Short Paper: Towards Information Flow Reasoning about Real-World C Code

Samuel Gruetter*
Massachusetts Institute of Technology
gruetter@mit.edu

Toby Murray
University of Melbourne
toby.murray@unimelb.edu.au

ABSTRACT

Strangely, despite much recent success proving information flow control (IFC) security for C programs, little work has investigated how to prove IFC security directly against C code, as opposed to over an abstract specification. We consider what a suitable IFC logic for C might look like, and propose a suitable *continuation-passing* style IFC security definition for C code. We discuss our ongoing work implementing these ideas in the context of an existing full-featured, sound program verification framework for C, the Verified Software Toolchain, supported by the verified C complier CompCert.

1 INTRODUCTION

Despite its age, C remains one of the most popular programming languages ever created. Modern languages like Rust aside, C continues to be indispensable for domains such as operating system kernels, device drivers, and embedded/real-time systems. It is also the de facto *lingua franca* of programming languages, in which the foreign function glue of almost all higher-level languages is written.

C has also played host to some of history's deepest software verification efforts. For instance the seL4 [23] microkernel's proof of correctness down to its ARM assembly [34] exploits the relatively close semantic gap from its C source [35, 36] to its gcc-produced binary. On the other hand, the CertiKOS [20] kernel's assembly-level verification leverages the CompCert [24] verified C compiler, as does much other recent work [1, 9, 18].

At the same time, perhaps nowhere else has the promise of software verification found more resonance than via the dream of verified security [26]. In this category, *verified information flow security* [30] has remained under constant study for the past 40 years, and has recently delivered a number of artifacts with formally verified information flow guarantees, including kernels like seL4 [28] and CertiKOS [14], but also conference management systems [22] and social network platforms [6].

It is perhaps curious therefore that there has been relatively little study of logics for proving general information flow control (IFC) theorems of C code. Indeed, while both are implemented in C, seL4 and CertiKOS each avoided such a logic by proving information flow security over higher level, abstract specifications. The resulting security theorems were then transferred to the actual implementations using relatively expensive proofs of refinement.[1]

In this paper, we consider the problem of proving IFC directly on C code. Such reasoning is required, for instance, when the IFC property is tightly coupled to the primitive memory layout, the way the program manipulates pointers or to how it performs raw memory accesses, as in the case of device drivers, embedded Multi-Level Secure (MLS) devices [17] and cross domain appliances amongst others. In doing so, we answer two basic questions:

- What would an IFC logic for C look like?
- How might we phrase a formal IFC definition for C?

Naturally, any IFC definition and logic for C should be phrased over a trustworthy C semantics that incorporates as many of C's language features as possible. Ideally, that semantics should be implemented by a trustworthy compiler. Moreover, the logic should be built to enable the re-use of existing logics and machinery for proving the functional correctness of C code. This last point is important: the security of interesting code that controls information flows often rests on the code's functional correctness. seL4 is a large-scale example, whose information flow security proofs made use of host invariants already proved in a Hoare logic [13].

Fig. 1 provides a minimal example, whose security is inherently tied to its functional correctness (see Sec. 2.1). This code fragment is indicative of MLS input processing code for a cross-domain system [7]. It processes a list, `in`, of data packets, each of which carries a boolean label `isSecret` indicating the sensitivity of the data it contains. The `unzip()` function takes the input list apart, prepending all the secret packages onto the output list `high` and the non-secret packages onto the output list `low`, respectively.

Sec. 2 provides an overview of the ingredients we argue are required to specify and reason about such code. Firstly (Sec. 2.1), an IFC logic here would require a means to reason about pointers and heap accesses, and in particular to reason *locally* about memory-updates via such pointers to facilitate compositional reasoning. Secondly, note that the classification of the data in the `payload` and `size` fields of each data packet is dictated by the packet's `isSecret` field. Thus (Sec. 2.2) any logic for reasoning about this kind of code must support *value-dependent classification* [25, 27, 37].

What does it mean for a C program to satisfy IFC security? A formal definition of IFC security needs to account for exceptional control-flow, and so be able to handle the effects of C statements like `break` and `continue` and early exit via `return`. A natural way to phrase ordinary program correctness (e.g. the validity of Hoare triples) for such is to adopt a *continuation passing style* definition [2] based on small-step operational semantics. Sec. 3 presents the first continuation-passing style IFC definition.

*The work was carried out while the author was visiting the University of Melbourne.

[1] For seL4, proving refinement was ∼5 times more expensive than confidentiality [23].

