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 compiler 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 low value-dependent classification [25, 27, 37].

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, 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:

• 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, ln, 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.

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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5099-0/17/10...$15.00
https://doi.org/10.1145/3139337.3139345

1For seL4, proving refinement was ~5 times more expensive than confidentiality [23].
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->next) {
            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 data. 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 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 $P_1 \land P_2$ 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 $p \mapsto v \land q \mapsto w$ 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:

$$\text{highptr} \mapsto h \land \text{lowptr} \mapsto l$$

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 $\ldots$ 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 $[P_1, \ldots, P_n]$.

$$\forall h \ b \ v. \quad \{\text{highptr} \mapsto h \land \text{lowptr} \mapsto l \land [b = b, v = v]\} \quad \ldots$$

$$\{\text{highptr} \mapsto (b \land h[v : b] \land \text{lowptr} \mapsto (b \land l[v])\}$$

(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 show how to realise these intuitions on top of VST’s existing separation logic for C.

We extend the $\longrightarrow$ notation of separation logic to carry ghost information asserting the sensitivity of data in the heap. We write $p \longrightarrow v$ 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 $\langle Hi, Lo \rangle$ in which $Lo \sqsubseteq Hi$ but $Hi \nsubseteq Lo$. 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.

\[
\begin{align*}
\text{highptr} &\quad \text{Hi} h \quad \text{lowptr} \quad \text{Lo} l \\
\{ b = b, v = v, b :: Lo, v :: (b ? Hi : Lo) \} \\
\cdots
\end{align*}
\]

In Sec. 2, we introduced security awareness assertions of the form $p \sqsubseteq (v \sqsubseteq b)$ at the heap location denoted by $p$ and $v$’s sensitivity is $l$), and $b :: l$ (stack variable $b$ holds $l$-sensitivity data). Intuitively, the former combines both an ordinary separation logic assertion $p \sqsubseteq v$ 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 $s$ as triples $(e, k, m)$ 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 $s_1 \rightarrow s_2$ for the small-step reduction relation, and we write $s_1 \rightarrow k s_2$ for its transitive closure, and $s_1 \rightarrow n s_2$ to say that after $n$ steps, state $s_1$ transitions to state $s_2$. Moreover, we define execution until final state, written $s_1 \Downarrow s_2$, as $s_1 \rightarrow * s_2$ where the command to be executed in $s_2$ 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 $(e, k, m)$ and $(e', k', m')$ are called low-equivalent with respect to the stack classification function $N$ and the heap classification function $A$ if for all stack locations $\ell$ for which $N \ell = Lo$, $e \ell = e' \ell$ and for all heap locations $\ell$ for which $A \ell = Lo, e \ell = e' \ell$.

**Definition 3.2 (Meaning of IFC judgement, first attempt).** The meaning of $[P_1, N_1, A_1] \in [P_2, N_2, A_2]$ is: (1) The Hoare judgement $[P_1] \in [P_2]$ holds and (2) for all $s_1, s_1', s_2, s_2'$, if $P_1 s_1$ and $P_1 s_1'$ hold and in both $s_1$ and $s_1'$ the command to be executed is $c$ and $s_1$ is low-equivalent to $s_1'$ w.r.t $N_1$ and $A_1$, and $s_1 \Downarrow s_2$ and $s_1' \Downarrow s_2'$, then $s_2$ is low-equivalent to $s_2'$ 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 $s_1$ and $s_1'$, we cannot cause changes in $Lo$ values between $s_2$ and $s_2'$.

#### 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):

\[
\forall x. \quad \{ \text{sec} = x, \text{sec} :: \text{Hi}, \text{pub} :: \text{Lo} \} \\
\text{pub} = \text{sec} \\
\{ \text{sec} = x, \text{pub} = x, \text{sec} :: \text{Hi}, \text{pub} :: \text{Lo} \}
\]
While this program is clearly insecure, the security statement is in fact provable w.r.t Definition 3.2: to prove the universal quantification, we assume \( x \) to be an arbitrary, but fixed value, so the \( \text{Hi} \) variable \( \text{sec} \) cannot have different values in the states \( \sigma_1 \) and \( \sigma'_2 \) from Definition 3.2, and therefore, \( \text{pub} \) will always have the same value in \( \sigma_2 \) and \( \sigma'_2 \), 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 \( VST \) defines validity of Hoare triples (to denote normal code execution until the end of the code block, or premature exit via \( \text{brk} \)), optional value and returning an assertion, where the type \( \text{exitkind} \) function body, \( VST \)‘s postconditions are not just plain assertions (i.e., execution isn’t stuck). 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 \( \tau \) 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 \( \text{break} \) and \( \text{continue} \). 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 \( (P) \vdash (Q) \) with the following series of definitions:

**Definition 3.3 (Immediately safe).** State \( \sigma = (e, k, m) \) is immediately safe if \( k = \text{nil} \text{ or } \sigma \rightarrow \sigma_2 \) for some \( \sigma_2 \) (i.e. execution isn’t stuck).

