12-12-2024

Tag-based Security in C: Writing and Specifying Flexible Protection

Sean Noble Anderson
Portland State University

Tag-based Security in C: Writing and Specifying Flexible Protection

by

Sean Noble Anderson

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Andrew P. Tolmach, Chair
Benjamin Pierce
Mark Jones
James Hook

Portland State University
2024

Abstract

The C language is ubiquitous and insecure. Tag-based security policies offer a flexible toolkit for runtime protection, including policies that mitigate the effects of undetected programming bugs and those that enforce security properties of the program logic. But tag policies can be difficult to define, and the protection that they offer can be difficult to specify and to prove.

This dissertation builds from an assembly-level specification of stack safety, encompassing the essential control-flow structure of C and other high level languages, to a C source-level policy definition framework called Tagged C. Tagged C includes a variant C semantics parameterized by a user-defined tag policy, and an interpreter that provably implements that semantics. It is flexible enough to define a wide range of security policies, including memory safety, compartmentalization, and secure information flow. Finally, introduces a novel compartmentalization policy, CluMPS, with a formal specification and an implementation in Tagged C, verified by mechanized proof in Coq.

Dedication

In loving memory of Lois J. Anderson, 1937-2024,

and of Steven W. Anderson, 1960-2024.

Dad, it means the world that you were able to see me finish this.

Acknowledgments

Infinite thanks to Andrew for years of patience and wisdom. There's no way to say "I couldn't have done it without you" with enough force to feel honest. And even if somehow had, I would have learned far less along the way.

Thank you as well to my committee, and all of my co-authors on the papers that make up this dissertation. Benjamin: being both, you've been a big influence on my writing. Special thanks to:

- Allison—Tagged C would be a far less functional product without you breaking it every other week!
- Chris—For many, many whiteboard sessions. Your ideas are present in the foundations of the project.
- Wes, Allison, and Rob—whether brainstorming or breeze-shooting, our little research group has been a great help at keeping me energized through this process.

To all of the volunteers of the Graduate Employees Union who poured their blood, sweat, and tears into making it possible to live as a graduate assistant: Solidarity, now and forever.

Mark: in all my time as the union's grievance officer, I never once had to deal with complaints of toxicity or dysfunction within the Computer Science department.

That means so much to me, and I know how hard you worked under challenging circumstances to keep the department as healthy as possible.

This work was supported by the National Science Foundation under Grant No. 2048499, Specifying and Verifying Secure Compilation of C Code to Tagged Hardware.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-18-C-0011. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA).

Table of Contents

List of Tables

List of Figures

1 Introduction

The computing infrastructure that underpins the world is insecure. Code written in unsafe languages (e.g., C) may hide any number of programming bugs that go uncaught until they are exploited in the wild, especially memory errors. Even in safe languages, code might contain logic errors (SQL injection, input-sanitization flaws, etc.) that subvert its security requirements.

Although static analyses can detect and mitigate many insecurities, an important line of defense against undetected or unfixable vulnerabilities is runtime enforcement of *security policies* using a reference monitor [5]. A security policy restricts the behavior of the system, typically by halting a badly-behaved process, termed "failstop behavior." A policy can be any kind of runtime check, from a simple assertion ("at line X, variable Y should have value Z") to a sophisticated temporal logic formula.

This dissertation focuses on a class of policies that can be specified in terms of flow constraints on *metadata tags*. A tag annotates a value with information like type, provenance, ownership, or security classification, and a tag-based policy is defined solely in terms of the interaction of tags, without reference to the values that they are attached to. Notable policies that can be implemented in this way include:

- *Memory safety*, which restricts programs in memory unsafe languages to obey the spatial and/or temporal constraints of the language, turning unchecked errors into checked ones;

- *Information flow control* (IFC), in which data identified as being secret or sensitive is preventing from leaking on an external channel;
- *Compartmentalization*, in which programs are divided into components (compartments) with restricted access to data and other resources;
- *Mandatory access control*, which identifies "subjects" (possibly compartments, but also non-code entities such as users) and explicitly restricts their access to resources.

Policies defined using tags are well-suited to efficient hardware enforcement, and a number of important security concepts can be enforced in this style. Policies covered in this dissertation are not tied to a specific hardware implementation, or necessarily to hardware at all, but examining an exemplar will help us understand the power and limitations of potential implementations. I take as my exemplar the PIPE[1] (Processor Interlocks for Policy Enforcement) ISA extension [8,9], a proposed hardware back-end for tag-based policies that has been realized in FPGA form.

PIPE is a programmable hardware mechanism that associates large (word-sized) metadata tags with every word of memory and every register. At each step of execution, while the ALU processes the operands of the current instruction, the tags associated with those operands are processed by a module called the "tag management unit" (TMU). The TMU, implemented in hardware as a cache or lookup table into a set of software-defined rules, consults those rules to (1) determine whether the operation should proceed, sending an interrupt if not, and (2) compute updated tags

[1]Variants of PIPE have been called PUMP [27] or SDMP [67] and marketed commercially under the names Dover CoreGuard and Draper Inherently Secure Processor.

to associate with the outputs of the operation. These rules collectively define a state machine operating on the tags in the system, which is termed a "micro-policy" [9], a concrete instantiation of the sorts of policies mentioned above.

PIPE is a good exemplar because it is very flexible. Because PIPE tags are so large, they can encode complex data structures. It is even feasible to layer multiple policies on top of one another by taking the Cartesian product of their tags. And because tags are inaccessible to normal execution, tag policies in a PIPE implementation are protected from subversion by application code.

However, PIPE also exemplifies the challenges in the definition, specification, and verification of tag policies. Tag policies are challenging to write due to their very flexibility and because they must currently be written with deep knowledge of the machine code to which they are attached.

Since subtle errors might enable a policy to be compromised, it is vital that policies be validated. But first they must be specified. What does the policy actually set out to do? What protection does it offer? In many cases, there is no standard specification for the kind of security that a policy hopes to enforce. Even in cases where there is a proposed formal specification, such as memory safety [10], a given policy may not precisely match it. Once defined and specified, the policy can be validated either by testing or formal verification. Verification rules out the possibility of bugs too subtle to show up in testing. But randomized property-based testing can increase confidence in a policy when proof is infeasible.

This dissertation contains novel policy definitions, specifications, and verification work. It also aims to make the tasks of defining, specifying, and verifying policies easier with a powerful new source-level policy-definition framework, Tagged C.

1.1 Tag Policies By Example

Here we walk through two example policies implemented in a PIPE-style system to understand how they are written, why doing so is challenging, and why defining policies at the source level is a big step forward.

The first, a simplified "spatial" stack safety policy, protects the data of suspended stack frames from being accessed by other function activations. This prevents assembly code from breaking the source-level abstraction that a function's local variables are isolated from other calls. The second is an "information flow control" (IFC) policy. Unlike stack safety, IFC is not concerned with assembly code breaking a source-level abstraction, but with a property of the source program itself: that observing its execution cannot reveal some information of interest.

The goal is not to introduce these policies on a deep techical level, but to give an intuition for how they might be attached to an assembly program and why this is a complex process. Both examples are defined at the assembly level as micro-policies, as is typical in the state-of-the-art [8,37,67], even though IFC would be more natural to express at the source level.

In brief, a micro-policy consists of a collection of rules, each associated with a family of opcodes. Almost all policies need to distinguish individual special instructions via tags on their values in memory, since many opcodes can play different roles that need to be treated differently in the policy. Defining a micro-policy in this style requires knowledge of both the assembly language of the host ISA and the behavior of the compiler, so that the policy designer can identify which instructions serve special purposes. Some policies require the binary to be rewritten with additional

instructions whose primary purpose is moving and manipulating tags.

Example: Spatial Stack Safety Figure 1.1 shows how a single function entry sequence must be modified to support a (simplified) spatial stack safety policy. The purpose of this policy is to prevent loads and stores to stack frames other than that of the active function. The policy is conceptually simple: each location in a stack frame is identified by the depth of the frame, and the stack pointer is tagged with the depth of the current function. Loads and stores of stack addresses must use a pointer that matches that of the location, i.e., the current stack pointer or a pointer derived from it. For simplicity, this version does not attempt to protect deallocated frames (which may share the depth of the active frame), so it only offers spatial and not temporal protection.

Figure 1.1a shows a typical entry sequence in assembly for a function whose frame (including saved return address) is sixteen bytes. It simply allocates those bytes by decreasing the stack pointer, then stores the return address to the stack. Later it will attempt to store data elsewhere in the frame, then load it. Figure 1.1b gives a sense of how this code might be instrumented with tags, and Figure 1.1c describes some of the rules that act on these tags. The entry sequence is given special tags to enforce that it runs from beginning to end and only following a call, and instructions are added to initialize tags on the stack frame.

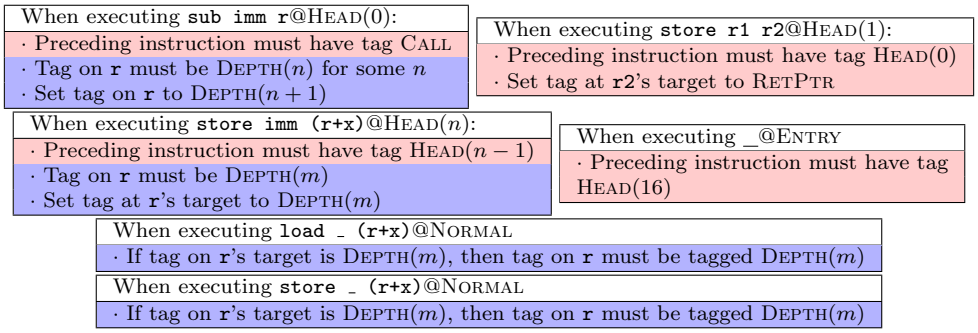
A significant subset of the policy's rules are dedicated to bookkeeping, in this case mostly for purposes of ensuring that the entry sequence executes in order (red). Only the lines in blue deal with the main focus of the policy: tagging the frame and the stack pointer with the current depth of the call stack, and enforcing that

```
0: sub sp 16 sp          Allocate sixteen bytes
8: store ra sp           Save return address
...
32: store 42 (sp+8)      Store to stack in body
...
64: load r0 (sp+8)       Load from stack in body
```

(a) Initial generated code

```
0: sub sp 16 sp          @ Head(0)
8: store ra sp           @ Head(1)
16: store 0 (sp+8)       @ Head(2)
24: nop                  @ Entry
...
48: store 42 (sp+8)      @ Normal
...
80: load r0 (sp+8)       @ Normal
```

(b) Tagged and expanded for policy



(c) Associated policy rules

Figure 1.1: Example: Adding Stack Safety policy at call

a stack address can only be written through the stack pointer at the same depth. Yet they must all work together with precision, or any security guarantees may be compromised.

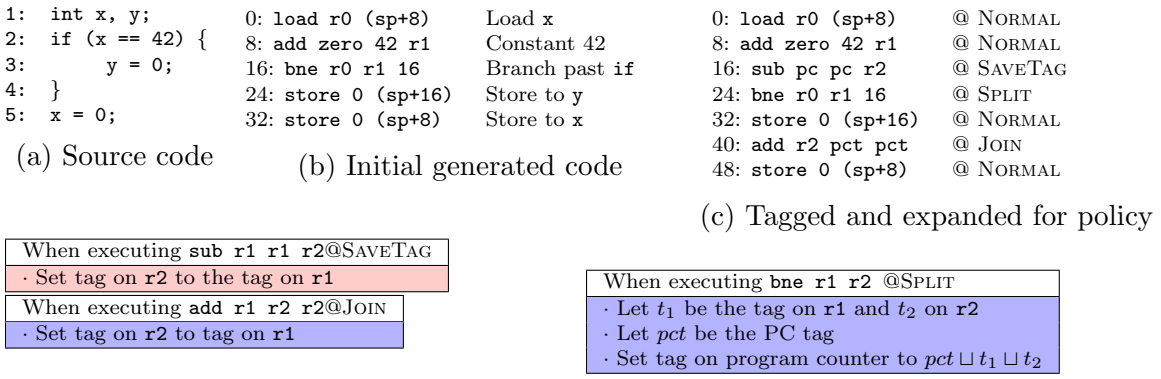
And which guarantees are those? Stack safety is commonly thought of as "temporal," that is, also protecting deallocated data from future accesses. But in the interest of simplicity, this example only offered spatial safety. If this is meant to be more than an illustrative example, spatial stack safety should be formally defined so that the policy can be shown to enforce it. Then it will be up to the user whether that level of security is sufficient. A more realistic policy might not sacrifice protection for simplicity, but it still might do so for performance reasons.

Example: IFC The example in Figure 1.2 illustrates another scenario that requires the assembly to be rewritten. This is an *information flow control* (IFC) problem, in which `x` is a secret and `y` is a public variable. The goal is that an observer who can see the value of `y` during execution not be able to deduce anything about the initial value of `x`. The assignment at line 3 of Fig. 1.2a would reveal to an observer that `x` is 42! To prevent this, an IFC policy might track when execution is in a state that depends on a secret value, which in this case lasts until line 5. That control-flow dependency is tracked by a tag on the program counter itself, the *PC tag*.

The policy uses a binary tag with values H (high security/secret) and L (low security/public), and uses the operator $\cdot \cup \cdot$ to take the higher of two tags. When control flow depends on a secret, the policy sets the PC tag to H; since the plan is to later restore it to its current level, that level has to be saved with an extra instruction, the `sub` at line 16 of Fig. 1.2c. Now `r2` is storing the old PC tag. Then, suppose the policy designer determines that join point of the expression is at line 40. They insert an `add` instruction to restore the tag on `r2` to the program counter tag.

All the policy designer really needs to know to define this policy is that the PC tag is saved at the split point and is available to be restored at the join point. But they are forced to consider how the PC tag is saved, in which register, whether that register is in use, which instruction to use to restore it, and so on.

These assembly transformations can be automated, given relevant annotations from the compiler, but the process amounts to an ad hoc compiler pass. It would be better to do it at the source level. Then the compiler (or interpreter) can handle most of the bookkeeping, leaving the policy developer free to focus on the rules that are relevant to the policy at hand.



(a) Source code

(b) Initial generated code

(c) Tagged and expanded for policy

Figure 1.2: Example: Adding IFC policy at if statement

Moving the task of policy definition to the source level has benefits for specification and validation of policies like IFC that aim to enforce security concepts expressible at the source level. Some policies are much easier to express at the source level. IFC is one example: the if statement that Fig. 1.2 relies upon is a source-level construct, and in a more realistic context the inputs and outputs that the policy is concerned with will be more natural to reason about in the source code. Another example is heap safety, since the heap does not exist in assembly. A source-level heap-safety specification can refer to the source semantics' notion of a heap.

As for validation, it is hard to prove properties of assembly code, even for ISAs that have formalizations. Many features of the source language are lost in assembly, such as structured control flow, function arguments and returns, and type information. In the case of IFC, it matters that the code in question is a compiled `if` statement rather than a hand-written assembly branch, because a `bne` instruction is not guaranteed to reach any future instruction the way the `if` statement is. Proofs about assembly programs are also non-portable across architectures and compilers.

1.1.1 Towards Source-level Tags

As these examples illustrate, much of the complexity of hardware tagging lies in ad hoc, non-portable assembly modifications and bookkeeping, which also contribute to challenges in specification and validation. The natural solution is to lift tags to the level of a source language wherever possible. In this dissertation that source language is C, and the instrumented variant is Tagged C. Figure 1.3 sketches an idealized outline of such a system. The source language (blue) has semantics that are parameterized by a policy definition. Tagged C works by annotating the semantics with a selection of *control points*: locations in the semantics where execution checks a specific tag rule, each serving a single purpose. This empowers policy designers to write tag rules knowing exactly how it will affect the program without needing to disambiguate common opcodes.

The source language might compile to tag-aware assembly code (red), be executed by an interpreter (top green), or be compiled into instrumented code as part of a more complex toolchain. Any back-end needs to provide a minimal degree of protection for the abstractions of the source language. Since this work focuses on C, the key abstraction that needs to be protected is that of function calls and returns. In other words, the "baseline policy" is stack safety, which might be enforced by an assembly-level micro-policy, the implicit guarantees of a software implementation, or some other mechanism entirely. Other assembly-level policies might be attached as well.

A compiler should be proven not only to produce correct code, but to preserve the behavior of any source policy, while an interpreter should also be proven sound and complete with respect to the source semantics.

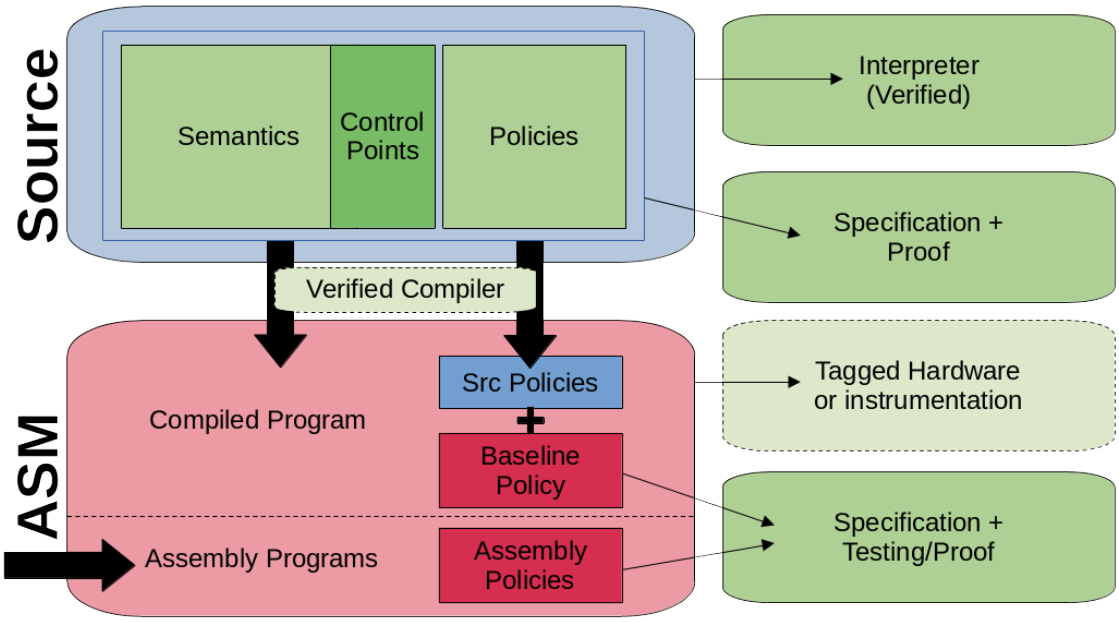


Figure 1.3: A Source-Level Tag Ecosystem

The green boxes represent important technical components of this system. The interpreter enables actual running of programs portably, not being reliant on actual tagged hardware or even on a compiler backend for the target architecture. Any given policy needs both a specification and to be validated against that specification, ideally with a source-level proof or, at the assembly-level, with proof or testing.

Tagged C approximates this idealized system, instantiating the semantics and interpreter for the C language. This dissertation also contains the specification for a family of assembly-level policies, stack safety, and the specification and verification of a source-level compartmentalization policy. The Tagged C compiler is out of scope, as is any work with real hardware.

1.2 Overview

This dissertation is about tag-based policies for C. It begins at the assembly level, where I confront a challenging specification problem: what is stack safety, exactly? After all, "the stack" is not a clearly defined language concept, but a loosely specified component of a system's ABI that is relied on by many different higher-level abstractions. I identify a specific stack safety property as an underlying assumption for all of my source-level work.

The second and largest part of this dissertation is Tagged C, a source-level specification framework that allows engineers to describe policies in terms of familiar C-level concepts. Tagged C allows policies to be defined at the source level via a fixed interface that never requires rewriting code or thinking about assembly. Where assembly instructions can serve different roles and must be distinguished for tag purposes, each Tagged C *control point* serves one clear role. The policy designer needs little knowledge of how Tagged C might be compiled, and need not deal with portions of a policy that would be colored red in Figure 1.1.

Finally, I put Tagged C to the test in performing source-level specification and verification of a novel compartmentalization policy. I specify compartmentalization in the form of an abstract semantics that is compartmentalized by construction, where compartments' local data are isolated entirely in separate address spaces, each of which behaves the same as Tagged C's memory model. I prove a simulation property between the abstract semantics and Tagged C with the appropriate policy, mechanized in Coq [74].

1.2.1 Topics and Contributions

Chapter 2 introduces related work and discusses how Tagged C fits into the larger runtime monitoring ecosystem. Chapters 3-7 are the body of the dissertation, and Chapter 8 concludes. The rest of this section breaks down the three topics in more detail, my contributions to each, and where they can be found in the dissertation.

Specifying Stack Safety In Chapter 3 I propose new formal characterizations of stack safety using concepts from language-based security, in the form of collections of trace properties at increasing levels of complexity.

The simplest model of stack safety is elegant and illustrates the core concepts, but does not account for many higher-level language features. At the other extreme, I sketch a stack safety model that takes into account a high-level view of pointer provenance: so high-level that it is better implemented at the source level. In the middle, a straightforward "public-private" data protection scheme is just right for the baseline protection of Tagged C.

These stack safety properties also capture the behavior of the "lazy" stack safety policies studied by Roessler and DeHon [67], which permit functions to write into one another's frames (intuitively a violation) but taint the written locations so that their owner cannot access them later. Laziness is a performance optimization which security implications were previously unclear; no prior characterization of stack safety can express this style of safety.

My contributions are:

- A novel characterization of stack safety as a conjunction of security properties:

confidentiality and integrity for callee and caller, plus well-bracketed control-flow. The properties are parameterized over a notion of external observation, allowing them to characterize lazy enforcement mechanisms.

- An extension of these core definitions to describe a realistic setting with argument passing on the stack, callee-saves registers, and tail-call elimination. The model is modular enough that adding these features is straightforward.

- Validation of a published enforcement mechanism, *Lazy Tagging and Clearing*, via property-based random testing; I find that it falls short, and propose and validate a fix.

This chapter was first published as:

It is a joint work with Roberto Blanco, Leonidas Lampropoulos, Benjamin Pierce, and Andrew Tolmach. I developed and formalized the properties from scratch. This work required significant pedagogical effort to make the properties comprehensible, which was shared by the full team. The initial testing framework was mostly written by Leonidas, and covered caller integrity in the simplest program model. Roberto and I worked jointly on the extension to confidentiality and various advanced program features.

Tagged C In chapter 4, I introduce Tagged C, a C variant whose semantics are parameterized by an arbitrary tag-based policy. Tagged C takes the form of a variant C semantics parameterized by tags attached to its data and rules triggered during

execution at a set of predefined *control points*. Control points correspond to significant execution events, such as function calls, expression evaluation, and pointer-based memory accesses.

Chapter 5 expands on Tagged C by exploring a handful of useful policies written in it, and chapter 6 digs deeper into the details of the Tagged C design.

The current implementation of Tagged C is an interpreter based on that of CompCert C [49]. This is sufficient to test small programs. Ultimately I expect for Tagged C to be compiled to a PIPE target by injecting the source policy's tag rules as a payload into a predefined assembly-level policy that handles the bookkeeping.

The Tagged-C semantics (also based on CompCert C) gives a formal definition of what each control point does. This means that properties of a policy may be proven in terms of how source programs behave when run under it. Such proofs are much easier than their assembly equivalents, and portable across architectures and implementations.

My contributions are:

- The design of a comprehensive set of *control points* at which the C language interfaces with a tag-based policy. These expand on prior work by encompassing the full C language while being powerful enough to enable a range of policies even in the presence of C's more challenging constructs (e.g., `goto`, conditional expressions, etc.).

- Tagged C policies enforcing: (1) compartmentalization; (2) memory safety, with realistic memory models that support varying kinds of low-level idioms; and (3) secure information flow.

- A full formal semantic definition for Tagged C, formalized in Coq, describing how the control points interact with programs, and an interpreter, implemented and verified against the semantics in Coq and extracted to OCaml.

This work is primarily mine. Chapters 4 and 5 were originally published as a joint work with Andrew Tolmach and Allison Naaktgeboren:

Some technical details related to how Tagged C might be compiled are also published in Chhak et al. [16], and some details of Tagged C's baseline memory model in Tolmach et al. [76], both joint works with CHR Chhak and Andrew Tolmach. The original content is updated to reflect further development, and Chapter 6 contains a detailed discussion of the design decisions that inform the current development.

Compartmentalization Finally, Chapter 7 presents a novel compartmentalization policy and its specification. Both are novel, and improve upon the state-of-the art in tag-based compartmentalization by allowing objects to be shared between compartments via passed pointers, without the overhead of protecting every object individually. The policy definition, its specification, and its proof are all concrete contributions on their own, and together they serve to demonstrate that Tagged C is a suitable setting in which to perform the entire define-specify-validate cycle.

My contributions are:

- A formal model of C compartmentalization in the form of an abstract machine that supports sharing between compartments while keeping their memories isolated by construction.

- A novel compartmentalization policy for Tagged C that supports cross-compartment sharing with fewer constraints on available tags than similar systems from the literature.
- A proof that the compartmentalization policy is safe with respect to the abstract semantics.

This work is not yet submitted for publication.

2 Runtime Enforcement with Tags and Beyond

Insecure computer systems are a ubiquitous and intractable problem, which has spawned a host of solution approaches with varying degrees of rigor and practicality. The security industry at large is built around a "find-and-fix" approach: engineers study systems from the perspective of an attacker, find vulnerabilities, and report them to the system's owner or maintainer to be fixed. This creates the fundamental structural problem that software is vulnerable to attack by any bad actor who finds an exploitable bug before it is fixed.

Various forms of static analysis can mitigate this risk by detecting code that deviates from a standard set of practices before that code is put into production. But static analysis is inherently imprecise, and may identify code that violates the standard but is not actually vulnerable. Especially in large legacy codebases, it may be prohibitively expensive to update potentially vulnerable code to follow secure practices unless the risk is clear and immediate. Any significant changes to legacy code can risk regression. And static analysis is of limited use against a vulnerability that involves poor security logic implemented in perfectly safe code.

Runtime enforcement is a last line of defense against undetected or unfixable vulnerabilities. This chapter reviews the broader field of runtime security enforcement, especially with hardware accleration.

2.1 Runtime Verification and Aspect-oriented Programming

Tagged C originates as a tool to improve usability of a family of secure hardware mechanisms, but it also has connections to the field of *runtime verification* (RV). An RV system monitors the runtime behavior of a program against some external specification. In an RV system the user supplements a program's source code with a list of *pointcuts*, which define a set of program events, such as "when the program calls `f` with argument 42" or "when the program reads input from a file with a name matching the regex `foo*.txt`." The user then defines some constraint on the traces of events that the system should produce. Early RV systems define these constraints in terms of temporal logic formulae (e.g., Chabot et al. [15]), regular expressions (e.g., Ball and Rajamani [11]), or context-free grammars. Of these, temporal logic is the most expressive, but not intuitive for programmers. Later systems allow the user to write arbitrary state machines that respond to trace events, for example the C verification system Rmor [35].

More recent systems borrow concepts from aspect-oriented programming (AOP) [43]. An AOP language distributes *join points*[1] throughout its semantics, and the programmer separately writes *advice* in the form of additional code that should execute before or after various join points according to a pointcut specification. The compiler or runtime *weaves* the advice together with the main code. Adding AOP to an RV context allows RV state machines to be programmed in an AOP style [36].

Tagged C can be understood as such a system that supports very low-level and fine-grained events (e.g., individual arithmetic operations and casts). This granularity

[1] Not to be confused with the control-flow graph join points discussed in Section 5.3.

gives Tagged C much more power to define policies related to the internal behavior of a program: memory safety, information flow, etc. The RV and AOP approaches are generally better suited to specifications of externally visible behavior, although some systems support very general forms of event definition based on matching syntactic patterns in code, and might be able to achieve a similar level of granularity [30].