PLAS'17, October 30, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-5099-0/17/10…$15.00
https://doi.org/10.1145/3139337.3139345

```
typedef struct node {
  bool isSecret;
  unsigned int size;
  void * payload;
  struct node * next;
} node;

void unzip(node * in, node ** high, node ** low) {
  while (in) {
    node * next = in->next;
    if (in->isSecret) {
      in->next = *high;
      *high = in;
    } else {
      in->next = *low;
      *low = in;
    }
    in = next;
  }
}
```

Figure 1: A hypothetical fragment of packet processing code.

Our continuation passing style IFC definition is inspired by ideas from the Verified Software Toolchain [4] (VST). The VST provides a sound program logic for proving functional correctness of C code, built on top of CompCert's C semantics. Statements proved using VST hold for the compiled code emitted by CompCert by virtue of VST's soundness theorems and CompCert's correctness theorems, all proved in Coq. VST also includes considerable automation for easing proofs about C programs [18]. In Sec. 4, we explain our ongoing work formalising the ideas of Sec. 2 atop the VST. By situating our work on VST, we hope to produce the first sound IFC logic for C code, backed by a verified C compiler.

2 LOGICAL INGREDIENTS

2.1 Separation Logic

Before we can talk about whether or not the code in Fig. 1 is secure, we must first reason about its functional correctness. For instance, if the pointers `high` and `low` are invalid this code's behaviour could be undefined. But more than knowing whether a pointer is valid, when reasoning about a statement like `*high = in;` we need to know that the two pointers `high` and `low` don't alias, otherwise this statement would inadvertently switch the `low` pointer (meant to point to data of low sensitivity) to point to a list of high sensitivity secret packets. More broadly, the security of this code is inherently tied to its correct functioning, and we can't talk about its security in the absence of its functional correctness.

Separation logic [32] has become the dominant program logic for reasoning about the correctness of programs with pointers, and can be viewed as an extension of Hoare logic [21]. For some program or statement S, precondition predicate P and postcondition predicate Q, the Hoare triple $\{P\}\,S\,\{Q\}$ states that if S is executed

```
void write_labeled_val(int v, bool b,
                       int* highptr, int* lowptr) {
  if (b)
    *highptr = *highptr | v;
  else
    *lowptr = *lowptr | v;
}
```

Figure 2: A simpler example.

from a state satisfying P, if it terminates the resulting state will satisfy Q. Separation logic extends this by firstly requiring that S is not allowed to reach an error state or get stuck (e.g. by dereferencing an invalid pointer) during its execution, and by extending the language of the predicates P and Q to ease reasoning about heap-manipulating programs especially with regards to aliasing.

If p denotes an address and v a value, the primitive predicate $p \mapsto v$ states that the value v lies at location p in the *heap* (the addressable part of memory). The compound separation logic predicate $P_1 \ast P_2$ denotes that predicates P_1 and P_2 both hold, and additionally that the parts of the heap to which P_1 and P_2 refer respectively do not overlap. Thus the compound predicate $p \mapsto v \ast q \mapsto w$ states not only that v and w live at p and q respectively, but also that p and q do not alias.

Consider the similar but simpler code in Fig. 2. Here the local variable `b` dictates the classification of variable `v`. `highptr` and `lowptr` point to respectively `Hi` and `Lo` integers. Its correct functioning relies on `highptr` and `lowptr` being valid pointers that do not alias. We can express this precondition as follows, namely that there exist some values h and l for which:

$$\texttt{highptr} \mapsto h \ast \texttt{lowptr} \mapsto l$$

Here h and l are logical (i.e. meta) variables that represent the values at the heap locations `highptr` and `lowptr` respectively.

We can describe the functionality of Fig. 2 (i.e. the results of calling this function) by writing a suitable separation logic Hoare triple. The postcondition in any such triple needs to talk about the final values in the heap locations `highptr` and `lowptr` and relate those to the *initial* values of the variables `b` and `v`. The standard approach is to use logical variables to capture the values of these program variables in the precondition, which can then appear in the postcondition. Doing so yields the following separation logic Hoare triple for this function, where we explicitly quantify over the logical variables h, l, b and v (since they may take on any value) and write … to abbreviate the function's body. We also abbreviate the conjunction of assertions $P_1, \ldots, P_n$ that each talk only about local variables but not about the heap as $[P_1, \ldots, P_n]$.

$$\begin{array}{l} \forall h\ l\ b\ v. \\ \qquad \{\texttt{highptr} \mapsto h \ast \texttt{lowptr} \mapsto l \ast [\texttt{b} = b, \texttt{v} = v]\} \\ \qquad\qquad\qquad \ldots \\ \qquad \{\texttt{highptr} \mapsto (b\ ?\ h|v : h) \ast \texttt{lowptr} \mapsto (b\ ?\ l : l|v)\} \end{array} \tag{1}$$

Notice how the precondition has the logical variables capture the initial values of the program variables and heap locations, and then the postcondition refers to those logical variables to talk about how the heap has been updated. For a boolean b and expressions e

and e', we write $b ~?~ e : e'$ as shorthand for the ternary if-expression that evaluates to e when b is true, and to b' otherwise.

