The Meaning of Memory Safety

Arthur Azevedo de Amorim\textsuperscript{1} Cătălin Hritcu\textsuperscript{2} Benjamin C. Pierce\textsuperscript{1}

\textsuperscript{1}University of Pennsylvania \quad \textsuperscript{2}Inria Paris

Abstract—We propose a rigorous characterization of what it means for a programming language to be memory safe, capturing the intuition that memory safety supports local reasoning about state. We formalize this principle in two different ways. First, we show how a small memory-safe imperative language validates a noninterference property: parts of the state that are not reachable from a given part of the program can neither affect nor be affected by its execution. Second, we show how to take advantage of memory safety to extend separation logic, a framework for reasoning about heap-manipulating programs, with a variant of its frame rule. Our new rule is stronger because it applies even when parts of the program are buggy or malicious, but also weaker because it requires a stricter form of separation between parts of the program state. We also consider a number of pragmatically motivated variations of memory safety and the reasoning principles they support. As an application of our characterization, we evaluate the security of a previously proposed dynamic monitor for memory safety of heap-allocated data.

1 Introduction

Memory safety—and the plethora of catastrophic vulnerabilities that arise in its absence [46]—are common concerns among system designers. But what is memory safety, exactly? Intuitions abound, but translating them into satisfying formal definitions is surprisingly difficult [21].

In large part, this difficulty stems from the prominent role that informal, everyday intuition assigns, in discussions of memory safety, to a range of errors related to memory misuse—buffer overruns, double frees, etc. Characterizing memory safety in terms of the absence of these errors is tempting, but this falls short for two reasons. First, there is often disagreement on which behaviors qualify as errors. For example, many real-world C programs intentionally rely on unrestricted pointer arithmetic [30], although it may yield undefined behavior according to the language standard [22] §6.5.6. Second, from the perspective of security, the critical issue is not the errors themselves, but rather the fact that, when such errors occur in unsafe languages like C, the program’s ensuing behavior is determined by obscure, low-level factors such as the compiler’s choice of run-time memory layout, often leading to exploitable vulnerabilities. By contrast, in memory-safe languages such as Java, programs can still attempt to access arrays out of bounds, but such mistakes lead to sensible, predictable outcomes.

Thus, rather than attempting a definition in terms of a list of bad things that cannot happen, we aim to formalize memory safety in terms of reasoning principles that programmers can soundly apply when coding in a memory-safe setting—or, conversely, that programmers should not naively apply in non-memory-safe settings, because doing so can lead to serious bugs and vulnerabilities. Specifically, to give a formal account of memory safety, as opposed to more inclusive terms such as “type safety,” we focus on reasoning principles that are directly related to mutable state—i.e., those that are common to a wide range of stateful abstractions, such as records, tagged or untagged unions, local variables, closures, arrays, call stacks, objects, compartments, and address spaces.

What sort of reasoning principles? One source of inspiration comes from separation logic [39], a formal system for proving properties of programs that manipulate heap data structures. Like Hoare logic and related systems, separation logic manipulates program specifications of the form \{p\} \ c \ \{q\}, which roughly read as follows: if program c is run on an initial heap that satisfies precondition p, and c terminates, then the final heap satisfies postcondition q. What makes separation logic special is local reasoning about state: its proofs guarantee that programs access only a limited region of the heap described by their pre- and postconditions, while everything else is left untouched. This discipline means that we can extend a program’s specification with arbitrary invariants about the rest of the heap, as captured by a proof principle known as the frame rule

\[
\begin{align*}
\{p\} \ c \ \{q\} \\
\{p \ast r\} \ c \ \{q \ast r\},
\end{align*}
\]

where \(p \ast r\) is the so-called separating conjunction of the assertions p and r. The rule can be read as follows: “Suppose c satisfies the specification \(\{p\} \ c \ \{q\}\)—that is, it is started in a state where some part of the heap satisfies p, and c terminates, then this part of the heap satisfies q. Then, if c is started in an initial heap where some part satisfies p and a disjoint part of the heap satisfies an arbitrary assertion r, this second part will still satisfy r after c.”

The locality discipline imposed by separation logic is closely related to typical guidelines for memory-safe programming such as avoiding out-of-bounds accesses, because both restrict the potential effects of each piece of code to a clearly delimited part of the heap. For example, if a jpeg decoding subroutine is free of memory-safety violations and...
we can see from the program text that it only has access to the input and output image buffers, we know that the decoder cannot tamper with other parts of the state.

We can express such program constraints as specifications in separation logic. For example, a simple specification of the jpeg decoder might be \{p\} decoder \{true\}, where \(p\) is a separation-logic assertion saying that the variables in and out point to heap-allocated arrays of appropriate size. This specification is quite loose, in that it does not guarantee that the contents of the output buffer after running decoder bear any particular relation to the contents of in beforehand; however, it does imply that, whatever decoder does, it only affects the part of the heap described by \(p\). This means that we can apply the frame rule to reason about the decoder’s potential effects on the rest of the heap. For example, if the next step after decoding is to render the contents of the output buffer into a window object, and if this window object is disjoint from the in and out buffers, then the frame rule tells us that running decoder will not affect the well-formedness of the window object. The frame rule thus embodies a fundamental reasoning principle associated with memory-safe programs.

An inherent characteristic of standard separation logic is that safety is enforced manually: the program correctness proof must show that each memory operation accesses only the local state as described by its pre- and postconditions. This is what makes separation logic useful for modular reasoning in unsafe settings such as C: although any part of a C program could, in principle, access any part of the state, we can still reason locally about memory-safe programs. However, pervasive manual verification of the safety of every memory access seems prohibitively expensive for large code bases—especially in the presence of third-party libraries or plugins, over which we have little control.

In a setting where memory safety is enforced automatically, we can do much better. Suppose that the jpeg decoder above is a library written in Java. Even though we may not know anything precise about its input-output behavior, we can still correctly reason that running it cannot have any effect on a window object that it cannot reference. A simple reachability check showing that the window object is inaccessible replaces the detailed reasoning about individual memory operations demanded by separation logic. Our aim in this paper is to formalize this kind of reasoning.

Our first contribution is to formalize local reasoning principles that are valid for an ideal notion of memory safety, using a concrete imperative language as the basis of our discussion (introduced in §3). We show three frame theorems (Theorems 3.1, 3.3 and 3.4) that explain how the execution of a piece of code is affected by extending the heap on which it starts running. We use these results to derive a noninterference property (Corollary 3.5), guaranteeing that code cannot affect or be affected by regions of memory that it does not have the capability to access through the pointers it possesses. In §3.4, we show how these results yield a variant of the original frame rule of separation logic (Theorem 3.7). The two variants have complementary strengths and weaknesses: while the original rule applies to unsafe settings like C, but requires comprehensively verifying individual memory accesses, our variant does not require proving that every access is correct, but demands a stronger notion of separation between parts of the program state. These results have been verified with the Coq proof assistant; our machine checked proofs are available at: [https://github.com/arthur/a-memory-safe-language]

Our second contribution (§4) is to evaluate pragmatically motivated relaxations of the ideal notion of memory safety discussed above. These models differ from the ideal one by exploring various trade-offs between safety, performance, flexibility, and backwards compatibility, or by taking physical aspects such as time into account. We argue that these variants can be broadly classified into two groups according to reasoning principles they support. The stronger group gives up on some secrecy guarantees provided by memory safety, but still ensures that pieces of code cannot modify the contents of parts of the heap they do not have permission to access with their pointers. The weaker group, on the other hand, leaves gaps that completely invalidate reachability-based reasoning.

Our third contribution (§5) is to demonstrate how our characterization can apply to more realistic settings, by analyzing the security of a recently proposed dynamic monitor enforcing heap memory safety for low-level code [16], [6]. We prove that the abstract machine modeling the monitor satisfies a noninterference property similar to the one we show for the language of §2; then since the monitor is a valid implementation of the abstract machine, it inherits this noninterference property, modulo memory exhaustion issues discussed in §4. These proofs are also done in Coq and are available at: [https://github.com/micro-policies/memory-safety](https://github.com/micro-policies/memory-safety)

We discuss related work on memory safety and stronger reasoning principles for similar enforcement mechanisms in §6 and conclude in §7. While memory safety has been formally investigated in the past (e.g. [33], [44]), our characterization is the first phrased in terms of reasoning principles that are valid when memory safety is enforced automatically. We thus hope that these reasoning principles can serve as good criteria for formally evaluating practical mechanisms for enforcing memory safety. Moreover, our definition is directly targeted at memory safety and does not rely on additional features such as full-blown capabilities, closures, objects, module systems, etc. Since these features tend to depend on some form of memory safety anyway, we could see our characterization as a common core of reasoning principles that underpin all of them.

## 2 An Idealized Memory-Safe Language

Our discussion of memory safety begins with a concrete case study: a simple imperative language with manual memory management. It features several mechanisms for controlling the effects of memory misuse, ranging from the most conventional, such as bounds checking for spatial safety, to
more uncommon ones, such as assigning unique identifiers to every allocated block for ensuring temporal safety.

Choosing a language with manual memory management may appear odd, given that memory safety is often associated with garbage collection. We made this choice for two reasons. First, most of the discussion on memory safety is motivated by its absence from low-level languages like C that also rely on manual memory management. There is a vast body of research that tries to reconcile such languages with memory safety, and, as stated earlier, we hope that our account can help inform it. Second, we wanted to stress the mechanisms used to enforce memory safety, especially because they might have complementary advantages and shortcomings. For example, manual memory management as done in C can lead to more memory leaks; running with a garbage collector might result in slow, unpredictable performance; and specialized type systems for managing memory \[44, 40\] are more complex.

