The Meaning of Memory Safety

Arthur Azevedo de Amorim
University of Pennsylvania

Cătălin Hriţcu
Inria Paris

Benjamin C. Pierce
University of Pennsylvania

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.

CCS CONCEPTS

• **Security and privacy** → *Formal security models*; *Logic and verification*; *Information flow control*;

KEYWORDS

memory safety, formal security definition, separation logic

1 INTRODUCTION

Memory safety—and the plethora of catastrophic vulnerabilities that arise in its absence [38]—are common concerns among system designers. But what *is* memory safety, exactly? Intuitions abound, but translating them into satisfying formal definitions is surprisingly difficult [18].

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 *mis*use—buffer overruns, double frees, etc. Characterizing memory safety in terms of the absence of these errors is tempting, but it falls short for two reasons. First, there is often disagreement on what program behaviors qualify as errors. For example, many real-world C programs intentionally rely on unrestricted pointer arithmetic [25], although it may yield undefined behavior according to the language standard [20, §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 may 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* [35], 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 its 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*

$$\frac{\{p\}\ c\ \{q\}}{\{p * r\}\ c\ \{q * r\}},$$

where $p * r$ 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, if 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 $\{\mathsf{true}\}$, 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.

A crucial feature of standard separation logic is that safety is enforced *manually*: a program correctness proof must check that each individual 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, comprehensive manual verification of the safety of every memory access may be prohibitively expensive for large codebases and is sometimes even impossible—for instance, if our jpeg decoder comes from a third-party library whose source code is unavailable, then we cannot use separation logic to prove *any* nontrivial specification of a program that calls it.

In a setting where memory safety is enforced *automatically*, we can do much better. For example, if the decoder is a Java library, 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, whether or not we can see its source code. A straightforward 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 idealized notion of memory safety, using a concrete imperative language as the basis of our discussion (introduced in Section 2). We show three frame theorems (Theorems 3.2.1, 3.2.3 and 3.2.4) 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 (Theorem 3.3.1), 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 Section 3.4, we discuss how these results yield a variant of the original frame rule of separation logic (Theorem 3.4.2). Both 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 technical results have been verified with the Coq proof assistant [40]; these machine checked proofs are available at: https://github.com/arthuraa/memory-safe-language

Our *second contribution* (Section 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 most 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.

We discuss related work on memory safety and reasoning principles for similar enforcement mechanisms in Section 5, and conclude in Section 6.

There is already a substantial literature on formal characterizations of memory safety, but we believe that ours is the first definition phrased in terms of reasoning principles that are valid when memory safety is enforced automatically, and that does not rely on additional mechanisms such as full-blown capabilities, closures, objects, etc. Since languages that offer strong reasoning principles and security guarantees tend to rely on some form of memory safety, we could see our principles as a common core that underpins all of these systems. Conversely, we hope that the reasoning principles identified here can also serve as good criteria for evaluating mechanisms for enforcing memory safety.

2 AN IDEALIZED MEMORY-SAFE LANGUAGE

Our discussion of memory safety is based on 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 that our characterization does not depend fundamentally on 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 turned on might bring high performance costs; and specialized type systems for managing memory [17, 36] are more complex.

We begin by giving a brief overview of the language and its semantics. In Section 3, we will explore the reasoning principles it supports.

2.1 Language Overview

The syntax is summarized in Figure 1. Expressions e include variables $x \in \mathsf{var}$, numbers $n \in \mathbb{Z}$, Booleans $b \in \mathbb{B}$, an invalid pointer nil, and various operations, both binary (arithmetic, logic, etc.) and unary (extracting the offset of a pointer). The form $[e]$ represents an access to the pointer denoted by 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

Command	Description
$x \leftarrow e$	Assign to local variable
$x \leftarrow [e]$	Read from heap location
$[e_1] \leftarrow e_2$	Assign to heap location
$x \leftarrow \mathsf{alloc}(e)$	Allocate heap block of size e
$\mathsf{free}(e)$	Free block starting at e
skip	Do nothing
if e then c_1 else c_2	Conditional
while e do c end	Loop
$c_1; c_2$	Sequencing

Figure 1: Syntax

$$
\begin{aligned}
s \in \mathcal{S} &\triangleq \mathcal{L} \times \mathcal{M} && \text{(states)}\\
l \in \mathcal{L} &\triangleq \mathsf{var} \rightharpoonup_{\text{fin}} \mathcal{V} && \text{(local stores)}\\
m \in \mathcal{M} &\triangleq \mathbb{I} \times \mathbb{Z} \rightharpoonup_{\text{fin}} \mathcal{V} && \text{(heaps)}\\
v \in \mathcal{V} &\triangleq \mathbb{Z} \uplus \mathbb{B} \uplus \{\mathsf{nil}\} \uplus \mathbb{I} \times \mathbb{Z} && \text{(values)}\\
O &\triangleq \mathcal{S} \uplus \{\mathsf{error}\} && \text{(outcomes)}
\end{aligned}
$$

$\mathbb{I} \triangleq$ some countably infinite set

$X \rightharpoonup_{\text{fin}} Y \triangleq$ partial functions $X \rightharpoonup Y$ with finite domain

Figure 2: Program states and values.

offset $n \in \mathbb{Z}$. The offset is always 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.) This structured memory model is widely used in the literature, as in the CompCert verified C compiler [23], for instance.