Tagged C also has a small, fixed set of control points, as opposed to arbitrary point cuts, and specifies a monitoring action (perhaps a no-op) for every potential join point. This is a side-effect of how tags work, but it hopefully reduces the burden on the user, who no longer needs to define pointcuts at all.

Unlike most AOP and RV systems, Tagged C's tag rules are constrained to inspect only tags, not arbitrary parts of program state, which limits their expressiveness. Also, tag rules are evaluated separately from the system being monitored and so cannot be used to "correct" bad behavior; all they can do is cause a failstop. These limitations follow from the ultimate goal of implementing Tagged C using efficient PIPE hardware.

2.2 Safer Cs

A number of systems use compiler techniques to enforce heap safety for C, especially spatial safety, in which live objects are only accessed through valid pointers. These include C Cured [20], Safe ANSI C [64], and Softbound [62], each of which combine static analysis to rule out most illegal memory accesses with runtime checks on those accesses that cannot be statically guaranteed safe. Some, like CETS, from Nagarakatte et al. [63], go further and enforce temporal safety, in which objects that

have been freed are not accessed afterward (i.e., no "use after free" errors are possible.) Such "Safer Cs" also includes works like Cyclone [40] and Checked C [73], which modify the C semantics or even syntax to create a memory-safe dialect for C.

In all cases, runtime enforcement is necessary. While various compiler techniques have been used to keep this as efficient as possible, it is a significant overhead, leading more recent work to focus on hardware acceleration. Architectures aiming at supporting these compilers include HardBound [26], Watchdog [60] [61], and the current state-of-the-art, CHERI [79]. CHERI can be used for more than just accelerating bounds checking, as can PIPE, as discussed below.

2.3 General-purpose Hardware Enforcement

The current state of the art in hardware enforcement has two active research areas: tagged architectures and capability machines. Tagged architectures such as PIPE [28] and ARM memory-tagging extension (MTE) [31] are generally similar or identical to normal ISAs, but with the addition of hardware tags that carry metadata used by the processor to track security information and potentially halt execution (or enter a privileged recovery mode). Capability machines directly provide architectural support for capabilities: unforgeable tokens of authority that encode both the location of the memory they point to, and the permissions to access it (and no memory beyond it.) These often take the form of "fat pointers," as in CHERI [79], where a pointer carries bounds information to the object that it points to, and the hardware automatically checks those bounds before allowing it to be dereferenced.

Memory Safety and Related Policies in PIPE PIPE has targeted memory safety from very early on, including concrete policies that are not expressly related to any particular source semantics [1]. By contrast, Azevedo de Amorim et al. [9] define correctness of a memory safety micro-policy according to the C standard as formalized in CompCert [49], a verified C compiler that also provides the state-of-the-art formalization of the standard. This is an instance of tags being used to target a specific semantics, in this case a fairly strict one. More theoretically, Azevedo de Amorim et al. [10] use PIPE as a back-end in their analysis of memory safety as a property, not tied to any particular language, but consistent with CompCert. Memory safety is important enough that it has warranted dedicated work in optimizing its performance, such as Li et al. [52], who devised a system for reclaiming the tags used to uniquely identify allocations.

The definitive PIPE-based work on Stack Safety is from Roessler and DeHon [67]. They propose a number of variations on stack safety, protecting both the control stack and stack-allocated memory, but do not support safe passing of pointers to stack objects. In Tagged C, the control stack is implicitly protected. But address-taken locals, including large arrays, are not. My memory safety policies will protect stack objects, including applying the same notion of provenance to their pointers that applies to heap objects.

Compartmentalization in PIPE Compartmentalization is another of the micro-policies implemented by Azevedo de Amorim et al. [9]; in this case, a low-level dynamic compartmentalization system. Their micro-policy is parameterized by compartment information and by restrictions on how compartments should interact (Manda-

tory Access Control) and does not permit memory to be shared between compartments. Meanwhile Roessler et al. [66] implement a form of static low-level compartmentalization informed by typical behavior of a program during recorded test runs, which does allow memory-sharing between compartments if that sharing occurs during the test runs. In both cases, compartments are sets of instructions, with restrictions on how control may transfer between compartments and which addresses a given compartment may access.

The more similar style of compartmentalization to that discussed in this paper has been implemented primarily as part of secure compilation work. Secure compilation is often presented in terms of separate compilation and linking – there is a component that is compiled by the (hopefully) secure compiler, linked against an adversarial *context* that might come from anywhere. This gives it a close relationship with compartmentalization, which is made explicit in Juglaret et al. [41] as *secure compartmentalizing compilation*. Their target machine has compartmentalization built-in, explicitly referencing Azevedo de Amorim et al.'s tag-based implementation as a possible back-end.

In turn, tag-based compartmentalized back-ends play a crucial role in Abate et al. [3], which defines and proves a secure compilation property for components that may contain *undefined behavior* (UB), and therefore become compromised dynamically, and in El Korashy et al. [29], who prove secure compilation in the presence of compartments that share memory by passing pointers.

ARM MTE Outside of PIPE, there is ARM memory-tagging extension (MTE) [31], which has only four-bit tags, and is intended to be used for a memory safety policy

in which allocations repeat colors, but adjacent allocations are always tagged differently. This protects against classic buffer overflows, but may not protect against more targeted attacks – or, at least, may be probabilistically vulnerable. Four bits provides only sixteen unique colors, which is very restrictive, so optimizations that safely reduce the number of tags are much more important. And the tag bits are not themselves protected, leaving the risk that what security there is can itself be undermined without careful handling [53]. MTE is not a suitable alternative to PIPE in general.

CHERI and Capability Machines CHERI [79] is the state-of-the-art capability machine. In addition to normal capabilities, CHERI supports *object capabilities* that associate data and control permission, such that jumping to their entry point is the only way to access the associated data. Object capabilities give CHERI a degree of generality because they can ensure that arbitrary code sequences are executed prior to accessing certain data. This has been used to implement both stack safety and compartmentalization [79]. It also allows for privileged code that tracks capabilities and revokes them when their objects are freed in order to handle temporal safety [80].

However, efficient revocation of capabilities in the CHERI model is challenging, as evidenced by the complexity of enforcing stack safety in CHERI. Because capabilities into the stack might be stored anywhere in the stack, and could later be used to subvert temporal stack safety, the entire empty part of the stack must be cleared on each return [70]. This challenge is the source of ongoing development of novel capability types: linear capabilities [71], which may be infeasible to implement in hardware, and capabilities with various restrictions on their storage or use [32]. This

is a natural consequence of the design decision that there is no security information attached to objects in memory; only to their capabilities, which may persist after the object is long gone.

An early application of the CHERI architecture provided an incrementally deployable compartmentalization model that was a hybrid of the CHERI back-end and MMU-based enforcement [79]. This is a different sort of flexibility than what Tagged C intends to offer: in the hybrid CHERI setting, the security being enforced (the equivalent of a policy) remains constant, and the hybridization is between two distinct back-ends. By contrast, Tagged C uses a common back-end, but can explore trade-offs between different policies. Despite this, it would be worth exploring whether a useful subset of Tagged C's control points could be implemented by a CHERI back-end.

System Assumptions Tagged C does not require hardware support, let alone any particular implementation of PIPE. But it is meant to be compatible, which means that it should have an efficient compilation path when targeting PIPE-like hardware. I make the following assumptions about potential hardware back-ends for Tagged C:

- Tags are not limited to pointers. There is a tag on the program counter, each register, and every word in memory. Tags on instructions are important as a means of parameterizing their behavior to match their corresponding meaning in the source.

- Tags are large. Tagged C does not restrict the size or complexity of tags, though a Tagged C compiler would. Specific policies might still be recommended based on their reduced usage of the tag space, placing fewer constraints on hardware,

but this is not built into the language.

- Tags are separate from the primary data, and cannot be directly modified.

In their work on compiling an early precursor to Tagged C, Chhak et al. pose the question of whether it can be safe to optimize away tag checks that are unnecessary for a given policy [16]. In this work I assume that this is plausible, so the checks built into Tagged C can cause overhead if necessary to complete their purpose.

3 Formalizing Stack Safety as a Security Policy

Functions in high-level languages (and related abstractions such as procedures, methods, etc.) are units of computation that invoke one another to define larger computations in a modular way. At a low level, each function activation manages its own local variables, spilled temporaries, etc., as well as information about the caller to which it will return. The *call stack* is the fundamental data structure used to implement functions, aided by an Application Binary Interface (ABI) that defines how registers are shared between activations. In this chapter, I offer a novel specification of *stack safety*: what does it mean to protect the stack from attack? [1]

From a security perspective, attacks on the call stack are attacks on the function abstraction itself. Indeed, the stack is an ancient [4] and perennial [18,38,57,58,72,77] target for low-level attacks, sometimes involving control-flow hijacking via corrupting the return address, sometimes memory corruption more generally.

The variety of attacks on the stack is mirrored in the range of software and hardware protections that aim to prevent them, including stack canaries [21], bounds checking [26, 62, 63], split stacks [44], shadow stacks [22, 69], capabilities [17, 33, 70, 71, 81], and hardware tagging [34, 67]. But enforcement mechanisms can be brittle, successfully eliminating one attack while leaving room for others. To avoid an endless game of whack-a-mole, we seek formal properties of safe behavior that can be proven,

[1] This chapter was originally published as joint work with Roberto Blanco, Leonidas Lampropoulos, Benjamin Pierce, and Andrew Tolmach. As such I have retained the use of "we" throughout. Chapter 1 gives a breakdown of who did what.

or at least rigorously tested. Such properties can be used as the specification against which enforcement can be validated—even enforcement mechanisms that do *not* fulfill a property can benefit from the ability to articulate why and when they may fail.

Many of the mechanisms listed above are fundamentally ill-suited for offering formal guarantees: they may impede attackers, but they do not provide universal protection. Shadow stacks, for instance, aim to "restrict the flexibility available in creating gadget chains" [69], not to categorically rule out attacks. Other mechanisms, such as SoftBound [62] and code-pointer integrity [44], do aim for stronger guarantees, but not formal ones. To our knowledge, the sole line of work making a formal claim to protect stack safety is the study of secure calling conventions by Skorstengaard et al. [71] and Georges et al. [33].

Some of the other mechanisms listed above should also be amenable to strong formal guarantees. In particular, Roessler and DeHon [67] present an array of tag-based micro-policies for stack safety that aim to offer universal protection. But the reasoning involved can be subtle: they include two micro-policy optimizations, Lazy Tagging and Lazy Clearing (likely to be deployed together, and the combination of which we hereafter refer to as Lazy Tagging and Clearing, or LTC). LTC allows function activations to write improperly into one another's stack frames, but ensures that the owner of the corrupted memory cannot access it afterward. This serves as an alternative to clearing the stack frame every time it is popped, which can be quite expensive. Under this policy, one function activation *can* corrupt another's memory—just not in ways that affect observable behavior. Therefore, LTC would not fulfill Georges et al.'s property (adapted to the tagged setting) [33]. But LTC does arguably enforce stack safety, or as Roessler and DeHon describe it informally,

a sort of data-flow integrity tied to the stack. A looser, more observational definition of stack safety is needed to fit this situation.

We propose here a formal characterization of stack safety based on the intuition of protecting function activations from each other and using the tools of language-based security [68] to treat function activations as security principals. We decompose stack safety into a family of properties describing the *integrity* and *confidentiality* of the caller's local state and the callee's behavior during (and after) the callee's execution, together with the *well-bracketed control flow* (WBCF) property articulated by Skorstengaard et al. [71].

Our properties are stated abstractly in the hope that they can also be applied to other enforcement mechanisms besides LTC. However, it does not seem feasible to give a universal definition of stack safety that applies to all architectures and compilers. While many security properties can be described purely at the level of a high-level programming language and translated to a target machine by a secure compiler, stack safety cannot be defined in this way since "the stack" is not explicitly present in the definitions of most source languages but rather is implicit in the semantics of features such as calls and returns.[2] But neither can stack safety be described coherently as a purely low-level property; indeed, at the lowest level, the specification of a "well-behaved stack" is almost vacuous because the ISA is not concerned with such questions as whether a caller's frame should be readable or writable to its callee. Those are the purview of high-level languages built atop the hardware stack.

Thus, any low-level treatment of stack safety must begin by asking: which high-

[2]Contrast Azevedo de Amorim et al.'s work on heap safety [10], where the concept of the heap figures directly in high-level language semantics and its security is therefore amenable to a high-level treatment.

level features are supported in a given setting using the stack, and how does their presence influence the expectation of well-bracketed control flow, confidentiality, and integrity? We begin with a simple system with very few features, then move to a more realistic one supporting tail-call elimination, argument passing on the stack, and callee-save registers. Our properties are factored so that the basic structure of each of our five properties remains constant while the presence or absence of different features leads to subtler differences in how they behave.

We demonstrate the usefulness of our properties for distinguishing between correct and incorrect enforcement using QuickChick [23, 45], a property-based random testing tool for Coq. Indeed, we find that the published version of LTC is flawed in a way that undermines both integrity and confidentiality. After correcting this flaw, however, LTC satisfies all of our properties. Further, we modify LTC to protect the features of our more realistic system and apply random testing to validate this extended protection mechanism against the extended properties.

In summary, we offer the following contributions:

- We give a novel characterization of stack safety as a conjunction of security properties—confidentiality and integrity for callee and caller—plus well-bracketed control-flow. The properties are parameterized over a notion of external observation, allowing them to characterize lazy enforcement mechanisms.

- We extend these core definitions to describe a realistic setting with argument passing on the stack, callee-saves registers, and tail-call elimination. The model is modular enough that adding these features is straightforward.

- We validate a published enforcement mechanism, *Lazy Tagging and Clearing*,

via property-based random testing, find that it falls short, and propose and validate a fix.

The following section offers a brief overview of our framework and assumptions. Section 3.2 walks through a function call in a simple example machine, discusses informally how each of our properties applies to it, and motivates the properties from a security perspective. Section 3.3 formalizes the machine model, its *security semantics*, and the stack safety properties built on these. Section 3.4 describes how to support an extended set of features. Section 3.5 gives an (untested) proof-of-concept formalization of a more sophisticated, heap-like sharing model based on pointer provenance. Section 3.6 describes the micro-policies that we test, Section 3.7 the testing framework itself, and Section 3.8 related work.

The accompanying artifact [3] contains formal definitions (in Coq) of our properties, plus our testing framework. It does not include proofs, since we use Coq primarily for the QuickChick testing library and to ensure that our definitions are unambiguous. Formal proofs are left as future work.

3.1 Framework and Assumptions

Stack safety properties need to describe the behavior of machine code, but they naturally talk about function activations and stack contents—abstractions that are typically not visible at machine level. To bridge this gap, our properties are defined in terms of a *security semantics* layered on top of the standard execution semantics of the machine. The security semantics identifies certain state transitions of the machine as

[3] https://github.com/SNoAnd/stack-safety

security-relevant operations, which update a notional *security context*. This context consists of an (abstract) stack of function activations, each associated with a *view* that maps each machine *state element* (memory location or register) to a *security class* (active, sealed, etc.) specifying how the activation can access the element. The action of a security-relevant operation on the context is defined by a mapping that characterizes how the operation's underlying machine code ought to implement the function abstraction in terms of the stack and registers.

Given the security classes of the elements of the machine state, we define high-level security properties—integrity, confidentiality, and well-bracketed control flow—as predicates that must hold on each call. These predicates draw on the idea of *variant* states from the theory of non-interference, plus a notion of *observable events*, which might include specific function calls (e.g., system calls that perform I/O), writes to special addresses representing memory-mapped regions, etc. For example, to show that certain locations are kept secret, it suffices to compare executions starting at machine states which vary at those locations and check that their traces of observable events are the same. This structure allows us to talk about the eventual impact of leaks or memory corruption without reference to internal implementation details and, in particular, to support lazy enforcement by flagging corruption of values only when it can actually impact visible behavior.

We introduce these properties by example in Section 3.2 and formally in Section 3.3. In the remainder of this section we introduce the underlying semantic framework in more detail.

Machine Model We assume a conventional ISA (e.g., RISC-V, x86-64, etc.), with registers including a program counter and stack pointer. We make no particular assumptions about the provenance of the machine code; in particular, we do not assume any particular compiler. If the machine is enhanced with enforcement mechanisms such as hardware tags [28, 34] or capabilities [81], we assume that the behavior of these mechanisms is incorporated into the basic step semantics of the machine. For any given mechanism we suppose an abstract notion of "compatible" states; in tagged hardware, two states are "compatible" if their memories and registers have identical tags, even if their values differ. In general, compatible states should have identical security behavior under the particular enforcement mechanism. Failstop behavior by enforcement mechanisms is modeled as stepping to the same state (and thus silently diverging).

Security Semantics A security semantics extends the core machine model with additional context about the identities of current and pending functions (which act as security principals) and about their security requirements on registers and memory. This added context is purely notional; it does not affect the behavior of the core machine. The security context evolves dynamically through the execution of security-relevant operations, which include calls, returns, and frame manipulation. Our security properties are phrased in terms of this context, often as predicates on future states ("when control returns to the current function, X must hold...") or as relations on traces of future execution (hyper-properties).

Security-relevant operations abstract over the implementation details of the actions they take. Since the same machine instruction may be used by a compiler for

different purposes, we assume that the compiler or another trusted source has provided labels to identify the security-relevant purpose of each instruction, if any. For instance, in the tagged RISC-V architecture that we use in our examples and tests, calls and returns are conventionally performed using the `jal` ("jump-and-link") and `jalr` ("jump-and-link-register") instructions, but these instructions might also be used for other things.

These considerations lead to an annotated version of the machine transition function, written $m \xrightarrow{\bar{\psi},e} m'$, where m and m are machine states, e is an optional externally observable event, and $\overline{\psi}$ is a list of security-relevant operations—necessarily a list because a single step might perform multiple simultaneous operations. This is then lifted into a transition between pairs of machine states and contexts by applying a transition function parameterized by the operation. We decompose this function into rules associated with each operation and introduce them as needed. The most important of these rules describe call and return operations. A call pushes a new view onto the context stack and changes the class of the caller's data to protect it from the new callee; a return reverses these steps. Other operations signal how parts of the stack frame are being used to store or share data, and their corresponding rules alter the classes of different state elements accordingly.

Exactly which operations and rules are needed depends on what code features we wish to support. The set of security-relevant operations (Ψ) covered here is given in Table 3.1. A core set of operations covering calls, returns, and local memory is introduced in the example in Section 3.2 and formalized in Section 3.3. An extended set covering simple memory sharing and tail-call elimination is described in Section 3.4 and tested in Section 3.7. The remaining operations are needed for the capability-

Operation $\psi \in \Psi$	Parameters	Sections
call	target address, argument registers	3.2,3.3
	stack arguments (base, offset & size)	3.4,3.7
return		3.2,3.3
alloc	offset & size	3.2,3.3
	public flag	3.4,3.7
dealloc	offset & size	3.2,3.3
tailcall	(same as for **call**)	3.4,3.7
promote	register, offset & size	3.5
propagate	source register/address	3.5
	destination register/address	3.5
clear	target register/address	3.5

Table 3.1: Security-relevant operations and their parameters, with the sections where they are first defined or used. Entries in light grey do not appear in our examples, but are part of our testing. Dark grey entries are not tested.

based model in Section 3.5.

Views and Security Classes The security context consists of a stack of *views*, where a view is a function mapping each state element to a *security class*—one of *public*, *free*, *active*, or *sealed*.

State elements that are outside of the stack—general-purpose memory used for globals and the heap, as well as the code region and globally shared registers—are always labeled *public*. We place security requirements on some *public* elements for the purposes of the well-bracketed control flow WBCF property, and a given enforcement mechanism might restrict their access (e.g., by rendering code immutable). But, for integrity and confidentiality purposes, they are considered accessible at all times.

When a function is newly activated, every stack location that is available for use but not yet initialized is *free*. From the perspective of the caller, the callee has no

obligations regarding its use of free elements.

Arguments are marked *active*, meaning that their contents may be used safely. When a function allocates memory for its own stack frame, that memory will also be *active*. Then, on a call, *active* elements that are not being used to communicate with the callee will become *sealed*—i.e., reserved for an inactive principal and expected to be unchanged when it becomes active again.

Instantiating the Framework Conceptually, the following steps are needed to instantiate the framework to a specific machine and coding conventions: (i) define the base machine semantics, including any hardware security enforcement features; (ii) identify the set of security-relevant operations and rules required by the coding conventions; (iii) determine how to label machine instructions with security-relevant operations as appropriate; (iv) specify the form of observable events.

Threat Model and Limitations When our properties are used to evaluate a system, the threat model will depend on the details of that system. However, there are some constraints that our design puts on any system. In particular, we must trust that the security-relevant operations have been correctly labeled. If a compiled function call is not marked as such, then the caller's data might not be protected from the callee; conversely, marking too many operations as calls may cause otherwise safe programs to be rejected.

We do not assume that low-level code adheres to any single calling convention or is being used to implement any particular source-language constructs. Indeed, if the source language is C, then high-level programs might contain undefined behavior,

in which case they might be compiled to arbitrary machine code. In general, it is impossible to distinguish buggy machine code from an attacker. In examples, we often identify one function or another as an attacker, but our framework does not require any static division between trusted and untrusted code, and we aim to protect even buggy code.

This is a strong threat model, but it does omit some important aspects of stack safety in real systems: in particular, it does not address concurrency. Hardware and timing attacks are also out of scope.

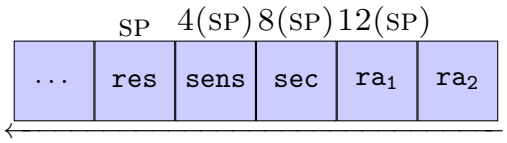
3.2 Properties by Example

In this section, we introduce our security properties by means of small code examples, using a simple set of security-relevant operations for calls, returns, and private allocations.

Figure 3.1 gives C code and possible corresponding compiled 64-bit RISC-V code for a function `main`, which takes an argument `secret` and initializes a local variable `sensitive` to contain potentially sensitive data. Then `main` calls another function `f`, and afterward it performs a test on `sensitive` to decide whether to output `secret`. Since `sensitive` is initialized to 0, the test should always fail, and `main` should instead output the return value of `f`. Output is performed by writing to the special global `out`, and we assume that such writes are the only observable events in the system.

The C code is compiled using the standard RISC-V calling conventions [65]. In particular, the function's first argument and its return value are both passed in `a0`. Memory is byte addressed, and the stack grows towards lower addresses. We assume

```
volatile int out;
void main(int secret) {
    int sensitive = 0;
    int res = f();
    if (sensitive == 42)
        out = secret;
    else
        out = res;
}
```




```
    0:  addi sp,sp,-20      | alloc (−20, 20)
    4:  sd ra,12(sp)        |
    8:  sw a0,8(sp)         |
   12:  sw zero,4(sp)       |
   16:  jal f,ra            | call ε
   20:  sw a0,0(sp)         |
   24:  lw a4,4(sp)         |
   28:  li a5,42            |
   32:  bne a4,a5,L1        |
   36:  lw a0,8(sp)         |
   40:  sw a0,out           |
   44:  j L2                |
L1, 48:  lw a0,0(sp)        |
   52:  sw a0,out           |
L2, 56:  ld ra,12(sp)       |
   60:  addi sp,sp,20       | dealloc (0, 20)
   64:  jalr ra             | return
```

Figure 3.1: Example: C and assembly code for `main` and layout of its stack frame (the stack grows to the left).

that `main` begins at address 0 and its callee `f` at address 100. The annotations in the right-hand column are security-relevant operations, described further below. The assembly is a simplified but otherwise typical compilation of the source code into RISC-V; its details are less important than the positions of the security-relevant operations.

Now, suppose that `f` is actually an attacker seeking to leak `secret`. It might do so in a number of ways, shown as snippets of assembly code in Figure 3.2. Leakage is most obviously viewed as a violation of `main`'s *confidentiality*. In Figure 3.2a, `f` takes an offset from the stack pointer, accesses `secret`, and directly outputs it. More subtly, even if it is somehow prevented from outputting `secret` directly, `f` can instead return its value so that `main` stores it to `out`, as in Figure 3.2b. Beyond simply reading `secret`, the attacker might overwrite `sensitive` with 42, guaranteeing that

```
100:  lw a4,8(sp)  |
104:  sw a4,out    |
108:  li a0,1      |
112:  jalr ra      | return
```

(a) Leaking `secret` directly

```
100:  lw a4,8(sp)  |
104:  mov a0,a4    |
108:  nop          |
112:  jalr ra      | return
```

(b) Leaking `secret` indirectly

```
100:  li a5,42     |
104:  sw a5,4(sp)  |
108:  li a0,1      |
112:  jalr ra      | return
```

(c) Attacking `sensitive`

```
100:  addi ra,ra,16 |
104:  nop           |
108:  nop           |
112:  jalr ra       | return
```

(d) Attacking control flow

```
100:  addi sp,sp,8  |
104:  nop           |
108:  nop           |
112:  jalr ra       | return
```

(e) Attacking stack pointer integrity

Figure 3.2: Example: assembly code alternatives for `f` as an attacker.

`main` publishes its own secret unintentionally (Fig. 3.2c); this does not violate `main`'s confidentiality, but rather its *integrity*. In Figure 3.2d, the attacker arranges to return to the wrong instruction, thereby bypassing the check and publishing `secret` regardless; this violates the program's *well-bracketed control flow* (WBCF). In Figure 3.2e, a different attack violates WBCF, this time by returning to the correct program counter but with the wrong stack pointer. (We pad some of these variants with `nop`s just so that all the snippets have the same length, which keeps the step numbering uniform in Fig. 3.3.)

The security semantics for this program is based on the security-relevant events noted in the right columns of Figures 3.1 and 3.2, namely execution of instructions that allocate or deallocate space (specified by an SP-relative offset and size), make a

call (with a specified list of argument registers), or make a return.

Our security semantics attaches a security context to the machine state, consisting of a view V and a stack σ of pending activations' views. Figure 3.3 shows how the security context evolves over the first few steps of the program. (The formal details of the security semantics are described in Section 3.3, and the context evolution rules are formalized in Fig. 3.7.) Execution begins at the start of `main`, with the program counter (PC) set to zero and the stack pointer (SP) at address 1000. State transitions are numbered and may be labeled with a security operation, written $\downarrow \psi$, between steps.

The initial view V_0 maps all stack addresses below SP to *free* and the remainder of memory to *public*. The sole used argument register, `a0`, is mapped to *active*; other caller-save registers are mapped to *free* and callee-save registers to *sealed*. Step 1 allocates a word each for `secret`, `sensitive`, and `res`, as well as two words for the return address; this has the effect of marking those bytes *active*. We use $V[\![\ldots]\!]$ to denote updates to V.