Separation logic [32] has rules similar to those of Hoare logic, but we elide discussing them here in the interest of brevity.

2.2 Value-Dependent Classification

Moving on from its functional correctness, but staying with the simpler example of Fig. 2, we now consider how to specify security. We will work at the level of intuition for the moment and then later show how to realise these intuitions on top of VST's existing separation logic for C.

We extend the $\mapsto$ notation of separation logic to carry *ghost* information asserting the sensitivity of data in the heap.[2] We write $p \overset{l}{\mapsto} v$ to denote that v resides at the location denoted by p, and that v's sensitivity is at the security level given by expression l (which, like all others, may mention logical variables). As elsewhere in the literature, security labels are drawn from a lattice, where $l \sqsubseteq l'$ means that label l' denotes higher sensitivity than label l, and we restrict our attention to the two-point lattice $\{\mathsf{Hi}, \mathsf{Lo}\}$ in which $\mathsf{Lo} \sqsubseteq \mathsf{Hi}$ but $\mathsf{Hi} \not\sqsubseteq \mathsf{Lo}$. For a local variable x we write $x :: l$ to denote that x holds data whose sensitivity is at the level denoted by l. The absence of an assertion $x :: l$ implies that x's level is Hi.

Given these ingredients, we can give the function in Fig. 2 a *security-aware* [15] specification as follows.

$$\left\{ \begin{array}{c} \texttt{highptr} \overset{\mathsf{Hi}}{\longmapsto} h \, * \, \texttt{lowptr} \overset{\mathsf{Lo}}{\longmapsto} l \, * \\ [\mathsf{b} = b, \mathsf{v} = v, \mathsf{b} :: \mathsf{Lo}, \mathsf{v} :: (b ~?~ \mathsf{Hi} : \mathsf{Lo})] \end{array} \right\} \\ \cdots \\ \{\texttt{highptr} \overset{\mathsf{Hi}}{\longmapsto} (b ~?~ h|v : h) \, * \, \texttt{lowptr} \overset{\mathsf{Lo}}{\longmapsto} (b ~?~ l : l|v)\} \tag{2}$$

Note that the precondition specifies the value-dependent classification of variable v, in terms of the value b of variable b.

2.3 Discussion

The two ingredients we argue for above, namely separation logic reasoning with value-dependent classification, were fused together in prior work of Costanzo and Shao [15] in the context of a toy imperative language. The ideas we sketched in Sec. 2.2 are very reminiscent of their logic. Our purpose in introducing them is not to argue for their novelty but instead that they are the right combination for performing IFC reasoning over C code.

Such reasoning, and any such logic, necessarily rests on a precise definition of what statements like that in Formula 2 mean, with respect to the semantics of the underlying programming language. In the following section, we explain why a standard definition of IFC security is ill suited to C's semantics, and so introduce the first *continuation-passing* style definition of IFC security instead.

3 THE IFC STATEMENT

This section presents our continuation-passing style IFC definition, which ultimately gives the precise meaning to statements like that of Formula 2. But first, we will present a more natural, "direct" style definition and discuss its two flaws, to motivate the shape of the IFC definition we then propose.

[2] Concurrent separation logics [31] use ghost information to track heap permissions.

In Sec. 2, we introduced *security aware* assertions of the form $p \overset{l}{\mapsto} v$ (v lies at the heap location denoted by p and v's sensitivity is l), and $\mathsf{b} :: l$ (stack variable b holds l-sensitivity data). Intuitively, the former combines both an ordinary separation logic assertion $p \mapsto v$ and a sensitivity assertion about the (value at the) heap location denoted by p; while the latter is a sensitivity assertion about a stack variable b. In fact, as we explain later in Sec. 4.1, both of these kinds of assertions are simply syntactic sugar for special cases of *security-aware specification triples* of the form (P, N, A). Such a triple should be thought of as the primitive, security-aware counterpart to a Hoare separation logic pre- or post-condition. Here P is a plain separation logic assertion, N is a function from *nonaddressable* stack variable names to sensitivity labels, and A is a function from *addressable* heap locations to sensitivity labels. P tracks what is true during a program's execution, while N and A track respectively the sensitivity of stack and heap.