We write $[\![c]\!](s)$ to denote the outcome of running a program c on an initial state s; this outcome can be either a final state s', representing successful termination, or a 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 Appendix A; 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 effect of various kinds of memory misuse by treating them as run-time errors that immediately 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 *temporal* and *spatial* 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 created at allocation time, where they are chosen to be fresh with 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:

$$
[\![e_1 + e_2]\!](s) \triangleq \begin{cases} n_1 + n_2 & \text{if } [\![e_i]\!](s) = n_i \\ (i, n_1 + n_2) & \text{if } [\![e_1]\!](s) = (i, n_1) \\ & \text{and } [\![e_2]\!](s) = n_2 \\ \mathsf{nil} & \text{otherwise} \end{cases}
$$

For simplicity, we stipulate that nonsensical combinations, such adding two pointers together, simply result in the nil value. A real implementation might represent block identifiers with hardware tags and use a monotonically increasing counter to assign identifiers to new blocks [5]; if enough tags are available, we can assume that every identifier will be fresh.

Pointers Cannot be Observed. Because of the freshness condition mentioned above, in the abstract semantics of our language block identifiers can reveal information about the entire program state. For example, if identifiers are chosen according to a monotonic counter, and the language allowed inspecting identifiers as integers, then code could learn how many blocks have been allocated during the execution of the program just by allocating a new one. A concrete implementation would encounter similar issues related to the choice of physical addresses where a new blocks are allocated. Such side channels constitute a clear violation of locality; thus, our language does not offer any means for programs to extract this information. (Such issues are commonplace in systems that combine dynamic allocation and information-flow control [10].) We discuss the consequences of allowing programs to extract this information in Section 4.2.

Note that we don't have to prevent programs from extracting *all* the information associated with a pointer: besides using pointers to load from memory, programs can also safely extract their offsets and compare pointers for equality. Our Coq development also shows that it would be possible for programs to soundly compute the size of a memory block, although for simplicity we do not discuss this feature in the paper.

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 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 Section 4.3.

3 REASONING WITH MEMORY SAFETY

With the 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* (3.2.1, 3.2.3, and 3.2.4) describes how the execution of a piece of code is affected by extending the initial state it runs on. These in turn imply a noninterference property, Theorem 3.3.1, guaranteeing that program execution is independent of memory regions it cannot access—that is, those that correspond to block identifiers that that a piece of code does not possess. Later, in Section 3.4, 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 3.4.2).

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

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

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

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

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

$$\begin{aligned} \mathsf{ids}(l, m) \triangleq\ & \mathsf{blocks}(l, m) \\ & \cup \{i \mid \exists x, n, l(x) = (i, n)\} \\ & \cup \{i \mid \exists p, n, m(p) = (i, n)\} \end{aligned}$$

We write $\mathsf{vars}(s)$ for the set of local variables defined in s:

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

We write $\mathsf{fv}(c)$ 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 π^{-1}; that is,

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

Given a permutation π 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 π. The definition of this renaming operation is straightforward.[1]

Finally, we write $X \# Y$ to indicate that sets X and Y are disjoint: $X \cap Y = \emptyset$. When $X = \mathsf{supp}(\pi)$, this simply means that $\pi(i) = i$ for all $i \in Y$.

[1] It can be derived formally by viewing $\mathcal{S}$ as a *nominal set* over $\mathbb{I}$ [31, 32] obtained by combining products, disjoint unions, and partial functions.

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 3.2.1 (Frame OK). *Suppose that* $[\![c]\!](s_1) = s_1'$, *that* $\mathsf{fv}(c) \subseteq \mathsf{vars}(s_1)$, *and that* $\mathsf{blocks}(s_1) \# \mathsf{blocks}(s_2)$. *Then there exists a permutation* π *such that*

$$[\![c]\!](s_1 \cup s_2) = \pi \cdot s_1' \cup s_2 \qquad \mathsf{blocks}(\pi \cdot s_1') \# \mathsf{blocks}(s_2).$$

The second premise, $\mathsf{fv}(c) \subseteq \mathsf{vars}(s_1)$, 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 that initial state with s_2. The third premise, $\mathsf{blocks}(s_1) \# \mathsf{blocks}(s_2)$, means that s_1 and s_2 define disjoint parts of the heap. Finally, the conclusion of the theorem intuitively states (1) that the execution of c does not affect the extra state s_2 at all and (2) that the rest of the result is almost the same as s_1', except for a permutation of block identifiers. (This permutation is needed to avoid clashes between the block identifiers present in s_2 and those chosen during the evaluation of c on s_1 when allocating new blocks.)

The proof of this theorem relies crucially on the facts that programs cannot inspect pointer values and that memory can grow indefinitely (a common assumption in formal models of memory). 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 π, we obtain the same outcome, up to some additional permutation π' that again accounts for different choices of fresh block ids.

Theorem 3.2.2 (Renaming states). *Let* s *be a state,* c *a command, and* π *a permutation. Then there exists* π' *such that:*

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

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 3.2.3 (Frame Loop). *Suppose* $[\![c]\!](s_1) = \bot$, *that* $\mathsf{fv}(c) \subseteq \mathsf{vars}(s_1)$, *and that* $\mathsf{blocks}(s_1) \# \mathsf{blocks}(s_2)$. *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. 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 ($\mathsf{ids}(s_1) \# \mathsf{blocks}(s_2)$) preventing any pointers in the original state s_1 from referencing the new blocks in s_2. Notice that, since $\mathsf{blocks}(s) \subseteq \mathsf{ids}(s)$, this premise subsumes the similar ones in the preceding results.

Theorem 3.2.4 (Frame Error). *Suppose that* $[\![c]\!](s_1) = \mathsf{error}$, $\mathsf{fv}(c) \subseteq \mathsf{vars}(s_1)$, *and* $\mathsf{ids}(s_1) \# \mathsf{blocks}(s_2)$. *Then* $[\![c]\!](s_1 \cup s_2) = \mathsf{error}$.

