P; S; G_s, G |- () : unit


------------------------------- [T-Buf]
 P; S; G_s, G |- (b, n) : S(b)


 P; S; G_s, G       |- e1 : buf t1
 P; S; G_s, G       |- e2 : int
 P; S; G_s, G, x:t1 |- e : t
---------------------------------------------------- [T-Readbuf]
 P; S; G_s, G |- let x:t1 = readbuf e1 e2 in e : t


 P; S; G_s, G |- e1 : buf t1
 P; S; G_s, G |- e2 : int
 P; S; G_s, G |- e3 : t1
 P; S; G_s, G |- e  : t
------------------------------------------------------ [T-Writebuf]
 P; S; G_s, G |- let _ = writebuf e1 e2 e3 in e : t

 P; S; G_s, G          |- e1 : t
 P; S; G_s, G, x:buf t |- e2 : t2
------------------------------------------------------ [T-Newbuf]
 P; S; G_s, G |- let x = newbuf n (e1:t) in e2 : t2


 P; S; G_s, G |- e1 : buf t
 P; S; G_s, G |- e2 : int
------------------------------------- [T-Subbuf]
 P; S; G_s, G |- subbuf e1 e2 : buf t


 P; S; G_s, G |- e : t
---------------------------------- [T-Withframe]
 P; S; G_s, G |- withframe e : t


 P; S; G_s, G |- e : t
---------------------------- [T-Pop]
 P; S; G_s, G |- pop e : t
```

```
     P; S; G_s, G |- e   : int
     P; S; G_s, G |- e_i : t
    ------------------------------------------ [T-If]
     P; S; G_s, G |- if e then e1 else e2 : t


     P(d) = fun (x:t1) -> _:t \/ G_s(d) = t1 -> t
     P; S; G_s, G      |- e1 : t1
     P; S; G_s, G, x:t |- e2 : t2
    ------------------------------------------- [T-App]
     P; S; G_s, G |- let x:t = d e1 in e2 : t2


     P; S; G_s, G      |- e1 : t
     P; S; G_s, G, x:t |- e2 : t2
    ------------------------------------------ [T-Let]
     P; S; G_s, G |- let x:t = e1 in e2 : t2


     P; S; G_s, G |- e_i : t_i
    ------------------------------------------------ [T-Rec]
     P; S; G_s, G |- { f_i : e_i } : { f_i : t_i }


     P; S: G_s, G|- e : { f_i : t_i }
    ---------------------------------- [T-Proj]
     P; S; G_s, G |- e.f_i : t_i


** Program typing   P; S; G_s |- P'


    ----------------------- [P-Emp]
     P; S; G_s |- .


     G_s             |- t_i
     P; S; G_s, y:t1 |- e : t2
     P; S; G_s       |- P'
    ------------------------------------------- [P-Fun]
     P; S; G_s |- let d = fun(y:t1) -> e:t2, P'


-------------------------------------
 Notion of equivalence-modulo-secrets
-------------------------------------

 Equivalence of values:
----------------------
```

```
A lax relation typing-wise.

V_int           = { (n, n) }
V_unit          = { ( (), () ) }
V_{ f_i : t_i } = { ({f_i = v_i}, {f_i = v_i'}) | forall i. (v_i, v_i') \in V_(t_i) }
V_(buf t)       = { ( (b, n), (b, n) ) }
V_a             = { (v1, v2) }


Equivalence of heaps:
---------------------

Again, a lax relation typing-wise. Further, S can contain type variables.

S |- H1 =eq= H2 iff

1. dom H1 = dom H2

2. forall (b, n) \in dom H1.
     let buf t = S (b, n)
     forall 0 <= i < n. (H1[(b, n)][i], H2[(b, n)][i]) \in V_t


Gen function for values:
------------------------

Let v1 is closed, v2 is closed, and (v1, v2) \in V_t (t can have type variables)

Gen v1 v2 t = v, p1, p2, G


Gen n n int    = (n, ., ., .)

Gen () () unit = ((), ., ., .)