We begin by giving a brief overview of the language and its semantics. In §3, we will explore the reasoning principles enabled by its safety.

2.1 Language Overview

The syntax is summarized in Figure 1. Expressions \(e\) include variables \(x \in \text{var}\), numbers \(n \in \mathbb{Z}\), Booleans \(b \in \mathbb{B}\), an invalid pointer \(\text{nil}\), and various operations, both binary (arithmetic, logic, etc.) and unary (extracting the offset of a pointer). We write \([e]\) for dereferencing the pointer \(e\).

Programs operate on states consisting of two components: a local store, which maps program variables to values, and a heap, which maps pointers to values (Figure 2). Pointers are not bare integers, but rather pairs \((i, n)\) of a block identifier \(i \in \mathbb{I}\) and an offset \(n \in \mathbb{Z}\). The offset is relative to the corresponding block, and the identifier \(i\) need not bear any direct relation to the physical address that might be used in a concrete implementation of this language on a conventional flat-memory machine. (That is, we can equivalently think of the heap as a map associating each block identifier with a separate array of heap cells.) Similar structured memory models are widely used in the literature, as in the CompCert verified C compiler \[28\], for instance.

We write \([c](s)\) to denote the outcome of running a program \(c\) in an initial state \(s\); this outcome can be either a successful final state \(s'\) or a fatal run-time error. Note that \([c]\) is partial, to account for non-terminating computations. Similarly, \([e](s)\) denotes the value resulting from evaluating the expression \(e\) on the state \(s\), where expression evaluation is total and has no side effects. The formal definition of these functions is left to the Appendix; we just single out here a few aspects that will have a crucial effect on the security properties discussed later.

Illegal Memory Accesses Lead to Errors The language controls the effects of various kinds of memory misuse by treating them as run-time errors that stop execution as soon as they occur. This contrasts with typical C implementations, where such errors lead to unpredictable undefined behavior in compiled code. The main errors are caused by reads, writes, and frees to the current memory \(m\) using invalid pointers—that is, pointers \(p\) such that \(m(p)\) is undefined. Such pointers typically arise either by offsetting an existing pointer outside its bounds or by freeing a structure on the heap (which turns all other pointers to that block in the program state into dangling ones). In common parlance, this discipline ensures both spatial and temporal memory safety.

Block Identifiers are Capabilities Pointers can only be used to access memory corresponding to their block identifiers, which effectively act as capabilities. Block identifiers are set at allocation time, where they are chosen to be fresh for respect to the entire current state (i.e., the new block identifier is not associated with any pointers defined in the current memory, stored in local variables, or stored on the heap). Once assigned, identifiers are immutable, making it impossible to fabricate a pointer to an allocated block out of thin air. This can be seen, for instance, in the semantics of addition and subtraction, which allow pointer arithmetic but do not affect identifiers:

\[
[e_1 + e_2](s) = \begin{cases} 
  n_1 + n_2 & \text{if } [e_1](s) = n_1 \\
  (i, n_1 + n_2) & \text{if } [e_1](s) = (i, n_1) \\
  \text{nil} & \text{otherwise}
\end{cases}
\]
counter to generate identifiers for new blocks (as done by Dhawan et al. [16]; see §5.1); if enough tags are available, we can assume that every identifier will be fresh.

**Block Identifiers Cannot be Observed** Because of the freshness condition above, block identifiers can reveal information about the entire program state. For example, if they are chosen according to a monotonic counter, knowing what identifier was assigned to a new block allows us to deduce how many allocations have been performed. A concrete implementation would encounter similar issues related to the choice of physical addresses where new blocks are allocated. (Such issues are commonplace in systems that combine dynamic allocation and information-flow control [13].) For this reason, our language keeps identifiers opaque and inaccessible to programs; they can only be used to reference values in memory, and nothing else. We discuss a more permissive approach and its consequences in §4.2.

Note that hiding block identifiers doesn’t mean we have to hide everything associated with a pointer: besides using pointers to access memory, programs can also safely extract their offsets and test if two pointers are equal (which only pointers to access memory, programs can also safely extract.

**New Memory is Always Initialized** Whenever a memory block is allocated, all of its contents are initialized to 0. (The exact value does not matter, as long as it is some constant that is not a previously allocated pointer.) This is important for ensuring that allocation does not leak secrets present in previously freed blocks; we return to this point in §4.3.

### 3 Reasoning with Memory Safety

With this language definition in hand, we now turn to the local reasoning principles that it supports. Intuitively, these principles allow us to analyze the effect of a piece of code by restricting our attention to a smaller portion of the program state. A first set of frame theorems (3.1 3.3 and 3.4) describes how the execution of a piece of code is affected by extending the initial state on which it runs. These in turn imply a noninterference property, Corollary 3.5 guaranteeing that program execution is independent of inaccessible memory regions—that is, those that correspond to block identifiers that a piece of code does not possess. Finally, in §3.6 we discuss how the frame theorems can be recast in the language of separation logic, leading to a new variant of its frame rule (Theorem 3.7).

#### 3.1 Preliminaries

If $s_1 = (l_1, m_1)$ and $s_2 = (l_2, m_2)$ are program states, we define a new state $s_1 \cup s_2$ by taking the pointwise union

$$s_1 \cup s_2 = (l_1, m_1) \cup (l_2, m_2) \triangleq (l_1 \cup l_2, m_1 \cup m_2),$$

where the (left-biased) union of finite partial functions is

$$(f \cup g)(x) \triangleq \begin{cases} f(x) & \text{if } x \in \text{dom}(f) \\ g(x) & \text{otherwise}; \end{cases}$$

We write $\text{blocks}(s)$ for the set of block identifiers defined in the memory of $s$:

$$\text{blocks}(l, m) \triangleq \{ i \in \mathbb{I} \mid \exists n, (i, n) \in \text{dom}(m) \}$$

We write $\text{ids}(s)$ for the set of block identifiers appearing somewhere in $s$:  

$$\text{ids}(l, m) \triangleq \text{blocks}(l, m) \cup \{ i \mid \exists x, n, l(x) = (i, n) \} \cup \{ i \mid \exists p, n, m(p) = (i, n) \}$$

We write $\text{vars}(s)$ for the set of local variables defined in the state $s$,  

$$\text{vars}(l, m) \triangleq \text{dom}(l),$$

and $\text{vars}(c)$ for the set of local variables that occur in the program $c$.

A permutation (of block identifiers) is a function $\pi : \mathbb{I} \to \mathbb{I}$ that has a two-sided inverse $\pi^{-1}$; that is,

$$\pi \circ \pi^{-1} = \pi^{-1} \circ \pi = \text{id}_\mathbb{I}.$$  

Given a permutation $\pi$ and a state $s$, we define a new state $\pi \cdot s$ that is like $s$ except that all of its identifiers are renamed by $\pi$. The definition of this renaming operation is straightforward.

Finally, we write $X \not\equiv Y$ to indicate that sets $X$ and $Y$ are disjoint: $X \cap Y = \emptyset$.

#### 3.2 Basic Properties of Memory Safety

The first frame theorem states that, if a program terminates successfully when run on some initial state, then we can extend that initial state without affecting execution.

**Theorem 3.1 (Frame OK).** Suppose that $\llbracket c \rrbracket(s_1) = s'_1$, that $\text{vars}(c) \subseteq \text{vars}(s_1)$, and that $\text{blocks}(s_1) \not\equiv \text{blocks}(s_2)$. Then there exists a permutation $\pi$ such that $\llbracket c \rrbracket(s_1 \cup s_2) = \pi \cdot s'_1 \cup \text{blocks}(\pi \cdot s'_1) \not\equiv \text{blocks}(s_2)$.

The second premise, $\text{vars}(c) \subseteq \text{vars}(s_1)$, guarantees that all the variables needed to run $c$ are already defined in $s_1$, implying that their values do not change once we extend initial state with $s_2$. The third premise, $\text{blocks}(s_1) \not\equiv \text{blocks}(s_2)$, means that the memories of $s_1$ and $s_2$ store disjoint regions. Finally, the conclusion of the theorem states that (1) the execution of $c$ does not affect the extra state $s_2$ and (2) the rest of the result is almost the same as $s'_1$, except for a permutation of block identifiers.

Permutations are needed to avoid clashes between block identifiers in $s_2$ and those assigned to regions allocated by $c$ when running on $s_1$. For instance, suppose that the execution of $c$ on $s_1$ allocated a new block, and that this block was assigned some identifier $i \in \mathbb{I}$. If the memory of $s_2$ already had a block corresponding to $i$, $c$ would have to choose a different identifier $i'$ for allocating that block when running on $s_1 \cup s_2$. This change requires replacing all occurrences of $i$ by $i'$ in the result of the first execution.

1. It can be derived formally by viewing $S$ as a nominal set over $\mathbb{I}$ [29], obtained by combining products, disjoint unions, and partial functions.
which can be achieved with a permutation that swaps these two identifiers.