It is this last frame theorem that requires that pointers be unforgeable and that every memory operation check for pointer validity before running. Along with the stronger premise $\mathsf{ids}(s_1) \# \mathsf{blocks}(s_2)$, these features guarantee that the execution of c will fail even when s_2 is added. It would still be possible to prove the first two frame theorems in a weaker setting where these checks are not present (see Section 4).

3.3 Memory Safety and Noninterference

We now turn to our noninterference property, which guarantees that 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 3.3.1 (Noninterference). *Let s_1, s_{21}, s_{22}, and s' be program states and c a program. Suppose that $\mathsf{fv}(c) \subseteq \mathsf{vars}(s_1)$, that $\mathsf{ids}(s_1) \# \mathsf{blocks}(s_{21})$ and $\mathsf{blocks}(s_1) \# \mathsf{blocks}(s_{22})$, and that $[\![c]\!](s_1 \cup s_{21}) = s'$. Then there is a state s_1' and permutations π_1 and π_2 such that*

$$\begin{aligned} s' &= \pi_1 \cdot s_1' \cup s_{21} \\ [\![c]\!](s_1) &= s_1' && \mathsf{blocks}(\pi_1 \cdot s_1) \# \mathsf{blocks}(s_{21}) \\ [\![c]\!](s_1 \cup s_{22}) &= \pi_2 \cdot s_1' \cup s_{22} && \mathsf{blocks}(\pi_2 \cdot s_1) \# \mathsf{blocks}(s_{22}). \end{aligned}$$

Because of the form of s', we see that the program indeed cannot affect the contents of s_{21}; furthermore, having the unreachable part of the initial state be s_{21} or s_{22} can only affect the choice for new identifiers in the reachable portion of the result (that is, $\pi_1 \cdot s_1'$ or $\pi_2 \cdot s_1'$), and nothing else.

Proof. Consider the result of executing c on s_1. It can't be the case that $[\![c]\!](s_1) = \bot$ or that $[\![c]\!](s_1) = \mathsf{error}$, because otherwise we would be able to apply Theorems 3.2.3 and 3.2.4 to conclude that $[\![c]\!](s_1 \cup s_{21}) = \bot$ or that $[\![c]\!](s_1 \cup s_{21}) = \mathsf{error}$, contradicting the hypothesis that $[\![c]\!](s_1 \cup s_{21}) = s'$. Therefore, there must be a state s_1' such that $[\![c]\!](s_1) = s_1'$. Thanks to Theorem 3.2.1, we can conclude that $[\![c]\!](s_1 \cup s_{2i}) = \pi_i \cdot s_1' \cup s_{2i}$ for a pair of permutations π_1 and π_2, and hence that $s' = \pi_1 \cdot s_1' \cup s_{21}$. □

3.4 Memory Safety and Separation Logic

We now explore the close connection between the principles identified above 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 $\mathcal{S}$). For our language, the meaning of such a $\{p\}\ c\ \{q\}$ specification could be roughly stated as

$$\forall s \in p, \mathsf{fv}(c) \subseteq \mathsf{vars}(s) \Rightarrow [\![c]\!](s) \in q \cup \{\bot\}.$$

That is, if we start c in a state that satisfies 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 3.2.1 and 3.2.3. 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.[2]

Theorem 3.4.1. *Let p, q, and r be predicates over states and c be a command. Suppose that*

$$\forall l_1\ l_2\ m, (\forall x \notin \mathsf{modvars}(c),\ l_1(x) = l_2(x)) \Rightarrow (l_1, m) \in r \Rightarrow (l_2, m) \in r,$$

where $\mathsf{modvars}(c)$ 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

$$\frac{\{p\}\ c\ \{q\}}{\{p * r\}\ c\ \{q * r\}}\ \textsc{Frame}$$

where $p * r$ denotes the separating conjunction *of p and r—the predicate over states defined as*

$$\{(l, m_1 \cup m_2) \mid (l, m_1) \in p, (l, m_2) \in r, \mathsf{blocks}(l, m_1) \# \mathsf{blocks}(l, m_2)\}.$$

The definition of separation-logic specifications above requires that either the program execution terminates successfully (and satisfies the postcondition) or it diverges, which require separation logic proofs to completely exclude memory errors. 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:

$$\frac{}{\{p\}\ c\ \{\mathsf{true}\}}\ \textsc{Taut}$$

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 a memory-safe language—in which errors have a sensible, predictable semantics, as opposed to wild undefined behavior—we can formulate a variant of separation logic that allows looser specifications. We now consider specifications of the form $\{p\}\ c\ \{q\}_e$, defined as

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

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

$$\frac{\{p\}\ c\ \{q\}}{\{p\}\ c\ \{q\}_e}$$

Because errors are no longer prevented, the modified Taut rule

$$\frac{}{\{p\}\ c\ \{\mathsf{true}\}_e}\ \textsc{Taut}$$

becomes sound, since the true postcondition now means that any outcome whatsoever is acceptable. Unfortunately, there is a price to pay for allowing errors: they compromise the soundness of the

[2] Technically, the proof of the frame rule requires a slightly stronger notion of specification than the one defined above, accounting for permutations of allocated identifiers; our Coq development has a more precise statement.

frame rule. The reason, intuitively, is that preventing run-time errors has an additional purpose in separation logic: it forces programs to act locally—that is, to access only the parts of the heap that are described by its pre- and postconditions. To see why, consider a program c defined as $x \leftarrow [y]$. This program clearly yields an error when run on an empty heap, implying that the triple

$$\{\mathsf{emp}\}\ c\ \{x = 0\}_e$$

is valid, where the predicate emp holds of any state with an empty heap and $x = 0$ holds of states whose local store maps x to 0. Now consider what happens if we try to apply an analog of the frame rule to this triple using the frame predicate $y \mapsto 1$, which holds in states where y holds a pointer to the unique defined location on the heap, which stores the value 1. After some simplification, we arrive at the specification

$$\{y \mapsto 1\}\ c\ \{x = 0 \wedge y \mapsto 1\}_e,$$

which clearly does not hold, since executing c on a state satisfying the precondition leads to a successful final state mapping x to 1.