At step 5, the current principal's record is pushed onto the inactive list. The callee's view is updated from the caller's such that all *active* memory locations become *sealed*. (For now we assume no sharing of stack memory between activations; data is passed only through argument registers, which remain active. In the presence of memory sharing, some memory would remain active, too.) Function `f` does not take any arguments; if it did, any registers containing them would be mapped to *active*, while any non-argument, caller-saved registers are mapped to *free*. In the current example, only register `a0` changes security class. All callee-save registers remain *sealed* for all calls, so if, in the example, we varied the assembly code for `main` so that

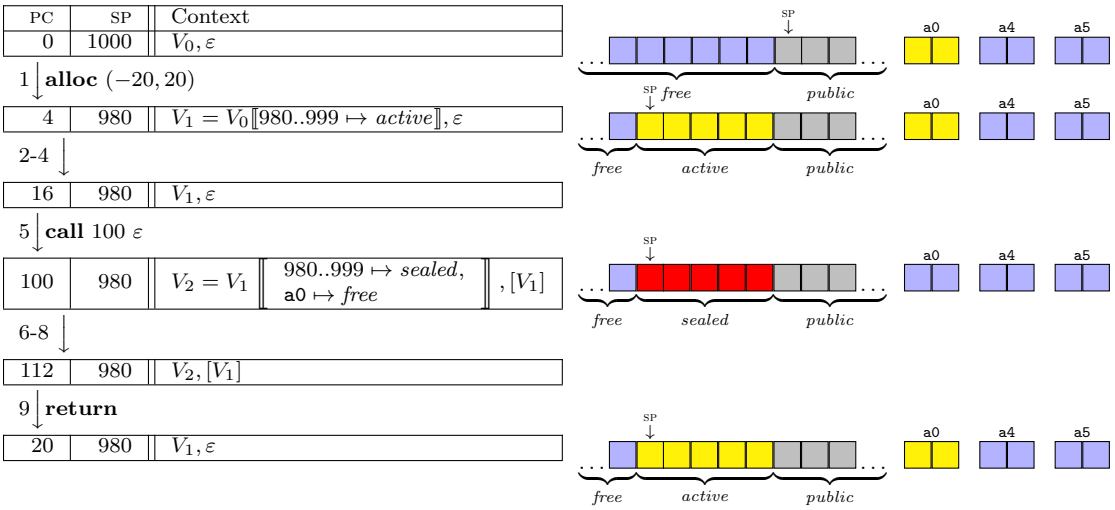


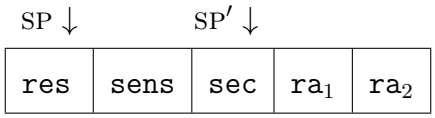
Figure 3.3: Execution of example up through the return from `f`. In stack diagrams, addresses increase to the right, stack grows to the left, and boxes represent 4-byte words.

`sensitive` was stored in a callee-save register (e.g., `s0`) rather than in memory, its security class would still be *sealed* at the entry to `f`. At step 9, `f` returns and the topmost inactive view, that of `main`, is restored.

We now show how this security semantics can be used to define notions of confidentiality, integrity, and correct control flow in such a way that many classes of bad behavior, including the attacks in Figure 3.2, are detected as security violations.

Well-Bracketed Control Flow To begin with, if `f` returns to an unexpected place (i.e., PC $\neq$ 20 or SP $\neq$ 980), we say that it has violated WBCF. WBCF is a relationship between call steps and their corresponding return steps: just after the return, the program counter should be at the next instruction below the call, and the stack pointer should have the same value that it had before the call. Both of these are essential for security. In Figure 3.2d, the attacker adds 16 to the return

address and then returns; this bypasses the `if`-test in the code and outputs `secret`. In Figure 3.2e, the attacker returns with $\text{SP}' = 988$ instead of the correct $\text{SP} = 980$. In this scenario, given the layout of `main`'s frame,



`main`'s attempt to read `sensitive` may instead read part of the return address, and its attempt to output `res` will instead output `secret`.

Before the call, the program counter is 16 and the stack pointer is 980. So we define a predicate on states that should hold just after the return: $\mathit{Ret}\ m \triangleq m[\text{PC}] = 20 \wedge m[\text{SP}] = 980$. We can identify the point just after the return (if a return occurs) as the first state in which the pending call stack is smaller than it was just after the call. WBCF requires that, if m is the state at that point, then $\mathit{Ret}\ m$ holds. This property is formalized in Table 3.2, line 1.

Stack Integrity Like WBCF, stack integrity defines a condition at the call that must hold upon return. This time the condition applies to all of the memory in the caller's frame. In Figure 3.3 we see the lifecycle of an allocated frame: upon allocation, the view labels it *active*, and when a call is made, it instead becomes *sealed*. Intuitively, the integrity of `main` is preserved if, when control returns to it, any *sealed* elements are identical to when it made the call. Again, we need to know when a caller has been returned to, and we use the same mechanism of checking the depth of the call stack. In the case of the call from `main` to `f`, the *sealed* elements are the addresses 980 through 999 and callee-saved registers such as the stack pointer.

Note that callee-saved registers often change during the call—but if the caller accesses them after the call, it should find them restored to their prior value.

While it would be simple to define integrity as "all sealed elements retain their values after the call," this would be stricter than necessary. Suppose that a callee overwrites some data of its caller, but the caller never accesses that data (or only does so after re-initializing it). This would be harmless, with the callee essentially using the caller's memory as scratch space, but the caller never seeing any change.

For a set of elements K, a pair of states m and n are *K-variants* if their values an only disagree on elements in K. We say that the elements of K are *irrelevant* in m if they can be replaced by arbitrary other values without changing the observable behavior of the machine. All other elements are *relevant*.[4]

We define *caller integrity* (CLRI) as the property that every relevant element that is *sealed* under the callee's view is restored to its original value at the return point. (This property is formalized in Table 3.2, line 2).

In our example setting, the observation trace consists of the sequence of values written to `out`. In Figure 3.2c the states before and after the call differ in the value of `sensitive`. Figure 3.4 shows the states before and after the call, which disagree on the value at `sensitive`. If we consider a variant of the original return state in which `sensitive` is 0 (orange) as opposed to 42 (blue), that variant will eventually output 1, while the actual execution outputs 5. This means that `sensitive` is relevant.

To be more explicit, similar to WBCF, we define a predicate on states *Int* such that *Int* m holds if all relevant sealed addresses in m have the same values as following

[4]This story is slightly over-simplified. If an enforcement mechanism maintains additional state associated with elements, such as tags, we don't want that state to vary. This is touched on in Section 3.3.4.

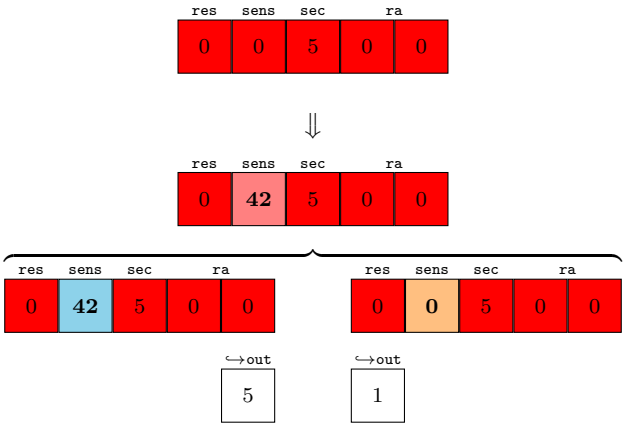


Figure 3.4: Integrity Violation: `sensitive` changed, and if varied, changes future outputs

the call. Then stack safety requires that *Int* hold on the state following the matching return, which is reached by step 9. Here `sensitive` has obviously changed, but it is relevant, which makes this a property violation.

Caller Confidentiality We treat confidentiality as a form of non-interference as well: the confidentiality of a caller means that its callee's behavior is dependent only on publicly visible data, not the caller's private state. This also requires that the callee initialize memory before reading it. As we saw in the examples, we must consider both the observable events that the callee produces during the call and the changes that the callee makes to the state that might affect the caller after the callee returns.

Consider the state after step 5, shown at the top of Figure 3.5, with the attacker code from Figure 3.2a and the assumption that `secret` has the value 5. We take a variant state over the set of elements that are *sealed* in V_2 (orange), and compare it to the original (blue). During the execution, the value of `secret` is written to

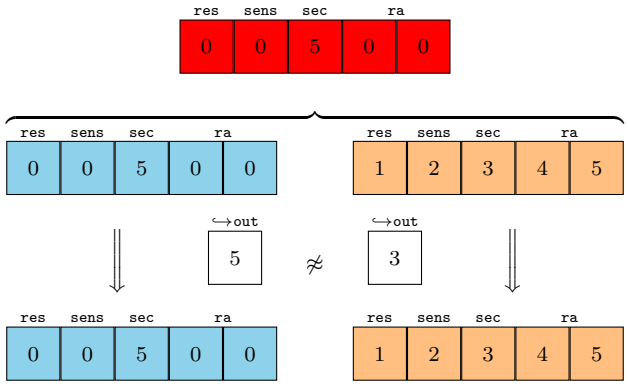


Figure 3.5: Internal Confidentiality Violation

the output, and the information leak is evidenced by the fact that the outputs do not agree—the original outputs 5, while the variant outputs 3. This is a violation of *internal confidentiality* (formalized in Table 3.2, line 3a).

But, in Figure 3.2b, we also saw an attacker that exfiltrated the secret by reading it and then returning it, in a context where the caller would output the returned value. Figure 3.6 shows the behavior of the same variants under this attack, but in this case, there is no output during the call. Instead the value of `secret` is extracted and placed in `a0`, the return value register.

At the end of the call, we can deduce that every element on which the variant states disagree must carry some information derived from the original varied elements. In most cases, that is because the element is one of the original varied elements and has not changed during the call, which does not represent a leak. But in the case of `a0`, it has changed during the call, *and* the return states do not agree on its value. This represents data that has been leaked, and should not be used to affect future execution. Unless `a0` happens to be irrelevant to the caller, this example is a violation

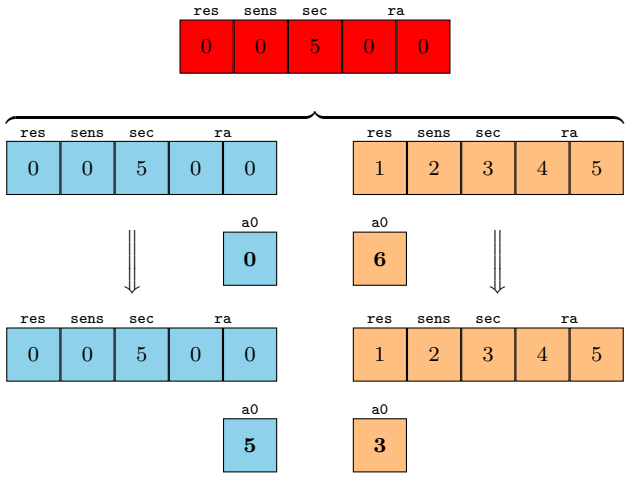


Figure 3.6: Return-time Confidentiality Violation

of what we term *return-time confidentiality* (formalized in Table 3.2, line 3b).

Structurally, return-time confidentiality resembles integrity, but now dealing with variants. We begin with a state immediately following a call, m. We consider an arbitrary variant state, n, which may vary any element that is *sealed* or *free*, i.e., any element that is not used legitimately to pass arguments. Caller confidentiality therefore can be thought of as the callee's insensitivity to elements in its initial state that are not part of the caller-callee interface.

We define a binary relation *Conf* on pairs of states, which holds on eventual return states m' and n' if all relevant elements are *uncorrupted* relative to m and n. An element is *corrupted* if it differs between m' and n', and it either changed between m and m' or between n and n'.

Finally, we define *caller confidentiality* (CLRC) as the combination of internal and return-time confidentiality (Table 3.2, line 3).

The Callee's Perspective We presented our initial example from the perspective of the caller, but a callee may also have privilege that its caller lacks, and which must be protected from the caller. Consider a function that makes a privileged system call to obtain a secret key, and uses that key to perform a specific task. An untrustworthy or erroneous caller might attempt to read the key out of the callee's memory after return, or to influence the callee to cause it to misuse the key itself!

Where the caller's confidentiality and integrity are concerned with protecting specific, identifiable state—the caller's stack frame—their callee equivalents are concerned with enforcing the expected interface between caller and callee. Communication between the principals should occur only through the state elements that are designated for the purpose: those labeled *public* and *active*.

Applying this intuition using our framework, *callee confidentiality* (CLEC) turns out to resemble CLRI, extended to every element that is not marked *active* or *public* at call-time. The callee's internal behavior is represented by those elements that change over the course of its execution, and which are not part of the interface with the caller. At return, those elements should become irrelevant to the subsequent behavior of the caller.

Similarly, in *callee integrity* (CLEI), only elements marked *active* or *public* at the call should influence the behavior of the callee. It may seem odd to call this integrity, as the callee does not have a private state. But an erroneous callee that performs a read-before-write within its stack frame, or which uses a non-argument register without initializing it, is vulnerable to its caller seeding those elements with values that will change its behavior. The fact that well-behaved callees have integrity by definition is probably why callee integrity has not typically been discussed in stack

safety studies such as Georges et al. [33].

3.3 Formalization

We now give a formal description of our machine model, security semantics, and properties. Our definitions abstract over: (i) the details of the target machine architecture and ABI, (ii) the set of security-relevant operations and their effects on the security context, (iii) the set of observable events, and (iv) a notion of value compatibility.

3.3.1 Machine

The building blocks of a machine are *words* and *registers*. Words are ranged over by w and, when used as addresses, a, and are drawn from the set $\mathcal{W}$. Registers in the set $\mathcal{R}$ are ranged over by r, with the stack pointer given the special name SP; some registers may be classified as caller-saved (CLR) or callee-saved (CLE). Along with the program counter, PC, these are referred to as *state elements* k in the set $\mathcal{K} ::= \{\text{PC}\} \cup \mathcal{W} \cup \mathcal{R}$.

A *machine state* $m \in \mathcal{M}$ is a map from state elements to a set $\mathcal{V}$ of *values*. Each value v contains a *payload* word, written $|v|$. We write $m[k]$ to denote the value of m at k and $m[v]$ as shorthand for $m[|v|]$. Depending on the specific machine being modeled, values may also contain other information relevant to hardware enforcement (such as a tag). When constructing variants (see Section 3.3.4) this additional information should not be varied. To capture this idea, we assume a given *compatibility* equivalence relation $\sim$ on values, and lift it element-wise to states. Two values should be compatible if their non-payload information (e.g., their tag) is identical.

The machine has a step function $m \xrightarrow{\bar{\psi},e} m'$. Except for the annotations over the arrow, this function just encodes the usual ISA description of the machine's instruction set. The annotations serve to connect the machine's operation to our security setting: $\bar{\psi}$ is a list of security-relevant operations drawn from an assumed given set Ψ, and e is an (potentially silent) observable event; these are described further below.

3.3.2 Security semantics

The security semantics operates in parallel with the machine. Each state element (memory word or register) is given a *security class* $l \in \{public, active, sealed, free\}$. A *view* $V \in VIEW$ maps elements to security classes. For any security class l, we write $l(V)$ to denote the set of elements k such that $V\ k = l$. The *initial view* V_0 maps all stack locations to *free*, all other locations to *public*, and registers based on which set they belong to: *sealed* for callee-saved, *free* for caller-saved except for those that contain arguments at the start of execution, which are *active*, and *public* otherwise.

A (security) *context* is a pair of the current activation's view and a list of views representing the call stack (pending inactive principals), ranged over by σ.

$$c \in C ::= VIEW \times list\ VIEW$$

The initial context is $c_0 = (V_0, \varepsilon)$.

Section 3.2 describes informally how the security context evolves as the system performs security-relevant operations. Formally, we combine each machine state with

a context to create a *combined state* $s = (m, c)$ and lift the transition to $\Longrightarrow$ on combined states. At each step, the context updates based on an assumed given function $Op : \mathcal{M} \to C \to \Psi \to C$. Since a single step might correspond to multiple operations, we apply Op as many times as needed, using *foldl*.

$$\frac{m \xrightarrow{\overline{\psi},e} m' \qquad \mathit{foldl}\ (\mathit{Op}\ m)\ c\ \overline{\psi} = c'}{(m, c) \xRightarrow{\overline{\psi},\ e} (m', c')}$$

A definition of Op is most convenient to present decomposed into rules for each operation. We have already seen the intuition behind the rules for **alloc**, **call**, and **ret**. For the machine described in the example, the Op rules would be those found in Figure 3.7. Note that Op takes as its first argument the state *before* the step.

3.3.3 Events and Traces

We abstract over the events that can be observed in the system, assuming just a given set *EVENTS* that contains at least the element τ, the silent event. Other events might represent certain function calls (i.e., system calls) or writes to special addresses representing memory-mapped regions. A *trace* is a nonempty, finite or infinite sequence of events, ranged over by $\mathcal{E}$. We use "$\cdot$" to represent "cons" for traces, reserving "::" for list-cons.

We are particularly interested in traces that end just after a function returns. We define these in terms of the depth d of the security context's call stack σ. We write $d \hookrightarrow s$ for the trace of execution from a state s up to the first point where the stack depth is smaller than d, defined coinductively by the rules in Figure 3.8a.

$$range\ r\ off\ sz\ m \triangleq \{m[r] + i | off \leq i < off + sz\}$$

$$\text{Alloc}\frac{K = range\ \text{SP}\ off\ sz\ m \cap free(V) \qquad V' = V[\![a \mapsto active \mid a \in K]\!]}{Op\ m\ (\textbf{alloc}\ off, sz)\ (V, \sigma) = (V', \sigma)}$$

$$\text{Dealloc}\frac{K = range\ \text{SP}\ off\ sz\ m \cap active(V) \qquad V' = V[\![a \mapsto free \mid a \in K]\!]}{Op\ m\ (\textbf{dealloc}\ off, sz)\ (V, \sigma) = (V', \sigma)}$$

$$\text{Call}\frac{V' = \lambda k. \begin{cases} free & \text{if } k \in CLR \\ public & \text{if } k \in \overline{r_{args}} \\ sealed & \text{if } k \in \mathcal{W} \text{ and } k \in active(V) \\ V(k) & \text{otherwise} \end{cases}}{(Op\ m\ (\textbf{call}\ a_{target}\ \overline{r_{args}})\ (V, \sigma) = (V', V :: \sigma)}$$

$$\text{Return}\frac{}{Op\ m\ \textbf{return}\ (\_, (V, \sigma')) = (V, \sigma')}$$

Figure 3.7: Basic Operations

$$\text{Done}\frac{|\sigma| < d}{d \hookrightarrow (m, (V, \sigma)) = \tau}$$

$$\text{Step}\frac{|\sigma| \geq d \qquad d \hookrightarrow (m', c') = \mathcal{E} \qquad (m, (V, \sigma)) \xRightarrow{\overline{\psi}, e} (m', c')}{d \hookrightarrow (m, (V, \sigma)) = e \cdot \mathcal{E}}$$

(a) "Produces trace" relation

$$\text{SimRefl}\frac{}{\mathcal{E} \approx \mathcal{E}} \qquad \text{SimEvent}\frac{\mathcal{E}_1 \approx \mathcal{E}_2}{e \cdot \mathcal{E}_1 \approx e \cdot \mathcal{E}_2}$$

$$\text{SimLeft}\frac{\mathcal{E}_1 \approx \mathcal{E}_2}{\tau \cdot \mathcal{E}_1 \approx \mathcal{E}_2} \qquad \text{SimRight}\frac{\mathcal{E}_1 \approx \mathcal{E}_2}{\mathcal{E}_1 \approx \tau \cdot \mathcal{E}_2}$$

(b) Trace Similarity

Figure 3.8: Relations on Traces

When $d = 0$, the trace will always be infinite because the machine never halts; in this case we omit d and just write $\hookrightarrow s$.

Two event traces $\mathcal{E}_1$ and $\mathcal{E}_2$ are *similar*, written $\mathcal{E}_1 \approx \mathcal{E}_2$, if the sequence of non-silent events is the same (Fig. 3.8b. That is, we compare up to deletion of τ events. Note that this results in an infinite silent trace being similar to any trace. So, a trace that silently diverges due to a failstop will be vacuously similar to all other traces.

3.3.4 Variants, corrupted sets, and "on-return" assertions

Two (compatible) states are variants with respect to a set of elements K if they agree on the value of every element not in K. Our notion of non-interference involves comparing the traces of such K-variants. We use this to define sets of irrelevant elements. Recall that $\sim$ is a policy-specific compatibility relation.

Definition 1. The *difference set* of two machine states m and m', written $\Delta(m, m')$, is the set of elements k such that $m[k] \neq m'[k]$.

Definition 2. Machine states m and n are *K-variants*, written $m \approx_K n$, if $m \sim n$ and $\Delta(m, n) \subseteq K$.

Definition 3. An element set K is *irrelevant* to state (m, c), written $(m, c) \parallel K$, if for all n such that $m \approx_K n$, $\hookrightarrow (m, c) \approx \hookrightarrow (n, c)$.

When comparing the behavior of variant states, we need a notion of how their differences have influenced them.

Definition 4. The *corrupted set* $\bar{\Diamond}(m, m', n, n')$ is the set $(\Delta(m, m') \cup \Delta(n, n')) \cap \Delta(m', n')$.

$$\text{Returned}\frac{|\sigma| < d \quad P\ m}{(d \uparrow P)\ (m, (V, \sigma))} \qquad \text{Step}\frac{|\sigma| \geq d \quad (d \uparrow P)\ (m', c') \quad (m, (V, \sigma)) \overset{\bar{\psi}, e}{\Longrightarrow} (m', c')}{(d \uparrow P)\ (m, (V, \sigma))}$$

(a) "On Return" $\uparrow$ Operator

$$\text{Returned}\frac{|\sigma_1| < d \quad |\sigma_2| < d \quad m_1\ R\ m_2}{(m_1, (V_1, \sigma_1))\ (d \Uparrow R)\ (m_2, (V_2, \sigma_2))}$$

$$\text{Left}\frac{|\sigma_1| \geq d \quad (m_1, (V_1, \sigma_1)) \overset{\bar{\psi}, e}{\Longrightarrow} (m_1', c_1') \quad (m_1', c_1')\ (d \Uparrow R)\ (m_2, (V_2, \sigma_2))}{(m_1, (V_1, \sigma_1))\ (d \Uparrow R)\ (m_2, (V_2, \sigma_2))}$$

$$\text{Right}\frac{|\sigma_2| \geq d \quad (m_2, (V_2, \sigma_2)) \overset{\bar{\psi}, e}{\Longrightarrow} (m_2', c_2') \quad (m_1, (V_1, \sigma_1))\ (d \Uparrow R)\ (m_2', c_2')}{(m_1, (V_1, \sigma_1))\ (d \Uparrow R)\ (m_2, (V_2, \sigma_2))}$$

(b) "When Both Return" $\Uparrow$ Operator

Figure 3.9: Logical Operators

If we consider two execution sequences, one from m to m' and the other from n to n', then $\bar{\Diamond}(m, m', n, n')$ is the set of elements that change in one or both executions and end up with different values. Intuitively, this captures the effect of any differences between m and n, i.e., the set of values that are "corrupted" by those differences.

Our "on-return" assertions are defined using a second-order logical operator $d \uparrow P$ (Fig. 3.9a), pronounced "P holds on return from depth d," where P is a predicate on machine states. This is a coinductive relation similar to "weak until" in temporal logic—it also holds if the program never returns from depth d.

Similarly, we give a analogous binary relation for use in confidentiality. We define $\Uparrow$ (Fig. 3.9b) so that $(m, c)\ (d \Uparrow R)\ (m', c')$ holds if R holds on the first states that return from depth d after (m, c) and (m', c'), respectively. Once again, $\Uparrow$ is coinductive.

1	WBCF $\triangleq(\lvert\sigma'\rvert \uparrow \mathit{Ret})\ (m', (V', \sigma'))$
	where $\mathit{Ret}\ m'' \triangleq m''[\text{SP}] = m[\text{SP}]$ and $m''[\text{PC}] = m[\text{PC}] + sz$
	where sz is the size of instruction at $m[\text{PC}]$
	for all calls $(m, (V, \sigma)) \Longrightarrow (m', (V', \sigma'))$
2	CLRI $\triangleq(\lvert\sigma\rvert \uparrow \mathit{Int})\ (m, (V, \sigma))$
	where $\mathit{Int}\ m' \triangleq m' \parallel (\mathit{sealed}(V) \cap \Delta(m, m'))$
	for all call targets $(m, (V, \sigma))$
3	CLRC $\triangleq\forall n$ s.t. $m \approx_K n$, where $K = \mathit{sealed}(V)$
3a	$\lvert\sigma\rvert \hookrightarrow (m, (V, \sigma)) \simeq \lvert\sigma\rvert \hookrightarrow (n, (V, \sigma))$
3b	and $(m, (V, \sigma))\ (\lvert\sigma\rvert \Uparrow \mathit{Conf})\ (n, (V, \sigma))$
	where $(m'\ \mathit{Conf}\ n') \triangleq m' \parallel \bar{\Diamond}(m, n, m', n')$
	for all call targets $(m, (V, \sigma))$
4	CLEC $\triangleq(\lvert\sigma\rvert \uparrow \mathit{CConf})\ (m, (V, \sigma))$
	where $\mathit{CConf}\ m' \triangleq m' \parallel (\Delta(m, m') - (\mathit{public}(V) \cup \mathit{active}(V))$
	for all call targets $(m, (V, \sigma))$
5	CLEI $\triangleq\forall n$ s.t. $m \approx_K n$, where $K = \mathcal{K} - (\mathit{public}(V) \cup \mathit{active}(V))$
5a	$\lvert\sigma\rvert \hookrightarrow (m, (V, \sigma)) \simeq \lvert\sigma\rvert \hookrightarrow (n, (V, \sigma))$
5b	and $(m, (V, \sigma))\ (\lvert\sigma\rvert \Uparrow \mathit{CInt})\ (n, (V, \sigma))$
	where $(m'\ \mathit{CInt}\ n') \triangleq m' \parallel \bar{\Diamond}(m, n, m', n')$
	for all call targets $(m, (V, \sigma))$

Table 3.2: Properties

3.3.5 Properties

Finally, the core property definitions are given in Table 3.2, arranged to show their commonalities and distinctions. Each definition gives a criterion quantified over states s that immediately follow call steps. If an execution includes a transition $s' \overset{\bar{\psi}}{\Longrightarrow} s$ where **call** $a\ \bar{r} \in \bar{\psi}$, then s is the target of a call. As a shorthand, we write that each property is defined by a criterion that must hold "for all call targets s," or, in the case of WBCF, "for all call steps $s \Longrightarrow s'$."

1. WBCF Given a call step $(m, (V, \sigma)) \Longrightarrow (m', (V', \sigma'))$, we define the predicate *Ret* to hold on states m'' whose stack pointer matches that of m and whose program

counter is at the next instruction. A system enjoys WBCF if, for every call transition, *Ret* holds just after the callee returns (i.e., the call stack shrinks).

2. CLRI When the call target is $(m, (V, \sigma))$, we define the predicate *Int* to hold on states m' if any element that is sealed in V and is in the difference set between m and m' is irrelevant. A system enjoys CLRI if, for every call, *Int* holds just after the corresponding return.

3. CLRC When the call target is $(m, (V, \sigma))$, we begin by taking an arbitrary n that is a K-variant of m, where K is the set of sealed elements in V. We require that two clauses hold. On line 3a, the behavior of a trace from $(m, (V, \sigma))$ up to its return must match that of $(n, (V, \sigma))$. On line 3b, we define a relation *Conf* that relates states m' and n' if their corrupted set (relative to m and n) is irrelevant, and require that it hold just after the returns from the callees that start at $(m, (V, \sigma))$ and $(n, (V, \sigma))$. A system enjoys CLRC if both clauses hold for every call.

4. CLEC We consider the callee's private behavior to be any changes that it makes to the state outside of legitimate channels—elements marked *active* or *public*. The remainder should be kept secret, which is to say, irrelevant to future execution. Similar to CLRI, given a call target $(m, (V, \sigma))$, we define a predicate *CConf* to hold on states m' if the difference set between m and m', excluding *active* or *public* locations, is irrelevant. A system enjoys CLEC if, for every call, *CConf* holds just after the corresponding return.