Gen { f_i = v_i } { f_i = v'_i } { f_i : t_i } = ({f_i = v''_i}, p1, p2, G)
  where forall i. Gen v_i v'_i t_i = (v''_i, p1_i, p2_i, G_i)
  and   p1 = compose_for_all_i p1_i
  and   p2 = compose_for_all_i p2_i
  and   G  = compose_for_all_i G_i

  (compose is a simple append in each case)

Gen (b, n) (b, n) (buf t) = ((b, n), ., ., .)

Gen v1 v2 a = (x, x |-> v1, x |-> v2, x:a)


Gen function for heaps:
-----------------------
```

```
Let H1 is closed, H2 is closed, and S |- H1 =eq= H2

Gen H1 H2 S = (H, p1, p2, G)

s.t. dom H = dom H1 (= dom H2)
     H[(b, n)][i] = v,
     where Gen H1[(b, n)][i] H2[(b, n)][i] t = v, p1_b_n_i, p2_b_n_i, G_b_n_i
     and p1 = compose of all p1_b_n_i and p2 = compose of all p2_b_n_i
     and G = compose of all G_b_n_i

-------------------------------------------------------------------------------------
Remark: Gen does not preserve sharing, but we recover sharing after the substitition.
Further, substitution is just a proof device.
-------------------------------------------------------------------------------------


Assumption on secrets manipulating functions:
---------------------------------------------

If

(1) G_s (f) = t1 -> t2, where G_s |- t1 and G_s |- t2  (f has type t1 -> t2
                                                        in the signature,
                                                        t1 and t2 can have
                                                        type variables)

(2) S |- H1 =eq= H2    (H1 and H2 are closed heaps (see 6, 7 below)
                        and related under S)

(3) P; S[q]; G_s[q] |- v1 : t1[q]    (v1 is a closed value of type t1[q])

(4) P; S[q]; G_s[q] |- v2 : t1[q]    (so is v2)

(5) (v1, v2) \in V_t1    (v1 and v2 are related)

(6) P; S[q]; G_s[q] |- H1    (H1 is well-typed heap, and closed)

(7) P; S[q]; G_s[q] |- H2    (so is H2)

Then

(a) P, P_s |- (H1, (P_s (f)) v1) -m->tr (H1', v1')    (f v1 terminates in m steps)

(b) P, P_s |- (H2, (P_s (f)) v2) -n->tr (H2', v2')    (f v2 terminates in n steps
                                                       with same trace as f v1)

(c) S' |- H1' =eq= H2'    (output heaps are equivalent modulo secrets)

(d) P; S'[q]; G_s[q] |- v1' : t2[q]
```

```
(e) P; S'[q]; G_s[q] |- v2' : t2[q]

(f) (v1', v2') \in V_t2    (v1' and v2' are equivalent modulo secrets)

(g) P; S'[q]; G_s[q] |- H1'

(h) P; S'[q]; G_s[q] |- H2'


----------------------------------------
Remark: a separate lemma can show m = n
----------------------------------------


-------------------------------------
Lemma: Gen-function-lemma-for-values
-------------------------------------

If

(1) G_s |- t

(2) G_s |- q

(3) P; S[q]; G_s |- v1 : t[q]

(4) P; S[q]; G_s |- v2 : t[q]

(5) (v1, v2) \in V_t

Then

(a) Gen v1 v2 t = v, p1, p2, G

(b) P; S; G_s, G |- v : t

(c) S; G_s, G |-_q p1

(d) S; G_s, G |-_q p2


-----------------------------------
Lemma: Gen-function-lemma-for-heaps
-----------------------------------

If

(1) G_s |- q

(2) P; S[q]; G_s |- H1
```

```
(3) P; S[q]; G_s |- H2

(4) S |- H1 =eq= H2

Then

(a) Gen H1 H2 S = (H, p1, p2, G)

(b) P; S; G_s, G |- H

(c) S; G_s, G |-_q p1

(d) S; G_s, G |-_q p2


----------------------------
Substitution-lemma-for-Low*
----------------------------

If

(1) P; S; G_s, G, x:t |- e : t'

(2) P; S; G_s, G |- v : t

Then

(a) P; S; G_s, G |- e[v/x] : t'


-----------------------
Lemma: Value-inversion
-----------------------

(a) If P; S; G_s |- v : int,   then v = n.