To salvage the frame rule, we need to adapt it to take errors into account. Fortunately, the reachability properties of memory safety provide a solution: instead of enforcing locality by preventing errors, we can take advantage of the fact that memory operations in a memory-safe language are automatically local—in particular, local to the block identifiers that the program possesses.

Theorem 3.4.2. *Under the same assumptions as Theorem 3.4.1, the following rule is sound*

$$\frac{\{p\}\ c\ \{q\}_e}{\{p \triangleright r\}\ c\ \{q \triangleright r\}_e}\ \textit{SafeFrame}$$

where $p \triangleright r$ denotes the isolating conjunction *of p and r, defined as*

$$\{(l, m_1 \cup m_2) \mid (l, m_1) \in p, (l, m_2) \in r, \\ \mathsf{ids}(l, m_1)\ \#\ \mathsf{blocks}(l, m_2)\}.$$

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 3.2.4 in addition to Theorems 3.2.1 and 3.2.3 (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 Section 3.2 can themselves be considered as a form of local reasoning: to reason about a program, it suffices to restrict attention to the parts of the state that the program 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 Section 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 [14].Although the code fragment that enabled that attack was essentially just manipulating an array, an out-of-bounds read from that array was enough to leak private data from completely unrelated parts of the program state. This illustrates how programmers can be fooled into incorrect local reasoning about code in a memory-unsafe language and thus form incorrect conclusions about its possible effects, leading to a serious vulnerability.

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. Some systems allow programmers to convert pointers to integers or test the equality of physical pointer addresses; some do not prevent accessing arrays without checking bounds; and some do not prevent using pointers after they are freed—relaxations that are often made for improved programming flexibility or for performance. In languages that generally aim to be memory safe, such devices are generally seen as trading relatively benign glimpses of the language's underlying implementation details (such as being able to read the previous contents of uninitialized memory or compare pointers with $\leq$) in return for significant performance 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 [29], for example, adds spatial memory-safety checks for C programs, but does not provide protection against bugs caused by erroneous uses of free.

In this section, we enumerate some common relaxed models of memory safety and evaluate how they affect the reasoning principles and security guarantees of Section 3. These relaxations fall into two rough categories: those that give up on some secrecy guarantees while maintaining integrity ones, and those that give up on both secrecy and integrity.However, this distinction should be taken with a grain of salt, since there are cases where integrity depends on secrecy in practice; for example, if a system grants privileges to change the state of some component when accessed with the right password, then 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 local reasoning about memory becomes impossible in general, 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 (among many other serious attacks [1, 16, 26, 39]). 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 [1, 8, 16].

4.2 Observing Pointers

The language of Section 2 maintains a complete separation between pointers and other kinds of values. In reality, this separation is often only enforced in one direction. For example, some tools for enforcing memory safety of C programs [11, 29] allow pointer-to-integer casts, a feature required by many low-level C idioms [9, 25]; additionally, languages considered as memory safe often include features that break this separation—e.g., the default implementation of `hashCode()` in Java, or OCaml's unsafe `Obj.magic` primitive. To model such features, we can extend the syntax of expressions with a cast operator

$$e ::= \cdots \mid \mathsf{cast}(e) \mid \cdots$$

and assume that we have some function $[\![\mathsf{cast}]\!] : \mathbb{I} \times \mathbb{Z} \to \mathbb{Z}$ for converting a pointer to an integer, which we use to define the semantics of cast:

$$[\![\mathsf{cast}(e)]\!](s) = [\![\mathsf{cast}]\!]([\![e]\!](s)) \qquad \text{if } [\![e]\!](s) \in \mathbb{I} \times \mathbb{Z}$$

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 are relevant when defining the semantics of $[\![\mathsf{cast}]\!]$. The three frame theorems (3.2.1, 3.2.3, and 3.2.4) are thus lost, because the state of unreachable parts of the heap may influence integers observed by the program. An important consequence is that secrecy is not valid 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.

Theorem 4.2.1 (Integrity-only Noninterference). *Let s_1, s_2, and s' be states and c a command such that* $\mathsf{fv}(c) \subseteq \mathsf{vars}(s_1)$, $\mathsf{ids}(s_1) \# \mathsf{blocks}(s_2)$, *and* $[\![c]\!](s_1 \cup s_2) = s'$. *Then we can find* $s_1' \in \mathcal{S}$ *such that* $s' = s_1' \cup s_2$.

The crucial difference between this and the stronger noninterference result of Theorem 3.3.1 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 Section 1. 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. 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.

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, suppose that the language implemented pointer equality by comparing their physical addresses and ignoring their block identifiers—something we explicitly disallow in our language (as explained in Appendix A), but which is usual in efficient implementations. If the attacker knows 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 know 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,

$$y \leftarrow \mathsf{alloc}(1); \mathsf{if}\ x + 1729 = y\ \mathsf{then}\ \ldots\ \mathsf{else}\ \ldots,$$

to learn the location of the new pointer assigned to y, and thus infer something about the global state.

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 [11, 29]. Even languages that emphasize safety and clean semantics often allow access to uninitialized memory in some cases—e.g., OCaml's `Bytes.create` primitive, Node.js's `Buffer.allocUnsafe`.