The proof of Theorem 3.1 relies crucially on the facts that programs cannot inspect block identifiers, that memory can grow indefinitely (a common assumption in formal models of memory), and that memory operations only succeed on valid pointers. Because of the renaming of identifiers, we also need the following result, which shows that the exact choice of block identifiers does not matter. Formally, if we permute the initial state \( s \) of a command \( c \) with any permutation \( \pi \), we obtain the same outcome, up to some additional permutation \( \pi' \) that again accounts for different choices of fresh identifiers.

**Theorem 3.2 (Renaming states).** Let \( s \) be a state, \( c \) a command, and \( \pi \) a permutation. There exists \( \pi' \) such that:

\[
\|c\| (\pi \cdot s) = \begin{cases} 
\text{error} & \text{if } \|c\| (s) = \text{error} \\
\bot & \text{if } \|c\| (s) = \bot \\
\pi' \cdot \pi \cdot s' & \text{if } \|c\| (s) = s'
\end{cases}
\]

A similar line of reasoning yields a second frame theorem, which says that we cannot make a non-terminating execution into a terminating one by extending its initial state.

**Theorem 3.3 (Frame Loop).** Suppose \( \|c\| (s_1) = \bot \), that \( \text{vars}(c) \subseteq \text{vars}(s_1) \), and that \( \text{blocks}(s_1) \neq \text{blocks}(s_2) \). Then \( \|c\| (s_1 \cup s_2) = \bot \).

The third frame theorem shows that extending the initial state also preserves erroneous executions. Its statement is similar to the previous ones, but with a subtle twist. In general by extending the state of a program with a new block, we might turn an erroneous execution into a successful one—if the error was caused by accessing a pointer whose identifier matches that new block. To avoid this, we need a different premise (\( \text{ids}(s_1) \neq \text{blocks}(s_2) \)) preventing any pointers in the original state \( s_1 \) from referencing the new blocks in \( s_2 \)—which is only useful because our language prevents programs from forking pointers to existing regions. Since \( \text{blocks}(s) \subseteq \text{ids}(s) \), this premise is stronger than the analogous ones in the preceding results.

**Theorem 3.4 (Frame Error).** Suppose that \( \|c\| (s_1) = \text{error} \), \( \text{vars}(c) \subseteq \text{vars}(s_1) \), and \( \text{ids}(s_1) \neq \text{blocks}(s_2) \). Then \( \|c\| (s_1 \cup s_2) = \text{error} \).

### 3.3 Memory Safety and Noninterference

The consequences of memory safety we have analyzed so far are intimately tied to the notion of noninterference \([20]\) used in information-flow control. In its most widely understood sense, noninterference is a secrecy guarantee: varying the secret inputs of a computation has no effect on its public outputs. Sometimes, however, it is also used to describe integrity guarantees: low-integrity inputs to a computation have no effect on its high-integrity outputs. In fact, both guarantees apply to unreachale memory in our language: the execution of a piece of code is independent of memory regions that were already allocated when it started executing but that it cannot access (because they are associated with block identifiers that it does not possess). By “independent,” we mean that the piece of code (1) cannot modify these inaccessible regions (preserving their integrity), and (2) cannot learn anything meaningful about these regions, not even their presence (preserving their secrecy).

**Corollary 3.5 (Noninterference).** Let \( s_1 \), \( s_2 \), and \( s_22 \) be states and \( c \) a program. Suppose that \( \text{vars}(c) \subseteq \text{vars}(s_1) \), that \( \text{ids}(s_1) \neq \text{blocks}(s_21) \) and \( \text{ids}(s_1) \# \text{blocks}(s_22) \). When running \( c \) on the extended states \( s_1 \cup s_2 \) and \( s_1 \cup s_22 \), there is only one of the following three possibilities:

- Both executions loop: \( \|c\| (s_1 \cup s_21) = \|c\| (s_1 \cup s_22) = \bot \);
- Both executions terminate with an error: \( \|c\| (s_1 \cup s_21) = \|c\| (s_1 \cup s_22) = \text{error} \); or
- Both executions successfully terminate without interfering with the inaccessible portions \( s_21 \) and \( s_22 \). Formally, there exists a state \( s_1' \) and permutations \( \pi_1 \) and \( \pi_2 \) such that \( \|c\| (s_1 \cup s_21) = \pi_1 \cdot s_1' \cup s_21 \), and \( \text{ids}(\pi_1 \cdot s_1' \cup s_21) \# \text{blocks}(s_22) \), for \( i = 1, 2 \).

**Proof.** Consider the result of executing \( c \) on \( s_1 \). If \( \|c\| (s_1) = \bot \), we apply Theorem 3.3 twice using \( s_21 \) and \( s_22 \) as the unreachable states (recall that \( \text{ids}(s_1) \# \text{blocks}(s_2) \) implies \( \text{blocks}(s_1) \# \text{blocks}(s_2) \)). If \( \|c\| (s_1) = \text{error} \), it suffices to apply Theorem 3.4 twice. And finally, if \( \|c\| (s_1) = s_1' \), we just apply Theorem 3.1 twice.

Noninterference is often formulated in terms of an indistinguishability relation on program states, which expresses that one state can be obtained from the other by varying its secrets. We could have equivalently phrased the above result in a similar way. Recall that the hypothesis \( \text{ids}(s_1) \# \text{blocks}(s_2) \) means that memory regions stored in \( s_2 \) are unreachale via \( s_1 \). Then, we could call two states “indistinguishable” if the reachable portions are the same (except for a possible renaming of block identifiers).

In \([34]\), we will see that the connection with noninterference provides a good benchmark for comparing different flavors of memory safety.

### 3.4 Memory Safety and Separation Logic

We now explore the close connection between the principles identified above, especially with respect to integrity, and the local reasoning facilities of separation logic. In separation logic, we are interested in proving specifications of the form \( \{p\} c \{q\} \), where \( p \) and \( q \) are predicates over program states (subsets of \( S \)). For our language, the meaning of such a \( \{p\} c \{q\} \) specification could be roughly stated as

\[
\forall s \in p, \text{vars}(c) \subseteq \text{vars}(s) \Rightarrow \|c\| (s) \in q \cup \{\bot\}.
\]

That is, if we start \( c \) in a state satisfying \( p \), then the program will either diverge or terminate in a final state that satisfies \( q \), but it will not trigger a run-time error. Part of the motivation
for precluding errors is that in unsafe settings like C they yield undefined behavior, destroying all hope of verification.

The power of separation logic for local reasoning and modular verification comes from the frame rule, a consequence of Theorems 3.1 and 3.3. The rule intuitively says that a verified program can only affect a well-defined portion of the state, with all other memory regions left untouched.

**Theorem 3.6.** Let \( p, q, \) and \( r \) be predicates over states and \( c \) be a command. Suppose that

\[
\forall l_1, l_2, m, (\forall x \notin \text{modvars}(c), l_1(x) = l_2(x)) \Rightarrow (l_1, m) \in r \Rightarrow (l_2, m) \in r,
\]

where \( \text{modvars}(c) \) is the set of all variables that appear in \( c \) as the destination of some assignment. (In other words, suppose that \( r \) does not depend on the local variables modified by \( c \).) Then, the following rule is sound

\[
\frac{p \land q}{p \land q} \quad \text{frame}.
\]

where \( p \land r \) denotes the separating conjunction of \( p \) and \( r \)—the predicate over states defined as

\[
\begin{align*}
\{(l, m_1 \cup m_2) \mid (l, m_1) \in p, (l, m_2) \in r, \\
\text{blocks}(l, m_1) \neq \text{blocks}(l, m_2)\}.
\end{align*}
\]

Separation-logic specifications require that executions terminate successfully (and satisfy the postcondition) or diverge—memory errors are completely ruled out by proof. However, this makes it difficult to use separation logic for partial verification: proving any property, no matter how simple, of a nontrivial program requires detailed reasoning about its internals. Even the following seemingly vacuous rule is unsound in separation logic:

\[
\frac{p}{p \land \{\text{true}\}} \quad \text{taut}.
\]

For a counterexample, take \( p \) to be true and \( c \) to be some arbitrary memory read \( x \leftarrow [y] \). If we run this program on an empty heap, which trivially satisfies the precondition, we obtain an error, contradicting the meaning of the triple.

Fortunately, in our memory-safe language—in which errororshave a sensible, predictable semantics, as opposed to wild undefined behavior—we can formulate a variant of separation logic that allows looseness in the postconditions. We now consider specifications of the form \( \{p\} c \{ q \} e \), defined as

\[
\forall s \in p, \text{vars}(c) \subseteq \text{vars}(s) \Rightarrow [q] (s) \in q \cup \{\bot, \text{error}\}.
\]

These specifications are weaker than their conventional counterparts presented above, leading to a subsumption rule:

\[
\frac{p \land q}{p \land q} \quad \text{taut}.
\]

Because errors are no longer prevented, the taut rule

\[
\frac{p \land \{\text{true}\} e}{p \land \{\text{true}\} e} \quad \text{taut}.
\]

Intuitively, the isolating conjunction guarantees that the heap fragment satisfying \( r \) is unreachable from the rest of the program state. The proof is very similar to the one for the original frame rule, but it relies on Theorem 3.4 in addition to Theorems 3.1 and 3.3 (which is the reason why the separating conjunction is not enough).

**3.5 Discussion**

As hinted by their connection with the frame rule, the frame theorems of 3.2 can themselves be considered as a form of local reasoning: to reason about a command, it suffices to restrict attention to the parts of the state that it can reach. Furthermore, the only thing we have to do is to calculate what this reachable portion is; how the program uses it is not
important. In a more realistic language, reachability might be inferred automatically from additional information such as typing. But even here it can probably be accomplished by a simple check of the program text.