5. CLEI Callee integrity means that the caller does not influence the callee outside of legitimate channels. The caller's influence can be seen internally, or in corrupted data on return, just like the caller's secrets would be under CLRC. So, for a call target $(m, (V, \sigma))$, we take an arbitrary n that is a K-variant of m, where K is the set of elements that are not *active* or *public*. The remainder of the property is identical to CLRC.

3.4 Extended Code Features

The system modeled in Sections 3.2 and 3.3 is very simple. It does not permit functions to pass pointers to stack-allocated data to one another, a feature that is very common in C code and frequently relied upon by compilers. Realistic systems also use tail-calls and callee-save registers. Fortunately our framework is designed to make it easy to add support for more realistic code features.

To support argument passing on the stack, we just add new parameters to the existing security-relevant operations, and refine how they update the security context. The remainder of the properties do not change at all. To add tail-calls, we add and define a new operation, and since it is a kind of call, we add it to the definition of call targets. The rules for the extended security semantics are given in Figure 3.10.

3.4.1 Sharing Stack Memory

The examples in Section 3.2 present a vision of stack safety in which the interface between caller and callee is in the registers that pass arguments and return values. This is frequently not the case in a realistic setting. Arguments may be passed on

the stack because there are too many to pass in registers, as variadic arguments, or because they are composite types that inherently have pass-by-reference semantics. The caller may also pass a stack-allocated object by reference in the C++ style, or take its address and pass it as a pointer.

We refine our call operation to make use of the information that we have about which stack memory locations contain arguments. The new annotation $\overline{sa}$ is a set of triples of a register, an offset from the value of that register, and a size. We first define the helpful set *passed* $\overline{sa}$ m, then extend the call operation to keep all objects in *passed* marked as *active* and seal everything else.

Using this mechanism, a call-by-value argument passed on the stack at an SP-relative offset is specified by the triple $(\text{SP}, \mathit{off}, \mathit{sz})$. In this case, only the immediate callee gains access to the argument location. A C++-style call-by-reference argument where the reference is passed in r is instead specified by the triple $(r, 0, \mathit{sz})$. Such a call-by-reference argument could be passed through multiple calls, provided that it is in $\overline{sa}$ each time.

When the compiler uses the stack to pass arguments, barring the same memory being shared further down the stack, it is reasonable to assume that only the immediate recipient of the shared memory can access it. But when an object's address is taken and escapes, the resulting pointer can legally be passed arbitrarily far down the stack, stored to memory, and generally treated like any other pointer, save that it will be deallocated along with the rest of its frame. How much protection is appropriate for these pointers?

In the extreme, the source language (if any) can tell us exactly how pointers should or should not behave. If the source language is C, there is an implicit notion

of pointer provenance that defines a pointer as valid if it is derived from an object's base pointer by a series of accepted operations. Loads and stores through invalid pointers are undefined behavior, which we could turn into failstop behavior. But we have no guarantee that C is the source language or that the source language even has a notion of pointer provenance. If it does, then transplanting that notion into our security semantics could be very complex. Section 3.5 sketches how the property would be constructed, by painstakingly tracking step-by-step which registers and address hold valid pointers. Because the property in question is really a feature of the source language, simulating it in the target is quite complicated.

Public-private Memory Instead, for purposes of specifying low-level memory protection without making assumptions about the source language, we prefer a simpler model in which we simply classify an object as *public* anytime its address is taken directly and passed as a pointer, and give it no protection against access by other functions. We extend the **alloc** operation with a Boolean flag, where **t** indicates that the allocation is public, and **f** that it is private. If space for multiple objects is allocated in a single step, that step can make multiple allocation operations, each labeled appropriately. Public objects are labeled *public* rather than *active*, so they are never sealed at a call.

This is the model that our testing applies to, described in detail in Figure 3.10. This model generalizes that presented in Figure 3.7, which can be recaptured by instantiating **call** with $\overline{sa}$ as the empty set, and assuming that all instances of **alloc** carry the flag **f**.

$$\text{AllocF}\frac{K = \mathit{range}\ \text{SP}\ \mathit{off}\ \mathit{sz}\ m \cap \mathit{free}(V) \qquad V' = V[\![a \mapsto \mathit{active} \mid a \in K]\!]}{\mathit{Op}\ m\ (\mathbf{alloc\ f}\ (\mathit{off}, \mathit{sz}))\ (V, \sigma) = (V', \sigma)}$$

$$\text{AllocT}\frac{K = \mathit{range}\ \text{SP}\ \mathit{off}\ \mathit{sz}\ m \cap \mathit{free}(V) \qquad V' = V[\![a \mapsto \mathit{public} \mid a \in K]\!]}{\mathit{Op}\ m\ (\mathbf{alloc\ t}\ (\mathit{off}, \mathit{sz}))\ (V, \sigma) = (V', \sigma)}$$

$$\text{Dealloc}\frac{K = \mathit{range}\ \text{SP}\ \mathit{off}\ \mathit{sz}\ m \cap (\mathit{active}(V) \cup \mathit{public}(V)) \qquad V' = V[\![a \mapsto \mathit{free} | a \in K]\!]}{\mathit{Op}\ m\ (\mathbf{dealloc}\ (\mathit{off}, \mathit{sz}))\ (V, \sigma) = (V', \sigma)}$$

$$\mathit{push}\ V\ \overline{r}\ K \triangleq \lambda k. \begin{cases} \mathit{free} & \text{if } k \in CLR \\ \mathit{public} & \text{if } k \in \overline{r_{args}} \\ \mathit{sealed} & \text{if } k \in \mathcal{W} \text{ and} \\ & k \in \mathit{active}(V) - K \\ V(k) & \text{otherwise} \end{cases} \qquad \mathit{passed}\ \overline{sa}\ m \triangleq \bigcup_{(r, \mathit{off}, \mathit{sz}) \in \overline{sa}} \mathit{range}\ r\ \mathit{off}\ \mathit{sz}\ m$$

$$\text{Call}\frac{K = \mathit{passed}\ \overline{sa}\ m \qquad V' = \mathit{push}\ V\ \overline{r_{args}}\ K}{\mathit{Op}\ m\ (\mathbf{call}\ a_{target}\ \overline{r_{args}}\ \overline{sa})\ (V, \sigma) = (V', V :: \sigma)}$$

$$\text{Tailcall}\frac{K = \mathit{passed}\ \overline{sa}\ m \qquad V' = \mathit{push}\ V\ \overline{r_{args}}\ K}{\mathit{Op}\ m\ (\mathbf{tailcall}\ a_{target}\ \overline{r_{args}}\ \overline{sa})\ (V, \sigma) = (V', \sigma)}$$

Figure 3.10: Operations supporting tail calls and argument passing on stack.

3.4.2 Tail Calls

The rule for a tail call is similar to that for a normal call. We do not push the caller's view onto the stack, but replace it outright. This means that a tail call does not increase the size of the call stack, and therefore, for purposes of our properties, all tail calls will be considered to return simultaneously when the eventual **return** operation pops the top of the stack.

Since the caller will not be returned to, it does not need integrity, but it should still enjoy confidentiality. We set its frame to *free* rather than *sealed* to express this. In Table 3.2, we replace "call targets" with "call or tail call targets" in CLRC, CLEC, and CLEI.

3.5 Provenance, Capabilities, and Protecting Objects

What if we want to express a finer-grained notion of safety, in which stack objects are protected unless the function that owns them intentionally passes a pointer to them? This can be thought of as a *capability*-based notion of security. Capabilities are unforgeable tokens that grant access to a region of memory, typically corresponding to valid pointers to that region. As such, this capability safety relies on some preexisting notion of pointer validity, i.e., *pointer provenance*. Memarian et al.'s PVI [54] (provenance via integer) memory model is a good option: it annotates pointers with the identity of the object they first pointed to, and propagates the annotation when the pointer is copied and when operations are performed on it. See Section 5.1 for more details. This constitutes a substantial addition to the security context, which is why this enhancement is more speculative than the others, and untested.

$$\text{Promote}\frac{\rho' = \rho[r_{dst} \mapsto \mathit{range}\ r_{base}\ \mathit{off}\ \ \mathit{sz}]}{\mathit{Op}\ m\ (\textbf{promote}\ r_{dst}\ (r_{base}, \mathit{off}, \mathit{sz}))\ (V, \sigma, \rho) = (V, \sigma, \rho')}$$

$$\text{Clear}\frac{\rho' = \rho[k \mapsto \emptyset]}{\mathit{Op}\ m\ (\textbf{clear}\ k)\ (V, \sigma, \rho) = (V, \sigma, \rho')}$$

$$\text{Propagate}\frac{\rho' = \rho[k_{dst} \mapsto \rho[k_{src}]]}{\mathit{Op}\ m\ (\textbf{propagate}\ k_{src}\ k_{dst})\ (V, \sigma, \rho) = (V, \sigma, \rho')}$$

$$\mathit{reach}\ k\ \rho \triangleq \{k' | \mathit{base} \leq k' < \mathit{bound} \text{ where } \rho[k] = (\mathit{base}, \mathit{bound})\}$$
$$\mathit{reach}^*\ K\ \rho \triangleq \bigcup_{k \in K} \{k\} \cup \mathit{reach}^*\ (\mathit{reach}\ k\ \rho)\ \rho$$

$$\text{Call}\frac{K = \mathit{passed}\ \overline{sa}\ \cup\ \overline{r_{args}} \qquad K' = \mathit{reach}^*\ K\ \rho \qquad V' = \mathit{push}\ V\ \overline{r_{args}}\ K'}{\mathit{Op}\ m\ (\textbf{call}\ a_{target}\ \overline{r_{args}}\ \overline{sa})\ (V, \sigma, \rho) = (V', V :: \sigma, \rho)}$$

$$\text{Tailcall}\frac{K = \mathit{passed}\ \overline{sa}\ \cup\ \overline{r_{args}} \qquad K' = \mathit{reach}^*\ K\ \rho \qquad V' = \mathit{push}\ V\ \overline{r_{args}}\ K'}{\mathit{Op}\ m\ (\textbf{call}\ a_{target}\ \overline{r_{args}}\ \overline{sa})\ (V, \sigma, \rho) = (V', \sigma, \rho)}$$

Figure 3.11: Operations supporting provenance-based protection of passed objects

We can model the provenance model as a trio of additional security-relevant operations: PROMOTE, which declares a register to contain a valid pointer, PROPAGATE, which transmits the provenance of a pointer from one element to another, and CLEAR, which clears the provenance (for instance, when a pointer is modified in place in a way that makes it invalid).

In addition to the normal call stack, our security context will carry a map ρ from elements to memory regions, represented as a base and a bound $c = (V, \sigma, \rho)$. Most existing operations are extended to preserve the value of ρ, while the new operations and the call operation work as seen in Figure 3.11.

This essentially generalizes the above notion of passing: we will consider a caller to have intentionally passed an object if that object is reachable by a capability that has been passed to the callee. Reachability includes capabilities passed indirectly, by being stored in an object that is in turn passed. We define the set of reachable addresses using *reach*$*$, the transitive closure of elements that can be reached from the arguments of the call. The call operation in this setting will seal only objects that are not in *reach*$*$ nor the previously defined *passed*.

In the resulting property, once an object is sealed (because its capability has not been passed to a callee), subsequent nested calls can never unseal it. On the other hand, an object that is passed via a pointer may be passed on indefinitely.

3.6 Enforcement

We implement and test two micro-policies inspired by Roessler and DeHon [67]: *Depth Isolation* without lazy optimizations (DI) and with both Lazy Tagging and Lazy Clearing optimizations (LTC). (The connection between our properties and Roessler and DeHon's work is discussed below.) They share a common structure: each function activation is assigned a "color" n representing its identity. Stack locations belonging to that activation are tagged STACK n, and while the activation is running, the tag on the program counter (PC tag) is PC n. Stack locations not part of any activation are tagged UNUSED.

In DI, n always corresponds to the depth of the stack when the function is called. A function must initialize its entire frame upon entry in order to tag it, then clear the frame before returning. During normal execution, the micro-policy rules only

permit stack-based load and store operations when the target memory is tagged *with the same depth* as the current PC tag, or, for store operations, if the target memory is tagged UNUSED. When a stack object has its pointer taken, the pointer is tagged the same, and on loads and stores through such a pointer the rules compare the tag on the base register to that on the target memory, rather than the PC tag.

In LTC, a function neither initializes the frame at entry nor clears it at exit; instead, it simply sets each location's tag to the PC tag when that location is written. It does not check if those writes are legal! If the PC tag is PC n, then any stack location that recieves a stack-based store will be tagged STACK n. On a stack-based load, the micro-policy failstops if the source memory location is tagged UNUSED or STACK n for some n that doesn't match the PC tag. Once again, stores and loads based on registers other than the stack pointer behave similarly, but using their base register's tag.

To implement this discipline, *blessed instruction sequences* (seen in Fig. 1.1) appear at the entry and exit of each function, which manipulate tags as just described while performing the usual tasks of saving/restoring the return address to/from the stack and adjusting the stack pointer. A blessed sequence uses further tags to guarantee that the full sequence executes from the beginning—no jumping into the middle.

Outside of blessed sequences, instruction tags are used to denote arithmetic operations that take the addresses of stack objects. For example, in Figure 3.1, the variable `secret` is at 8(SP), so an instruction `addi a2,sp,8` would be tagged to indicate that it is creating a pointer to `secret`. This corresponds to the **promote** operation described above. The micro-policy's rules in this case would tag `a2` with the current depth. (A policy aiming to protect individual objects would need to further refine

the instruction tag to indicate that this instruction specifically generates a pointer to `secret`.)

Applicability to Testing Roessler & DeHon [67] Roessler and DeHon (henceforward *R&D*) differentiate between memory safety policies (without lazy optimization) and *data-flow integrity* policies (with lazy optimization). Our properties are phrased in terms of data flow, and we apply them to both optimized and non-optimized Depth Isolation. *R&D* do not attempt to define explicit formal properties, but they do list the behaviors that they expect their data-flow integrity policies to prevent, namely: reads from sealed objects (our ClrC); writes to sealed objects if they are later read (our ClrI); and reads from deallocated objects (our CleC). They also note that Lazy Clearing prevents uninitialized reads, which corresponds roughly to our CleI.

R&D note a flaw in Depth Isolation: because function activations are identified by depth, a dangling pointer into a stack frame might be usable when a new frame is allocated at the same depth. Our testing does not discover this flaw, because we do not test address-taken objects, but it discovers a related flaw under Lazy Tagging and Clearing that does not require an object's address to be taken. If an activation reads a location that was previously written by an earlier activation at the same depth, it will violate callee confidentiality. If that location was in a caller's frame, it also violates caller integrity and confidentiality.

R&D propose addressing the dangling-pointer issue by tracking both the depth of the current activation and the static identity of the active function. This would not eliminate all instances of this issue, but it would require the confidentiality-violating

activation to be of the same function that wrote the data in the first place, which is a significantly higher bar. We propose instead tracking every activation uniquely, which should eliminate the issue entirely—and does in our tests.

Protecting Registers R&D do not need to protect registers, since they include the compiler in their trusted computing base. But we target threat models that do not do so. In particular, CLRI requires callee-saved registers to be saved and restored properly. We extend DI and LTC so that callee-saved registers are also tagged with the color of the function that is using them. In DI they are tagged as part of the entry sequence, while in LTC they are tagged when a value is placed in them.

3.7 Validation through Random Testing

There are several ways to evaluate whether an enforcement mechanism enforces the above stack safety properties. Ideally, such validation would be done through formal proof over the semantics of the enforcement-augmented machine. However, while there are no fundamental barriers to producing such a proof, it would be considerable work to carry out for a full ISA like RISC-V and complex enforcement mechanisms like Roessler and DeHon's micro-policies. We therefore choose to systematically *test* their *Depth Isolation* and *Lazy Tagging and Clearing* micro-policies. We specifically test a variant that we call *Lazy Per-Activation Tagging and Clearing*.

We use a Coq specification of the RISC-V architecture [14], extend it with a run-time monitor implementing a stack safety micro-policy, and test it using QuickChick [45], a randomized property-based testing framework. QuickChick works by generating

random programs, executing them, and checking that they fulfill our criteria.

Such testing is sound—it will not produce false positives—but necessarily incomplete. We might test a flawed policy but fail to generate a program that exploits the flaw. Additionally, detecting violations of noninterference-style properties is dependent on choosing appropriate variant states, so it is possible to generate a dangerous program but have it pass the test due to variant selection. We increase our confidence in our test coverage by *mutation testing*, in which we intentionally inject flaws into the policies and demonstrate that testing can find them.

3.7.1 Test Generation

To use QuickChick, we develop random test-case generators that produce an initial RISC-V machine state tagged appropriately for the micro-policy (see Section 3.6), including a code region containing a low-level program. The generators also produce meta-information about how program instructions map to security-relevant operations, which would normally be provided by the compiler.

Our generators build on the work of Hriţcu et al. [37], which introduced *generation by execution*, a technique that produces programs that lead to longer executions—and hopefully towards more interesting behaviors as a result. Each step of generation by execution takes a partially instantiated machine state and attempts to generate an instruction that makes sense locally (e.g., jumps go to a potentially valid code location, loads read from a potentially valid stack location). The generator repeats this process for an arbitrary number of steps, or until it reaches a point where the machine cannot step any more. Each time it generates a call or return, it places the

appropriate policy tags on the relevant instruction(s) and records the operation.

We extend Hriţcu et al.'s technique with additional statefulness to avoid early failstops. For example, immediately after a call, we increase the probability of generating code that initializes any stack-allocated variables. To allow for potential attack vectors to manifest, the generator periodically relaxes those constraints and generates potentially ill-formed code, such as failing to initialize variables, writing outside of the current stack frame, or attempting an ill-formed return sequence,

3.7.2 Property-based Testing

Once a test program is generated, QuickChick tests it against a property. A typical hyperproperty testing scheme might do this by generating a pair of initial variant states, executing them to completion, and comparing the results. We extend this procedure to handle the nested nature of confidentiality.

For our setup to naïvely test the confidentiality of every call, it would need to create a variant state at each call point, execute it until return, then generate a post-call variant based on any tainted values. The post-call variant would execute alongside the "primary" execution until the test is finished. This results in tracking a number of variant executions that is linear in the total number of calls!

For better performance, we instead maintain a single "corruption variant" (CV) that tracks all post-return corruption. That is, the first time a function returns, suppose that K_1 is its corrupted set. We set all components in K_1 to random values in the CV and continue. On the second return, with corrupted set K_2, we do the same. The CV potentially differs from the original execution in all of $K_1 \cup K_2$, which

is more efficient than tracking two separate variant states, one that differs from the original in K_1 and the other in K_2.

This means that at any given time we need only simulate (1) the original execution; (2) the CV; and (3) one variant execution for each call on the call stack, a total that is linear in the depth of nested calls. That is much more efficient. This approach makes it much faster to test longer executions with many consecutive calls, albeit at the cost of making it harder to identify which call is the source of a failure.

3.7.3 Mutation Testing

To ensure the effectiveness of testing against our formal properties, we use *mutation testing* [39] to inject errors (mutations) in a program that should cause the property of interest (here, stack safety) to fail, and ensure that the testing framework can find them. The bugs we use for our evaluation are either artificially generated by us (deliberately weakening the micro-policy in ways that we expect should break its guarantees), or actual bugs that we discovered through testing our implementation. We elaborate on some such bugs below.

For example, when loading from a stack location, *Depth Isolation* needs to enforce that the tag on the location being read is STACK n for some number n and that the tag of the current PC is PC n for the same depth n. We can relax that restriction by omitting the check (bug *LOAD_NO_CHECK*). Similarly, when storing to a stack location, the correct micro-policy needs to ensure that the tag on the memory location is either UNUSED or has again the same depth as the current PC tag. Relaxing that constraint causes violations to the integrity property (bug *STORE_NO_CHECK*).

In addition to intentional mutations, our testing catches errors in our own implementation of the enforcement mechanism, including one interesting bug where the initial function's frame included space allocated for its return address, but this uninitialized (and therefore UNUSED-tagged) space was treated as private data but left unprotected. We added this to our set of mutations as *HEADER_NO_INIT*.

The original LTC micro-policy, implemented as *PER_DEPTH_TAG*, fails in testing when data is leaked between sequential calls. To round out our mutation testing we also check *LOAD_NO_CHECK*, equivalent to its counterpart in depth isolation, and a version where stores succeed but fail to propagate the PC tag, *STORE_NO_UPDATE*.

Mutation testing is commonly evaluated in terms of mean-time-to-failure (MTTF). It is expected that tests will fail, but a good property-based testing scheme should find injected bugs quickly. MTTF for each bug can be found in Table 3.3, along with the average number of tests it took to find the failure. Note that *HEADER_NO_INIT* took significantly longer than most, being a "real" bug and not an intentional mutation. In general, by generating tests that involve many loads and stores, we find mutations in a matter of seconds. Experiments were run on a desktop machine equipped with i7-4790K CPU @ 4.0GHz with 32GB RAM.

3.8 Related Work

The centrality of the function abstraction and its security are behind the many software and hardware mechanisms proposed for its protection [17, 21, 22, 26, 33, 34, 44, 62, 63, 67, 69–71, 81]. Many enforcement techniques focus purely on WBCF; others combine this with some degree of memory protection, chiefly focusing on integrity.

Bug	Property Violated	Ave. MTTF (s)	Tests
LOAD_NO_CHECK	Confidentiality	24.2	13.3
STORE_NO_CHECK	Integrity	26.9	26
HEADER_NO_INIT	Integrity	69.5	76.3
PER_DEPTH_TAG	Integrity	10.5	82
PER_DEPTH_TAG	Confidentiality	16.85	88
LOAD_NO_CHECK	Integrity	8.82	34.3
LOAD_NO_CHECK	Confidentiality	22.55	127
STORE_NO_UPDATE	Integrity	6.96	101
STORE_NO_UPDATE	Confidentiality	17.34	11

Table 3.3: MTTF for finding bugs in erroneous micro-policies: DI (top) and LTC (bottom)

Roessler and DeHon's *Depth Isolation* and *Lazy Tagging and Clearing* [67] both offer protections corresponding to WBCF, ClrI, and ClrC, though they do not give a formal description of this. They are generally not concerned with protecting callees.

To our knowledge, the only other line of work that aims to characterize the security of the stack in a rigorous manner is the StkTokens-Cerise family of CHERI-enforced secure calling conventions [33, 70, 71]. The authors of these works define stack safety using an overlay semantics, which connects the target machine semantics to an abstract machine that is stack safe by construction. They also prove properties of individual programs, phrased as logical relations.

In their initial paper, Skorstengaard et al. define stack safety informally as the combination of WBCF and "local state encapsulation" [70]. They describe the latter as an integrity property, roughly equivalent to ClrI. But when this concept is formalized as an abstract machine in StkTokens, that machine appears to enjoy confidentiality as well, and to be equivalent to ClrI *and* ClrC) [71]. Their overlay semantics (1) on call mints new a stack frame from a capability representing the

available stack space, and (2) on return merges the current frame back into the stack capability, under the assumption that there are no capabilities left on the stack. This should prevent callees from reading caller data, but the accompanying unary logical relation does not capture confidentiality per se.

The later paper from this group, Georges at al. [33], was inspired by the properties presented in this chapter to extend their formalism to include confidentiality through a binary logical relation. When checking if our properties applied to their old calling convention, they noted that it did not enforce CLEC, and made sure that their new version would in addition to building it into their formalism[5]. To do so, they redesign the overlay semantics to actually pop stack frames on return. This demonstrates a benefit of our choice to explicitly state properties in security terms: specifying security is hard, and when the spec takes the form of a "correct by construction" machine, it is easy to neglect a non-obvious security requirement.

In terms of direct feature comparison with Georges et al. [33], with the addition of confidentiality to their formalism, we are roughly at parity in terms of the expressiveness of our properties. We have additionally proposed callee-integrity, but it is probably the least practical of our properties. We extend our model to tailcalls, which they do not, and to the passing of pointers to stack objects. They discuss stack objects and the interaction between stack and heap, but their calling convention does not guarantee safety in the presence of pointer passing without additional checks. We test a limited degree of pointer passing, which does not guarantee memory safety for the passed pointer but which does not undermine the security of its frame, and we offer an untested formalism for memory-safe passing of pointers. On the other hand,

[5] A. L. Georges, personal communication.

their properties are validated by proof, while ours are only tested.

3.9 The Wider Context

Although this chapter addresses stack safety at the assembly level, it is haunted by the spectre of C. Most high-level languages implement their abstractions using a stack, but C's lack of memory safety renders it particularly vulnerable to stack-based attacks. As we add features to the security semantics it increasingly begins to resemble a model of compiled C rather than raw assembly, culminating with a notion of pointer provenance that is out of place at this level of abstraction.

Machine code is the wrong level of abstraction to specify what is essentially a construct of the source language. And while this chapter does not go into much detail about the micro-policies under test (they are not novel), they did need to be reimplemented in the test setup. That process was tedious and complex.

There is a better way: decompose the overall security policy into a single common assembly-level policy, namely the public-private split discussed in Section 3.4.1, and whatever source-level policy (or policies) are appropriate for a given code base. The public-private split is the appropriate level of granularity. Objects must be shareable sometimes, but a restriction on how shareable objects are actually shared and used is properly a source policy. The next chapter focuses on how best to define those source policies for the C language.

4 Flexible Runtime Security Enforcement with Tagged C

Many essential technologies rely on new and old C code. Operating systems (Linux, Windows, OSX, BSD), databases (Oracle, sqlite3), the internet (Apache, NGNIX, NetBSD, Cisco IOS), and the embedded devices that run our homes and hospitals are built in and on C. The safety of these technologies depends on the security of their underlying C codebases. Insecurity can arise from C undefined behavior (UB) such as memory errors (e.g., buffer overflows, use-after-free, double-free), from defined code that contains logic errors (e.g., SQL injection, input-sanitization flaws), or from larger-scale architectural flaws (e.g., over-provisioning access rights).

C's ubiquity and insecurity makes it an important target for tag-based enforcement. But PIPE policies can be difficult for a C engineer to write: their tags and rules are defined in terms of individual machine instructions and ISA-level concepts. In practice, they depend on reverse engineering the behavior of specific compilers. Moreover, some security policies can only be expressed in terms of high-level code features that are not preserved at machine level, such as function arguments, structured types, and structured control flow.

To address these problems, I introduce a *source-level* specification framework, *Tagged C*, which allows engineers to describe policies in terms of familiar C-level concepts, with tags attached to C functions, variables and data values, and rules triggered at *control points* that correspond to significant execution events, such as function calls, expression evaluation, and pointer-based memory accesses. Control

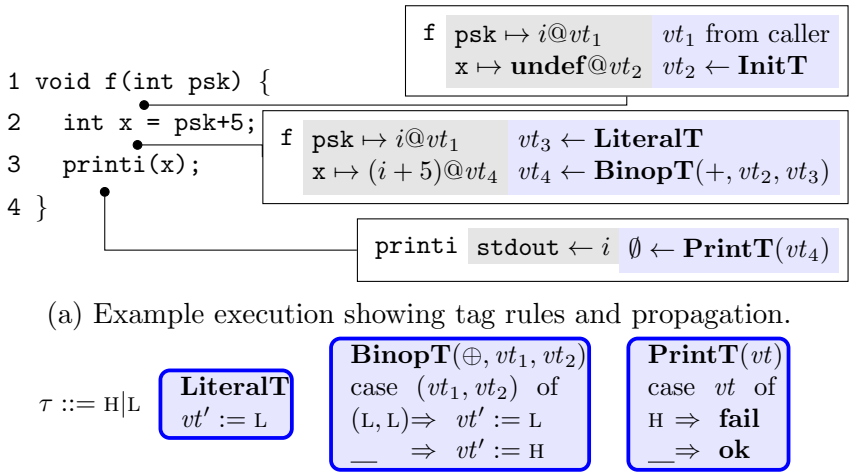
points resemble "join points" in aspect-oriented programming, but the "advice" in this case can only manipulate tags, not data.