As usual, 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 becomes even more severe if old pointers or other kinds of capabilities are allowed to occur in re-allocated memory (in a way that the program can use), since they can be used to access 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 facet of temporal memory safety 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 Section 3) and obtaining good performance. 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 from section 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. Systems targeted solely at spatial memory safety [11, 29] drop this requirement.

To see how the results of Section 3 are affected in such a setting, consider the program

$$x \leftarrow \mathrm{alloc}(1); z \leftarrow (y = x),$$

and suppose we run it on a state where y holds a dangling pointer—that is, an already freed pointer whose block id does not appear in the domain of the heap. Depending on the behavior of the allocator and the state of the memory, the pointer assigned to x could potentially be equal to y. Since this outcome depends on the entire state of the system, not just the reachable memory, Theorems 3.2.1, 3.2.3 and 3.2.4 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 from certain parts of the code. 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.

In practice, dangling pointers can have disastrous consequences for overall system security, independently of the freshness issues just described, freeing a pointer more than once can violate allocator invariants, enabling practical attacks [38].

4.5 Infinite Memory

In our idealized language, memory can grow indefinitely and allocation never fails, whereas in reality memory is a finite resource and allocation is therefore partial. This means that Theorem 3.2.1 can never hold in a real programming language as is, since executing a program in an extended state can cause it to run out of memory. However, it seems possible to weaken that result to hold in a more realistic setting. For instance, if the allocator we consider were partial, we could explicitly require, in addition to the other hypotheses of Theorem 3.2.1, that executing the program in the extended state does not result in an error. We might therefore hope to enforce secrecy and integrity modulo memory-exhaustion errors.

How this limitation affects security in practice will depend on the particular system under consideration. An adversary that can cause a system to allocate large regions of memory could launch a denial-of-service attack, compromising a system's availability. Furthermore, every memory-exhaustion error that is observed leaks one bit of information about the state of the entire system; depending on how such errors are handled, they could potentially be used by adversaries to learn about data that should remain hidden. In Java for instance, running out of memory triggers an `OutOfMemoryError` exception, which can be caught and handled without causing the program to terminate. This means memory exhaustion can be used as a fairly high-bandwidth channel.

This discussion suggests that, if such vulnerabilities are a real concern for a system, they need to be treated with special care. One approach would be to provide a mechanism to limit the amount of memory an untrusted component can allocate (as done for instance by Yang and Mazières [41]), 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 global denial-of-service attacks are prevented). A more speculative idea is to develop *quantitative* versions of the 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 Physics

Our crucial reality ignored by our model is that computers are physical artifacts: anything they compute affects (and can be affected by) the real world. Data stored in "unreachable" parts of the heap may have an effect on the cache, which in turn has an influence on the time a computation takes, allowing one to infer properties of the rest of the heap; observing the energy consumption of a virtual machine can reveal information about other supposedly isolated virtual machines running along it [19]; and fast, repeated reads of the same memory address can change the value stored in neighboring locations [21].

5 RELATED WORK

The present work lies at the intersection of two large bodies of past research: one on reasoning principles for programs, the other on characterizing memory safety formally. We review the most closely related work in these areas, focusing on four conceptual threads: (1) work that aims to characterize memory safety (or its absence) in terms of some form of abstract execution model, (2) work on principles for reasoning about memory-safe programs or languages, (3) formal definitions and proofs of memory safety for low-level languages with explicit memory management, and (4) reasoning principles for systems offering stronger guarantees than just memory safety, such as object capabilities.

Memory Safety as an Execution Model. In a popular blog post [18], Hicks characterized memory safety in terms of a relatively high-level execution model that ensures that pointers behaves as capabilities. Much of the formal work on memory safety defining execution models is similar in spirit. Work on formalizing the SoftBound [29] and CETS [28] transformation passes for LLVM proves that they

enforce an execution model that is aimed at capturing temporal and spatial memory safety. Languages such as Cyclone [17] and CCured [30] feature execution models that are similar in spirit, but that additionally rely on more sophisticated type properties than the ones enforced by the C type system; Rust [36] is a more recent design, and a corresponding formalization is underway. Mezzo [33] is a functional programming language with a richer type system, guaranteeing in particular the absence of data races from programs [6]. The PUMP architecture for hardware-accelerated metadata tag processing [13] supports a memory-safety monitor that implements a higher-level machine with a block-structured memory [5]. Finally, popular formalizations of the C language [22, 23] attempt to rigorously characterize sources of undefined behavior in C—in particular, instances of memory misuse that trigger such undefined behavior. All of these characterizations share many similarities with the imperative language studied in this paper, though they differ in that they do not try to analyze the local reasoning principles that arise by executing programs with an execution model that prevents memory safety violations.

Reasoning Principles for Memory Safety. Separation logic [35, 42] has been a significant 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 Section 3.4, our reasoning principles are reminiscent of the frame rule, but they take advantage of pointer reachability properties available in settings where memory safety is enforced automatically to guarantee access locality. Furthermore, we analyze how practically-motivated relaxations of memory safety affects these reasoning principles.

Other past work has also obtained guarantees similar to the frame rule. The soundness result for L^3 [4], a linear calculus featuring strong updates and aliasing control, validates frame-like reasoning principles. Yarra [37] is an extension of C that allows programmers to protect data structures by marking their types as *critical*, and thus to reason about their integrity. Agten *et al.* [3] showed how to strengthen the guarantees of separation logic in the presence of unverified code, by dynamically checking that the required separation-logic assertions are valid at its interface. These papers do not discuss secrecy guarantees and do not use reachability-based isolation to characterize memory safety.

Morrisett *et al.* [27] 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. 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.