(b) If P; S; G_s |- v : buf t, then v = (b, n)


---------------------------
Subject reduction of Low*
---------------------------

If

(1) P; S; G_s |- P    (program is well typed)

(2) P; S; G_s, G |- e : t    (e has type t)

(3) .; S[q]; G_s[q] |-  P_s    (P_s is a well typed implementation
                                of the interface G_s)
```

```
(4) P, P_s |- H, e -->tr H', e'

(5) P; S; G_s |- H

Then, exists S' \superset S s.t.

(a) P; S'; G_s, G |- e': t

(b) P; S'; G_s |- H'


Proof: Low* type system is a simply-typed type system, so the proofs go through
in an unsurprising manner.

Also, the type system ensures subject reduction, but does not ensure progress
since it does not track the buffer bounds.


----------------------------------------
Lemma: Equivalence-modulo-secrets-values
----------------------------------------

If

(1) P; S; G_s, G_f |- v : t    (v is well-typed against the secrets interface)

(2) S; G_s, G_f |-_q p    (p is a well-typed substitution of secret variables under
instantiations of type variables q)

(3) S; G_s, G_f |-_q p1    (p1 is also a well-typed substitution)

Then

(a) (v[p], v[p1]) \in V_t    (the substituted values are related)


----------------------------------------
Lemma: Equivalence-modulo-secrets-heaps
----------------------------------------

(Explanation same as the above lemma)

If

(1) P; S; G_s, G_f |- H

(2) S; G_s, G_f |-_q p    (p is a well-typed substitution of secret variables under
instantiations of type variables q)

(3) S; G_s, G_f |-_q p1    (p1 is also a well-typed substitution)
```

```
Then

(a) S |- H[p] =eq= H[p1]


-------------------------------
Lemma: Substituted-value-typing
-------------------------------

If

(1) P; S; G_s, G_f |- v : t    (v is well-typed)

(2) S; G_s, G_f |-_q p    (substitution p is well-typed
                           with type variables instantiations q)

Then

(a) P; S[q]; G_s[q] |- v[p] : t[q]    (substituted value is well-typed and closed)


-------------------------------
Lemma: Substituted-heap-typing
-------------------------------

If

(1) P; S; G_s, G_f |- H    (H is well-typed)

(2) S; G_s, G_f |-_q p    (substitution p is well-typed
                           with type variables instantiations q)

Then

(a) P; S[q]; G_s[q] |- H[p]    (substituted heap is well-typed and closed)


-------------------------------------
Lemma: Non-secret-value-substitution
-------------------------------------

If

(1) S; G_s, G_f |-_q p

(2) P; S; G_s, G_f |- v[p] : t

Then

(a) If t = int, then v = n.
```

```
(b) If t = buf t, then v = (b, n)


-----------------------------------
Theorem: Secret-independent-traces
-----------------------------------

(Also assume G_s |- q, G_s and q do not vary with program execution)

If

(1) P; S; G_s, G_f |- (H, e) : t    (only secret type variables are free)

(2) S; G_s, G_f |-_q p    (substitution of secret typed variables is well-typed)

(3) S; G_s, G_f |-_q p1    (p1 is some other substitution that is also well-typed)

(4) .; S[q]; G_s[q] |-  P_s    (P_s is a well typed implementation
                                of the interface G_s)

Then either e is a value, or
exists m > 0, n > 0,
exists S' \superseteq S, G_f' \superseteq G_f, H', e', p' \superseteq p,
       p1' \superseteq p1 s.t.

(a) P, P_s |- (H, e)[p] -m->tr (H1, e1)    (it takes m steps, emitting the trace tr)

(b) (H', e')[p'] = (H1, e1)    (the resulting H1 and e1 are some template
                                with some substitution)

(c) P; S'; G_s, G_f' |- (H', e') : t    (H' and e' are well-typed)

(d) S'; G_s, G_f' |-_q p'     (the resulting substitution is well-typed)

(e) P, P_s |- (H, e)[p1] -n->tr (H', e')[p1']    (running on p1 is basically
                                                   same trace and same template
                                                   with different substitution
                                                   for secrets)

(f) S'; G_s, G_f' |-_q p1'     (the second resulting substitution
                                is also well-typed)

-------------------------
Proof: by induction on e
-------------------------

Case e = let x:t1 = e1 in e2
------------------------------

Subcase 1: e1 is not a value:
```

```
------------------------------

Inverting [E-Let] with (1), we get:

(5) P; S; G_s, G_f |- e1 : t1

Applying I.H. on (5), (2), and (3):

(6) P, P_s |- (H, e1)[p] -m->tr (H1, e11)

(7) (H', e1')[p'] = (H1, e11)

(8) P; S'; G_s, G_f' |- (H', e1') : t1

(9) S; G_s, G_f' |-_q p'

(10) P, P_s |- (H, e1)[p1] -n->tr (H', e1')[p1']

(11) S; G_s, G_f' |-_q p1'

Choose m = m, n = n.
Choose S' = S', G_f' = G_f', H' = H', e' = let x = e1' in e2, p' = p', p1' = p1'

And, choose the evaluation context LE = let x = <> in e2

Using [Step], the evaluation context stepping rule, we get (a):

(12) P, P_s |- (H, let x = e1 in e2)[p] -m->tr (H1, let x = e11 in e2[p])

Consider (H', let x = e1' in e2)[p'], since p' \supserseteq p, e2[p'] = e2[p],
hence we get (b).

(c) follows from applying [T-Let] with (8), and weakening lemmas.

(d) follows from (9).

(e) follows similar to deriving (a) above.

(f) follow from (11).


Subcase 2: e1 is a value
-------------------------

(5) e[p] = let x = e1[p] in e2[p]

Using [E-Let]:

(6) P, P_s |- (H[p], let x = e1[p] in e2[p]) --> (H[p], e2[p][e1[p]/x])

Choosing m = 1, H1 = H[p], e1 = e2[p][e1[p]/x], tr = ., we get (a).
```

```
Choosing H' = H, e' = e2[e1/x], p' = p, we get (b).

Inverting [T-Let] with (1), we get:

(7) P; S; G_s, G_f |- e1 : t'

(6) P; S; G_s, G_f, x:t' |- e2 : t

Since e1 is a value, Using Lemma [Substitution-lemma-for-Low*], we get:

(8) P; S; G_s, G_f |- e2[e1/x] : t

Since H' = H, we get (c).

Since p' = p, we get (d) from (2).

(e) follows similar to above evaluation for p with n = 1
(p really did not play any role in the evaluation above).

(f) follows from (3), since p1' = p1.


Case e = let x = f e1 in e2
-----------------------------

Subcase 1: e1 is not a value:
-----------------------------

Inverting [T-App] with (1):

(5) P; S; G_s, G_f |- e1 : t1

Applying I.H. on (5), (2), and (3):

(6) P, P_s |- (H, e1)[p] -m->tr (H1, e11)

(7) (H', e1')[p'] = (H1, e11)

(8) P; S'; G_s, G_f' |- (H', e1') : t1

(9) S; G_s, G_f' |-_q p'

(10) P, P_s |- (H, e1)[p1] -n->tr (H', e1')[p1']

(11) S; G_s, G_f' |-_q p1'

Choose m = m, n = n.
Choose S' = S', G_f' = G_f', H' = H', e' = let x = f e1' in e2, p' = p', p1' = p1'

And, choose the evaluation context LE = let x = f <> in e2
```

```
Using [Step], the evaluation context stepping rule, we get (a):

(12) P, P_s |- (H, let x = f e1 in e2)[p] -m->tr (H1, let x = f e11 in e2[p])

Consider (H', let x = f e1' in e2)[p'], since p' \supseteq p, e2[p'] = e2[p],
hence we get (b).

(c) follows from applying [T-App] with (8), and weakening lemmas.

(d) follows from (9).

(e) follows similar to deriving (a) above.

(f) follow from (11).


Subcase 2: e1 is a value
------------------------

Subsubcase 1: f \in P:
----------------------

Inverting [T-App] with (1):

(5) P(f) = fun (x:t1) -> e:t

(6) P; S; G_s, G_f |- e1 : t1

(7) P; S; G_s, G_f, x:t |- e2 : t2

Using [App], we get:

(8) P, P_s |- (H, let x = f e1 in e2)[p] -0->. (H[p], let x = e[e1[p]/x] in e2[p])

which gives us (a).