We model program states σ as triples $\langle e, k, m\rangle$ of a variable environment e, a continuation stack k which is simply a list of commands to be executed, and a heap memory m. We write $\sigma_1 \to \sigma_2$ for the small-step reduction relation, and we write $\sigma_1 \to^* \sigma_2$ for its transitive closure, and $\sigma_1 \to_n \sigma_2$ to say that after n steps, state σ_1 transitions to state σ_2. Moreover, we define execution until final state, written $\sigma_1 \Downarrow \sigma_2$, as $\sigma_1 \to^* \sigma_2$ where the command to be executed in σ_2 is the empty command, which means that execution is done (and hasn't got stuck along the way).

3.1 Semantics of IFC judgement: First attempt

Definition 3.1 (Simple low-equivalence). Two states $\langle e, k, m\rangle$ and $\langle e', k', m'\rangle$ are called low-equivalent with respect to the stack classification function N and the heap classification function A if for all stack locations ℓ for which $N~\ell = \mathsf{Lo}$, $e~\ell = e'~\ell$ and for all heap locations ℓ for which $A~\ell = \mathsf{Lo}$, $m~\ell = m'~\ell$.

Definition 3.2 (Meaning of IFC judgement, first attempt). The meaning of $\{P_1, N_1, A_1\}~c~\{P_2, N_2, A_2\}$ is: (1) The Hoare judgement $\{P_1\}~c~\{P_2\}$ holds and (2) for all σ_1, σ_1', σ_2, σ_2', if: $P_1~\sigma_1$ and $P_1~\sigma_1'$ hold and in both σ_1 and σ_1' the command to be executed is c, and σ_1 is low-equivalent to σ_1' w.r.t N_1 and A_1, and $\sigma_1 \Downarrow \sigma_2$ and $\sigma_1' \Downarrow \sigma_2'$, then: σ_2 is low-equivalent to σ_2' w.r.t N_2 and A_2.

Proving this statement for a particular program c would then prove (termination-insensitive) information flow security for that program in the sense that Hi data does not influence the values of Lo data, because if we vary the values of Hi data between σ_1 and σ_1', we cannot cause changes in Lo values between σ_2 and σ_2'.

3.2 Problems with the first attempt

This direct style definition suffers two problems. First, it doesn't admit quantifying over logical variables to connect values of the precondition with values of the postcondition, as we did in Formula 1. Consider the following example (presented in the notation from Sec. 2, for ease of exposition):

$$\forall x.~\{[\mathsf{sec} = x, \mathsf{sec} :: \mathsf{Hi}, \mathsf{pub} :: \mathsf{Lo}]\}$$

```
pub = sec;
```

$$\{[\mathsf{sec} = x, \mathsf{pub} = x, \mathsf{sec} :: \mathsf{Hi}, \mathsf{pub} :: \mathsf{Lo}]\}$$

Here x is a logical variable that captures the initial value of `sec`. While this program is clearly insecure, the security statement is in fact provable wrt Definition 3.2: to prove the universal quantification, we assume x to be an arbitrary, but *fixed* value, so the `Hi` variable `sec` cannot have different values in the states σ_1 and σ_1' from Definition 3.2, and therefore, `pub` will always have the same value in σ_2 and σ_2', which makes the statement true.

So we see that the way we combined universal quantification with our definition of information flow security is flawed, because it results in a vacuous information flow security statement.

Therefore, we have to give control over the quantification to the IFC judgement, rather than adding it on the outside. We achieve this later in Sec. 3.4 by parameterising all pre- and postconditions by a logical variable x, which can be any user-specified tuple type. The single variable x will contain a tuple, and so might be thought of as a tuple of logical variables which the security-aware assertion (P, N, A) can then refer to.

Since P links x to values on the stack and heap, allowing the classification functions N and A to depend on x allows for the same kind of value-dependent classification that we argued for in Sec. 2.2.

Besides this problem of quantification and logical variables, the second problem is that Definition 3.2 does not deal with premature exits such as **`break`** and **`continue`**. While it might be possible to deal with them and retain a direct style definition, by appropriately enriching the notion of a final execution state [33], VST's experience shows that adopting "continuation-passing" style definitions [2, 3] can be simpler without sacrificing expressivity. Since the shape of our continuation passing style IFC definition is inspired by VST's, we will explain that one first.

3.3 VST's continuation-passing style definition

VST defines validity of Hoare triples $\{P\}\, c\, \{Q\}$ with the following series of definitions:

Definition 3.3 (Immediately safe). State $\sigma = \langle e, k, m\rangle$ is immediately safe if $k = \mathsf{nil}$ or $\sigma \to \sigma_2$ for some σ_2 (i.e. execution isn't stuck).