Benton and Tabereau [7] show how to compile a simple higher-order language while validating a semantic interpretation of types. Their interpretation guarantees, among other things, that executing that code in some memory context preserves arbitrary invariants on that context. Furthermore, since the interpretation is *relational*, it can express both secrecy and integrity guarantees, which are similar to our noninterference result (even though our result is not formulated relationally) . One difference with our work is that they do not consider features that typically pose problems in non-memory-safe settings, such as manual memory management and array bounds checking.

Abadi and Plotkin [2] study the effectiveness of address randomization against attacks exploiting memory safety vulnerabilities by proving a full-abstraction result for a core calculus, guaranteeing that observationally equivalent programs in the calculus are equivalent with high probability when compiled. One difference between these approaches and ours is that they guarantee that *any* high-level reasoning principles that can be expressed in terms of observational equivalence are still available after applying a given transformation or technique, whereas we propose to study exactly which reasoning principles memory safety provides.

Memory Safety for Low-level Languages . Memory safety is a common concern in low-level systems and languages, leading to a variety of techniques for bringing its benefits to that setting. Approaches include: Cyclone [17], a language featuring a region-based type system for doing safe manual memory management; CCured [30], a program transformation that adds temporal memory safety to C by refining its pointer type to distinguish between various degrees of pointer safety; Ivory [15] an embedding of a similar "safe-C type system" into Haskell; SoftBound [29], an instrumentation technique for C programs for enforcing spatial memory safety, including the prevention of sub-object bounds violations; CETS [28], 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. These techniques explore different points of the design space, varying significantly in terms of the provided guarantees, applicability and performance impact. All of them include soundness proofs relating them to formal notions of memory safety.

Beyond Memory Safety. The principles studied in this paper show that we can provide quite strong guarantees when a state component is unreachable from a given piece of code. In many cases, unfortunately, this notion of separation is too strong: we seldom try to keep program components completely isolated, but try to limit how they are allowed to interact, forcing them to do so using interfaces, capabilities, inter-process communication, etc. Finding reasoning principles for such strong linguistic mechanisms has been one of the main problems in programming-languages research, and recent years have seen substantial advances in applications of those principles for security. Although some of these characterizations are similar (or quite stronger) to ours, the linguistic mechanisms they target are not as widespread as the ones we consider: general stateful references. For instance, Maffeis *et al.* [24] capture similar reachability and isolation-based reasoning principles for reasoning about capability systems with read and write access permissions; Devriese *et al.* [12] take advantage of stronger interfaces for controlling access to state components to provide a system of logical relations and a notion of effect parametricity that subsumes Maffeis *et al.*'s principles; and Rajani *et al.* [34] study the relationship between capability systems, access control, and confused-deputy attacks. Moreover, they devise a property characterizing memory safety, or to establish the same kind of contrast with the error-preventing behavior of separation logic (although the logical relations of Devriese *et al.* are very close in spirit to separation logic).

6 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 unsafe ones.

The language used in this paper has a simple storage model, with just global variables and flat, heap-allocated arrays. Realistic languages, 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 languages 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 [11, 29] add spatial memory safety to C while allowing programmers to *downgrade 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 model to take into account the reasoning enabled by this feature.

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 (for instance, for programs with a plugin mechanism).

ACKNOWLEDGMENTS

We are grateful to Antal Spector-Zabusky, 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).

A LANGUAGE DEFINITION

This appendix defines the language of Section 2 more formally. Figure 3 summarizes the syntax of programs and repeats the definition of program states. The syntax is standard for a simple imperative language with pointers.

Figure 4 defines expression evaluation, $[\![e]\!] : \mathcal{S} \to \mathcal{V}$. 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

$$
\begin{array}{rcll}
\oplus & ::= & + \mid \times \mid - \mid = \mid \leq \mid \text{and} \mid \text{or} & \text{(operators)} \\
e & ::= & x \in \text{var} \mid b \in \mathbb{B} \mid n \in \mathbb{Z} & \text{(expressions)} \\
 & \mid & e_1 \oplus e_2 \mid \text{not } e \mid \text{offset } e \mid \text{nil} & \\
c & ::= & \text{skip} \mid c_1; c_2 & \\
 & \mid & \text{if } e \text{ then } c_1 \text{ else } c_2 & \\
 & \mid & \text{while } e \text{ do } c \text{ end} & \\
 & \mid & x \leftarrow e \mid x \leftarrow [e] \mid [e_1] \leftarrow e_2 & \text{(commands)} \\
 & \mid & x \leftarrow \text{alloc}(e) \mid \text{free}(e) &
\end{array}
$$

$$
\begin{array}{rcl}
s \in \mathcal{S} & \triangleq & \mathcal{L} \times \mathcal{M} \quad \text{(states)} \\
l \in \mathcal{L} & \triangleq & \text{var} \rightharpoonup_{\text{fin}} \mathcal{V} \quad \text{(local stores)} \\
m \in \mathcal{M} & \triangleq & \mathbb{I} \times \mathbb{Z} \rightharpoonup_{\text{fin}} \mathcal{V} \quad \text{(heaps)} \\
v \in \mathcal{V} & \triangleq & \mathbb{Z} \uplus \mathbb{B} \uplus \{\text{nil}\} \uplus \mathbb{I} \times \mathbb{Z} \quad \text{(values)} \\
O & \triangleq & \mathcal{S} \uplus \{\text{error}\} \quad \text{(outcomes)}
\end{array}
$$

$\mathbb{I} \triangleq$ some countably infinite set

$X \rightharpoonup_{\text{fin}} Y \triangleq$ partial functions $X \rightharpoonup Y$ with finite domain