For example, consider the hypothetical jpeg decoder from §4.1 As discussed there, we would like to guarantee that the decoder cannot tamper with an object that it cannot reference—a window object, a whitelist of trusted websites, etc. The frame theorems give us a means to do so, provided that we are able to show that the object is indeed unreachable. The noninterference result additionally implies that the jpeg decoder cannot directly extract any information from these unreachable objects, such as passwords or private keys.

Many real-world attacks involve direct violations of the reasoning principles we have articulated. For example, consider the infamous Heartbleed attack on OpenSSL [17], which used out-of-bounds reads from a buffer to leak data from completely unrelated parts of the program state—potentially stealing sensitive information like private keys. Given that the code fragment that enabled that attack was just manipulating an innocuous array, a programmer could easily be fooled into believing (as probably many have) that that snippet could not possibly access sensitive information, allowing that vulnerability to remain unnoticed for years.

Finally, our new frame rule only captures the fact that a command cannot influence the heap locations that it cannot reach, while our noninterference result (Corollary 3.5) captures not just this integrity aspect of memory safety, but also a secrecy aspect. We hope that future research will explore the connection between the secrecy aspect of memory safety and (relational) program logics.

4 Relaxing Memory Safety

So much for formalism. What about reality?

Strictly speaking, the strong security properties we have formulated above do not hold in any real system. This is partly due to fundamental physical limitations—real systems run with finite memory, and they interact with their users in various ways that transcend inputs and outputs, notably through time. A more interesting reason is that real systems typically do not impose all the restrictions we have relied on for the proofs of these properties. Languages that aim to be memory safe generally offer relatively benign glimpses of their implementation details (such as being able to read the previous contents of uninitialized memory, compare pointers with $\leq$, or extract physical addresses from pointers) in return for significant performance or flexibility gains in some situations. In other systems, the concessions are more fundamental, to the extent that it is harder to clearly delimit what part of a program is unsafe: the SoftBound transformation [33], for example, adds spatial memory-safety checks for C programs, but does not provide protection against bugs caused by erroneous uses of free by itself; a related transformation, CETS [34], is required to enforce temporal memory safety.

In this section, we enumerate some common relaxed models of memory safety and evaluate how they affect the reasoning principles and security guarantees of §3. Some relaxations, such as allowing pointers to be forged out of thin air, completely give up on the strong reachability-based reasoning we proposed above. Others, however, retain strong reasoning principles for integrity while giving up on some secrecy, allowing aspects of the global state of a program to be observed. For example, a system with finite memory (§4.5) may leak some information about its total memory consumption, and a system that allows pointer-to-integer casts (§4.2) may leak information about its memory layout. Naturally, the distinction between integrity and secrecy should be taken with a grain of salt, since the former depends on the latter in many practical situations; for example, if a system grants privileges to change the state of some component when accessed with the right password, a secrecy violation can be escalated to an integrity violation!

4.1 Forging Pointers

Many real-world C programs rely on using integers as pointers. If this idiom is permitted without restrictions, then robust local reasoning is compromised, since every part of memory might be reachable by any part of the program. It is thus not surprising that systems that strive for memory safety either prevent this kind of pointer forging or else restrict it to well-delimited unsafe fragments.

More insidiously, and perhaps surprisingly, similar dangers also lurk in the stateful abstractions even of some systems that are widely regarded as “memory safe.” In JavaScript, for example, it is possible for code to access arbitrary global variables by indexing an associative array with a string, a feature that enables many serious attacks [1], [19], [47], [31]. One might argue that global variables in JavaScript are “memory unsafe” because they fail to validate local reasoning: the fact that a particular part of a JavaScript program does not explicitly mention a given global variable does not imply that running this code cannot change this variable or the things it points to. Re-enabling local reasoning requires imposing very strong restrictions on the allowed programming style [10], [1], [19].

4.2 Observing Pointers

The language of §3 maintains a complete separation between pointers and other kinds of values. In real systems, this separation is often only enforced in one direction. For example, some tools for enforcing memory safety of C programs [33], [14] allow pointer-to-integer casts [25], a feature required by many low-level C idioms [11], [30]. Additionally, languages considered as memory safe often include features that break this separation—e.g., the default implementation of `hashCode()` in Java. To model such features, we can extend the syntax of expressions with a cast operator

\[ e ::= \cdots | \text{cast}(e) | \cdots \]

and assume that we have some function $\text{cast} : I \times Z \to Z$ for converting a pointer to an integer, which we use to define
the semantics of cast:
\[ [[\text{cast}(e)](s) = [[\text{cast}]](\|e\|(s)) \quad \text{if} \quad \|e\|(s) \in \mathbb{I} \times \mathbb{Z} \]

Note that the language we introduced originally included an offset operator for extracting the offset of a pointer. Their definitions are similar, but have crucially different consequences: while offset does not depend on the block identifier, allocation order, or other low-level details of the language implementation (such as the choice of physical addresses when allocating a block), all of these could be relevant when defining the semantics of cast. The three frame theorems \(1, 2, 3\) and \(3, 4\) are thus lost, because the state of unreachable parts of the heap may influence integers observed by the program. An important consequence is that the secrecy is weakened in this language: an attacker could exploit pointers as a side-channel to learn secrets about data it doesn’t have direct access to.

Nevertheless, we can still guarantee the integrity of unreachable parts of the program state: if a program does not hold any pointers to an allocated block, the contents of that block will not change at the end of the execution. (This result was also proved in Coq.)

**Theorem 4.1** (Integrity-only Noninterference). Let \(s_1, s_2,\) and \(s'\) be states and \(c\) a command such that \(\text{vars}(c) \subseteq \text{vars}(s_1),\) \(\text{idss}(s_1) \# \text{blocks}(s_2),\) and \(\|e\|(s_1 \cup s_2) = s'.\) Then we can find \(s_1' \in S\) such that \(s' = s_1' \cup s_2\) and \(\text{idss}(s_1') \# \text{blocks}(s_2).\)

The crucial difference between this and the stronger noninterference result of Corollary 3.5 is that, if pointer-to-integer casts are prohibited, we know that changing the contents of the unreachable portion \(s_2\) has no effect on the reachable portion, \(s_1'.\) If these casts are allowed, then changing \(s_2\) can influence \(s_1'\) in arbitrary ways: not only can the contents of this final state change, but the execution can also loop forever or terminate in an error.

To understand why this is the case, consider the jpeg decoder of 11. Suppose that the decoder is being used inside of a web browser, but that it does not have the required pointers to learn the address that the user is currently visiting. Suppose that there is some relation between the memory consumption of the program and that website, and that there is some correlation between the memory consumption and the identifier assigned to a new block. Then, by allocating a new block and converting it to a integer, the decoder might be able to infer useful information about the website that the user is visiting 23. Thus, if \(s_2\) denoted the part of the browser’s state responsible for storing that location, changing its contents would have a nontrivial effect on \(s_1',\) the part of the state that the decoder does have access to. We could speculate that, in a reasonable system, this channel can only reveal information about the layout of unreachable regions, and not their contents. Indeed, we conjecture this for the variant of our language considered in this sub-section.

Finally, it is worth mentioning that simply excluding the cast operation from the language might not be enough for preventing this sort of secrecy violations in practice. For instance, recall that our language takes both offsets and block identifiers into account to test if two pointers are equal. For performance reasons, we could have chosen a different design that only compares pointers using their physical addresses at run time, completely discarding block identifiers. If attackers know the physical address of a pointer that they own—which could happen, for instance, if they know that pointer was the first to be allocated, and know enough about the implementation of the allocator to determine where that pointer will live—they can use pointer arithmetic (which is generally harmless and allowed in our language) to discover the address of other pointers. If \(x\) holds the pointer they control, they can run, for instance,

\[ y \leftarrow \text{alloc}(1); \text{if} \ x + 1729 = y \text{ then} \ldots \text{ else} \ldots,\]

to learn the location of the new pointer assigned to \(y,\) and thus draw conclusions about the global state of the program.

**4.3 Uninitialized Memory**

Traditionally, memory-safe languages require variables and objects to be initialized before they are used. But this can degrade performance for some applications, leading many systems to drop this feature—including not only standard implementations of C, but also implementations that provide some memory-safety guarantees [33], [14]. Even languages that emphasize safety allow access to uninitialized memory in some cases—e.g., OCaml’s Bytes.create primitive, Node.js’s Buffer.allocUnsafe.

The problem with uninitialized memory is that it breaks the abstraction of the program state as consisting solely of the local variables and allocated objects: the entire memory becomes relevant to the execution of a program, and reasoning locally becomes much harder. By inspecting old values living in uninitialized memory, an attacker can learn about parts of the program state they shouldn’t have access to, a direct violation of secrecy. This issue would become even more severe in a hypothetical system that allowed old pointers or other capabilities to occur in re-allocated memory in a way that the program can use, since they could yield access to restricted resources directly, leading to potential integrity violations as well. (The two examples given above—OCaml’s Bytes.create primitive and Node.js’s Buffer.allocUnsafe—do not suffer from this issue, because any preexisting pointers in re-allocated memory are treated as bare bytes that cannot be used to access memory.)