Formally, Tagged C is defined as a variant C semantics that instruments ordinary execution with control points. At each control point, the semantics consults a user-defined set of tag rules to propagate tags and potentially halt execution. In the limiting case where no tag rules are defined, the semantics is similar to that of ordinary C, except that the memory model is very concrete; data pointers are just integers, and all globals, heap-allocated objects, and address-taken stack objects are allocated in the same integer-addressed memory space. Pointer arithmetic that would be undefined in standard C is defined in Tagged C: any properly-aligned integer can be used as a pointer into the memory space. Out-of-bounds loads and stores might fail, but if successful they present a consistent view of memory.

I build the Tagged C formal semantics on top of the CompCert C semantics, which is formalized as part of the CompCert verified compiler [49, 50]. Although motivated by PIPE, Tagged C is not tied to any particular enforcement mechanism. For prototyping and executing example policies, I provide a pure software implementation: a reference interpreter [1], also based on that of CompCert, written in the Gallina functional language of the Coq Proof Assistant [12]. Tag types and rules are also written directly in Gallina.

The choice of control points and their associations with tag rules, as well as the tag rules' signatures, form the essence of Tagged C's design. I have validated this design on the three classes of policies explored in Chapter 5, and, outside of a few known limitations related to `malloc`, structs, and unions (Section 6.4), I believe it

[1] Available at `https://github.com/SNoAnd/Tagged-C`



(a) Example execution showing tag rules and propagation.

(b) Secure Information Flow (pt. 1)

Figure 4.1: Tag Rules and Instantiation

is sufficiently expressive to describe most other flow-based policies, although further experience is needed to confirm this.

4.1 Metadata Tags and Policies, by Example

Consider a straightforward security requirement for a program that handles sensitive passkeys: "do not leak passkeys on insecure channels." This is an instance of a broad class of *secure information flow* (SIF) policies. Suppose the code on the left in Figure 4.1a is part of such a system, where `psk` is expected to be a passkey and `printi` prints an integer to an insecure channel, so `f` indirectly performs a leak via the local variable `x`. Let's walk through how a monitor defined in Tagged C could detect such a leak. (Of course, this particular leak could also be easily found using static analysis, but more complex ones might not.)

First we'll consider how the policy works as a whole. In Tagged C, all values carry

a metadata tag. Whenever execution reaches a control point, it consults an associated tag rule, to check whether the next execution step should be allowed to continue and if so, to update the tags. A policy consists of a tag type definition and instantiations of the tag rules for every control point. For a simple SIF policy like this one, the tag type is an enumeration containing H (high security) and L (low security). In this case, the input `psk` arrives in `f` with the tag H. This tag will be propagated along with the value through variable accesses, assignments, and arithmetic, according to generic rules that are not specific to this program. Finally, a special rule tied to the library function `printi` will check if the tag is H and, since it is, cause a failstop.

To understand how Tagged C is parameterized by a policy, we first see in Figure 4.1a the policy-independent framework under which tag rules are triggered in this program: the initial tag on `psk` (vt_0) and the tag given to the literal 5 by **LiteralT** are combined via the **BinopT** tag rule, and the result is passed to **PrintT** during the call. Figure 4.1a maps three points in the execution of `f` to descriptions of the corresponding program states, with the input value and all tags treated symbolically. In each state, the first column (white) shows the active function, the second (gray) gives the symbolic values and tags of variables in the local environment, and the third (blue) shows the rules that produce those tags. We write $v@vt$ for the value v tagged with vt. Identifiers are subscripted with the identifier namespace, e.g. $\texttt{f}_{fun}$ is the function name `f`. **undef** denotes an uninitialized value.

The SIF policy described informally above is implemented by instantiating the tag rules as shown in Figure 4.1b. The resulting behavior is best understood by mentally "weaving" together the two figures. Suppose `f` is called with an argument value $i@\text{H}$. The tag vt_3 on the constant `5` come from **LiteralT**, which tags all constants as L. (In

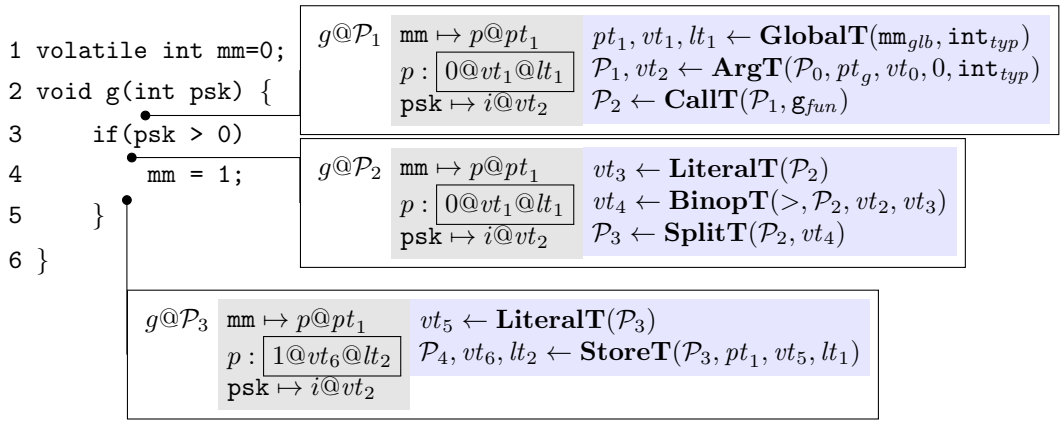


Figure 4.2: Second example showing tag rules and tag propagation.

tag rules, the assignment operator := denotes an assignment to the named tag-rule output, by convention written as primed metavariables t'.) The result of the addition on Line 2 is tagged by **BinopT** as the higher of the two inputs, so vt_4 is H. Finally, inside `printi`, **PrintT** is invoked to check if vt_4 should be printed. It has no output tags but can still failstop if its argument is H, which is the case in this execution.[2]

As a second example, Figure 4.2 steps through the execution of a function `g` that introduces two new wrinkles: the policy needs to keep track of metadata associated with addresses and with the program's control-flow state. Suppose `mm` is a memory-mapped device register that can be read from outside the program, so we want to avoid storing the passkey there; therefore we need a way to monitor stores to memory. Furthermore, although this code does not leak the passkey directly, it does so indirectly: since the store to `mm` is conditional on testing `psk`, an outside observer of `mm` can deduce one bit of the key (an *implicit flow* [24]).

[2]For simplicity, we omit showing tag rules that play no interesting role in this example: **AccessT** and **AssignT**, which are triggered each time a variable is read and assigned, respectively, as well as **ArgT** and **CallT**, which are triggered by the call itself.

In addition to tags on values, Tagged C attaches tags to memory locations (*location tags*, ranged over by lt) and tracks a special global tag called the PC tag (ranged over by $\mathcal{P}$, and attached to the function name in these diagrams). That makes three types of tags in total: value tags in the set τ_V, location tags in the set τ_L, and control tags in τ_C. Tagged C initializes the tags on `mm` with the **GlobalT** rule. Prior to the call to `g`, the **ArgT** tag rule processes the tag on `psk`, identifying it by its index (0) within the parameters. Then, as execution enters `g`, the PC tag at the point of call, $\mathcal{P}_0$, is fed to **CallT** to determine a new PC tag inside of `g`. The `if`-statement consults the **SplitT** rule to update the PC tag inside of its branch based on the value tag of the expression `psk < 0`. Once inside the conditional, when the program assigns to `mm`, it must consult the **StoreT** rule.

Figure 4.3 shows an instantiation of these rules that extend the previous SIF policy. In this case all three tag types are the same, $\tau_V = \tau_L = \tau_C = \{\text{L}, \text{H}\}$. The rule for globals initializes the location tag of `mm` to L, as a low-security output channel, and marks all other addresses H. **CallT** sets the PC tag to L on entry to each function. Whenever execution branches on a high-tagged value, the PC tag will be set to H. We modify the previous rules so that all expressions propagate the higher of the PC tag and the relevant value tag(s). This is shown for the updated **BinopT** in Figure 4.3. When an assignment is to a memory location, the store rule will check the tag on that location against the value being written, and failstop if a high value would be written to a low location. For this program, **SplitT** will set the PC tag to H, as it branches on a value derived from `psk`; then, at the write to `mm`, **StoreT** will fail rather than write to a low address in a high context. For the purposes of this example, the context remains high until `g` returns, though we will discuss an alternative to this

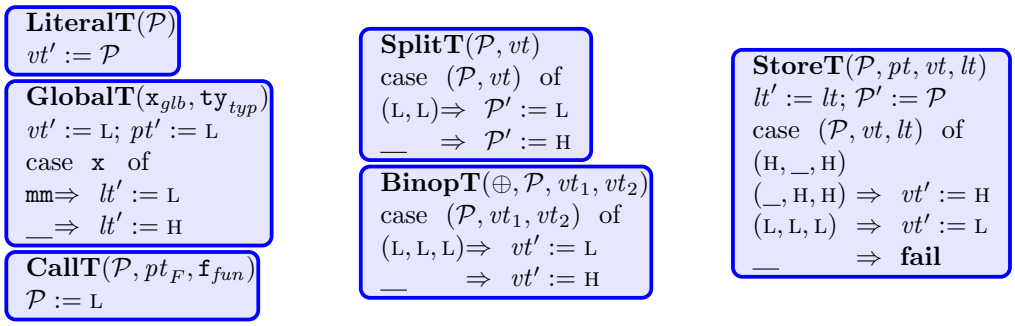


Figure 4.3: Tag rule instantiations for secure information flow (pt. 2)

later.

4.2 Tagged C Language

Tagged C contains almost all features of full ISO C 99.[3] Its semantics is based on that of CompCert C [49], a formalization of the C standard into a small-step reduction semantics. Tagged C's semantics differs from CompCert C's in two key respects: it has tags, and its memory model is concrete.

4.2.1 Base Semantics

Tagged C's semantics are based on those of CompCert C, so it will be helpful to briefly examine the latter. CompCert C uses a smallstep reduction semantics to evaluate expressions with side-effects. An expression is in either the "left-hand" or "right-hand" position. Left-hand expressions that are fully reduced are *lvalues*, which represent memory addresses. Fully reduced right-hand expressions are *rvalues*, which

[3]It inherits the limitations of CompCert C: the semantics assume a single-threaded environment, `setjump` and `longjump` are unsupported, and there are no variable-length arrays.

$$
\begin{aligned}
STATE_{CompCert} ::= \; & \mathcal{S}\,(m, le \mid s \gg k) \\
\mid \; & \mathcal{E}\,(m, le \mid e \gg k) \\
\mid \; & \mathcal{C}\,(m \mid f, args \gg k) \\
\mid \; & \mathcal{R}\,(m \mid v \gg k)
\end{aligned}
$$

(a) Base CompCert C

$$
\begin{aligned}
STATE_{TaggedC} ::= \; & \mathcal{S}\,(m, le, te \mid s \gg k \;@\mathcal{P}) \\
\mid \; & \mathcal{E}\,(m, le, te \mid e \gg k \;@\mathcal{P}) \\
\mid \; & \mathcal{C}\,(m \mid f, vt_F, args \gg k \;@\mathcal{P}) \\
\mid \; & \mathcal{R}\,(m \mid v@vt \gg k \;@\mathcal{P}) \\
\mid \; & \mathcal{F}\,(err)
\end{aligned}
$$

(b) Tagged C

Figure 4.4: Execution states

carry an element of the *val* type.

Figure 4.4a lists the states of a program. A normal state $\mathcal{S}\,(m, le \mid s \gg k)$ has the statement s "under focus," with memory and environment m and le and continuation k. (The '|' and '≫' symbols are just separators.) The continuation is effectively a stack of operations that will be performed later, including future statements to be executed, the contexts within which to place the result of a fully evaluated expression, and the destination that a call will return to. An expression state $\mathcal{E}\,(m, le \mid e \gg k)$ is similar, but with an expression under focus instead. A call state $\mathcal{C}\,(m, le \mid f, args \gg k)$ is about to enter the function f with a list of argument values *args*. A return state $\mathcal{R}\,(m, le \mid v \gg k)$ is about to return from some function with a return value v.

Tagged C has the same states, with some extra data and with an additional failstop state, $\mathcal{F}\,(err)$, which carries the error information that caused the failstop. All other states have an additional environment, the *temporary environment* (te), which contains private variables. They also track a PC tag, $\mathcal{P}$. A call state also records the tag on the function pointer that was called. The return value v and parameters *args* are now an atom (pair of a value and tag) and list of atoms, respectively.

In Tagged C an lvalue can be one of several kinds of locations. A memory location with accompanying lvalue-tag, written $\underline{p@pt}$, a function's location $\underline{id@pt}$, or a private

variable $\underline{\mathtt{x}}$. An rvalue is always an atom, a pair of a value and a tag, $v@vt$. The tags on either are drawn from a set of *val_tags*, with the pt and vt metavariables simply differentiating their roles.

An expression e might left-reduce to an lvalue l, written $e \Rightarrow_{\text{LH}} l$, or right-reduce to an atom $e \Rightarrow_{\text{RH}} v@vt$. In addition, either class of reduction can instead fail, $e \Rightarrow_k \bot(err)$.

Memory Unlike CompCert C, Tagged C has no memory-related UB. CompCert C models memory as a collection of disjoint blocks, and treats each variable as having its own block [51]. Instead, Tagged C places all public data in a single flat address space, and pointers are simply integer offsets into this space. Some unallocated addresses may be reserved by the compiler, in which case attempts to access them cause an explicit failstop, rather than UB.

This model is strong enough to support a reasonable notion of semantics-preserving compilation, without making any commitment about fine-grained memory safety, which is intentionally left for explicit tag policies to specify (see Section 5.1). In an implementation that compiles to PIPE, the private data can be protected by a small number of "built-in" tags.

Additional protection must be provided by the Tagged C policy. Policies that protect public data, whether they do so uniformly by enforcing memory safety or take a more targeted approach, typically distinguish ownership of a byte of memory via its location tag.

$$\frac{v_1 \langle\oplus\rangle v_2 = v'}{m, le, v_1 \oplus v_2 \Rightarrow_{\text{RH}} m, le, v'}$$

(a) CompCert C

$$\frac{v_1 \langle\oplus\rangle v_2 = v' \qquad vt' \leftarrow \textbf{BinopT}(\oplus, \mathcal{P}, vt_1, vt_2)}{\mathcal{P}, m, le, te, v_1@vt_1 \oplus v_2@vt_2 \Rightarrow_{\text{RH}} \mathcal{P}, m, le, te, v'@vt'}$$

(b) Tagged C

Figure 4.5: Binary operation, without and with tags

Tags Tagged C's values and states are annotated with metadata tags, and its reductions contain control points, which are hooks within the operational semantics at which the tag policy is consulted and either tags are updated, or the system failstops. Tagged C relies on a fixed number of predefined control points, kept small in order to simplify and organize the task of the policy designer. A control point consists of the name of a *tag rule* and the bindings of its inputs and outputs.

For example, Figure 4.5 shows the expression step reduction for binary operations in CompCert C vs. Tagged C.

In the "tagless" version, a binary operation $\oplus$ is applied to two subexpressions that are values. It reduces to a single value expression based on the semantics of the operation, $\langle\oplus\rangle$. On the right, all value expressions now additionally carry tags, and the tag on the result is determined by the **BinopT** rule.

A *policy* consists of a type definition for each of the three sets of tags (τ_V, τ_L, and τ_C) and an instantiation of each tag rule. The tag rule itself is instantiated as a partial function; if a policy leaves a tag rule undefined on some inputs, then those inputs violate the policy, sending execution into a special failstop state. The names and signatures of all the tag rules, and their corresponding control points, are summarized in Table 4.1. Motivating examples of their use appear in Chapter 5.

4.3 Control Points

The Tagged C control points and their signatures follow a basic design philosophy, expanded as needed to implement specific policies. The key principle is that changing a value always (potentially) changes its tag, and the dependencies between values must be reflected in dependencies between tags.

In the context of these principles, the PC tag represents dependencies between the current executing statement or expression, and the rest of the program state. With the exception of **GlobalT**, which is invoked at initialization-time rather than runtime, all tag rules take the PC tag as an argument, and all impure expressions have the potential to update it.

In this scheme, pure expressions take as arguments the PC tag $\mathcal{P}$ and any operand tags, and produce a tag for the result of the expression (**LiteralT** for constants, **UnopT** and **BinopT** for operations, **FieldT** for struct and union fields, and **AccessT** and **LoadT** as described below). Impure expressions additionally produce a new PC tag (**AssignT**, **StoreT**, and **ExprSplitT**).

The distinction between **AccessT** and **LoadT**, and between **AssignT** and **StoreT**, corresponds to private (non-address-taken) and public (allocated in public memory) variables. All reads of variables invoke **AccessT**, and all assignments invoke **AssignT**, so that the behavior of the variable itself is independent of where it is stored. Public variables additionally use the **LoadT** and **StoreT** rules to add restrictions to how variables in memory are accessed.

The **ExprSplitT** tag rule updates the PC tag when an expression branches based on a value; it is paired with **ExprJoinT**, which updates the PC tag again when the

Rule Name	Inputs	Outputs	Control Points
CallT	$\mathcal{P}, pt_F, \mathtt{f}_{fun}$	$\mathcal{P}'$	Update PC tag at call
ArgT	$\mathcal{P}, pt_F, vt, n, \mathtt{arg}_{typ}$	$\mathcal{P}', vt'$	Per argument at call
RetT	$\mathcal{P}_{CLE}, \mathcal{P}_{CLR}, pt_F, vt, \mathtt{ret}_{typ}$	$\mathcal{P}', vt'$	Return value and PC tag
LoadT	$\mathcal{P}, pt, vt, \overline{lt}$	vt'	Memory loads
CoalesceT	$\overline{vt}$	vt'	Multi-byte loads
StoreT	$\mathcal{P}, pt, vt, lt$	$\mathcal{P}', vt', lt'$	Memory stores
EffectiveT	$\overline{vt}$	vt'	Multi-byte stores
AccessT	$\mathcal{P}, vt$	vt'	Variable accesses
AssignT	$\mathcal{P}, vt_1, vt_2$	$\mathcal{P}', vt'$	Variable assignments
UnopT	$\odot, \mathcal{P}, vt$	vt'	Unary operation
BinopT	$\oplus, \mathcal{P}, vt_1, vt_2$	vt'	Binary operation
LiteralT	$\mathcal{P}$	vt'	Applied to literals
InitT		vt'	Applied to fresh variables
SplitT	$\mathcal{P}, vt, \mathtt{L}_{lbl}$	$\mathcal{P}'$	Statement control splits
LabelT	$\mathcal{P}, \mathtt{L}_{lbl}$	$\mathcal{P}'$	Labels/arbitrary code points
ExprSplitT	$\mathcal{P}, vt$	$\mathcal{P}'$	Expression control splits
ExprJoinT	$\mathcal{P}, vt$	$\mathcal{P}', vt'$	Join points in expressions
GlobalT	$\mathtt{x}_{glb}, \mathtt{ty}_{typ}$	pt', vt', lt'	Program initialization
FunT	$\mathtt{f}_{fun}, \mathtt{ty}_{typ}$	pt'	Program initialization
LocalT	$\mathcal{P}, \mathtt{x}_{lcl}, \mathtt{ty}_{typ}$	$\mathcal{P}', pt', lt'$	Stack allocation (per var.)
DeallocT	$\mathcal{P}, \mathtt{ty}_{typ}$	$\mathcal{P}', vt', lt'$	Stack deallocation (per var.)
MallocT	$\mathcal{P}, pt_f$	$\mathcal{P}', pt', vt', lt_{header}$ $lt_{body}, lt_{padding}$	Call to `malloc`
FreeT	$\mathcal{P}, pt, vt, \overline{lt}$	$\mathcal{P}', lt'$	Call to `free`
ClearT	$\mathcal{P}, pt, lt$	lt'	Call to `free`
FieldT	$\mathcal{P}, pt, \mathtt{ty}_{typ}, \mathtt{x}_{glb}$	pt'	Structure/union field access
CastToPtrT	$\mathcal{P}, vt, lt^*, \mathtt{ty}_{typ}$	vt'	Cast from pointer to scalar
CastOtherT	$\mathcal{P}, vt, \mathtt{ty}_{typ}$	vt'	Cast from scalar to pointer

Table 4.1: Full list of tag-rule signatures and control points. Different classes of identifiers are ranged over with the metavariables: $\mathtt{f}_{fun}$, function identifiers; n, the index of a function argument; $\mathtt{x}_{glb}$, global variable names; $\mathtt{L}_{lbl}$, labels; and $\mathtt{ty}_{typ}$, types.

branches have rejoined. Similarly, **SplitT** updates the PC during branching statements. The **LabelT** rule can changes the PC tag at any labeled point in execution, and handles join points following branch statements.

ArgT checks and updates the tags on arguments prior to a call, based on the tag of the function pointer and the index and type of each argument. Then **CallT** updates the PC tag as control transfers to the callee, given the callee's name and the tag on the function pointer through which it is called. **RetT** handles the equivalent checks on return. It takes both the caller's PC tag from before the call ($\mathcal{P}_{CLR}$), the callee's ($\mathcal{P}_{CLE}$), and the tag on the return value, and it updates the PC tag as well as the tag on the return value.

Newly initialized variables are tagged according to the **InitT** rule, as well as **LocalT** if they are public locals; the lt' tag returned by the latter is used to tag the memory occupied by the variable. Similarly, global variables are initialized before runtime based on the **GlobalT** rule. **DeallocT** returns an lt' tag used to re-tags the memory of deallocated locals.

The heap equivalents of **LocalT** and **DeallocT** are **MallocT** and **FreeT**. Again, the lt' tags returned by these functions are used to tag and re-tag the allocated memory. Other library functions have the tags of their results tagged by the **ExtCallT** tag rule.

The cast rules are specialized based on whether the original type or the new type is a pointer, or both, because casts to and from pointers can make use of the location tags at their targets. This enables a PNVI memory safety policy (Section 5.1), and more generally policies that keep track of the correspondence between pointers and their targets.

This design philosophy sometimes leads to expressions with no control points. The `*` and `&` operators, which turn lvalues into rvalues and vice versa without actually changing their values, do not need control points under this model.

There is always the chance that new policies might arise for which the current set of control points proves to be inadequate. There is no conceptual reason why control points cannot be added or given modified signatures as needed, but extending the interpreter and (eventually) the compiler would be non-trivial. Care needs to be taken in designing control points that are amenable to compilation for PIPE: Tag rule evaluation has a complicated interaction with compiler optimization [16], and some potentially useful tag rule signatures (such as updating tags on operation inputs to enforce non-aliasing of pointers) would require the compiler to generate extra instructions to work around limitations of the PIPE hardware.

4.4 Implementation

The current Coq-based implementation of Tagged C consists of a formal semantics and a matching interpreter written in Coq's Gallina language. These are based on the `Csem` and `Cexec` modules from the CompCert compiler (version 3.10) [48]. The interpreter can be extracted to a stand-alone OCaml program and further compiled to a native executable.

5 Tagged C Policies

This chapter discusses concrete policy implementations and how they motivate Tagged C's control point design. Memory safety policies inform our requirements for memory tags and type casts. Compartmentalization policies depend on the call- and return-related control points, to keep track of the active compartment. Secure information flow policies expose the many places where the user may need to reference identifiers from their program in the policy itself. Taken together, these example policies illustrate Tagged C's breadth of application.

5.1 Memory Safety

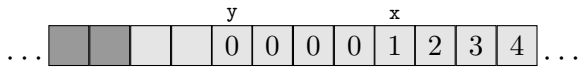
Tagged C can be used to enforce memory safety with respect to different *memory models*—formal or informal descriptions of how C should handle memory. This section first discusses how Tagged C treats variables in and out of memory, before any policies. Then it shows how to enforce the CompCert C memory model and two models proposed by Memarian et al. [54] for the purposes of supporting low-level idioms in the presence of compiler optimization, focusing in particular on how each policy handles casts from pointers to integers and back.

Public-Private Separation Following Tolmach et al. [76], Tagged C distinguishes between public and private data. Variables can be private if they are scalar locals that do not have their addresses taken; address-taken locals, local arrays, globals,

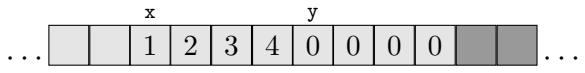
```
int f(int i) {
2:  char[4] x;
3:  int y = 0;
4:
5:  x[i] = 42;
6:  return y;
}
```

(a) Example code

(b) Private



(c) Public 1



(d) Public 2

Figure 5.1: Public vs. Private

and heap-allocated objects are all public. Private variables are stored in their own environment separate from memory. This corresponds full stack safety as described in 3.4.1.

The code in Figure 5.1a can behave differently depending on where `y` ends up stored. In Figure 5.1b, `y` is private, while `x` as an array is public. The dark gray boxes represent bytes that are reserved for the compiler's use[1]. If `f` tries to access them then the system will failstop, but if `f` returns it will always return 0 because there is no way to modify `y`. In Figure 5.1c and Figure 5.1d, two of many possible allocation patterns, `x[i]` might alias with a byte of `y`. If `i` is 4 or -4, respectively, it would overwrite the first byte with 42.