Since e1 has no free secret variables,

(9) let x = e[e1[p]/x] in e2[p] = (let x = e[e1/x] in e2)[p]

Choose H' = H, e' = let x = e[e1/x] in e2, and p' = p to get (b).

Choose S' = S, G_f' = G_f

Using substitution lemma with weakening:

(10) P; S; G_s, G_f |- e[e1/x] : t (recall e1 is a value)

Applying [E-Let] with (10) and (7):
```

```
(11) P; S; G_s, G_f |- let x = e[e1/x] in e2 : t2

And since H' = H, we get (c).

(d) follows since G_f' = G_f and p' = p.

(e) follows since m = n = 0, and tr = . .

(f) follows since p1' = p1.


Subsubcase 2: f:t1 -> t2 \in G_f
---------------------------------

Inverting (1):

(5) P; S; G_s, G_f |- H

(6) P; S; G_s, G_f |- e1 : t1

(7) P; S; G_s, G_f, x:t2 |- e2 : t

Using Lemma [Substituted-value-typing] with (2) and (6):

(8) P; S[q]; G_s[q] |- e1[p] : t1[q]

Using Lemma [Substituted-value-typing] with (3) and (6):

(9) P; S[q]; G_s[q] |- e1[p1] : t1[q]

Using Lemma [Substituted-heap-typing] with (2) and (5):

(10) P; S[q]; G_s[q] |- H[p]

Using Lemma [Substituted-heap-typing] with (3) and (5):

(11) P; S[q]; G_s[q] |- H[p1]

Using Lemma [Equivalence-modulo-secrets-heaps] with (5), (2), and (3):

(12) S |- H[p] =eq= H[p1]

Using Lemma [Equivalence-modulo-secrets-values] with (6), (2), and (3):

(13) (e1[p], e2[p]) \in V_t1

Using assumption about secret manipuating functions with
H1 = H[p], H2 = H[p1], v1 = e1[p], v2 = e1[p1]
and (12), (8), (9), (13), (10), and (11):

(14) P, P_s |- (H[p], f e1[p]) -m->tr (H1', v1')
```

```
(15) P, P_s |- (H[p1], f e1[p1]) -n->tr (H2', v2')

(16) S' |- H1' =eq= H2'

(17) P; S'[q]; G_s |- v1' : t2[q]

(18) P; S'[q]; G_s |- v2' : t2[q]

(19) (v1', v2') \in V_t2

(20) P; S'[q]; G_s |- H1'

(21) P; S'[q]; G_s |- H2'

Using Lemma [Gen-function-lemma-for-values] with (assume), (17), (18), (19):

(22) Gen v1' v2' t2 = v, p', p1', G

(22a) P; S; G_s, G |- v : t2

(23) S; G_s, G |-_q p'

(24) S; G_s, G |-_q p1'

Using Lemma [Gen-function-lemma-for-heaps] with (assume), (20), (21), (16):

(25) Gen H1 H2 S' = H', p_h, p_h1, G_h

(25a) P; S; G_s, G_h |- H'

(26) S; G_s, G_h |-_q p_h

(27) S; G_s, G_h |-_q p_h1


For theorem consequences, choose:

m = m, n = n, S' = S', G_f' = G_f \union G \union G_h, H' = H',
e' = let x = v in e2, p' = p \union p' \union p_h, p1' = p1 \union p1' \union p_h1

(a), (b), and (e) follow from [Context] rule of semantics and (22) and (25) above.

(d) and (f) follow from (23), (24), (26), and (27).

(c) follows from (22a) and (7) with [T-Let] and heap typing from (25a).


Case e = let _ = writebuf e1 e2 e3 in e
-----------------------------------------
```

```
Subcase 1: One of e1, e2, e3 is not a value: These are context cases proven similar
to the context cases of let and application above.