**4.4 Dangling Pointers and Freshness**

Another crucial issue is the treatment of dangling pointers—those that reference objects that have already been freed. Dangling pointers are problematic because there is an inherent tension between giving them a sensible semantics (for instance, one that validates the properties of §3) and obtaining good performance and predictability. Languages with garbage collection avoid the issue by ensuring that dangling pointers never occur—heap storage is freed only when there are no pointers to it left. In the simple language from §2 besides giving a well-defined behavior to the use of dangling pointers (aborting execution with an error), we
imposed strong freshness requirements in the allocation rule, mandating not only that the identifier assigned to the new block not correspond to any existing block, but also that it not be present anywhere else in the program state.

To see how the results of §3 are affected in a setting where freshness is not enforced, consider the program

\[ x \leftarrow \text{alloc}(1); z \leftarrow (y = x), \]

and suppose we run it on a state where \( y \) holds a dangling pointer. Depending on the behavior of the allocator and the state of the memory, the pointer assigned to \( x \) could be equal to \( y \). Since this outcome depends on the entire state of the system, not just the reachable memory, Theorems 3.1, 3.3 and 3.4 now fail. Furthermore, an attacker with detailed knowledge of the allocator implementation could potentially launder secret information by testing pointers for equality. Weakening freshness guarantees can also have implications for integrity, since it becomes much harder to guarantee that memory blocks are properly isolated. For instance, a newly allocated block might be reachable through a dangling pointer that is controlled by an attacker, allowing them to access that block even if they were not supposed to.

Some practical solutions for memory safety use mechanisms similar to our language’s, where each memory location is tagged with an identifier describing the region it belongs to [12], [16]. Pointers are tagged similarly, and when a pointer is used to access memory, a violation is detected if its identifier does not match the location’s. However, for performance reasons, the number of possible identifiers might be limited to a relatively small number, such as 2 or 4 [12] or 16 [49]. In addition to the above problems, since multiple live regions can share the same identifier in such schemes, it might be possible for buffer overflows to lead to violations of secrecy and integrity as well.

Although we framed our discussion in terms of block identifiers, the issue of freshness can manifest itself in other ways. For example, many systems for spatial memory safety work by enriching pointers with base and bounds information. In some of these [14], [33], dealing with dangling pointers is treated as an orthogonal issue, and it is possible for the allocator to return a new memory region that overlaps with the range of a dangling pointer, in which case the new region will not be properly isolated from the rest of the state.

Finally, dangling pointers can have disastrous consequences for overall system security, independently of the freshness issues just described: freeing a pointer more than once can break allocator invariants, enabling attacks [46].

4.5 Infinite Memory

Our idealized language allows memory to grow indefinitely. But real languages run on finite-memory machines, and cannot allocate new memory when programs run out of space. Besides enabling denial-of-service attacks, finite memory has consequences for the secrecy properties studied earlier. Corollary 3.5 does not hold in a real programming language as is, because an increase in memory consumption can cause a previously successful allocation to fail. By noticing this difference, a piece of code could learn something about the entire state of the program—not just its reachable portions. How problematic this is in practice, however, will depend on the particular system under consideration.

A potential solution is to force programs that run out of memory to terminate immediately—for example, by making alloc in our language cause an execution error if an outOfMemory predicate holds of the current program state. Though this choice might be bad from an availability standpoint, it is probably the most benign in terms of secrecy. We should be able to prove an error-insensitive variant of Corollary 3.5, where the only significant interaction that unreachable memory can have with a piece of code is to turn a successful execution or infinite loop into an error, or vice versa. Similar issues arise for information-flow mechanisms that often cannot prevent secrets from influencing the termination behavior of programs, leading to termination-insensitive notions of noninterference.

Unfortunately, even an error-insensitive result might be too strong for real systems, which often make it possible for attackers to extract multiple bits of information about the global state of the program—something that had already been noted in the information-flow literature [5]. Java, for example, does not force termination when memory runs out, but triggers an exception that can be caught and handled by user code, which is then free to record the event and probe the allocator with a different test. And most languages do not operate in batch mode like ours does, merely producing a single answer at the end of execution; rather, their programs continuously interact with their environment through inputs and outputs, allowing them to communicate the exact amount of memory that caused an error.

This discussion suggests that, if size vulnerabilities are a real concern for a system, they need to be treated with special care. One approach would be to use a mechanism to limit the amount of memory an untrusted component can allocate [50], so that exhausting the memory allotted to that component doesn’t reveal information about the state of the rest of the system (and so that also global denial-of-service attacks are prevented). A more speculative idea is to develop quantitative versions [42], [7] of the noninterference results discussed here, allowing us to analyze the behavior of a program on an extended state only if the total memory used by the program is below a certain limit.

4.6 Side-channel Attacks

As often done in the information-flow control literature, our noninterference result assumes the code does not use side-channels to learn information about unreachable memory regions. In practice, a malicious piece of code (e.g., a plugin) may infer secret information from the unreachable memory regions by for instance observing differences in its execution time caused by the contents of processor caches, which are normally shared by all the code. While the attacker model considered in this paper does not try to address such side-channel attacks, one should be able to use the previous research on the subject to protect against them or limit the damage they can cause [42], [7], [52], [43].
5 Case Study: A Memory-safety Monitor

To demonstrate the applicability of our proposed characterization, we use it to evaluate the security guarantees of a tag-based dynamic monitor recently proposed by Dhawan et al. to enforce heap memory safety for low-level code [16]. Prior work by Azevedo de Amorim et al. [6] has shown formally that a model of the monitor running on a idealized tag-based architecture correctly implements a higher-level abstract machine with built-in checks for memory safety—a bit more formally, every behavior of the monitor is also a behavior of the abstract machine. Building upon this work, we prove that this abstract machine satisfies a noninterference property similar to Corollary 4.5. We were also able to prove that a similar result holds for a lower-level machine that runs a so-called “symbolic” representation of the monitor—although we had to slightly weaken the result to account for memory exhaustion (cf. §4.5), since the machine that runs the monitor has finite memory, while the abstract machine has infinite memory. If we had a verified machine-code implementation of this monitor, it would be possible to prove a similar result for it as well.

5.1 Tag-based Monitor

We content ourselves with a brief overview of Dhawan et al.’s monitor [16], [6], since the formal statement of noninterference for it is more complex than the one for the abstract machine from §5.2 on which we will focus.

Following a proposal by Clause et al. [12], Dhawan et al.’s monitor enforces memory safety for heap-allocated data by checking and propagating metadata tags. Every memory location receives a tag that uniquely identifies the allocated region to which that location belongs (akin to block identifiers in §2), and pointers receive the tag of the region they are allowed to reference. The monitor assigns these tags to new regions by storing a monotonic counter in protected memory that is bumped on every call to malloc; with a large number of possible tags, it is possible to avoid the freshness pitfalls discussed in §4.4. When a memory access occurs, the monitor checks whether the tag on the pointer matches the tag on the location. If they do, the operation is allowed; otherwise, execution halts. The monitor instruments the allocator to make set up tags correctly. Its implementation achieves good performance using the PUMP, a hardware extension accelerating such micro-policies for metadata tagging [16].

5.2 Abstract Machine

The abstract machine of Azevedo de Amorim et al. [6] operates on two kinds of values: machine words \( w \), or pointers \( (i, w) \), which are pairs of an identifier \( i \in I \) and a machine word offset \( w \). We use \( W \) to denote the set of machine words, and \( V \) to denote the set of values. Machine states are triples \( (m, rs, pc) \), where

- \( m \in I \to fin V^* \) is a memory, which maps identifiers to lists of values (frames);
- \( rs \in R \to fin V \) is a register bank, mapping registers (elements of a finite set \( R \)) to values;
- \( pc \in V \) is the program counter.

The execution of the operation is specified by a stepping relation: \( s \to s' \) means that running a single instruction on the machine at state \( s \) yields a new state \( s' \). If there is no \( s' \) such that \( s \to s' \), we say that \( s \) is stuck, which means that a fatal error occurred during execution.

On every instruction, the machine checks if the current program counter is a pointer and, if so, tries to fetch the corresponding value in memory. The machine then ensures that this value is a word that correctly encodes an instruction and, if so, acts accordingly. The instructions of the machine, representative of typical RISC architectures, are summarized in Figure 3. Programs can perform binary operations (Binop), move values to and from memory (Load, Store), and branch (Jump, Jal, Bnz). The machine is in fact fairly similar to the language of §2. Some binary operations are overloaded to manipulate pointers; for example, adding a pointer to a word is allowed, and the result is obtained by adjusting the pointer’s offset accordingly. Accessing memory causes the machine to halt when the corresponding position is undefined.

In addition to these basic instructions, the machine possesses a set of special monitor services that can be invoked by user code as regular functions, using registers to pass in arguments and return values. There are two services alloc and free for managing memory, and one service eq for testing whether two values are equal. The reason for using separate monitor services instead of special instructions is to keep its semantics closer to the more concrete machine that implements it. While the instruction set includes an equality test, this instruction cannot replace the eq service, since it only takes physical addresses into account. As argued in §4.2, such physical address comparisons can be turned into a side channel: comparing out-of-bounds pointers to different blocks reveals information about the global state of the allocator. To prevent this, testing two pointers for equality directly using the corresponding machine instruction results in an error if the pointers have different block identifiers.