Figure 5.2 steps through the assignment at Line 5, with `x` located at address 100, `y` at address 104, and the parameter `i` private with value `4`. The initial expression

[1]Likely to hold a return pointer

State	Reduction	
$\mathcal{E}\left(m \mid *(\lVert \boxed{\texttt{x}} \rVert + \lVert \texttt{i} \rVert) = 42 \gg \_ \ @\mathcal{P}\right)$	$\dfrac{e[\texttt{x}] = 100@pt_x}{\mathcal{P}, m, e, te, \texttt{x} \Rightarrow_{\text{LH}} \mathcal{P}, m, e, te, \underline{100@pt_x}}$	(1)
$\mathcal{E}\left(m \mid *(\lVert \underline{100@pt_x} \rVert + \lVert \boxed{\texttt{i}} \rVert) = 42 \gg \_ \ @\mathcal{P}\right)$	$\mathcal{P}, m, e, te, \texttt{i} \Rightarrow_{\text{LH}} \mathcal{P}, m, e, te, \underline{\texttt{i}}$	(2)
$\mathcal{E}\left(m \mid *(\boxed{\lVert \underline{100@pt_x} \rVert} + \lVert \underline{\texttt{i}} \rVert) = 42 \gg \_ \ @\mathcal{P}\right)$	$\dfrac{pt'_x \leftarrow \textbf{AccessT}(\mathcal{P}, vt)}{\mathcal{P}, m, e, te, \lVert \underline{100@pt_x} \rVert \Rightarrow_{\text{RH}} \mathcal{P}, m, e, te, 100@pt'_x}$	(3)
$\mathcal{E}\left(m \mid *(100@pt_x + \boxed{\lVert \underline{\texttt{i}} \rVert}) = 42 \gg \_ \ @\mathcal{P}\right)$	$\dfrac{te[\texttt{i}] = 4@vt_i \quad vt' \leftarrow \textbf{AccessT}(\mathcal{P}, vt_i)}{\mathcal{P}, m, e, te, \lVert \underline{\texttt{i}} \rVert \Rightarrow_{\text{RH}} \mathcal{P}, m, e, te, 4@vt'}$	(4)
$\mathcal{E}\left(m \mid *(\boxed{100@pt_x + 4@vt_i}) = 42 \gg \_ \ @\mathcal{P}\right)$	$\dfrac{pt' \leftarrow \textbf{BinopT}(+, pt_4, vt_i)}{\mathcal{P}, m, e, te, 100@pt_x + 4@vt_i \Rightarrow_{\text{RH}} \mathcal{P}, m, e, te, 104@pt'}$	(5)
$\mathcal{E}\left(m \mid \boxed{*(104@pt')} = 42 \gg \_ \ @\mathcal{P}\right)$	$\mathcal{P}, m, e, te, *(104@pt') \Rightarrow_{\text{LH}} \mathcal{P}, m, e, te, \underline{104@pt'}$	(6)
$\mathcal{E}\left(m \mid \underline{104@pt'} = \boxed{42} \gg \_ \ @\mathcal{P}\right)$	$\dfrac{vt \leftarrow \textbf{LiteralT}(\mathcal{P})}{\mathcal{P}, m, e, te, 42 \Rightarrow_{\text{RH}} \mathcal{P}, m, e, te, 42@vt}$	(7)
$\mathcal{E}\left(m \mid \boxed{\underline{104@pt'} = 42@vt} \gg \_ \ @\mathcal{P}\right)$	$\dfrac{m[100] = \_@vt_{old}@lt \quad \mathcal{P}', vt' \leftarrow \textbf{AssignT}(\mathcal{P}, vt) \quad \mathcal{P}'', vt'', lt' \leftarrow \textbf{StoreT}(\mathcal{P}', vt') \quad m' = m[100 \mapsto 42@vt''@lt']}{\mathcal{P}, m, e, te, \underline{104@pt'} = 42@vt \Rightarrow_{\text{RH}} \mathcal{P}, m', e, te, 42@vt}$	(8)
$\mathcal{E}\left(m' \mid \texttt{skip} \gg \_ \ @\mathcal{P}'\right)$		

Figure 5.2: Line 5, step-by-step

state is $\mathcal{E}\left(m \mid *(\lVert \texttt{x} \rVert + \lVert \texttt{i} \rVert) = 42 \gg \_ \ @\mathcal{P}\right)$. Steps (1) and (2) lookup the lvalues of `x` and `i` by name. Since `x` is in memory, its lvalue is its address and comes with a tag, determined when it was allocated. Then Steps (3) and (4) access those lvalues. In the case of `x`, as an array its rvalue is just a pointer to the same location. Each variable access invokes the **AccessT** rule. Step (5) performs the addition, and Step (6) dereferences the resulting pointer to obtain the lvalue of `x[4]`. On the right-hand side of the assignment, Step (7) turns the integer literal `42` into an atom, tagging it with **LiteralT**. Finally, Step (8) performs the assignment, invoking both **AssignT** and **StoreT**. (A third rule, **EffectiveT**, is omitted from this example because the store is a single byte; see Section 6.1.1 for details.)

5.1.1 Protecting Memory

The basic idea for enforcing memory safety is a "lock and key" approach [9,19]. These rules are given in Figures 5.4 and 5.5. When an object is allocated, it is assigned a unique "color," and its memory locations, as well as its lvalue, are tagged with that color, written $\textsc{clr}(c)$. The default tag N indicates a non-pointer or non-allocated location. In this case, `x` and `y` are local variables, so their location tags and lvalue tags produced by invocations of **LocalT**. Then, when an lvalue is eventually used to load or store a variable in memory, the **LoadT** and **StoreT** rules compare the lvalue's tag with the tags on locations being accessed. The lvalue is valid if they match, otherwise the policy failstops. (The `assert` command failstops if its argument does not hold.)

Between allocation and access, an lvalue or pointer may be subject to other operations, especially pointer arithmetic. The **BinopT** rule determines whether binary operations are legal on its operands and, if so, whether the result remains a valid pointer. The exact rules for pointer validity vary between memory models. In general, a pointer that is taken out of bounds of its target is still valid as long as it is brought back to point at the correct memory before it is accessed.

The PC tag tracks the colors that have already been used to determine the next available color for new allocations. Allocations are illustrated in Figure 6.6.

Memory Model Policies While the idea of a valid pointer may seem obvious, the precise definition can vary. The C standard does not support arbitrary arithmetic on pointers or their integer casts. In practice, it is common for programs to violate the C standard to various degrees; see Figure 5.3. For example, if objects are known to

be aligned to 2^n-byte boundaries, the low-order n bits of pointers can be "borrowed" to store other data [55]. The possible presence of these low-level idioms means that there is no one-size-fits all memory safety policy. CompCert C's definition of a valid pointer allows the pointer to be cast into an integer and back, but only if its value does not change in the interim. This is very strict! Programs that use low-level idioms would failstop if run under a policy that enforces this.

Memarian et al.'s ([54]) first memory model, *provenance via integer* (PVI), treats memory as a flat address space, and pointers as integers with additional provenance information associating them to their objects. Pointers maintain this provenance even through casts to integers and the application of arithmetic operations. When cast back, the pointers will still be associated with the same object. This enables many low-level idioms, while still forbidding memory-safety violations like buffer overflows.

On the other hand, their second model, *provenance not via integer* (PNVI), clears the provenance of a pointer when it is cast to an integer. When an integer is cast to a pointer (whether or not it was previously derived from a pointer), it takes on the provenance of whatever it points to at that time. The security properties of this memory model are questionable, but it is a realistic option for a compiler to choose and can support useful idioms that PVI cannot.

The specific memory models (Fig. 5.5) behave differently when pointers are cast to integers and back. The CompCert C variant marks that the integer has been cast from a valid pointer, and restores that provenance when cast back. But **BinopT** will failstop if the integer is actually modified between the casts. PVI simply keeps the provenance and allows all operations between casts. PNVI accesses the memory pointed to by the cast pointer and takes its location tag.

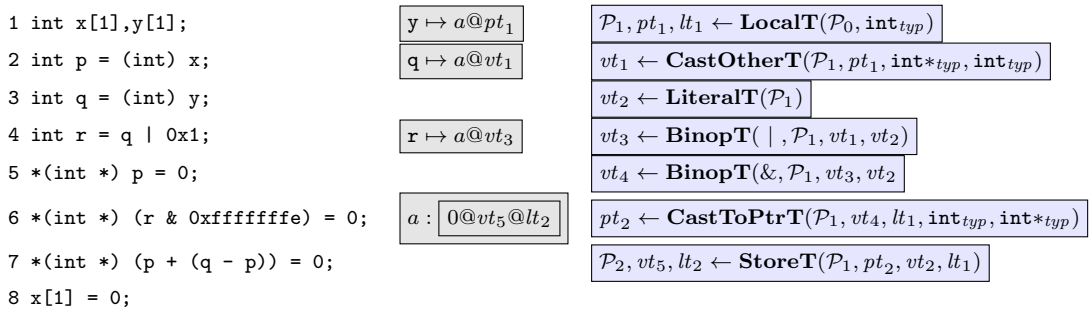


Figure 5.3: Memory safety and pointer casts, tracing `y`, `q`, and `r`. (Assume `int` and pointers are 32 bits. Some tag rules omitted.) Line (5) is always legal, (6) is illegal in CompCert C due to bitwise arithmetic not preserving provenance, (7) is also illegal in PVI due to combining provenance of multiple objects, and (8) is illegal in all models.

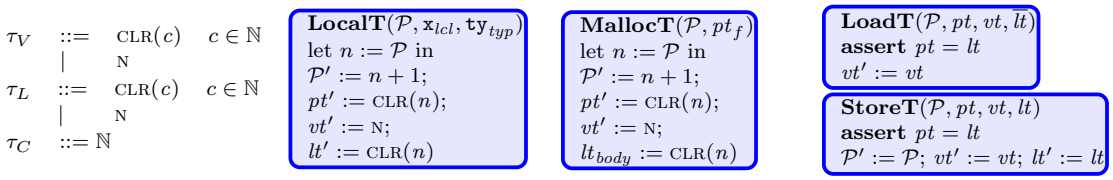


Figure 5.4: Generic Memory Safety Rules

Temporal Memory Safety The tag rules described so far only enforce spatial memory safety, but Tagged C can also enforce temporal safety. A full memory safety policy prevents use-after-free and double-free errors by either retagging a deallocated region, or using the PC tag to track the set of live objects and revoking permissions on an object as soon as it is freed.

5.2 Compartmentalization

In principle, the monitoring techniques in the previous section could be used to detect all unintended memory safety violations (albeit only at run time) and ultimately to fix them. In reality, the cost and risk of regressions may make it undesirable to fix

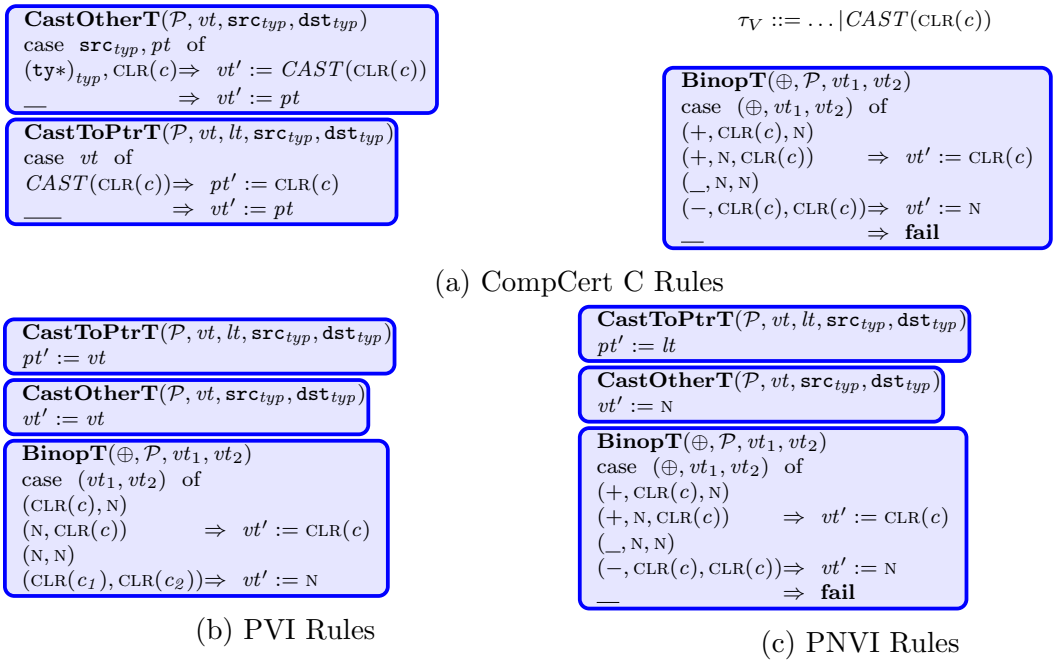


Figure 5.5: Specialized Memory Safety Rules

bugs in older code [13]. A compartmentalization policy can isolate potentially risky code, such as code with unfixed (or intentional) UB, from safety-critical code, and enforce the *principle of least privilege*. Even in the absence of language-level errors, compartmentalization can usefully restrict how code in one compartment may interact with another. External libraries are effectively required for most software to function, yet represent a supply-chain threat; isolating them prevents vulnerabilities in the library from compromising critical code, and limits the tools available to attackers in the event of a compromise.

Assume we have been given a compartmentalization policy, with at least two compartments, to add to the system after development. The compartments and what belongs in them are represented in the policy by a set of compartment identifiers,

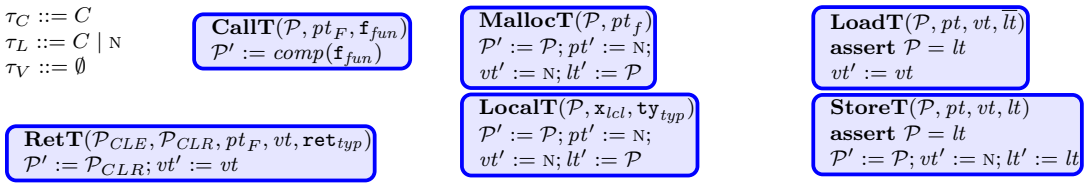


Figure 5.6: Simple Compartmentalization Policy

ranged over by C, and a map from function and global identifiers to compartments, written as $comp(id)$.

Implementation Compartmentalization requires the policy to keep track of the active compartment, which means keeping track of function pointers. In Tagged C, function pointers are the exception to the concrete memory model. They carry symbolic values that refer uniquely to their target functions. If `f` is located at the symbolic address α, then the expression `&f` evaluates to $\alpha@\mathtt{f}_{fun}$. When the function pointer is called, Tagged C invokes **CallT**$(\mathcal{P}, pt_F, \mathtt{f}_{fun})$, where pt is the function pointer's tag, to update the PC tag. On return, in addition to handling the return value (if any), **RetT**$(\mathcal{P}_{CLE}, \mathcal{P}_{CLR}, pt_F, vt, \mathtt{ret}_{typ})$ determines a new PC tag based on the one before the call ($\mathcal{P}_{CLR}$) and the one at the time of return ($\mathcal{P}_{CLE}$). In our compartmentalization policy (Fig. 5.6), we define a tag to be a compartment identifier or the default N tag. The PC tag always carries the compartment of the active function, kept up to date by the **CallT** and **RetT** rules.

Once the policy knows which compartment is active, it must ensure that compartments do not interfere with one another's memory. A simple means of doing so is given in Figure 5.6: any object allocated by a given compartment, whether on the stack or via `malloc`, is tagged with that compartment's identity, and can only be ac-

cessed while that compartment is active. This is very limiting, however! In practice, compartments need to be able to share memory, such as in the common case where libraries have separate compartments from application code. One solution is to allow compartments to share selected objects by passing their pointers, treating them as *capabilities*—unforgeable tokens of privilege. One such compartmentalization model is the focus of Chapter 7.

Compartmentalization Variants Using program-specific tags for globals and functions, a policy like the one above can be extended with a Mandatory Access Control (MAC) policy [46]. Here, a table explicitly identifies which compartments may call one another's functions, which global variables they can access, and with which other compartments they can share memory.

5.3 Secure Information Flow

Finally, we return to the family of *secure information flow* (SIF) [25] policies introduced in Section 4.1. SIF deals with enforcing higher-level security concerns, so it is useful even in code with no language-level errors.

In Figure 5.7, the program checks the format of the passkey `psk`, which is tagged H, and uses a switch statement to perform operations on it based on the result. As in Figure 4.2, this means that the policy should "raise" the PC tag to H to indicated that the program's control-flow depends on `psk`. After control reaches label `J`, the PC tag can be lowered again, because code execution from this point on no longer depends on `psk`. `J` is a *join point*: the point in a control-flow graph where all possible

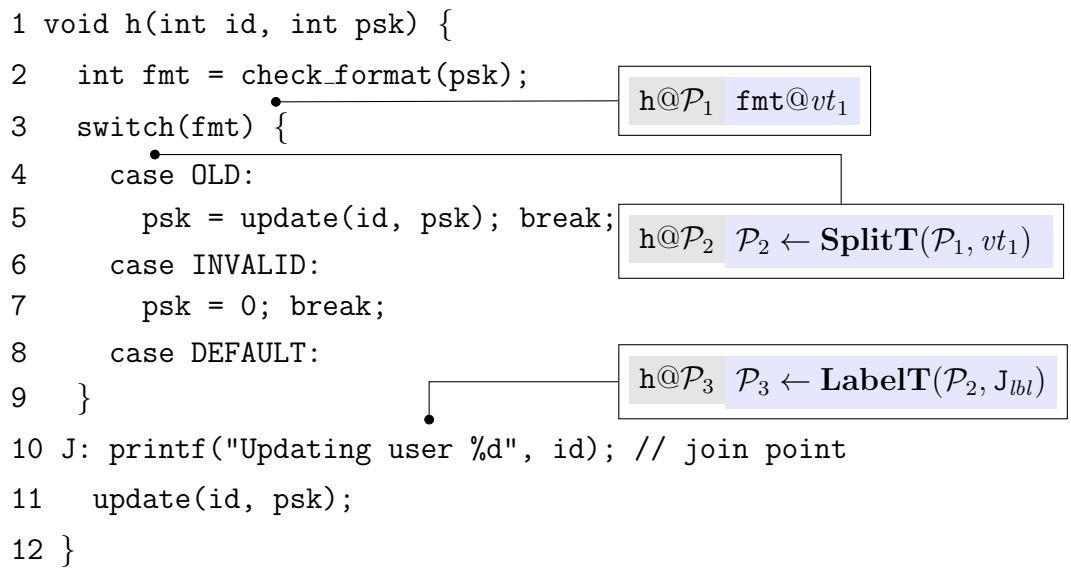


Figure 5.7: Not an Implicit Flow

$$\tau_V ::= \mathrm{H} \mid \mathrm{L} \qquad \tau_C ::= lbls$$

$$|t| \triangleq \begin{cases} \mathrm{L} & \text{if } t = \mathrm{L} \text{ or } t = \mathrm{PC}\ \mathtt{f}_{fun}\emptyset \\ \mathrm{H} & \text{otherwise} \end{cases}$$

$$t_1 \sqcup t_2 \triangleq \begin{cases} \mathrm{L} & \text{if } |t_1| = |t_2| = \mathrm{L} \\ \mathrm{H} & \text{otherwise} \end{cases}$$

SplitT$(\mathcal{P}, vt, \mathtt{L}_{lbl})$
case $\mathcal{P}, vt$ of
$lbls, \mathrm{H} \Rightarrow \ \mathcal{P}' := (lbls \cup \mathtt{L}_{lbl})$
$\_, \mathrm{L} \ \Rightarrow \ \mathcal{P}' := \mathcal{P}$

LabelT$(\mathcal{P}, \mathtt{L}_{lbl})$
let $lbls := \mathcal{P}$ in
$\mathcal{P}' := (lbls - \mathtt{L}_{lbl})$

Figure 5.8: SIF Conditionals

routes from the split to a return have re-converged, which can be identified statically as the immediate post-dominator of the split point [25].

The **SplitT** and **LabelT** tag rules support policies that reason about splits and joins. Every transition that tests a value as part of a conditional or loop contains a control point that invokes **SplitT**, passing the label for the corresponding joint point. (This label argument is optional, both because some policies may not care about join points, and because for some functions the only join point may be after the return.) This way, the policy can react when execution reaches that label via the **LabelT** rule.

The full SIF policy keeps track of the pending join points within the PC tag, and lowers the PC tag when execution reaches the corresponding join point. (A similar approach applies to conditional expressions, but we omit the details here.) Value tags are the usual L and H, and the PC carries a set of label identifiers to record the join points of tainted statement scopes. Initially, the PC tag is $\emptyset$, which corresponds to "low" security. The join operator, $\cdot \sqcup \cdot$, takes the higher of its arguments after reducing a PC tag into either H or L.

In order to use this version of SIF, the program must undergo a minor automatic transformation by the compiler or interpreter, introducing explicit labels at all join points that don't already have one. In the example, `J` becomes an explicit label in the code. The internal syntactic form of each conditional statement (`if`, `switch`, `while`, `do-while`, and `for`) carries this optional label. If the conditional branches on a high value, **SplitT** adds the label to the set in the PC tag. Later, when execution reaches a label, **LabelT** deletes it from the set. If the set is non-empty, there is at least one high split point that has not yet reached its join point, so we treat the PC tag as high. When execution reaches the last join point and the set is empty, the PC tag is treated as low, because it is no longer possible to deduce which path was taken.

SIF Variants SIF can cover many different policies. We have shown an instance of a confidentiality policy, but SIF can also support integrity ("insecure inputs do not affect secure data"), intransive policies ("data can flow from A to B and B to C, but not from A to C"), and policies with more than two security levels.

To give a couple of more realistic examples, an intransitive integrity policy could be used to protect against SQL injections by requiring unsafe inputs to pass through

a sanitizer before they can be appended to a query. Similarly, a more complex SIF policy could ensure that data at rest is always encrypted, by setting a low security level to the outputs of an encryption routine and a high level to the outputs of its corresponding decryption routine.

5.4 Combining Policies

$$
\begin{array}{l}
\mathbf{UnopT}(\odot, (\mathcal{P}_1, \mathcal{P}_2), (vt_1, vt_2)) \\
\text{let } vt_1' := \mathbf{UnopT}_A(\odot, \mathcal{P}_1, vt_1) \text{ in} \\
\text{let } vt_2' := \mathbf{UnopT}_B(\odot, \mathcal{P}_2, vt_2) \text{ in} \\
vt' := (vt_1', vt_2');
\end{array}
$$

Figure 5.9: Example Rule in Policy Product

Why limit protection to just one policy? Tagged C supports policies that reference other policies, the most useful of which is the "product" policy, which takes the Cartesian product of two policies. Figure 5.9 shows an example of how this works for the **UnopT** rule. If policy A has value tag type ${\tau_V}^A$ and policy B has ${\tau_V}^B$, then policy $A \times B$ should have the value tag type $\tau_V = {\tau_V}^A \times {\tau_V}^B$, and each rule will dispatch its counterpart in both policies, then combine the results. If either fails, the entire rule fails.

Other combinations are possible, including the "sum" or "tagged union" of policies. One related policy that turns out to be very useful is one that simply logs all rule invocations and their arguments, then dispatches into some other, less verbose policy.

6 Details and Lessons Learned

In this chapter I discuss the nuts and bolts of the Tagged C development and how it evolved over time.

To adapt CompCert, I replace the standard block-offset memory with a concrete one, leaving the block-based system to handle only function pointers. I rework global environments to separate (symbolic) function pointers from other (concrete) pointers. I also add a temporary environment to contain private variables, and semantics rules to deal with them. Most importantly, I thread the tags into the state and add control points to the relevant semantic rules. These changes appear in both the semantics and the corresponding interpreter code. The interpreter correctness proof also involved a lot of new proof automation work.

6.1 Control Points Under a Microscope

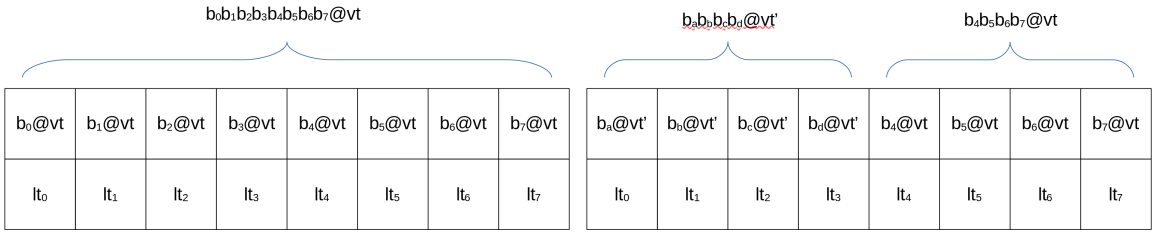
This section looks closely at a few more complicated control points and the motivation for their design. The interaction between tags and memory is not always obvious. As such, there are some interesting points to discuss about the the call and return structure, and Tagged C's handling of casts and of control flow are both strongly informed by the needs of specific policies. Finally, system and library calls pose a whole host of challenges.

6.1.1 Memory, Loads, and Stores

Location Tags The existence of location tags is a matter of convenience. Consider the PVI memory-safety policy in Section 5.1: tags are used to identify valid pointers by the allocation that produced them, and to identify the locations in memory that are associated with those allocations. When a value is loaded from memory, the policy needs to (1) check that it is in an allocation that is legal to read, and (2) propagate its pointer identity (or lack thereof) along with the value. In a world without location tags, this could be implemented by defining a value tag as a pair, one element of which always stays with the in-memory value while the other is propagated. It is simpler to differentiate the component that stays put from the one that is moved from the start.

Per-byte Tagging Many Tagged-C values occupy multiple bytes in memory, but conceptually each value has but a single value tag. This can cause unintuitive behavior if a value is partially overwritten, which can happen if a value of a smaller type is written to it, or in the case of a union whose fields have different sizes. In the concrete representation of memory, each byte has both a location tag and a value tag, and a multi-byte store copies the the value tag across each byte. Figure 6.1a shows an 8-byte value written in this way: each byte has a location tag and a value tag, but the value tags are all the same. If the first four bytes are overwritten with a new value, the result is Figure 6.1b: the latter bytes remain behind, along with their value tags.

Unlike the value tag, when location tags are written to memory they might not be uniform. It depends on the control point in question, and whether there is enough information to usefully differentiate them. This capability is intended to be used for



(a) 8 Bytes with Tags in Memory (b) Overwriting 4 Bytes

Figure 6.1: Partial Values in Memory

substructural memory safety or possibly to look for memory alignment issues.

The **CoalesceT** and **EffectiveT** tag rules determine how the policy treats mismatched value tag in multi-byte values. **CoalesceT** turns a list of value tags into a single value tag; in most policies it turns a list of identical tags into just one of that tag, and failstops if they are not identical. **EffectiveT** is used for assignments. Recall that **AssignT** takes as an argument the value tag of the value being overwritten. If there are multiple bytes, **EffectiveT** defines the "effective tag" of those bytes. **CoalesceT** and **EffectiveT** are separate because the appropriate way to treat mismatched tags may differ depending on whether they are being loaded or overwritten. Overwriting mismatched tags is generally safe, and if there is a reason not to overwrite them, it can be handled by **AssignT**, so **EffectiveT** does not fail.

Finally, loads and stores involving memory use the **LoadT** and **StoreT** rules to determine if the memory access is valid. These both take as parameters the tag on the pointer being accessed and all of the location tags in the footprint of the load or store. To summarize, the steps involved in a public load are:

1. The **CoalesceT** rule combines the value tags on the footprint into one tag.

2. The **LoadT** rule checks the relationship between the tag on the pointer be-

AccessT :	τ_C	PC tag			
	$\rightarrow\tau_V$	Tag on read value	**LoadT** :	τ_C	PC tag
	$\rightharpoonup\tau_V$	New tag on read value		$\rightarrow\tau_V$	Pointer tag
CoalesceT :	τ_C	PC tag		$\rightarrow\tau_V$	Coalesced tag
	$\rightarrow$*list* τ_V	Value tags, per byte		$\rightarrow$*list* τ_L	Location tags, per byte
	$\rightharpoonup\tau_V$	Single value tag		$\rightharpoonup\tau_V$	Tag on result value

Figure 6.2: Accessing Variables

ing accessed, the tag on the value being loaded, and the tags on the memory locations being accessed.

3. **AccessT** is invoked to determine the tag on the result value, just as in any other access.

And the steps involved in a public store are:

1. The **EffectiveT** rule interprets multiple value tags in memory into a single effective tag.

2. **AssignT** governs the relationship between the incoming value tag and the effective tag on the data being overwritten.

3. **StoreT** is invoked to determine the new tags on the memory locations being written to, based on the pointer tag, current location tags, and value being written.