Definition 3.4 (Safe). A state σ is safe if for all σ_2, if $\sigma \to^* \sigma_2$, then σ_2 is immediately safe.

Definition 3.5 (Guard). Predicate P guards continuation stack k, written $\{P\}\, k$, if for all e, m: $P\ \langle e, k, m\rangle$ implies $\langle e, k, m\rangle$ is safe.

To support **`break`**, **`continue`** and **`return`** before the end of the function body, VST's postconditions are not just plain assertions like the preconditions, but functions taking an `exitkind` and an optional value and returning an assertion, where the type `exitkind` is an enum with the four values $\mathsf{EK}_{\mathsf{nrm}}$, $\mathsf{EK}_{\mathsf{brk}}$, $\mathsf{EK}_{\mathsf{cont}}$, and $\mathsf{EK}_{\mathsf{ret}}$ (to denote normal code execution until the end of the code block, or premature exit via **`break`**, **`continue`** or **`return`**, respectively), and the optional value is used for the return value if there is one.

Definition 3.6 (Return guard). Postcondition R guards the continuation stack k, written $\{R\}\, k$, if for all exitkinds ek and values v, we have $\{R\ ek\ v\}\, k$.

For a command c and continuation stack (i.e. list of commands) k, $c :: k$ is the continuation stack whose head is c and whose tail is k.

Definition 3.7 (Meaning of Hoare judgement). The meaning of $\{P\}\, c\, \{R\}$ is: for all continuation stacks k, $\{R\}\, k$ implies $\{P\}\, (c :: k)$.

It might look like the above definition only talks about safety in the sense of absence of crashes but, in fact, it does guarantee functional correctness, because k could be any program which tests whether R holds, and crashes if it does not hold. Then, the above definition guarantees that after running c, R must hold.

3.4 Definition of the IFC judgement

We will now use this continuation-passing style for a definition of information flow security.

Definition 3.8 (Equivalent continuations). Two continuations (i.e. commands) c_1 and c_2 are called equivalent, written $c_1 \equiv_{\mathsf{cont}} c_2$, if they are equal or they are both a function body to be resumed after a return, of the same function, but with potentially different variable environments to be restored.

Definition 3.9 (Head-equivalent states). Two states $\sigma = \langle e, k, m\rangle$ and $\sigma' = \langle e', k', m'\rangle$ are called head-equivalent, written $\sigma \equiv_{\mathsf{head}} \sigma'$ if either both k and k' are the empty stack, or both are non-empty and their head (top) continuations are equivalent.

Definition 3.10 (Matching States). Two states σ_1 and σ_1' are called *matching*, written $\sigma_1 \equiv_{\mathsf{match}} \sigma_1'$, if for all n, σ_2, σ_2', if $\sigma_1 \to_n \sigma_2$ and $\sigma_1' \to_n \sigma_2'$, then $\sigma_2 \equiv_{\mathsf{head}} \sigma_2'$.

Matching can be thought of as some kind of low-equivalence, with the advantage that it does not need any classification functions, which are typically only available for the program state right before and right after the command in question, but not for intermediate states or future states.

In fact, low-equivalence between two memories for a bit stored at heap location ℓ can be encoded as follows using our notion of matching. Let k be a continuation stack whose program loads the bit at location ℓ and then branches on the value of that bit, executing some command c_0 if it is 0, or some different command c_1 (such that $c_0 \equiv_{\mathsf{cont}} c_1$ does *not* hold) if it is 1. Now if we have two variable environments e_1 and e_1', and two memories m_1 and m_1', and we want to say that after running some given command c, the bit at ℓ must be the same in both memories, we can express this as $\langle e_1, c :: k, m_1\rangle \equiv_{\mathsf{match}} \langle e_1', c :: k, m_1'\rangle$. If c terminates, it does so in a certain number of steps n, and after $n + 1$ steps, execution will be in k and branch on the value stored at ℓ, putting c_0 or c_1 on top of the continuation stack depending on the bit stored at ℓ, and since match requires the two continuation stack heads to be equivalent, it ensures that the values stored at ℓ are the same.

That is, we can append a "test continuation" k to the command c in question, which makes the matching proposition false if any equality we desire to hold does not hold.

We can use this intuition to define an IFC guard in a similar way as VST's guard. Such a guard now takes a logical variable x as an argument, as explained earlier in Sec. 3.2, as do all P, N, A, and quantifies over x *twice* (once for each execution) to avoid the aforementioned problems of vacuous security specifications.