5.3 Verifying Memory Safety

The proof of memory safety for this abstract machine mimics the one carried for the language in §4. We use similar notations as before: \( \pi \cdot s \) means renaming every identifier that appears in \( s \) according to the permutation \( \pi \), and \( ids(s) \) is the finite set of all identifiers that appear in the state \( s \).

A simple case analysis on the possible instructions yields analogs of Theorems 3.1 [32] and 3.4 [33] (we don’t include an analog of Theorem 3.3 because there isn’t a corresponding notion of looping for this machine). We show single-step versions for simplicity, but the results generalize easily to multiple steps.
Theorem 5.1. Let $\pi$ be a permutation, and $s$ and $s'$ be two machine states such that $s \to s'$. There exists another permutation $\pi'$ such that $\pi \cdot s \to \pi' \cdot s'$.

Theorem 5.2. Let $(m_1, rs, pc)$ be a state of the abstract machine, and $m_2$ a memory. Suppose that $ids(m_1, rs, pc) \neq dom(m_2)$, and that $(m_1, rs, pc) \to (m', rs', pc')$. Then, there exists a permutation $\pi$ such that

$$\text{ids}(\pi \cdot m', \pi \cdot rs, \pi \cdot pc) \neq \text{dom}(m_2)$$

and

$$(m_2 \cup m_1, rs, pc) \to (m_2 \cup \pi \cdot m', \pi \cdot rs', \pi \cdot pc').$$

Theorem 5.3. Let $(m_1, rs, pc)$ be a state of the abstract machine, and $m_2$ a memory. Suppose that $ids(m_1, rs, pc) \neq \text{dom}(m_2)$, and that $(m_1, rs, pc)$ is stuck. Then $(m_2 \cup m_1, rs, pc)$ is also stuck.

Once again, by combining these properties, we obtain a proof of noninterference.

Corollary 5.4 (Noninterference). Let $s = (m_1, rs, pc)$ be a state, and $m_{21}$ and $m_{22}$ two memories. Suppose that $ids(m_1, rs, pc) \neq \text{dom}(m_{2i})$ for $i = 1, 2$. When trying to run $s$ by adding the extra memories $m_{2i}$, only the following two possibilities can arise.

- Both $(m_{21} \cup m_1, rs, pc)$ and $(m_{22} \cup m_1, rs, pc)$ are stuck; or
- Both states successfully step without interfering with the inaccessible portions $m_{21}$ and $m_{22}$. Formally, there exists a state $s' = (m', rs', pc')$, and permutations $\pi_1$ and $\pi_2$ such that

$$(m_{21} \cup m_1, rs, pc) \to (m_{21} \cup \pi_1 \cdot m', \pi_1 \cdot rs', \pi_1 \cdot pc')$$

and

$$\text{ids}(\pi_i \cdot s') \neq \text{dom}(m_{2i}), \text{for } i = 1, 2.$$

5.4 Discussion

This noninterference result for the memory-safety monitor has an important difference compared to the original one (Corollary 3.3). In the memory-safe language, the notion of memory reachability is relative to a program's local variables. If we want to argue that part of the state is kept isolated from some code fragment, we just have to consider fragment's local variables—other parts of the program are still allowed to access the region. The memory-safety monitor, on the other hand, does not have an analogous notion: if a memory region becomes unreachable it becomes useless, since it will remain unreachable forever.

It seems that, from the standpoint of noninterference, heap memory safety taken in isolation is much weaker than the guarantees it provides in the presence of other linguistic features, such as local variables. Nevertheless, the properties studied above suggest several possibilities that we could explore for strengthening the mechanism and making its guarantees more useful. The most obvious one would be to use the mechanism as the target of a compiler for a programming language that provides additional (safe) abstractions for managing state, such as variables and a stack for procedure calls. A more modest approach from the point of view of formal verification would be to add other state abstractions to the mechanism itself. Besides variables and call stacks, if the mechanism made code immutable and separate from data, a simple check would suffice to tell whether a code segment stored in memory references a given privileged register. If the register is the only means of reaching a memory region, we should be able to soundly infer that that code segment is independent of that region.

On a last note, although the abstract machine we verified is fairly close to our original language, the dynamic monitor that implements it using tags is quite different (§5.1). In particular, the monitor works on a machine that has a flat memory model, and keeps track of free and allocated memory using a protected data structure that stores base and bounds information associated with every block. It was claimed that reasoning about this base and bounds information was the most challenging part of the proof that the monitor implements the abstract machine [6]. For this reason, we believe that this proof can be adapted to other enforcement mechanisms that rely solely on base and bounds information—for example, fat pointers [27], [14] or SoftBound [33]—while keeping a similar abstract machine as their specification, and thus satisfying a similar noninterference property. This gives us confidence that our memory safety characterization generalizes to other settings.

6 Related Work

The present work lies at the intersection of two areas of previous research: one on formal characterizations of memory safety, the other on reasoning principles for programs. We review the most closely related work in these areas.

Characterizing Memory Safety Many formal characterizations of memory safety originated in attempts to reconcile its benefits with low-level languages and systems. Generally, these works claim that a mechanism enforces memory safety by showing that it prevents or catches typical temporal and spatial violations. Examples in the literature include: Cyclone [44], a language featuring a region-based type system for safe manual memory management; CCured [35], a program transformation that adds temporal memory safety to C by refining its pointer type to distinguish between various degrees of safety; Ivory [18] an embedding of a similar “safe-C type system” into Haskell; SoftBound [33], an instrumentation technique for C programs for enforcing spatial memory safety, including the detection of bounds violations within an object; CETS [34], a compile-time pass for preventing temporal memory-safety violations in C programs, including accessing dangling pointers into freed heap regions and stale stack frames; the tag-based memory-safety monitor for the PUMP [16], [6], which formed the basis of our case study in §5; and languages like Mezzo [38] and Rust [43], whose guarantees extend to preventing data races [8]. Similar models of memory errors appear in popular formalizations of C [28], [26], which need to rigorously characterize sources of undefined behavior in the language—in particular, instances of memory misuse.

Either explicitly or implicitly, these works define memory errors as attempts to use a pointer to access a memory...
location that it was not meant to access—for example, a location that lies past its bounds, or one that has already been freed. This aspect was noted by Hicks [21], who, inspired by the work on SoftBound, proposed to define memory safety as an execution model that tracks what part of memory each pointer has access. Our characterization is complementary to these accounts, in that it is extensional: its data isolation properties allow us to reason directly about program observable behavior. Furthermore, as demonstrated by our application to the monitor of §5 and the discussions on §4, it can be easily adapted to various enforcement mechanisms and variations of memory safety.

**Reasoning Principles**  Separation logic [39], [51] has been an important source of inspiration for our work. The logic’s frame rule enables its local reasoning capabilities and imposes restrictions that are similar to those mandated by memory-safe programming guidelines. As discussed in §3.4, our reasoning principles are reminiscent of the frame rule, but use pointer reachability to guarantee access locality in settings where memory safety is enforced automatically. In separation logic, by contrast, locality needs to be guaranteed for each program individually by comprehensive proofs.

Several works have investigated similar reasoning principles for a variety of program analysis techniques, including static, dynamic, manual, or a mixture of those. Some of these are formulated as expressive logical relations associated with types, guaranteeing that programs are compatible with the framing of state invariants; representative works include: L3 [4], a linear calculus featuring strong updates and aliasing control; the work of Benton and Tabereau [9] on a compiler for a simple higher-order language; and the work of Devriese et al. [15] on object capabilities for a JavaScript-like language. Other developments are based on proof systems reminiscent of separation logic with rules that guarantee isolation; these include Yarra [41], an extension of C that allows programmers to protect the integrity of data structures marked as critical; the work of Agten et al. [3], which allows mixing unverified and verified components by instrumenting the program to check that required assertions hold at interfaces; and the logic of Swasey et al. [45] for reasoning about object-capability memory patterns.

Unlike our work, these works do not propose to use reachability-based isolation to define memory safety in a general setting, nor do they attempt to analyze how the reasoning principles they identify are affected by common variants of memory safety. Furthermore, many of these other reasoning principles—especially the logical relations—rely on encapsulation mechanisms such as closures, objects, or modules that go beyond plain memory safety. Memory safety only allows us to fully isolate state components, while encapsulation provides finer control, allowing some interaction between components, while guaranteeing the preservation of certain invariants associated with them. In this sense, one can see memory-safety reasoning as a special case of encapsulation reasoning. Nevertheless, it is a practically relevant special case that is interesting on its own, since when code makes use of an encapsulated component, one must argue explicitly that the invariants we care about are preserved by the privileged operations of that component; memory safety, on the other hand, guarantees that any invariants on unreachable parts of the memory are automatically preserved.

Maybe closer related to our work, Maffeis et al. [29] show that languages that satisfy their notion of “authority safety” guarantee isolation, in the sense that a component’s actions cannot influence the actions of another component with disjoint authority. Their notion of authority behaves similarly to the set of block identifiers accessible by a program in our language; however, they do not attempt to connect their notion of isolation to the frame rule, noninterference, or traditional notions of memory safety.

Morrisett et al. [32] state a correctness criterion for garbage collection based on program equivalence. Some of the properties they study are again similar to the frame rule, in that they describe the behavior of code running in an extended heap. However, they use this analysis to justify the validity of erasing unreachable state, rather than studying the possible interactions between the extra state and the program in terms of integrity and secrecy.