6.1.2 Calls and Returns

Function calls and returns are fairly complex, each taking two steps to setup and tear-down the callee's state. A call starts from an expression state, with a "call"

AssignT :	τ_C	PC tag	
	$\rightarrow\tau_V$	Tag on old value	
	$\rightharpoonup(\tau_C$	New PC tag	
	$\times\ \tau_V)$	Tag on new value	
EffectiveT :	$list\ \tau_V$	Value tags, per byte	
	$\rightharpoonup\tau_V$	Input to **AssignT**	

StoreT :	τ_C	PC tag
	$\rightarrow\tau_V$	Pointer tag
	$\rightarrow\tau_V$	Tag on value
	$\rightarrow list\ \tau_L$	Old loc. tags
	$\rightharpoonup(\tau_C$	New PC tag
	$\times\ \tau_V$	New tag on value
	$\times\ list\ \tau_L)$	New loc. tags

Figure 6.3: Assigning to Variables

$$\mathcal{E}\ (m, en, te \mid C[f@pt_f(42@vt)] \gg k\ @\mathcal{P})$$

Lookup f as $\texttt{f} : arg0_{typ} \rightarrow res_{typ}$
$\mathcal{P}', vt' \leftarrow \mathbf{ArgT}(\mathcal{P}, pt_f, vt, 0, arg1_{typ})$
Let $k' = Kcall\ g\ C\ \mathcal{P}\ fpt;\ entek$

$$\mathcal{C}\ (m \mid f, pt_f, 42@vt' \gg k'\ @\mathcal{P}')$$

$\mathcal{P}'' \leftarrow \mathbf{CallT}(\mathcal{P}, pt_f, \texttt{f})$
Allocate public locals in en' and m', tagged by **LocalT**, new PC tag $\mathcal{P}'''$
Initialize private locals in te'

$$\mathcal{S}\ (m', en', te' \mid \texttt{f}.body \gg k'\ @\mathcal{P}''')$$

Figure 6.4: First Step of Call

expression containing an lvalue (pointing to the function being called) and a list of rvalues (the arguments). It steps to a call state, in which the callee has been identified and the argument values implicitly cast to the types in the callee's signature. Then the call state steps to a normal state to execute the callee's body, allocating locals. Figure 6.4 shows the steps of a call from function `g` to `f` in some context C, with argument 42. The first step marshals the arguments based on the type of the function and prepares a continuation for eventual return. The second allocates local variables and then enters the body of the function.

The purpose of the **ArgT** rule is generally to ensure that arguments are safe to be passed to the callee. For instance, if a function is meant to have exclusive access

ArgT : τ_C	PC tag	**LocalT** : τ_C	PC tag
$\rightarrow\tau_V$	Tag on function pointer	$\rightarrow$*ident*	Name of local variable
$\rightarrow\tau_V$	Tag on argument	$\rightarrow$*type*	Variable type
$\rightarrow n$	Index of argument	$\rightharpoonup(\tau_C$	New PC tag
$\rightarrow$*type*		$\times\ \tau_V$	Lvalue tag
$\rightharpoonup(\tau_C$	New PC tag	$\times$ *list* τ_L	Location tags
$\times\ \tau_V)$	New tag on argument		
		DeallocT : τ_C	PC tag
CallT : τ_C	PC tag	$\rightarrow$*type*	Variable type
$\rightarrow\tau_V$	Tag on function pointer	$\rightharpoonup(\tau_C$	New PC tag
$\rightarrow\mathtt{f}_{fun}$	Function id	$\times\ \tau_V$	Value tag for cleared bytes
$\rightharpoonup\tau_C$	New PC tag	$\times\ \tau_L$	Location tag for cleared bytes

Figure 6.5: Tag rules for Calls and Returns

to some data via a specially tagged pointer, it should not pass that pointer to other functions (or perhaps the pointer should be stripped of its tag.) **ArgT** takes the tag on the function pointer, which can encode the identity of the callee, or some coarser information about it.

Then **CallT** represents the control-flow transition from caller to callee. It might seem worthwhile to divide this into two rules, one that executes during the transition to the call state, and one that does so during the transition from the call state to the callee. This proves to be redundant: there is no information available to the first rule that is not already available to the second.

LocalT associates the base-lvalue of a public variable with its allocated bytes, discussed above in Section 5.1. This is the high-level equivalent to the process described in Section 3.6 for taking the address of an object by adding an offset to the stack pointer. In a compiled program the relationship would be encoded in the tags on each instruction that takes the variable's address.

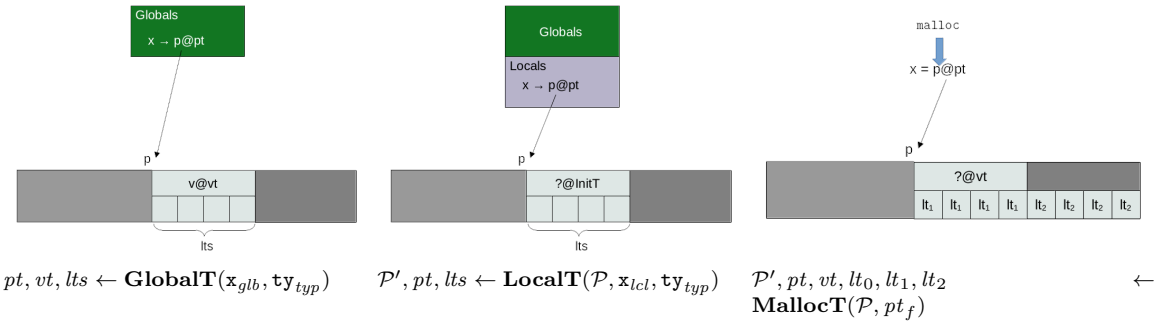
On return, locally allocated variables and arguments are deallocated; their memory locations are re-tagged by **DeallocT**. The continuation pops down to its highest-level call continuation, recovering the context C and the original PC tag. **RetT** fills a role similar to both **CallT** and **ArgT**, handling both the PC tag and the tag on the return value. Like **CallT**, this could be split into two rules, but one of them would always be redundant.

6.1.3 Allocation

The tags that protect memory are initialized by one of three tag rules, depending on whether the memory is allocated in a global variable, a local variable, or in the heap. Figure 6.6 shows some of the subtle differences between them.

A global variable is allocated at program initialization. This allocation invokes the **GlobalT** tag rule. Because globals are statically known at initialization, **GlobalT** provides an initial value tag and a list of individual location tags, which can be used to give very fine-grained control over the protection of global structs, for instance. It also defines the tag on the base lvalue of the variable in the environment. Because this happens before execution proper, the PC tag is not relevant and the rule cannot fail.

Local (i.e., stack) variables are allocated during function calls, and their tags are determined by **LocalT**. These variables are not initialized, as declarations that include initializations are separated by the parser into separate declarations and assignments. Their initial value tags are the default **InitT**. But their types and identities are known, so **LocalT** has access to this information, allowing it to produce a list of



$pt, vt, lts \leftarrow \mathbf{GlobalT}(\mathtt{x}_{glb}, \mathtt{ty}_{typ})$ $\mathcal{P}', pt, lts \leftarrow \mathbf{LocalT}(\mathcal{P}, \mathtt{x}_{lcl}, \mathtt{ty}_{typ})$ $\mathcal{P}', pt, vt, lt_0, lt_1, lt_2 \leftarrow \mathbf{MallocT}(\mathcal{P}, pt_f)$

Figure 6.6: Kinds of Allocations

different location tags.

Finally, heap objects are allocated by calls to `malloc`. They have no static identity and no type information associated with them, so the only meaningful distinction that can be made between their location tags are those that are part of the actual allocation, and any padding bytes reserved to maintain 8-byte alignment. In Figure 6.6 the four allocated bytes are tagged lt_1 and the padding bytes lt_2. The tag lt_0 is applied to any header or equivalent structure associated with the block by the `malloc` implementation, discussed in Section 6.1.

6.1.4 Control flow splits and joins

When control flow branches, whether as the result of a conditional or loop statement or a conditional expression, the value that determined the result of the branch influences every computation that happens until the branches rejoin. Policies that need to track this influence do so by changing the PC tag.

Conditional expressions are slightly simpler than statements, because they are guaranteed to rejoin once fully evaluated. These are ternary conditional expressions

ExprSplitT :		**ExprJoinT** :	
τ_C	PC tag	τ_C	PC tag
$\rightarrow\tau_V$	Tag on branch value	$\rightarrow\tau_V$	Tag on result
$\rightharpoonup\tau_C$	New PC tag	$\rightharpoonup(\tau_C$	New PC tag
		$\times\ \tau_V)$	New tag on result

SplitT :		**LabelT** :	
τ_C	PC tag	τ_C	PC tag
$\rightarrow\tau_V$	Tag on branch value	$\rightarrow$*ident*	Label name
$\rightarrow$*option ident*	Label of join point, if known	$\rightharpoonup\tau_C$	New PC tag
$\rightharpoonup\tau_C$	New PC tag		

Figure 6.7: Splits and Joins

(`a ? b : c`), logical-and (`&&`), and logical-or (`||`). The latter two split control flow because they can short-circuit: `0 && ...` will reduce to 0, while `1 && ...` will have to evaluate the right side of the expression. The **ExprSplitT** rule sets a new PC tag to represent the state inside of that "optional" control-flow path. When the resulting expression is fully evaluated to an rvalue, **ExprJoinT** triggers, setting the PC tag again and also determining the tag on the resulting atom. In Section 5.3, the PC tag forms a stack of "tainted" contexts, and these rules push and pop contexts from it.

Conditional and loop statements, on the other hand, might rejoin at a point arbitrarily distant from where they split. In the control-flow graph of the program, the branches rejoin at the immediate post-dominator of the split node [25]. This poses a problem for information-flow policies that want to lower the PC tag anytime a branch rejoins (for instance, see Section 5.3).

6.1.5 Casts

A policy may retag a value when it is explicitly cast to a different type. Tagged C distinguishes casts whose target types are pointers from other casts, because casting

from an integer to a pointer (or between pointer types) can have major security implications; see the PNVI memory safety policy (Section 5.1). Casts between non-pointer types are less obviously security-relevant, but the tag rule is included for completeness.

In the case of a cast to a pointer, Tagged C will attempt to read the location tags at the pointer's target, and provide them as an optional parameter to the **CastToPtrT** tag rule. If the tags cannot be read, for instance if the pointer is a null pointer, it passes a *None*.

Implicit casts do not trigger these rules. This is a trade-off between complexity and expressiveness: most implicit casts merely increase the size of an integer or float in order to use it in arithmetic with a larger value, or to store it in a variable or pass it as an argument whose type is larger. The converse (casting from a larger type to a smaller) could cause security flaws by creating the opportunity for integer overflows, but that does not happen implicitly. Implicit casts between pointers are limited to casts between `void*` and another type. There are no implicit casts from integers to pointers.

The cast rules are specialized based on whether the original type or the new type is a pointer, or both, because casts to and from pointers can make use of the location tags at their targets. This enables PNVI enforcement (Section 5.1), and more generally policies that keep track of the correspondence between pointers and their targets.

CastToPtrT :	τ_C	PC tag
	$\rightarrow \tau_V$	Tag on value
	$\rightarrow$ *option* (*list* τ_L)	Tags on memory at pointer location, if available
	$\rightarrow$ *type*	Source type
	$\rightarrow$ *type*	Target type
	$\rightharpoonup \tau_V$	Tag on resulting value

CastOtherT :	τ_C	PC tag
	$\rightarrow \tau_V$	Tag on value
	$\rightarrow$ *type*	Source type
	$\rightarrow$ *type*	Target type
	$\rightharpoonup \tau_V$	Tag on resulting value

Figure 6.8: Type cast rules

FieldT :	τ_C	PC tag
	$\rightarrow \tau_V$	Tag on struct/union's lvalue
	$\rightarrow$ *type*	Struct/union type
	$\rightarrow$ *ident*	Struct/union field
	$\rightharpoonup \tau_V$	Tag on result

Figure 6.9: Fields of Structs and Unions

6.1.6 Structs and Unions

Tagged C treats structs and unions as their concrete representations in memory, almost like arrays. This is a design inherited from CompCert C, but one that can be unsatisfying when tags enter the picture. Assigning one struct or union "into" another is tantamount to performing a `memcpy` on its representation in memory. There is no struct or union value separate from its pointer. That being the case, accessing the field of a struct or union is tantamount to pointer arithmetic, albeit adding a static offset. The **FieldT** rule does have access to the identifier of the field, potentially of use for substructural safety.

6.2 Modeling Special Functions

Some library functions cannot be modeled directly as Tagged C functions, typically because they contain assembly code or require special privileges. This includes system calls, I/O functions, and `malloc` and `free`, as well as functions written in assembly or for which source code is not available.

There are two considerations here. First, `malloc` and `free` are key components of the C language, as they govern the heap, so heap safety policies must manipulate tags based on their behavior. The same is true to a lesser degree for I/O functions, which play a crucial role in information flow policies. We term these *privileged functions*. Second, any function that is "out of sight" of the Tagged C semantics (termed *external functions*) might have internal behavior that violates a policy. How should Tagged C treat those cases?

6.2.1 Privileged Functions and their Subrules

`malloc` and `free` are the quintessential privileged functions: though they can be implemented in many ways, the C standard has a great deal to say about their behavior, and so do many policies. In Tagged C they are axiomatized within the semantics, and I have instantiated the axiomatization with two different implementations: one using an abstract "free list" and a concrete version where the free list is actually a data structure in the program's memory. The latter is useful for testing policies under a more realistic view of memory, as in Naaktgeboren et al. [59]. Other implementations are plausible, as long as they can associate each allocation with a tag.

In this section we'll examine the abstract free list and how the tag rules interact with it. It will soon become clear that `free` needs to check tags at two different points, resulting in two different tag rules that need to be invoked, **FreeT** and **ClearT**.

In general these are termed *subrules*: special-purpose rules that govern some subset of the behavior of a privileged function. In the current Tagged C development there are four subrules: **MallocT**, **FreeT**, **ClearT**, and **PrintT**.

`malloc` and MallocT Tagged C is parameterized by an allocator, and for purposes of the interpreter I instantiate it with a simple list allocator. The allocator state consists of an abstract list of headers, each containing a flag LIVE or FREE and a base address, size, and location tag (simulating that headers would, in a more realistic setting, live in memory.) Figure 6.10 shows a call to `malloc` seeking 32 bytes, which walks through the block list until it finds a suitable block (in this case, an exact fit). First, **MallocT** produces all of the tags that will be used in this process: a

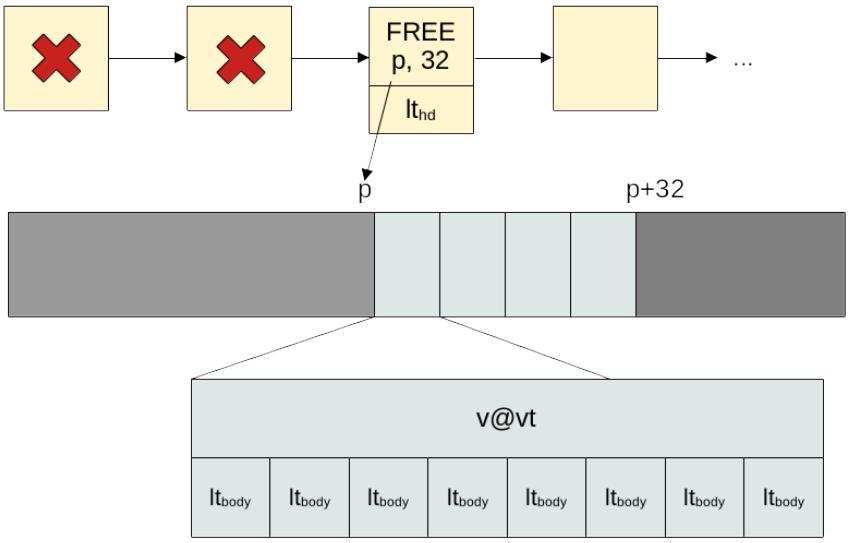


Figure 6.10: Allocating 32 bytes

new location tag for the header, another to be copied across the entire body of the allocated block, and a third to be copied across any padding bytes that may be necessary, as well as a value tag that will be attached to the pointer. `malloc` then flips the FREE flag to LIVE and sets its new location tag, and copies the other tags across memory as appropriate.

`free`, FreeT, and ClearT `free` is a little more complex. Notice that the semantics, at a call `free(x);`, doesn't immediately know whether `x` points to the start of a block, nor how large that block is. So the first step of `free` is to search through the header list to find a matching header. Once found, the question is: is it legal to free this block? A policy might restrict `free` to be called only with a valid pointer, or only from within a specific compartment, for instance. This is handled by **FreeT**.

MallocT : τ_C	PC tag
$\rightarrow \tau_V$	Function pointer tag
$\rightharpoonup (\tau_C$	New PC tag
$\times \tau_V$	Pointer tag
$\times \tau_V$	Initial tag on values in allocated block
$\times \tau_L$	Location tag on header
$\times \tau_L$	Location tag on the allocated bytes
$\times \tau_L)$	Location tag on any padding bytes (for 8 byte alignment)

FreeT : τ_C	PC tag
$\rightarrow \tau_V$	Function pointer tag
$\rightarrow \tau_V$	Argument pointer tag
$\rightarrow list\ \tau_L$	Header loc. tags
$\rightharpoonup (\tau_C$	New PC tag
$\times \tau_L)$	New header loc. tag

ClearT : τ_C	PC tag
$\rightarrow \tau_V$	Function pointer tag
$\rightarrow \tau_V$	Argument pointer tag
$\rightarrow \tau_L$	Location tag on byte
$\rightharpoonup (\tau_C$	New PC tag
$\times \tau_L)$	New loc. tag on byte

Figure 6.11: Rules for Malloc and Free

This ordering means that **FreeT** cannot do anything with the tags in the body of the block.

Once the freeing of the block is determined to be valid, `free` also knows how large it is, and can iterate across it, clearing its tags. In addition to setting the location tags to some default tag, this can be used to check if the contents of the cleared block contain secrets that should have been zeroed out, or generally to detect if something unexpected has happened, such as a write into padding bytes. With the addition of logging functionality (below) it can be useful to check for such anomalies and log them, even if they are not overtly dangerous. The location tag on each byte is given as an argument to **ClearT**, which produces a new tag for it. This is an operation that would be very likely to be optimized away if the policy doesn't need it. Figure 6.11 gives the detailed signatures for these rules.

`printf` and PrintT Functions that output data play an important role in information-flow policies, since by definition they produce externally visible results. **ArgT** is insufficient to gate values from leaking via `printf`, because not only can it output its arguments, but with a "%s" format string it can also dereference its arguments and output their contents as a string. It is infeasible to check at call time whether `printf` will output any given value, so instead Tagged C gives `printf` and other similar functions a subrule, **PrintT**. **PrintT** does not return any tags: its purpose is exclusively to failstop or allow execution to continue.

In addition to **PrintT**, calls to `printf` that use "%s" load data from memory, so they need to invoke the usual trio of **CoalesceT**, **AccessT**, and **LoadT**.

6.2.2 External Functions

Other special functions are less fundamental to the C language, but nevertheless need to be represented outside of the Tagged C syntax. In the development these are implemented in OCaml. The big question is: how do they interact with tags? First, let's consider what would be realistic for external code in a compiled setting.

External Functions in a Compiled, Hardware Enforced Tagged C In the current interpreted version of Tagged C, I have a great deal of power to implement external functions in whatever way is easiest, but it is important not to do anything that would be problematic in a more constrained hardware-enforced setting. So let's imagine an eventual, complete Tagged C ecosystem with a compiler to PIPE-style hardware.

At the assembly-level, compiled Tagged C tag rules are translated into low-level tag rules. In cases where the same instruction could represent multiple different source rules, they are distinguished by instruction tags. In this setting an external function is essentially a piece of arbitrary assembly with no instruction tags. (Though we might concede that we need to at least use instruction tags to identify its return instruction(s).)

It turns out that by setting the default behavior of the low-level rules (absent instruction tags) to match their high-level counterparts, at least some policies can still be enforced in this environment. If the default behavior of load and store instructions matches **LoadT** and **StoreT**, and value tags generally propagate as normal, then a memory safety policy can continue to be enforced inside of the external function. The same is true of information flow policies.

External Functions in Interpreted Tagged C In the interpreter, when external functions perform actions that can reasonably be recognized as control points (especially loads and stores), they invoke the appropriate tag rules directly from their OCaml definitions. For example, when the OCaml implementation of `printf` is called with the `%s` flag, it treats the corresponding argument as a pointer and loads from it until it reaches a null-terminator, and applies the **LoadT** tag rule for each load.

An external function might invoke multiple rules, or the same rule multiple times, just like a privileged function does with its subrules. And functions that are both, like `printf`, might combine some privileged subrules with standard rules.

6.3 Using, Breaking, and Fixing Tagged C

No design survives contact with the user, and Tagged C is no exception. Over the course of its development the design has shifted in response to practical experience developing policies. In particular, Naaktgeboren et al. [59] use Tagged C as part of their policy-informed fuzzing system, Pipe-Cleaner. Their feedback has informed many of the changes discussed in this section. Some features of Tagged C exist expressly to serve the needs of that project; others came into existence as an indirect result of their usage uncovering inadequate assumptions in the existing language.

Distinguishing Tag Types The separation of tags into three distinct types (value tags, location tags, and control tags) followed from early user feedback. In earlier versions, all three varieties of tags shared a single type. Writing policies with a single tag type is tedious and complicated, because most of the time policies do in fact use largely disjoint tags for these roles. With a single tag type, many tag rules begin by filtering out tag values that they should never actually encounter. It is much simpler to rule out those invalid values statically by separating the types.

Policy State and Logging Tagged C originally conceived of the PC tag as the only global state accessible to the policy. This is sufficient when the PC tag is keeping track of simple things like "how many colors have been used," a simple counter. But for tracking more significant information, such as a mapping between colors and the code locations at which they were allocated, it is not very realistic. By the same token, it is often helpful to be able to log arbitrary strings during execution, which is not reasonable to attach to a tag. Naaktgeboren et al. track sophisticated structures

and keep detailed logs to assist in root-cause analysis.

These needs motivate the inclusion of a *policy_state* parameter in policies, which is always accessible in tag rules, along with a log in the form of a list of strings. None of the policies discussed in this dissertation make use of the former, but the logging functionality has proven extremely helpful in debugging.

Recovery By user request, the Tagged C interpreter has a limited ability to call out of a tag rule into a recovery function in the extracted OCaml, which can read the entire state and could even modify it before returning to the tag rule. Naaktgeboren et al. use this feature to read segments of memory and log their contents, then continue execution. For example, their version of a memory safety policy does not failstop on loads from uninitialized memory, but instead logs the contents of memory surrounding the load to determine if there is a risk of deallocated data leaking through such an attack.

Fixed-length Tag Lists One practical issue that showed up early on is the fact that some tag rules produce lists of tags (mostly location tags). Those lists need to be the correct size for the memory region they are meant to tag. For some tag rules, like **MallocT**, I decided early on to just have them produce a single tag and copy it over the range being allocated. For those that might produce more complex lists of tags I added types as a parameter, but this still proved tedious to work with in practice. A lot of the time a designer just wants a list containing a single tag. Conversely, one might want to test that all tags in a list are the same. These operations became part of an *lt-op* interface built into the policy module type. Policy designers are

discouraged from operating directly on tag lists, and should use the tools provided in *lt-op* instead.

Passthrough Most policies focus on a subset of tag rules. A memory safety policy uses **LoadT** and **StoreT** extensively, but **RetT** simply passes the tag on *vt* on to *vt′*. It's burdensome for the policy designer to rewrite those identical rules on every policy, and creates opportunities for mistakes. The *passthrough* module provides reasonable defaults: tags move from inputs to outputs without changing.

6.3.1 Rejected Control Points

It's important that Tagged C have enough control points to implement a wide range of policies, but with too many rules the entire system would become overwhelming to work with. Paring down the set of tag rules has been an important part of the process. Here are some tag rules that existed in earlier versions of Tagged C, or that were proposed to be added and then rejected.

I initially proposed equivalents of **CallT** and **RetT** that trigger on the callee's side of calls and returns. Similarly, I proposed a distinct **ExtCallT** and **ExtRetT** which triggered on special functions. **ExtCallT** encapsulated all of the tag rules for all special functions that didn't have their own dedicated rules. This turned out to be insufficient to actually model special functions, so after switching to the system described below **ExtCallT** behaved identically to **CallT** with only slightly different parameters. Ultimately I deprecated **ExtCallT**, though it remains in the development for now.

Tagged C's treatment of the `*` (dereference) and `&` (address-taking) operators is not very intuitive. The dereference expression takes a right-hand value and turns it into a memory location—it does not access the value. Taking an address takes a memory location (which already has a tag associated with it) and converts it to a pointer. Early designs of Tagged C did have an **AddrT** rule, but there is no actual change in the address, so there is no point in executing a tag rule during them.

Originally, I implemented four different cast rules, one for each combination of casts between pointers and integers. Casts to and from pointer types both fetched location tags, and casts from one pointer type to another did so twice (since a cast between pointer types might change the footprint of the pointer). This was terribly complicated, and there doesn't appear to be much point in checking location tags when casting away from a pointer type.

6.4 Limitations

Distinguishing Calls Chapter 7 focuses on a single policy and its specification, with the interesting feature that it distinguishes some calls to `malloc` as being "local" to a compartment, while others are shared between compartments. The behavior of `malloc` should be the same in either case, but the result of **MallocT** should differ. There is no obviously correct way to do this in Tagged C. I take the approach of declaring two distinct identifiers that both resolve to the same function, but which have different function-pointer tags. Those tags parameterize the **MallocT** rule. A more principled solution might draw on pointcuts to identify specific calls to `malloc`. (Indeed, it would be interesting to investigate how pointcuts could apply to Tagged

C more generally!)

`malloc` and Substructural Safety It is unsatisfactory that `malloc` does not get access to type information; it takes just a size and returns a `void *`, which the caller must cast to a pointer of the desired type (at an arbitrary future point). Therefore, Tagged C cannot easily enforce substructural memory safety (i.e. protecting fields within a single struct from overflowing into each other) or other properties that call for allocated regions to be tagged according to their types. This is a well-known impediment to improving C memory safety; previous work (e.g. [56]) has often adopted non-standard versions of `malloc` that take more informative parameters. This is not satisfactory for protecting legacy code, but I do not yet see a good alternative.

Assigning Structs and Unions Treating structs and unions as pointers violates the design philosophy of Tagged C. The C standard allows structs and unions to be assigned just like other values, but that Tagged C turns such an assignment into a copy operation. This causes strange situations, especially with nested structs. What is the value tag of a nested struct that is neither the top-level pointer nor a bottom-level entry? What is the value tag of the "inactive" field of a union? More research is needed to find the best way to handle these types.

7 CluMPS: Verified Compartmentalization with Local and Shared Memory

7.1 Introduction

Tagged C provides a platform for writing policies and a formal semantics that captures the effects of each policy on the execution of programs. The semantics can be used to verify that a given policy has the desired behavior. This chapter applies this approach to verification to prove the correctness of a novel, fairly complex compartmentalization policy.

The CluMPS (*Clu*stered *M*emory *P*rotection with *S*haring) model of compartmentalization distinguishes between compartment-local memory and memory shared between compartments by memory-safe pointers. Because local allocations are isolated from other compartments, but not from one another, the Tagged C implementation of CluMPS can use fewer tags and therefore place fewer requirements on the underlying tagged hardware than similar systems from the literature [29]. The CluMPS specification is an abstract machine with separate memory regions for local and shared data. This machine gives definition to low-level idioms that would be UB in a memory-safe machine, while still protecting compartments from one another. It is formalized in Coq, along with a mechanized proof that Tagged C, running a CluMPS policy, simulates the CluMPS abstract machine.

```
#define LEN 1 // define pwd length
char* master_pwd;
extern void gets(char* buf);
extern void log(char* msg);
extern void __sys_fire_missiles();

int check_pwd(char* pwd) {
  for (int i = 0; i < LEN; i++) {
    if (pwd[i] != master_pwd[i])
      return 0; // bad password
  }
  return 1; // report success
}
```

```
void listen() {
  char pwd[LEN];
  while (1) {
    gets(pwd);
    // call external log library
    log("launch attempt");
    if (check_pwd(pwd)) {
      __sys_fire_missiles();
    }
  }
}
```

Figure 7.1: Example Program

7.1.1 Motivating Example

Consider a program (Figure 7.1) for firing missiles. It listens on standard input for a password, checks that input against the master password in the program's memory, and reports whether it was correct. If correct, it launches missiles with a system call. It also logs that it received an attempted launch before checking, using an off-the-shelf logging library.