Subcase 2: e1, e2, e3 are values
---------------------------------

Inverting (1):

(4) P; S; G_s, G_f |- H

(5) P; S; G_s, G_f |- e1 : buf t1

(6) P; S; G_s, G_f |- e2 : int

(7) P; S; G_s, G_f |- e3 : t1

Using Lemma [Substituted-value-typing] with (5) and (2):

(8) P; S[q]; G_s[q] |- e1[p] : buf t1[q]

Using Lemma [Substituted-value-typing] with (6) and (2):

(9) P; S[q]; G_s[q] |- e2[p] : int

Using Lemma [Substituted-value-typing] with (7) and (2):

(10) P; S[q]; G_s[q] |- e3[p] : t1[q]

Using Lemma [Value-inversion], we get:

(11) e1[p] = (b, n), actually e1 = (b, n)

(12) e2[p] = n', actually e2 = n'    (Lemma [Non-secret-value-substitution])

Using [E-Write]:

(13) P, P_s |- (H, let _ = write e1 e2 e3 in e)[p] ->write(b, n + n')
               (H[p][(b, n + n') |-> e3[p]], e[p])

(the step can get stuck if the index is out of bound or the frame has been popped,
in which case we just step to stuck value)

For (a) and (b) choose m = 1, tr = write(b, n + n'), H' = H[(b, n + n') |-> e3],
e' = e, p' = p, S' = S, G_f' = G_f

(c) simply follows from the inversion of typing judgment (1),
and new heap is also well-typed.

(d) follows from (2).

We can do similar analysis with p1 as above and get that:
```

```
(13) P, P_s |- (H, let _ = write e1 e2 e3 in e)[p1] ->write(b, n + n')
               (H[p1][(b, n + n') |-> e3[p1]], e[p1])

Note that e1 and e2 are independent of p or p1, as noted in (11) and (12) above.

So, choose n = 1 after which (e) follows. (f) follows from (3).


Case e = let x = newbuf n e1 in e2
-----------------------------------

Subcase 1: Either e1 or e2 is not a value: These are context cases proven
similar to the let and application cases above.

Subcase 2: Both e1 and e2 are values
-------------------------------------

Inverting (1):

(4) P; S; G_s, G_f |- H

(5) P; S; G_s, G_f |- e1 : t1

(6) P; S; G_s, G_f, x:buf t |- e2 : t

Using [E-NewBuf]:

(7) P, P_s |- (H[p], let x = n e1[p] in e2[p]) ->write (b, 0) ... (b, n - 1)
              (H[p][b |-> e1[p]^n], e2[p][(b, 0)/x])

Choose m = 1, n = 1, S' = S[b |-> buf t1], G_f' = G_f, H' = H[b |-> e1^n],
e' = e2[(b, 0)/x], p' = p, p1' = p1, tr = write (b, 0) ... (b, n - 1)

(a) and (b) follow from (7).

(c) follows since H' is well-typed in S', and e' is well-typed after applying
Lemma [Substitution-lemma-for-Low*].

(d) follows from (2).

(e) follows with p1' = p1, and the trace is independent of the substitution p and p'

(f) follows from (3).


Case e = let x = readbuf e1 e2 in e
-----------------------------------

Subcase 1: One of e1 or e2 is not a value: These are context cases proven
similar to the let and application cases above.
```

```
Subcase 2: Both e1 and e2 are values
------------------------------------

Inverting (1):

(4) P; S; G_s, G_f |- e1 : buf t1

(5) P; S; G_s, G_f |- e2 : int

(6) P; S; G_s, G_f, x:t1 |- e : t

Using Lemma [Substituted-value-typing] with (4) and (2), (5) and (2), and
Lemma [Substituted-heap-typing] with (6) and (2):

(7) P; S[q]; G_s[q] |- e1[p] : buf t1[q]

(8) P; S[q]; G_s[q] |- e2[p] : int

(9) P; S[q]; G_s[q] |- H[p]

Using Lemma [Non-secret-value-substitution] with (7) and (8):

(10) e1 = (b, n)

(11) e2 = n'

Using [E-Read]:

(12) P, P_s |- (H, let x = readbuf e1 e2 in e)[p] ->read (b, n + n')
                (H[p], e[p][H[p](b, n + n')/x])

(the step can get stuck if the index is out of bound or the frame has been popped,
in which case we just step to stuck value)

Choose m = 1, n = 1.
Choose S' = S, G_f' = G_f, H' = H, e' = e[H(b, n + n')/x], p' = p, p1' = p1

(a) and (b) follow from (12).

(c) follows since H' = H, and e' is well-typed
    after applying Lemma [Substitution-lemma-for-Low*].

(d) follows from (2).

(e) follows similar to (12).

(f) follows from (3).
```