Definition 3.11 (IFC guard). We write $\{\lambda x.\, (P, N, A)\}\ k\ k'$ if for all x, x', e, e', m, m', if $P\ x\ \langle e, k, m\rangle$ and $P\ x'\ \langle e', k', m'\rangle$ hold, and e is low-equivalent to e' w.r.t. $N\ x$ and $N\ x'$, and m is low-equivalent to m' w.r.t. $A\ x$ and $A\ x'$, then $\langle e, k, m\rangle \equiv_{\mathsf{match}} \langle e', k', m'\rangle$.

Return guards are defined straightforwardly: an IFC return guard $\lambda x\ k\ ek\ v.\ (P, N, A)$ takes also a continuation stack k, exitkind ek and possible return-value v and then yields a (P, N, A) triple.

Definition 3.12 (IFC return guard). For an IFC return guard $\mathcal{R}$, we write $\{\mathcal{R}\}\ k\ k'$ if for all ek and v, we have $\{\lambda x.\ \mathcal{R}\ x\ ek\ v\}\ k\ k'$.

Definition 3.13 (Meaning of IFC judgement, final version). For $\mathcal{P} = \lambda x.\ (P_1, N_1, A_1)$ and $\mathcal{R} = \lambda x\ k\ ek\ v.\ (P_2, N_2, A_2)$, the meaning of $\{\mathcal{P}\}\ c\ \{\mathcal{R}\}$ is: for all x, the VST judgement $\{P_1\ x\}\ c\ \{P_2\ x\}$ holds and for all k and k', $\{\mathcal{R}\}\ k\ k'$ implies $\{\mathcal{P}\}\ (c :: k)\ (c :: k')$.

3.5 Discussion

Note that this IFC definition imposes the restriction that branching on Hi data is not allowed, so that different continuation stack heads can be used as an indicator that values which are supposed to be equal are not. This definition is also in some sense *timing-sensitive* (unlike Definition 3.2 which was termination- and timing-insensitive), since our notion of matching compares two executions after the same number n of steps. While there is a growing body of code that is written purposefully to avoid branching on Hi data [10–12, 16], we could allow programs that branch on Hi data by using a different matching indicator, e.g. by asserting the two commands produce the same public output or have the same termination behaviour. We conjecture that doing so could also allow weakening the definition to become timing- or termination-insensitive, but leave this investigation for future work.

4 INSTANTIATION IN VST

We are currently implementing these ideas atop the VST. With the continuation passing IFC definition formalised, we have devised a set of IFC rules for the major syntactic constructs of C.[3] In the interests of brevity we defer to our working draft paper [19], and here just present one representative rule, for memory loads.

$$\textsc{ifc-load}\ \frac{\begin{array}{c} P \vdash (\llbracket \&e \rrbracket = p\ \wedge\ (p \mapsto v) * \top) \\ P \vdash (\mathsf{clsf\_expr}\ N\ e = \ell_1\ \wedge\ A\ p = \ell_2) \end{array}}{\left\{\begin{pmatrix} P \\ N \\ A \end{pmatrix}\right\}\ \mathtt{id}{=}e\ \left\{\mathsf{nret}\ \begin{pmatrix} \llbracket \mathtt{id} \rrbracket = v \wedge \exists v'.\ P[v'/\mathtt{id}] \\ N[\mathtt{id} := \ell_1 \sqcup \ell_2] \\ A \end{pmatrix}\right\}}$$

The statement $\mathtt{id}{=}e$ loads the value at heap address denoted by expression e into the stack variable $\mathtt{id}$. Expression e can refer only to stack variables, and might be e.g. an array access `a[i]`. $\vdash$ denotes entailment between separation logic predicates while nret expresses that the command terminates normally. The rule requires that the expression e denotes an address p at which lies some value v, whose sensitivity (given by A) is ℓ_2, while the sensitivity (denoted $\mathsf{clsf\_expr}\ N\ e$) of the expression e is ℓ_1. Then after this memory load, $\mathtt{id}$ has value v and contains data whose sensitivity is the least upper bound of ℓ_1 and ℓ_2. Observe how this rule captures traditional separation logic elements (tracked via P) while also tracking security information (via N and A).

Each of our IFC rules can be proved sound with respect to the continuation passing security definition, leveraging VST's existing machinery for reasoning about separation logic assertions. These proofs are currently in progress.

[3]Specifically of Clight, the formal front-end language of CompCert and into which C programs are translated for verification in VST.