Other works attempt to characterize protection schemes that are weaker than full memory safety. Juglaret et al. [24] propose a correctness criterion for compiling compartmentalized programs, which allows memory-safety violations to occur within each compartment, but bounds the effect of such violations on other compartments of the program. Their criterion is reminiscent of the traditional notion of full abstraction, which guarantees that contextually equivalent programs remain equivalent after compilation. Abadi and Plotkin [2] develop an address-space randomization scheme for a simple compiler, and prove a probabilistic full-abstraction result for it. While full abstraction and related properties guarantee that certain security properties of programs are preserved by compilation, these works do not consider whether these properties encompass the type of isolation guarantee analyzed here.

7 Conclusions and Future Work

We have explored the consequences of memory safety for reasoning about programs, formalizing intuitive principles that, we argue, capture the essential distinction between memory-safe systems and memory-unsafe ones. We showed how the reasoning principles we identified apply to a recent dynamic monitor for heap memory safety of low-level code.

The systems studied in this paper have a simple storage model: the language of §2 has just global variables and flat, heap-allocated arrays, while the monitor of §5 doesn’t even have variables or immutable code. Realistic programming platforms, of course, offer much richer stateful abstractions, including, for example, procedures with stack-allocated local variables as well as structured objects with contiguously allocated sub-objects. In terms of memory safety, these systems have a richer vocabulary for describing resources that programs can access, and programmers could benefit from isolation-based local reasoning involving these resources.
For example, in a memory-safe language with procedures and stack variables, it should be possible to assert that the behavior of a procedure depends only on the arguments that it is given, the global variables it uses, and the portions of the state that are reachable from these values; if the caller of that procedure has a private object that is not passed as an argument, it should not affect or be affected by the call. Additionally, languages such as C allow for objects consisting of contiguously allocated sub-objects for improved performance. Some systems [13, 14] add spatial memory safety to C while allowing programmers to down-grade capabilities—that is, narrow the range of a pointer so that it can’t be used to access outside of a sub-object’s bounds. It would be interesting to refine our idealized model of memory safety to take into account the reasoning enabled by these features. In the case of the monitor of [3] such considerations could lead to improved monitor designs or to incorporating the monitor inside a secure compiler.

Conversely, it would be interesting to derive finer security properties for relaxations of memory safety like the ones discussed in [4]. Some inspiration could come from the information-flow literature, where quantitative noninterference results provide bounds on the probability that some secret is leaked, the rate at which it is leaked, how many bits are leaked, etc. [7, 42].

The main goal of this work was to understand, formally, the benefits of memory safety for informal and partial reasoning, and to evaluate a variety of weakened forms of memory safety in terms of which reasoning principles they preserve. However, our approach may also suggest ways in which formal verification using proof assistants might be improved. One promising idea is to leverage the guarantees of memory safety to obtain formal proofs of program correctness modulo unverified code that could have errors, in contexts where complete verification is too expensive or not possible (e.g., for programs with a plugin mechanism).

Acknowledgments

We are grateful to Antal Spector-Zabik, Greg Morrisett, Justin Hsu, Michael Hicks, Yannis Juglaret, and Andrew Tolmach for useful suggestions on earlier drafts. This work is supported by NSF grants Micro-Policies (1513854) and DeepSpec (1521523) and ERC Starting Grant SECOMP (715753).

Appendix

This appendix defines the language of §2 more formally. Figure 4 summarizes the syntax of programs and repeats the definition of program states. The syntax is standard for a simple imperative language with pointers.

Figure 5 defines expression evaluation, \( [e] : \mathcal{S} \rightarrow \mathcal{V} \). Variables are looked up in the local-variable part of the state (for simplicity, heap cells cannot be dereferenced in expressions; the command \( x \leftarrow [e] \) puts the value of a heap cell in a local variable). Constants (booleans, numbers, and the special value nil used to simplify error propagation) evaluate to themselves. Addition and subtraction can be

\[
\oplus ::= + \mid \times \mid \cdot \mid \div \mid \leq \mid \text{and} \mid \text{or} \quad \text{(operators)}
\]

\[
e ::= \text{var} \mid \text{enum} \mid \text{bool} \mid \text{number} \mid \text{null} \quad \text{(expressions)}
\]

\[
c ::= \text{skip} \mid c_1 \mid c_2 \quad \text{(commands)}
\]

\[
\begin{align*}
\mathcal{S} & \triangleq \mathcal{T} \bowtie \mathcal{M} \quad \text{(states)} \\
\mathcal{T} & \triangleq \mathcal{V} \longrightarrow \mathcal{V} \quad \text{(local stores)} \\
\mathcal{M} & \triangleq \mathcal{I} \times \mathcal{I} \quad \text{(heaps)} \\
\mathcal{V} & \triangleq \mathcal{Z} \cup \mathcal{B} \cup \{\text{null}\} \cup \mathcal{I} \times \mathcal{Z} \quad \text{(values)} \\
\mathcal{O} & \triangleq \mathcal{S} \cup \{\text{error}\} \quad \text{(outcomes)}
\end{align*}
\]

\[
\begin{align*}
[x](i, m) & \triangleq \begin{cases} 
  l(x) & \text{if } x \in \text{dom}(l) \\
  \text{nil} & \text{otherwise}
\end{cases} \\
[b](s) & \triangleq b \\
[n](s) & \triangleq n \\
[\text{nil}](s) & \triangleq \text{nil}
\end{align*}
\]

\[
\begin{align*}
[e_1 + e_2](s) & \triangleq \begin{cases} 
  n_1 + n_2 & \text{if } [e_1](s) = n_1 \text{ and } [e_2](s) = n_2 \\
  (i, n_1 + n_2) & \text{if } [e_1](s) = (i, n_1) \text{ and } [e_2](s) = (i, n_2) \\
  & \text{or } [e_1](s) = n_1 \text{ and } [e_2](s) = (i, n_2) \\
  \text{nil} & \text{otherwise}
\end{cases} \\
[e_1 - e_2](s) & \triangleq \begin{cases} 
  n_1 - n_2 & \text{if } [e_1](s) = n_1 \text{ and } [e_2](s) = n_2 \\
  (i, n_1 - n_2) & \text{if } [e_1](s) = (i, n_1) \text{ and } [e_2](s) = (i, n_2) \\
  \text{nil} & \text{otherwise}
\end{cases} \\
[e_1 \times e_2](s) & \triangleq \begin{cases} 
  n_1 \times n_2 & \text{if } [e_1](s) = n_1 \text{ and } [e_2](s) = n_2 \\
  \text{nil} & \text{otherwise}
\end{cases} \\
e_1 = e_2(s) & \triangleq ([e_1](s) = [e_2](s)) \\
e_1 \leq e_2(s) & \triangleq \begin{cases} 
  n_1 \leq n_2 & \text{if } [e_1](s) = n_1 \text{ and } [e_2](s) = n_2 \\
  \text{nil} & \text{otherwise}
\end{cases} \\
e_1 \text{ and } e_2(s) & \triangleq \begin{cases} 
  b_1 \land b_2 & \text{if } [e_1](s) = b_1 \text{ and } [e_2](s) = b_2 \\
  \text{nil} & \text{otherwise}
\end{cases} \\
e_1 \text{ or } e_2(s) & \triangleq \begin{cases} 
  b_1 \lor b_2 & \text{if } [e_1](s) = b_1 \text{ and } [e_2](s) = b_2 \\
  \text{nil} & \text{otherwise}
\end{cases} \\
[\text{not } e](s) & \triangleq \begin{cases} 
  \neg b & \text{if } [e](s) = b \\
  \text{nil} & \text{otherwise}
\end{cases} \\
[\text{offset } e](s) & \triangleq \begin{cases} 
  n & \text{if } [e](s) = (i, n) \\
  \text{nil} & \text{otherwise}
\end{cases}
\]

Figure 4. Syntax and program states

Figure 5. Expression evaluation
bound\((f, \perp)\) ≜ \perp

bound\((f, \text{error})\) ≜ \text{error}

\[
\text{bound}(f,(I,l,m)) ≜ \begin{cases}
(I \cup I', l', m') & \text{if } f(I, m) = (I', l', m')
\text{error} & \text{if } f(I, m) = \text{error}
\perp & \text{otherwise}
\end{cases}
\]

if\((b, x, y)\) ≜ \begin{cases}
x & \text{if } b = \text{true}
y & \text{if } b = \text{false}
\text{error} & \text{otherwise}
\end{cases}

Figure 6. Auxiliary operators bind and if

applied both to numbers and to combinations of numbers and pointers (for pointer arithmetic); multiplication only works on numbers. Equality is allowed both on pointers and on numbers. Pointer equality compares both the block identifier and its offset, and while this is harder to implement in practice than just comparing physical addresses, this is needed for not leaking information about pointers (see §4.2). The special expression offset extracts the offset component of a pointer; we introduce it to illustrate that for satisfying our memory characterization pointer offsets do not need to be hidden (as opposed to block identifiers). The less-than-or-equal operator only applies to numbers—in particular, pointers cannot be compared. However, since we can extract pointer offsets, we can compare those instead.