**Definition 3.4 (Safe).** A state \( \sigma \) is safe if for all \( \sigma_2, \) if \( \sigma \rightarrow^+ \sigma_2 \), then \( \sigma_2 \) is immediately safe.

**Definition 3.5 (Guard).** Predicate \( P \) guards continuation stack \( k \), written \( (P) \vdash (k) \), if for all \( e, m : P (e, k, m) \) implies \( (e, k, m) \) is safe.

To support \( \text{break, continue and return} \) before the end of the function body, \( VST \)‘s postconditions are not just plain assertions like the preconditions, but functions taking an \( \text{exitkind} \) and an optional value and returning an assertion, where the type \( \text{exitkind} \) is an enum with the four values \( \text{Eknorm}, \text{Ekbyk, Ekcont, and EKret} \) (to denote normal code execution until the end of the code block, or premature exit via \( \text{break, continue or return} \), 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 \( (R) \vdash (k) \), if for all \( e, k, v \), we have \( (R \ ek v) \vdash (k) \).

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 \( (P) \vdash (R) \) is: for all continuation stacks \( k \), \( (R) \vdash (k) \) implies \( (P) \vdash (c :: k) \).

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 \( c_1 \equiv_{\text{cont}} c_2 \), 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 \( \sigma = (e, k, m) \) and \( \sigma' = (e', k', m') \) are called head-equivalent, written \( \sigma \equiv_{\text{head}} \sigma' \) 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 \( \sigma_1 \) and \( \sigma'_2 \) are called matching, written \( \sigma_1 \equiv_{\text{match}} \sigma'_2 \), if for all \( n, \sigma_2, \sigma'_2 \), if \( \sigma_1 \rightarrow^* \sigma_2 \) and \( \sigma'_1 \rightarrow^* \sigma'_2 \) then \( \sigma_2 \equiv_{\text{head}} \sigma'_2 \).

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 \( \ell \) can be encoded as follows using our notion of matching. Let \( k \) be a continuation stack whose program loads the bit at location \( \ell \) 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 \( c_0 \equiv_{\text{cont}} c_1 \) 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 \( \ell \) must be the same in both memories, we can express this as \( (e_1, c :: k, m_1) \equiv_{\text{match}} (e'_1, c :: k, m'_1) \). 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 \( \ell \), putting \( c_0 \) or \( c_1 \) on top of the continuation stack depending on the bit stored at \( \ell \), and since match requires the two continuation stack heads to be equivalent, it ensures that the values stored at \( \ell \) 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 \( P x (e, k, m) \) and \( P x' (e', k', m') \) 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 \( (e, k, m) \equiv_{\text{match}} (e', k', m') \).
We are currently implementing these ideas atop the VST. With the write programs are translated for verification in VST.

3. 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 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 that the two productions 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. In the interests of brevity we defer to our working draft paper [19], and here just present one representative rule, for memory loads.

\[ P \vdash (\text{lcl}_\text{expr}) = v \land (p \rightarrow v) + \top \]

\[ P \vdash (\text{cst}_\text{expr} N e = e_1 \land A p = A) \]

The statement \text{id}=e loads the value at heap address denoted by expression e into the stack variable \text{id}. Expression e can refer only to stack variables, and might be e.g. an array access \text{a[i]}.

\[ P \vdash (\text{lcl}_\text{expr} N e = e_1 \land A p = A) \]

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.

4.1 Implementation of annotated assertions
Sec. 2.2 introduced security-aware separation logic assertions, like \( p \vdash v \), while our IFC definition of Sec. 3 and the primitive rules like \text{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” \( \ast \) operator, defined as follows.

\[ (P_1, N_1, A_1) \ast (P_2, N_2, A_2) \equiv (P_1 \ast P_2, N_1 \sqcup N_2, A_1 \sqcup A_2) \]

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 \( b :: l \) is encoded as

\[ b :: l \equiv (\text{emp}, (\lambda \text{id}. \text{id} = b \text{ then } l \text{ else } \top), \top) \]

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

\[ p \vdash t \equiv (p \rightarrow v, \top, (\lambda a. (\text{if } p \leq a < p + \text{ size } t \text{ then } l \text{ else } Hi))) \]

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.

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

\[ \text{In fact, they also carry permission annotations inherited from VST’s concurrent separation logic, which we ignore here for simplicity.} \]