4.1 Implementation of annotated assertions

Sec. 2.2 introduced security-aware separation logic assertions, like $p \overset{l}{\mapsto} v$, while our IFC definition of Sec. 3 and the primitive rules like $\textsc{ifc-load}$ are phrased over triples (P, N, A). We now close the loop and show how to build the former in terms of the latter. Doing so will also allow us to derive more friendly IFC rules that talk in terms of the security-aware separation logic assertions.

Each security-aware assertion encodes a triple (P, N, A). Such triples are combined using a "lifted" $*$ operator, defined as follows.

$$(P_1, N_1, A_1)\ *\ (P_2, N_2, A_2)\ \equiv\ (P_1\ *\ P_2, N_1 \sqcap N_2, A_1 \sqcap A_2)$$

Here $f \sqcap g$ on functions f and g is the function $\lambda x.\ f\ x\ \sqcap g\ x$. Stack- and heap-classification functions, N and A respectively, are combined together by taking their greatest lower bound. This is to allow a stack (resp. heap) classification function to talk about only a sub-part of the stack (resp. heap) by returning Hi for everywhere outside that part. Note that separation logic ensures that the parts of the heap which P_1 and P_2 refer to do not overlap, so $\sqcap$ will always have a default Hi on one side and an actual label on the other side, but never two "competing" labels which would have to be combined with $\sqcup$ to be sound.

Thus a stack variable classification assertion $\mathsf{b} :: l$ is encoded as

$$\mathsf{b} :: l\ \equiv\ (\mathsf{emp}, (\lambda id.\ \textbf{if}\ id = \mathsf{b}\ \textbf{then}\ l\ \textbf{else}\ \mathsf{Hi}), \top)$$

where emp is the separation logic predicate that talks about no part of the heap and $\top$ here denotes the function $\lambda x.\ \top$. In our VST encoding, heap assertions carry extra type information t, inherited from VST, and are encoded as[4]

$$p \underset{t}{\overset{l}{\mapsto}} v\ \equiv\ (p \underset{t}{\mapsto} v, \top, (\lambda a.\ (\textbf{if}\ p \le a < p + \mathsf{size}\ t\ \textbf{then}\ l\ \textbf{else}\ \mathsf{Hi}))).$$

5 RELATED WORK

As far as we are aware, ours is the first formulation of a continuation passing style definition of IFC security.

As mentioned earlier in Sec. 2, our proposed logic is very similar in spirit to that of Costanzo and Shao [15]. They prove termination insensitive IFC for a simple imperative language with pointer arithmetic and aliasing, also based on Separation Hoare Logic. We work instead over C and take on its associated complexities. Their logic is deeply embedded whereas ours is shallowly embedded, in the style of VST. To prove their logic sound, they define an instrumented operational semantics that tracks the sensitivity of values, and prove simulation theorems with an ordinary semantics. In contrast the rules of our logic can be proved directly against the IFC definition.

The idea of leveraging an existing Hoare like logic for proving IFC security is well known. Murray et al. [29] build a shallowly embedded relational logic for IFC proofs atop an existing Hoare logic. Our work in the context of VST does something similar, albeit for C and a separation Hoare logic. Barthe et al. [5] verify IFC security using a Hoare like logic instead via the technique of *self composition*; yet different, Beringer [8] introduces *relational decomposition*, which reduces proofs of relational properties like IFC to proofs involving only program one execution by finding suitable witness relations between pairs of memories.

[4]In fact, they also carry permission annotations inherited from VST's concurrent separation logic, which we ignore here for simplicity.

REFERENCES

[1] Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Transactions on Programming Languages and Systems* 37, 2 (2015), 1–31.

[2] Andrew W. Appel and Sandrine Blazy. 2007. Separation Logic for Small-Step Cminor. In *Theorem Proving in Higher Order Logics*. Springer, 5–21.

[3] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers*. Cambridge University Press, New York, NY, USA.

[4] Andrew W. Appel and others. 2017. The Verified Software Toolchain. https://github.com/PrincetonUniversity/VST. (2017).

[5] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2011. Secure information flow by self-composition. *Mathematical Structures in Computer Science* 21, 6 (2011), 1207–1252.

[6] Thomas Bauereiß, Armando Pesenti Gritti, Andrei Popescu, and Franco Raimondi. 2016. CoSMed: A Confidentiality-Verified Social Media Platform. In *International Conference on Interactive Theorem Proving*. Springer, 87–106.

[7] Mark Beaumont, Jim McCarthy, and Toby Murray. 2016. The Cross Domain Desktop Compositor: Using Hardware-Based Video Compositing for a Multi-Level Secure User Interface. ACM Press, 533–545.