The definition of command evaluation employs an auxiliary partial function that computes the result of evaluating a program along with the set of block identifiers that were allocated during evaluation. Formally, \([c]: S \rightarrow O_+\), where \(O_+\) is an extended set of outcomes defined as \(\mathcal{P}_{\text{fin}}(l) \times S \cup \{\text{error}\}\). We then set

\[
[c](l,m) = \begin{cases}
(l', m') & \text{if } [c]_+(l,m) = (l', m')
\text{error} & \text{if } [c]_+(l,m) = \text{error}
\perp & \text{otherwise}
\end{cases}
\]

finalids\((l,m)\) = \(\text{ids}(l,m) \setminus I\) if \([c]_+(l,m) = (I', l', m')\) otherwise

To define \([c]_+\), we first endow the set \(S \rightarrow O_+\) with the partial order of program approximation:

\[
f \subseteq g \triangleq \forall s, f(s) \neq \perp \Rightarrow f(x) = g(x)
\]

This allows us to define the semantics of iteration (the rule for while \(e\) do \(c\) end) in a standard way using the Kleene fixed point fix operator.

The definition of \([c]_+\) appears in Figure 7 where several of the rules use a bind operator (Figure 6) to manage the “plumbing” of the sets of allocated block ids between the evaluation of one subcommand and the next. The rules for if and while also use an auxiliary operator if (also defined in Figure 6) that turns non-boolean guards into errors.

The evaluation rules for skip, sequencing, conditionals, while, and assignment are standard. The rule for heap lookup, \(x \leftarrow [e]\), evaluates \(e\) to a pointer and then looks it up in the heap, yielding an error if \(e\) does not evaluate to a pointer or if it evaluates to a pointer that is invalid, either because its block id is not allocated or because its offset is out of bounds. Similarly, the heap mutation command, \([e_1] \leftarrow e_2\), requires \(e_1\) evaluate to a pointer that is valid in the current memory \(m\) (i.e., such that looking it up in \(m\) yields something other than \(\perp\)). The allocation command \(x \leftarrow \text{alloc}(e)\) first evaluates \(e\) to an integer \(n\), then calculates the next free block id for the current machine state (\(\text{fresh}(\text{ids}(l,m))\)); it yields a new machine state where \(x\) points to the first cell in the new block and where a new block of \(n\) cells is added the heap, all initialized to 0. Finally, \(\text{free}(e)\) evaluates \(e\) to a pointer and yields a new heap where every cell sharing the same block id as this pointer is undefined.

References

[1] Caja. Attack vectors for privilege escalation. 2012.
[2] M. Abadi and G. D. Plotkin. On protection by layout randomization. ACM TISSEC. 15(2):8, 2012.
[3] P. Agten, B. Jacobs, and F. Piessens. Sound modular verification of C code executing in an unverified context. POPL. 2015.
[4] A. Ahmed, M. Fluett, and G. Morrisett. L_3: A linear language with locations. Fundam. Inform., 77(4):397–449, 2007.
[5] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. ESORICS. 2008.
[6] A. Azevedo de Amorim, M. Dénès, N. Giannarakis, C. Hrîtu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies: Formally verified, tag-based security monitors. Oakland S&P. 2015.
[7] M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. Oakland S&P. 2009.
[8] T. Balabonski, F. Pottier, and J. Protzenko. Type soundness and race freedom for Mezzo. FLOPS. 2014.
[9] N. Benton and N. Tabareau. Compiling functional types to relational specifications for low level imperative code. In A. Kennedy and A. Ahmed, editors, TLDI. 2009.
[10] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Defensive javascript - building and verifying secure web components. In A. Al-dini, J. Lopez, and F. Martinelli, editors, Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures. 2013.
[11] D. Chisnall, C. Rothwell, R. N. M. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roc, B. Davis, and P. G. Neumann. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. ASPLOS. 2015.
[12] J. A. Clause, I. Doudalis, A. Orso, and M. Pruvolovic. Effective memory protection using dynamic tainting. ASE. 2007.
[13] A. A. de Amorim, N. Collins, A. DeHon, D. Demange, C. Hrîtu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. Journal of Computer Security, 24(6):689–734, 2016.
[14] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hard-Bound. Architectural support for spatial safety of the C programming language. ASPLOS. 2008.
[15] D. Deviere, F. Piessens, and L. Birkedal. Reasoning about object capabilities with logical relations and effect parametricity. EuroS&P. 2016.
[16] U. Dhawan, C. Hrîtu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon. Architectural support for software-defined metadata processing. ASPLOS. 2015.
\[\text{[skip]}_+(l, m) \triangleq (0, l, m)\]
\[\text{[c}_1; \text{c}_2]_+(l, m) \triangleq \text{bind}([\text{c}_2]_+, [\text{c}_1]_+(l, m))\]
\[\text{[if } e \text{ then } c_1 \text{ else } c_2]_+(l, m) \triangleq \text{if}(\text{[f]}(l, m), [\text{c}_1]_+(l, m), [\text{c}_2]_+(l, m))\]
\[\text{[while } e \text{ do } c \text{ end}]_+ \triangleq \text{fix}(\Lambda f (l, m). ([\text{f]}(l, m), \text{bind}([\text{c}]_+, f(l, m)), (0, l, m)))\]
\[\text{[x } \leftarrow e\]_+(l, m) \triangleq (0, l, [x \mapsto \text{[e]}(l, m)], m)\]
\[\text{[x } \leftarrow [e]\]_+(s) \triangleq \begin{cases} (0, l, m[(i, n) \mapsto \text{[e]}_2(l, m)]) & \text{if } \text{[e]}_1(s) = (i, n) \text{ and } m(i, n) = v \\ \text{error} & \text{otherwise} \end{cases}\]
\[\text{[x } \leftarrow \text{alloc}(e\]_+(l, m) \triangleq \begin{cases} (i, l[x \mapsto (0, 0), m[i, k) \mapsto 0 | 0 \leq k < n]) & \text{if } \text{[e]}(l, m) = n \text{ and } i = \text{fresh}[\text{[ids]}(l, m)] \\ \text{error} & \text{otherwise} \end{cases}\]
\[\text{[free(}e\]_+(l, m) \triangleq \begin{cases} (0, l, m[i, k) \rightarrow \perp | k \in \mathbb{Z}) & \text{if } \text{[e]}(l, m) = (i, 0) \text{ and } m(i, n) \neq \perp \text{ for some } n \\ \text{error} & \text{otherwise} \end{cases}\]

Figure 7. Command evaluation with explicit allocation sets

[17] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of heartbleed. IMC. 2014.

[18] T. Elliott, L. Fike, S. Winwood, P. C. Hickey, J. Bielman, J. Sharp, E. L. Seidel, and J. Lauchbury. Gilt free Ivory Haskell. 2015.

[19] C. Fournet, N. Swamy, J. Chen, P.-É. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript POPL. 2013.

[20] J. A. Goguen and J. Meseguer. Security policies and security models In Symposium on Security and Privacy, 1982.

[21] M. Hicks. What is memory safety? [http://www.pl-enthusiast.net/2014/07/21/memory-safety/]. 2014.

[22] ISO. ISO C standard 1999. Technical report, ISO, 1999. ISO/IEC 9899:1999 draft.

[23] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints Oakland S&P. 2012.

[24] Y. Juglare, C. Hritcu, A. Azevedo de Amorim, B. Eng, and B. C. Pierce. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation ESORICS. 2016.

[25] J. Kang, C. Hur, W. Mansky, D. Garbusov, S. Zdancewic, and V. Vafeiadis. A formal C memory model supporting integer-pointer casts. PLDI. 2015.

[26] R. Krebbers. The C standard formalized in Coq PhD thesis, Radboud University Nijmegen, 2015.

[27] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security CCS. 2013.

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

[29] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications Oakland S&P. 2010.

[30] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell. Into the depths of C: elaborating the de facto standards PLDI. 2016.

[31] L. A. Meyerovich and V. B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. Oakland S&P, 2010.

[32] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management FPCA. 1995.

[33] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. Soft-Bound: highly compatible and complete spatial memory safety for C PLDI. 2009.

[34] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C JMM. 2010.

[35] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weaver. CCured: type-safe retrofitting of legacy software ACM Trans. Program. Lang. Syst., 27(3):477–526, 2005.

[36] A. M. Pitts. Nominal logic, a first order theory of names and binding. Information and Computation, 186:2003, 2002.

[37] A. M. Pitts. Nominal Sets: Names and Symmetry in Computer Science. Cambridge University Press, New York, NY, USA, 2013.

[38] F. Pottier and J. Potztenko. Programming with permissions in Mezzo. ICFP, 2013.

[39] J. C. Reynolds. Separation logic: A logic for shared mutable data structures LICS. 2002.

[40] The Rust programming language. [http://www.rust-lang.org]. 2017.

[41] C. Schlesinger, K. Pattabiraman, N. Swamy, D. Walker, and B. G. Mazi`eres. Eliminating cache-based timing attacks with instruction-based scheduling ESORICS. 2013.

[42] D. Swasey, D. Garg, and D. Dreyer. Robust and compositional verification of object capability patterns To appear at OOPSLA, 2017.
[46] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. *IEEE S&P*. 2013.

[47] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated analysis of security-critical javascript apis. *Oakland S&P*. 2011.

[48] A. Turon. Rust: from POPL to practice (keynote). *POPL*. 2017.

[49] C. Williams. Oracle’s Larry Ellison claims his Sparc M7 chip is hacker-proof – errr... *The Register*, 2015.

[50] E. Z. Yang and D. Mazières. Dynamic space limits for Haskell. *PLDI*. 2014.

[51] H. Yang and P. W. O’Hearn. A semantic basis for local reasoning. *FoSSaCS*. 2002.

[52] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. *PLDI*. 2012.