Figure 3: Syntax and program states

$$
\begin{array}{rcl}
[\![x]\!](l,m) & \triangleq & \begin{cases} l(x) & \text{if } x \in \text{dom}(l) \\ \text{nil} & \text{otherwise} \end{cases} \\
[\![b]\!](s) & \triangleq & b \\
[\![n]\!](s) & \triangleq & n \\
[\![\text{nil}]\!](s) & \triangleq & \text{nil} \\
[\![e_1 + e_2]\!](s) & \triangleq & \begin{cases} n_1 + n_2 & \text{if } [\![e_1]\!](s) = n_1 \text{ and } [\![e_2]\!](s) = n_2 \\ (i, n_1 + n_2) & \text{if } [\![e_1]\!](s) = (i, n_1) \text{ and } [\![e_2]\!](s) = n_2 \\ & \text{or } [\![e_1]\!](s) = n_1 \text{ and } [\![e_2]\!](s) = (i, n_2) \\ \text{nil} & \text{otherwise} \end{cases} \\
[\![e_1 - e_2]\!](s) & \triangleq & \begin{cases} n_1 - n_2 & \text{if } [\![e_1]\!](s) = n_1 \text{ and } [\![e_2]\!](s) = n_2 \\ (i, n_1 - n_2) & \text{if } [\![e_1]\!](s) = (i, n_1) \text{ and } [\![e_2]\!](s) = n_2 \\ \text{nil} & \text{otherwise} \end{cases} \\
[\![e_1 \times e_2]\!](s) & \triangleq & \begin{cases} n_1 \times n_2 & \text{if } [\![e_1]\!](s) = n_1 \text{ and } [\![e_2]\!](s) = n_2 \\ \text{nil} & \text{otherwise} \end{cases} \\
[\![e_1 = e_2]\!](s) & \triangleq & ([\![e_1]\!](s) = [\![e_2]\!](s)) \\
[\![e_1 \leq e_2]\!](s) & \triangleq & \begin{cases} n_1 \leq n_2 & \text{if } [\![e_1]\!](s) = n_1 \text{ and } [\![e_2]\!](s) = n_2 \\ \text{nil} & \text{otherwise} \end{cases} \\
[\![e_1 \text{ and } e_2]\!](s) & \triangleq & \begin{cases} b_1 \wedge b_2 & \text{if } [\![e_1]\!](s) = b_1 \text{ and } [\![e_2]\!](s) = b_2 \\ \text{nil} & \text{otherwise} \end{cases} \\
[\![e_1 \text{ or } e_2]\!](s) & \triangleq & \begin{cases} b_1 \vee b_2 & \text{if } [\![e_1]\!](s) = b_1 \text{ and } [\![e_2]\!](s) = b_2 \\ \text{nil} & \text{otherwise} \end{cases} \\
[\![\text{not } e]\!](s) & \triangleq & \begin{cases} \neg b & \text{if } [\![e]\!](s) = b \\ \text{nil} & \text{otherwise} \end{cases} \\
[\![\text{offset } e]\!](s) & \triangleq & \begin{cases} n & \text{if } [\![e]\!](s) = (i, n) \\ \text{nil} & \text{otherwise} \end{cases}
\end{array}
$$

Figure 4: Expression evaluation

$$\mathsf{bind}(f, \bot) \triangleq \bot \qquad \mathsf{bind}(f, \mathsf{error}) \triangleq \mathsf{error}$$

$$\mathsf{bind}(f, (I, l, m)) \triangleq \begin{cases} (I \cup I', l', m') & \text{if } f(l, m) = (I', l', m') \\ \mathsf{error} & \text{if } f(l, m) = \mathsf{error} \\ \bot & \text{otherwise} \end{cases}$$

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

Figure 5: Auxiliary operators bind **and** if

be 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 Section 4.2). The less-than-or-equal operator only applies to numbers—in particular, pointers cannot be compared.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 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, $[\![c]\!]_+ : \mathcal{S} \rightharpoonup \mathcal{O}_+$, where Here, $\mathcal{O}_+$ is an extended set of outcomes defined as $\mathcal{P}_{\mathsf{fin}}(\mathbb{I}) \times \mathcal{S} \uplus \{\mathsf{error}\}$. We then set

$$[\![c]\!](l, m) = \begin{cases} (l', m') & \text{if } [\![c]\!]_+(l, m) = (I, l', m') \\ \mathsf{error} & \text{if } [\![c]\!]_+(l, m) = \mathsf{error} \\ \bot & \text{if } [\![c]\!]_+(l, m) = \bot \end{cases}$$

$$\mathsf{finalids}(l, m) = \begin{cases} \mathsf{ids}(l, m) \setminus I & \text{if } [\![c]\!]_+(l, m) = (I, l', m') \\ \emptyset & \text{otherwise} \end{cases}$$

To define $[\![c]\!]_+$, we first endow the set $\mathcal{S} \rightharpoonup \mathcal{O}_+$ with the partial order of program approximation:

$$f \sqsubseteq g \quad \triangleq \quad \forall s, f(s) \neq \bot \Rightarrow f(x) = g(x)$$

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 operator fix.

The definition of $[\![c]\!]_+$ appears in Figure 6, where several of the rules use an auxiliary operator bind (Figure 5) 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 5) 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 that 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 $\bot$). The allocation command $x \leftarrow \mathsf{alloc}(e)$ first evaluates e to an integer n, then calculates the next free block id for the current machine state ($\mathsf{fresh}(\mathsf{ids}(l, m))$); 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, $\mathsf{free}(e)$ 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. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2015.

[4] A. Ahmed, M. Fluet, and G. Morrisett. L^3: A linear language with locations. *Fundam. Inform.*, 77(4):397–449, 2007.