[8] Lennart Beringer. 2011. Relational Decomposition. In *International Conference on Interactive Theorem Proving*. Springer, 39–54.

[9] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *USENIX Security Symposium*. 207–221.

[10] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. NaCl: Networking and Cryptography library. Available at: http://nacl.cr.yp.to/. (????).

[11] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. 2012. The Security Impact of a New Cryptographic Library. Santiago, CL, 159–176.

[12] Sandrine Blazy, David Pichardie, and Alix Trieu. 2017. Verifying Constant-Time Implementations by Abstract Interpretation. In *ESORICS*. 260–277.

[13] David Cock, Gerwin Klein, and Thomas Sewell. 2008. Secure Microkernels, State Monads and Scalable Refinement. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (Lecture Notes in Computer Science)*, Vol. 5170. 167–182. https://doi.org/10.1007/978-3-540-71067-7_16

[14] David Costanzo. 2016. *Formal End-to-End Verification of Information-Flow Security for Complex Systems*. Ph.D. Dissertation. Yale University. http://www.cs.yale.edu/homes/dsc5/thesis.pdf.

[15] David Costanzo and Zhong Shao. 2014. A Separation Logic for Enforcing Declarative Information Flow Control Policies. In *International Conference on Principles of Security and Trust*. Springer, 179–198.

[16] Florian Dewald, Heiko Mantel, and Alexandra Weber. 2017. AVR Processors as a Platform for Language-Based Security. In *ESORICS*.

[17] Duncan A Grove, Toby Murray, Chris A Owen, Chris J North, JA Jones, Mark R Beaumont, and Bradley D Hopkins. 2007. An overview of the Annex system. In *Annual Computer Security Applications Conference (ACSAC)*. 341–352.

[18] Samuel Gruetter. 2017. *Improving the Coq proof automation tactics of the Verified Software Toolchain, based on a case study on verifying a C implementation of the AES encryption algorithm*. Master's thesis. École Polytechnique Fédérale de Lausanne (EPFL), Switzerland.

[19] Samuel Gruetter and Toby Murray. 2017. VST-Flow: Fine-Grained Low-Level Reasoning about Real-World C Code. *arXiv:1709.05243 [cs]* (Sept. 2017). arXiv:1709.05243

[20] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *OSDI*.

[21] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580.

[22] Sudeep Kanav, Peter Lammich, and Andrei Popescu. 2014. A conference management system with verified document confidentiality. In *International Conference on Computer Aided Verification*. Springer, 167–183.

[23] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems* 32, 1 (Feb. 2014), 2:1–2:70. https://doi.org/10.1145/2560537

[24] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[25] Luísa Lourenço and Luís Caires. 2015. Dependent Information Flow Types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Mumbai, India, 317–328.

[26] Donald A MacKenzie. 2004. *Mechanizing proof: computing, risk, and trust*. MIT Press.

[27] Toby Murray. 2015. Short Paper: On High-Assurance Information-Flow-Secure Programming Languages. In *10th ACM Workshop on Programming Languages and Analysis for Security (PLAS)*. 43–48.

[28] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: from General Purpose to a Proof of Information Flow Enforcement. In *IEEE Symposium on Security and Privacy*. San Francisco, CA, 415–429. https://doi.org/10.1109/SP.2013.35

[29] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. 2012. Noninterference for Operating System Kernels. In *International Conference on Certified Programs and Proofs*. Springer, Kyoto, Japan, 126–142.

[30] Toby Murray, Andrei Sabelfeld, and Lujo Bauer. 2017. Guest editorial: Special issue on verified information flow security. *Journal of Computer Security* (2017). https://doi.org/10.3233/JCS-15801 To appear.

[31] Peter W. O'Hearn. 2004. Concurrency, and Local Reasoning *(LNCS)*, Vol. 3170.

[32] John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *LICS*. IEEE, 55–74.

[33] Norbert Schirmer. 2006. *Verification of Sequential Imperative Programs in Isabelle-HOL*. Ph.D. Dissertation. Technical University Munich.

[34] Thomas Sewell, Magnus Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Seattle, Washington, USA, 471–481.

[35] Harvey Tuch, Gerwin Klein, and Michael Norrish. 2007. Types, Bytes, and Separation Logic. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 97–108.

[36] Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. 2009. Mind the Gap: A Verification Framework for Low-Level C. In *22nd TPHOLs (LNCS)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.), Vol. 5674. Springer, Munich, Germany, 500–515.

[37] Lantian Zheng and Andrew C. Myers. 2007. Dynamic security labels and Static Information Flow Control. *International Journal of Information Security* 6, 2–3 (March 2007).