[5] A. Azevedo de Amorim, M. Dénès, N. Giannarakis, C. Hriţcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies: Formally verified, tag-based security monitors. In *36th IEEE Symposium on Security and Privacy (Oakland S&P)*. 2015.

[6] T. Balabonski, F. Pottier, and J. Protzenko. Type soundness and race freedom for Mezzo. In M. Codish and E. Sumii, editors, *Functional and Logic Programming – 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*. 2014.

[7] 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.

[8] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Defensive javascript - building and verifying secure web components. In A. Aldini, J. Lopez, and F. Martinelli, editors, *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*. 2013.

[9] D. Chisnall, C. Rothwell, R. N. M. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. *ASPLOS*. 2015.

[10] A. A. de Amorim, N. Collins, A. DeHon, D. Demange, C. Hritcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. *Journal of Computer Security*, 24(6):689–734, 2016.

[11] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. HardBound: Architectural support for spatial safety of the C programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. 2008.

[12] D. Devriese, F. Piessens, and L. Birkedal. Reasoning about object capabilities with logical relations and effect parametricity. *EuroS&P*, 2016.

[13] U. Dhawan, C. Hriţcu, 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.

[14] 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. In C. Williamson, A. Akella, and N. Taft, editors, *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*. 2014.

[15] T. Elliott, L. Pike, S. Winwood, P. C. Hickey, J. Bielman, J. Sharp, E. L. Seidel, and J. Launchbury. Guilt free Ivory. In B. Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*. 2015.

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

[17] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. *SIGPLAN Not.*, 37(5):282–293, 2002.

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

[19] H. Hlavacs, T. Treutner, J. Gelas, L. Lefèvre, and A. Orgerie. Energy consumption side-channel attack at virtual machines in a cloud. In *IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing, DASC 2011, 12-14 December 2011, Sydney, Australia*. 2011.

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

[21] Y. Kim, R. Daly, J. Kim, C. Fallin, J. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. 2014.

$$[\![\mathsf{skip}]\!]_+(l,m) \triangleq (\emptyset,l,m) \qquad [\![c_1;c_2]\!]_+(l,m) \triangleq \mathsf{bind}([\![c_2]\!]_+,[\![c_1]\!]_+(l,m)) \qquad [\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]_+(l,m) \triangleq \mathsf{if}([\![e]\!](l,m),[\![c_1]\!]_+(l,m),[\![c_2]\!]_+(l,m))$$

$$[\![\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{end}]\!]_+ \triangleq \mathsf{fix}(\lambda\, f\,(l,m).\ \mathsf{if}([\![e]\!](l,m),\mathsf{bind}([\![c]\!]_+,f(l,m)),(\emptyset,l,m))) \qquad [\![x \leftarrow e]\!]_+(l,m) \triangleq (\emptyset,l[x \mapsto [\![e]\!](l,m)],m)$$

$$[\![x \leftarrow [e]]\!]_+(s) \triangleq \begin{cases}(\emptyset,l[x\mapsto v],m) & \text{if } [\![e]\!](s)=(i,n) \text{ and } m(i,n)=v\\ \mathsf{error} & \text{otherwise}\end{cases}$$

$$[\![[e_1] \leftarrow e_2]\!]_+(s) \triangleq \begin{cases}(\emptyset,l,m[(i,n)\mapsto [\![e_2]\!](l,m)]) & \text{if } [\![e_1]\!](s)=(i,n) \text{ and } m(i,n)\neq\bot\\ \mathsf{error} & \text{otherwise}\end{cases}$$

$$[\![x \leftarrow \mathsf{alloc}(e)]\!]_+(l,m) \triangleq \begin{cases}(\{i\},l[x\mapsto(i,0)],m[(i,k)\mapsto 0 \mid 0\le k<n]) & \text{if } [\![e]\!](l,m)=n \text{ and } i=\mathsf{fresh}(\mathsf{ids}(l,m))\\ \mathsf{error} & \text{otherwise}\end{cases}$$

$$[\![\mathsf{free}(e)]\!]_+(l,m) \triangleq \begin{cases}(\emptyset,l,m[(i,k)\mapsto \bot \mid k\in\mathbb{Z}]) & \text{if } [\![e]\!](l,m)=(i,0) \text{ and } m(i,n)\neq\bot \text{ for some } n\\ \mathsf{error} & \text{otherwise}\end{cases}$$

Figure 6: Command evaluation with explicit allocation sets

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

[23] 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.

[24] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, 2010.

[25] 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. In C. Krintz and E. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016.* 2016.

[26] L. A. Meyerovich and V. B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, 2010.

[27] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture.* 1995.

[28] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C. In J. Vitek and D. Lea, editors, *ISMM.* 2010.

[29] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2009.

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

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

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

[33] F. Pottier and J. Protzenko. Programming with permissions in Mezzo. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*, 2013.

[34] V. Rajani, D. Garg, and T. Rezk. On access control, capabilities, their equivalence, and confused deputy attacks. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016.* 2016.

[35] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science.* 2002.

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

[37] C. Schlesinger, K. Pattabiraman, N. Swamy, D. Walker, and B. G. Zorn. Modular protections against non-control data attacks. *Journal of Computer Security*, 22(5):699–742, 2014.

[38] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy.* 2013.

[39] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated analysis of security-critical javascript apis. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA.* 2011.

[40] The Coq Development Team. *The Coq Reference Manual, version 8.4*, 2012. Available electronically at http://coq.inria.fr/doc.

[41] E. Z. Yang and D. Mazières. Dynamic space limits for Haskell. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2014.

[42] H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. In *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures.* 2002.