If the logging library has a vulnerability, it might be used to undermine the security of the whole system. Even if the library cannot call `__sys_fire_missiles` directly or corrupt control flow to reach it, it can still do harm. In a memory-unsafe setting such as baseline Tagged C (Chapter 4), an attacker hijacking the logging library could overwrite `master_pwd` to make it match the supplied password, or leak its value to be used in a future attempt.

One approach to making the system secure is to compartmentalize it. All of the above code can be gathered in one unit, which we will call A, and kept separate from the logging library, which belongs to its own compartment, B. We might further chose

to keep the standard library in yet another compartment, C. The compartments' memories are kept disjoint, except that `gets` must take a pointer to an array that will be accessible to both A and C. The special `__sys_fire_missiles` function is modeled as an external call, outside of all three compartments.

A division of code and data into compartments as described above is termed a *compartmentalization interface*. Such an interface must assign each of a program's functions and global variables to a compartment. The interface implicitly describes restrictions on how its compartments can interact with data: for instance, that `log` (code belonging to compartment B) may not read or write `master_pwd` (data in compartment A). These restrictions can then be used to reason about more complex dynamic behavior (termed a *property*) of the program as a whole.

In this example we might articulate a desired property of the program in English as, "This program only fires the missiles after receiving the valid password." This property would not hold on an unprotected program with a vulnerable library, but in the compartmentalization interface described above, it should—provided that the interface is correctly enforced and the system obeys the implicit restrictions on how compartments may interact.

The first step to proving correctness is to formalize those implicit restrictions via an abstract compartmentalized machine that keeps its compartments separate "by construction." Such a machine's execution will then satisfy all properties that depend on the separation. Quantifying over all programs and all compartmentalization interfaces, an implementation of CluMPS is correct if it preserves the properties satisfied by the abstract machine. This chapter describes such an abstract machine for the CluMPS compartmentalization model and a proof that Tagged C with a CluMPS

policy correctly enforces it.

7.1.2 Memory Granularity and Tagged Hardware

The desired property in the example above would hold in a memory safe system regardless of compartmentalization. But enforcing memory safety may not always be possible. The memory safety policy in Section 5.1 tags each individual object with a unique "color" tag, which places no restrictions on the number of available tags. For efficiency, real tagged hardware requires the "working set" of tags that are in use at any given time to remain fairly small (the precise number is hardware-dependent) [28].

CluMPS is designed to be enforced using fewer unique tags than full memory safety. Rather than give every allocation a unique tag, allocations that are declared as local to a given compartment can share a tag. The compartmentalization interface can therefore be tuned to fit within any particular hardware constraints on the number of tags.

The following analysis of existing compartmentalization models includes the question of whether they require a comparable number of tags. This is a tricky question because not all of them use tags for protection. They do not necessarily focus strongly on their underlying enforcement mechanisms at all. This question boils down to whether the system requires that all memory allocations are kept separate from one another, either because its source language separates them explicitly or because it is compiled to a memory-safe target. When the answer is yes, the model presented here should offer performance benefits over those less flexible alternatives.

7.1.3 Related Work

There are many compartmentalization mechanisms in the literature [47]. This section discusses some prominent formal characterizations of compartmentalization, and how they compare to the new characterization and proof described in this chapter.

Abate et al. [3] characterize compartmentalization in the presence of undefined behavior, treating UB as equivalent to compromise by an adversary. Their model does not support shared memory, and their policy places strong demands on potential hardware implementations: they assume "linear tags," a proposed style of tagging in which the policy can overwrite the tags on the operands of tag rules, used to maintain the uniqueness of tags by presenting them from being copied. Linear tags are unlikely to appear in realistic implementations, and mimicking them in software can be costly. They prove their theorems with a mix of mechanized and hand proofs.

Building on Abate et al.'s work, El-Korashy et al. [29] present a model of compartmentalization that does support shared memory. They use a much more restrictive specification than Abate et al., with a source language that is fully memory-safe and has no UB. They focus on enforcing their compartmentalized calling convention in a memory safe setting. They do not discuss underlying enforcement hardware, but as they assume a memory-safe target, the cost of full memory safety is implied. Their proofs are fully mechanized in Coq.

Thibault et al. [75] go deeper in their proof: they prove safe compilation of a compartmentalized version of CompCert C down to a compartmentalized assembly language. In the process they give up sharing. Their treatment of UB is similar to Abate et al., but they add new UB in the form of violations of the compartment

interfaces. Their proof is fully mechanized as well.

Compared to these works in total, my model supports cross-compartment sharing like El-Korashy et al., but should place fewer demands on the hardware than likely implementations of their system. It also supports a full C setting, though instead of a full compilation chain we attach our policy directly to the Tagged C source semantics. My abstract machine is also more precise about its treatment of UB than the others: it gives definition to some UB, and so can be used to reason about the behavior of compartments that contain UB but still display consistent internal behavior. Like the others, my proof is mechanized in Coq.

7.1.4 Contributions

This chapter's contributions are:

- The CluMPS abstract semantics, an abstract machine suitable for reasoning about compartmentalized C programs with safe memory sharing but with potential low-level idioms inside of compartments, formalized in Coq.
- A CluMPS policy for Tagged C that supports cross-compartment sharing with fewer demands on available tags than similar systems from the literature.
- A mechanized proof of forward and backward simulation between the CluMPS abstract semantics and the CluMPS policy.

7.2 Introducing the CluMPS Semantics

CluMPS is an abstract C semantics built on top of CompCert C, with its own memory model that supports a flexible and efficient model of compartmentalization. Each compartment gets its own local region of memory that is modeled as an isolated flat memory space, akin to Tagged C's memory model. Allocations within a local region can only be accessed by the compartment that owns them, but otherwise enjoy no memory safety. They can be thought of as separate allocations that are "clustered together," although they may not be contiguous within the overall address space. Memory can also be allocated in shareable regions that are accessed in a capability style by any compartment that holds a valid pointer to them.

Given a C program and a compartmentalization interface (below), the CluMPS semantics describes how the program should behave when compartmentalized. CluMPS can be used to reason directly about an individual program's behavior, and also serves as the specification for compartmentalization enforcement mechanisms—in this case, Tagged C with a relevant policy. If the target shares the behavior of the CluMPS abstract machine, then all reasoning about programs executed under the CluMPS machine applies when they are executed under Tagged C with the policy.

7.2.1 CluMPS Interfaces

The execution of a CluMPS program is parameterized by static mappings that collectively define the compartmentalization interface discussed above. The mapping from function identifiers `f` to compartment identifiers (drawn from the set $\mathcal{C}$ and ranged over by C) is written $comp(\texttt{f})$. The predicate $pub(\texttt{f})$ denotes that the function `f`

$C : \mathcal{C}$	Compartment identifiers
$id : ident$	Abstract identifiers of shared allocations
$r : \mathcal{R} ::= \mathbf{L}(C) \mid \mathbf{S}(id)$	Regions, local and shared
$comp : \mathtt{f}_{fun} \to \mathcal{C}$	Map from function identifiers to compartments
$pub \subseteq \mathtt{f}_{fun}$	Identification of public functions

Figure 7.2: Modeling CluMPS Interfaces

is public, that is, allowed to be called from outside of the compartment that owns it. These definitions are collected in Figure 7.2. Note that in this work, all code is assumed to be visible in the same compilation unit; while conceptually CluMPS could be used to model separate compilation, I do not provide a modular semantics for it.

At any given time, the compartment that contains the currently active function is considered the active compartment. The active compartment may call its own functions and public functions belonging to other compartments, but never private functions of other compartments.

Local (stack allocated) variables and globals `x` are statically designated as shared (written *shared*(`x`)) or local (*local*(`x`)). For heap-allocated objects, it is the call sites to malloc that are statically designated, by separating `malloc` into two functions; a call to `malloc_shared` yields a shared pointer, while normal `malloc` yields a local one. Choosing how to annotate variables can be done using escape analysis: objects whose references never escape can be allocated locally, whereas objects whose references escape to another compartment must be allocated as shared.

Local pointers should never escape: they cannot be passed to or returned from public functions, or stored in shared memory. Violations of these rules exhibit failstop behavior.

7.2.2 Safely Handling Pointers

Several idioms that are unsafe in normal C can be used safely in CluMPS. Section 5.1 discusses the PVI (provenance via integer) and PNVI (provenance not via integer) memory models of Memarian et al. [54]. Like a pointer in a PVI setting (Section 5.1), a CluMPS pointer retains its provenance when cast into an integer, and can be cast back after arbitrary arithmetic to obtain the original pointer, which still points to the same local or shared region.

Because local memory may contain multiple allocations, it can even be used to implement low-level structures that are defined in PNVI and not PVI. But unlike PNVI, local regions are isolated from each other and from shared regions, so such idioms will never accidentally access data outside of the original local region. A compartment can even forge a pointer into its own local region by casting an integer, with no risk of forging a pointer to a different region. This should be avoided when the integer in question comes from outside the compartment, to ensure that the new pointer does not alias important data.

Due to the restrictions on how local pointers can be passed and stored, a function that expects a shared pointer from outside of its compartment can be assured that it will never receive a local pointer into its own memory. That means that a pointer recieved from another compartment can be used with no risk that it actually points into some sensitive private data structure.

$$
\begin{array}{rll}
v : val ::= \ldots \mid \texttt{ptr}\ r\ a : int & & \text{Values extended with pointer value} \\
le : \mathtt{x}_{lcl} \rightharpoonup (\mathcal{R} \times int) & & \text{Environment maps variables to region, address}
\end{array}
$$

$$
\begin{array}{rlll}
m : mem = & \{contents & : int \to val; & \text{Memory arena: concrete memory} \\
& perms & : int \to \{\textsc{live}, \textsc{dead}\}; & \text{and permissions}
\end{array}
$$

$$
\begin{array}{rlll}
M : \mathcal{M} = & \{ms & : \mathcal{R} \rightharpoonup mem; & \text{Total memory tracks arenas for each region,} \\
& bases & : ident \rightharpoonup int; & \text{base addresses of shared arenas,} \\
& stk & : list(\mathcal{R} \times int \times int); & \text{stack metadata,} \\
& heap & : list(\mathcal{R} \times int \times int); & \text{heap metadata,} \\
& next & : nat\} & \text{and a counter for identifiers}
\end{array}
$$

$$
\begin{array}{rll}
STATE_{CluMPS} ::= & \mathfrak{s}\,(C : M, le, te \mid s \gg k) & \text{Normal state (executing statement)} \\
\mid & \mathfrak{e}\,(C : M, le, te \mid e \gg k) & \text{Expression state} \\
\mid & \mathfrak{c}\,(C : M \mid f, args \gg k) & \text{Call state} \\
\mid & \mathfrak{r}\,(C : M \mid v \gg k) & \text{Return state} \\
\mid & \mathfrak{f}\,(\mathbf{err}) & \text{Failstop}
\end{array}
$$

Figure 7.3: CluMPS States

7.2.3 CluMPS Abstract Semantics

Because of their common heritage in CompCert C, the abstract semantics for CluMPS is quite close to that of Tagged C. They differ in their memory models, in the presence of tags in Tagged C, and in CluMPS tracking the active compartment in its state. The memory models also have basic operations in common, but differ in their underlying representations and in the axiomatization of their allocators.

Some important types are defined in Figure 7.3. The set of values present in Tagged C is extended with pointer values, $\texttt{ptr}\ r\ a$, where r is a region and a a concrete address. The local environment, which maps local variable names to their

base address, is extended to also map them to a region. Program states in Figure 7.3 correspond to those in Figure 4.4, but there is no PC tag. Instead the state explicitly tracks the active compartment (ranged over by C).

A memory *arena* consists of a map from concrete integer addresses to values, and another map indicating whether each address is "live" or "dead". Each arena behaves just like a Tagged C memory. Each address always has a concrete value, but loading from it will failstop if the address is dead. In Tagged C, addresses are assumed to be live unless reserved by the compiler, whereas in CluMPS addresses are assumed to be dead unless explicitly allocated. This means that CluMPS does not support arbitrary out-of-bounds accesses the way Tagged C does, but it does support out-of-bounds accesses that nevertheless access live memory.

In CluMPS, multiple arenas are contained in a single *total memory*, M, which takes the place of the normal memory in Tagged C. Arenas are kept totally separate, guaranteeing that compartments' local arenas are never accessible to other compartments, and that shared arenas are only accessible via valid pointers. Because arenas have the same behavior as a Tagged C memory, code that respects the separation of compartments should behave the same in CluMPS as in Tagged C. The CluMPS semantics rules for reading and writing a total memory are given in Figure 7.4.

Allocating and Freeing Memory Rather than using a concrete allocator like that in Chapter 4, the CluMPS semantics are parameterized by an allocator that obeys a simple set of axioms, inspired by the axiomatization in Tolmach et. al [76]. This abstraction means that the correctness proof is portable across different allocators.

The semantics of heap allocations and frees are given in Figure 7.5. The abstract

$$M[r][a] : \mathcal{M} \to \mathcal{R} \to int \rightharpoonup val \triangleq$$
if $(M.ms\ r).perms\ a = \text{LIVE}$
then $(M.ms\ r).contents\ a$

$$M[r][a \mapsto v] : \mathcal{M} \to \mathcal{R} \to int \to val \rightharpoonup \mathcal{M} \triangleq$$
if $(M.ms\ r).perms\ a = \text{LIVE}$
then let $m = M.ms\ r$ in
let $m' = m[a \mapsto v]$ in
$M[ms \mapsto M.ms[r \mapsto m']]$

$$\text{ValOfL}\ \frac{M[C][a] = v}{\mathfrak{e}\,(C : M, le, te \mid *(\texttt{ptr}\ \mathbf{L}(C)\ a) \gg k) \longrightarrow \mathfrak{e}\,(C : M, le, te \mid v \gg k)}$$

$$\text{ValOfS}\ \frac{M[id][a] = v}{\mathfrak{e}\,(C : M, le, te \mid *(\texttt{ptr}\ \mathbf{S}(id)\ a) \gg k) \longrightarrow \mathfrak{e}\,(C : M, le, te \mid v \gg k)}$$

$$\text{AssignL}\ \frac{M' = M[C][a \mapsto v]}{\mathfrak{e}\,(C : M, le, te \mid *(\texttt{ptr}\ \mathbf{L}(C)\ a) = v \gg k) \longrightarrow \mathfrak{e}\,(C : M', le, te \mid v \gg k)}$$

$$\text{AssignS}\ \frac{v \neq \texttt{ptr}\ \mathbf{L}(\_)\ \_ \qquad M' = M[id][a \mapsto v]}{\mathfrak{e}\,(C : M, le, te \mid *(\texttt{ptr}\ \mathbf{S}(id)\ a) = v \gg k) \longrightarrow \mathfrak{e}\,(C : M', le, te \mid v \gg k)}$$

Figure 7.4: Reads and Writes

operation *heapalloc* chooses a base address for the new allocation, subject to the constraints of *disj* (but see below). Since *heapalloc* is parameterized by the identity of the region that it is allocating within, it is allowed to make decisions based on that information, such as clumping compartment-local allocations together to protect them using a page-table-based enforcement mechanism.

New allocations are guaranteed to be disjoint from existing live blocks, whether in the stack or heap. It is possible to construct a variant CluMPS semantics that instead only guarantees that new allocations are disjoint from existing blocks in the same region. This corresponds to the predicate *disj_weak*. From the perspective of safe compartmentalization, two compartments could contain blocks whose integer addresses overlap. The separation of their memories ensures that the blocks will not

$$\mathit{live}\ M\ r\ a \triangleq\ \exists\ r\ a_1\ a_2.\ ((r, a_1, a_2) \in M.\mathit{stk} \vee (r, a_1, a_2) \in M.\mathit{heap}) \wedge a_1 \leq a < a_2$$

$$\mathit{perm\_consistent}\ M \triangleq\ \forall\ r\ a.\ \mathit{live}\ M\ r\ a \leftrightarrow M.\mathit{ms}\ r\ a = \textsc{live}$$

$$\mathit{disj\_weak}\ M\ r\ a\ sz \triangleq\ \forall a'.\ a \leq a' < a + sz \rightarrow \neg \mathit{live}\ M\ r\ a'$$

$$\mathit{disj}\ M\ a\ sz \triangleq\ \forall\ r\ a'.\ a \leq a' < a + sz \rightarrow \neg \mathit{live}\ M\ r\ a'$$

$$\frac{\mathit{disj}\ M\ a\ sz \qquad M'.\mathit{stk} = M.\mathit{stk} \qquad \mathit{perm\_consistent}\ M' \qquad M'.\mathit{heap} = (r, a, a + sz) :: M.\mathit{heap}}{\mathit{heapalloc}\ M\ r\ sz = (a, M')} \qquad \frac{(r, a, a') \in M.\mathit{heap} \qquad M'.\mathit{stk} = M.\mathit{stk} \qquad \mathit{perm\_consistent}\ M' \qquad M'.\mathit{heap} = M.\mathit{heap} - (r, a, a')}{\mathit{heapfree}\ M\ r\ a = M'}$$

$$\text{StepMallocL}\ \frac{\mathit{heapalloc}\ M\ \mathbf{L}(C)\ sz = (a, M') \qquad f = \texttt{malloc}}{\mathfrak{c}\,(C : M \mid f, sz \gg k) \longrightarrow \mathfrak{r}\,(C : M' \mid \texttt{ptr}\ \mathbf{L}(C)\ a \gg k)}$$

$$\text{StepMallocS}\ \frac{\mathit{heapalloc}\ M\ \mathbf{S}(M.\mathit{next})\ sz = (a, M') \qquad f = \texttt{malloc\_share}}{\mathfrak{c}\,(C : M \mid f, sz \gg k) \longrightarrow \mathfrak{r}\,(C : M'[\mathit{next} \mapsto \mathit{id} + 1] \mid \texttt{ptr}\ \mathbf{S}(\mathit{id})\ a \gg k)}$$

$$\text{StepFree}\ \frac{\mathit{heap\_free}\ M\ r\ a = M'}{\mathfrak{c}\,(C : M \mid \texttt{free}, \texttt{ptr}\ r\ a) \gg k) \longrightarrow \mathfrak{r}\,(C : M' \mid k)}$$

Figure 7.5: Heap Allocation Steps

alias with one another. However, in implementation, it is not feasible for each region to have an entire 64-bit address space to itself. The proof covered in this chapter therefore uses *disj*.

Function Calls and Returns The CluMPS function call semantics differs from that of Tagged C in how it enforces the compartment interface, shown in Figure 7.6. First, as in Tagged C, a statement whose subexpression is a function call with fully evaluated arguments steps to an intermediate state. In addition to requiring that the arguments all be values, CluMPS requires that local pointers only be passed to internal (non-public) functions. Then the system steps to a call state, recording in the continuation the necessary information to return to the caller.

Once in the call state, the CluMPS semantics checks whether the call is cross-compartment, and if so, whether the target function is public. Together with the check on function arguments, this ensures that local pointers do not escape by being passed to other compartments. Similar checks occur on return values to prevent local pointers from escaping that way.

The step from the call state also allocates memory on the stack. Stack allocations (*stackalloc*) work similarly to heap allocations, except that the list $M.stk$ is used as a stack, pushing new allocations for the call, to later be popped in order by *stackfree* when the function returns. Disjointness and consistency requirements remain the same. Allocations are designated local or shared by a static flag, `L` or `S`, in the function's argument list.

On return, the contents of *le* are deallocated by *dealloc_vars*, and levels are popped from the continuation until the call is reached (definitions omitted). Then the caller's

environments are restored, and the return value placed in the call context. The latter step can not occur if the return value is a local variable and the new compartment differs from the old; instead the system failstops.

Arithmetic and Integer-Pointer Casts Tagged C has fully concrete pointers, represented by normal integers and distinguished only by their types. By contrast, CluMPS pointers are *semi-concrete*: they have a concrete value, but also carry additional abstract information in the form of the region that they point to. Such pointer values need not have pointer types. If cast to an integer, a pointer `ptr` r a retains its value until it is used in an integer operation, at which point the semantics interpret it as the integer value a. Whether the result of such an operation is an integer or a pointer depends on its operands, as shown in Figure 7.7.

If an operation involves only a single pointer value, the result will also be a pointer value. A binary operation between two pointer values, on the other hand, has no natural interpretation as a pointer, and therefore produces a plain integer. This is true for all operations, whether or not they have a sensible interpretation as pointer operations, to enable the widest possible range of behaviors.

If an integer with a pointer value is cast back to a pointer type, it retains its value and is once again a valid pointer. Otherwise, if an integer value is cast to a pointer, the result is always a local pointer to the active compartment. This means that a compartment can freely perform PNVI-style pointer arithmetic within its local memory, treating integers and local pointers as interchangeable. But it cannot create a pointer to a shared object that it does not already have access to, and errors in pointer arithmetic cannot accidentally corrupt other compartments. Forging a pointer

$$\text{StepCall}\ \frac{\begin{array}{c} k' = \mathit{call\_cont}\ C\ \mathit{ctx}\ \mathit{le}\ \mathit{te}\ k \qquad \forall e \in \mathit{args}.\ \mathit{is\_value}\ e \\ \mathit{public}(\mathtt{f}) \rightarrow \forall e \in \mathit{args}.\ e \neq \mathtt{ptr}\ \mathbf{L}(C)\ \_ \end{array}}{\mathfrak{s}\,(C : M, \mathit{le}, \mathit{te} \mid \mathit{ctx}[\mathtt{f}(\mathit{args})] \gg k) \longrightarrow \mathfrak{c}\,(C : M \mid \mathtt{f}, \mathit{args} \gg k')}$$

$$\frac{\mathit{disj}\ M\ a\ \mathit{sz} \qquad M'.\mathit{heap} = M.\mathit{heap} \qquad \mathit{perm\_consistent}\ M' \qquad M'.\mathit{stk} = (r, a, a + \mathit{sz}) :: M.\mathit{stk}}{\mathit{stackalloc}\ M\ r\ \mathit{sz} = (a, M')} \qquad \frac{M'.\mathit{heap} = M.\mathit{heap} \qquad \mathit{perm\_consistent}\ M' \qquad (r, a, a + \mathit{sz}) :: M'.\mathit{stk} = M.\mathit{stk}}{\mathit{stackfree}\ M\ r = M'}$$

$$\frac{\mathit{stackalloc}\ M'\ \mathbf{L}(C)\ \mathit{sz} = (a, M'') \qquad \mathit{alloc\_vars}\ M\ C\ \mathit{vars} = (\mathit{le}, M') \qquad \mathit{le}' = \mathit{le}[\mathtt{x} \mapsto (\mathbf{L}(C), a)]}{\mathit{alloc\_vars}\ M\ C\ ((\mathtt{L}, \mathtt{x}, \mathit{sz}) :: \mathit{vars}) = (\mathit{le}', M'')}$$

$$\frac{\mathit{stackalloc}\ M'\ \mathbf{S}(M'.\mathit{next})\ \mathit{sz} = (a, M'') \qquad \mathit{alloc\_vars}\ M\ C\ \mathit{vars} = (\mathit{le}, M') \qquad \mathit{le}' = \mathit{le}[\mathtt{x} \mapsto (\mathbf{S}(M'.\mathit{next}), a)]}{\mathit{alloc\_vars}\ M\ C\ ((\mathtt{S}, \mathtt{x}, \mathit{sz}) :: \mathit{vars}) = (\mathit{le}', M''[\mathit{next} \mapsto M'.\mathit{next} + 1])}$$

$$\text{StepEntry}\ \frac{\begin{array}{c} C' = \mathit{comp}(\mathtt{f}) \qquad C \neq C' \rightarrow \mathit{pub}(\mathtt{f}) \qquad \mathit{te} = \mathit{empty\_te} \\ (\mathit{le}, M') = \mathit{alloc\_vars}\ M\ C\ (\mathtt{f}.\mathit{vars} +\!\!+ f.\mathit{args}) \\ M'' = \mathit{init\_params}\ M'\ \mathit{args} \end{array}}{\mathfrak{c}\,(C : M \mid \mathtt{f}, \mathit{args} \gg k) \longrightarrow \mathfrak{s}\,(C' : M'', \mathit{le}, \mathit{te} \mid \mathtt{f}.\mathit{body} \gg k)}$$

$$\text{StepReturn}\ \frac{k' = \mathit{pop\_to\_call\_cont}(k) \qquad M' = \mathit{dealloc\_vars}\ M\ \mathit{dom}(\mathit{le})}{\mathfrak{s}\,(C : M, \mathit{le}, \mathit{te} \mid \mathtt{return}\ v \gg k) \longrightarrow \mathfrak{r}\,(C : M' \mid v \gg k')}$$

$$\text{StepExit}\ \frac{k = \mathit{call\_cont}\ C'\ \mathit{ctx}\ \mathit{le}\ \mathit{te}\ k' \qquad v = \mathtt{ptr}\ \mathbf{L}(C)\ \_ \rightarrow C = C'}{\mathfrak{r}\,(C : M \mid v \gg k) \longrightarrow \mathfrak{s}\,(C' : M, \mathit{le}, \mathit{te} \mid \mathit{ctx}[v] \gg k')}$$

Figure 7.6: Function Call and Return Semantics

$$\frac{}{\mathfrak{e}\,(C : M, le, te \mid \odot\texttt{ptr}\ r\ a \gg k) \longrightarrow \mathfrak{e}\,(C : M, le, te \mid \texttt{ptr}\ r\ \langle\odot\rangle a \gg k)}$$

$$\frac{}{\mathfrak{e}\,(C : M, le, te \mid \texttt{ptr}\ r\ a \oplus i \gg k) \longrightarrow \mathfrak{e}\,(C : M, le, te \mid \texttt{ptr}\ r\ (a\langle\oplus\rangle i) \gg k)}$$

$$\frac{}{\mathfrak{e}\,(C : M, le, te \mid i \oplus \texttt{ptr}\ r\ a \gg k) \longrightarrow \mathfrak{e}\,(C : M, le, te \mid \texttt{ptr}\ r\ (i\langle\oplus\rangle a) \gg k)}$$

$$\frac{}{\mathfrak{e}\,(C : M, le, te \mid \texttt{ptr}\ r_1\ a_1 \oplus \texttt{ptr}\ r_1\ a_2 \gg k) \longrightarrow \mathfrak{e}\,(C : M, le, te \mid (a_1\langle\oplus\rangle a_2) \gg k)}$$

Figure 7.7: Arithmetic Operations Involving Pointers

into local memory remains risky, especially if the integer that is cast into a pointer originates from outside of the compartment, but the risk is contained within the compartment.

7.3 Enforcing CluMPS in Tagged C

Chapter 4 describes how to define a Tagged C policy; this section introduces CluMPS(I), a Tagged C policy parameterized by a CluMPS interface I.

In CluMPS, a control tag (in τ_C) is a pair of a compartment identifier and a counter for abstract identifiers. A value tag (in τ_V) can represent a local or shared pointer, a function pointer, or a normal integer, written $\emptyset$. A location tag (in τ_L) can represent local or shared allocations, or unallocated memory ($\emptyset$).

$$\tau_V ::= \mathit{LPtr}(C) \mid \mathit{SPtr}(n : \mathit{nat}) \mid \mathit{FPtr}(\mathtt{f}_{\mathit{fun}}) \mid \emptyset$$

$$\tau_C ::= (C \times n : \mathit{nat})$$

$$\tau_L ::= \mathit{LMem}(C) \mid \mathit{SMem}(n : \mathit{nat}) \mid \emptyset$$

Memory protection is governed by the **LoadT** and **StoreT** tag rules (Fig. 7.8), which compare the pointer tag to the location tags on the target location. In the case of local memory, they also check that the active compartment matches, and for shared memory, that a local pointer is not being stored.

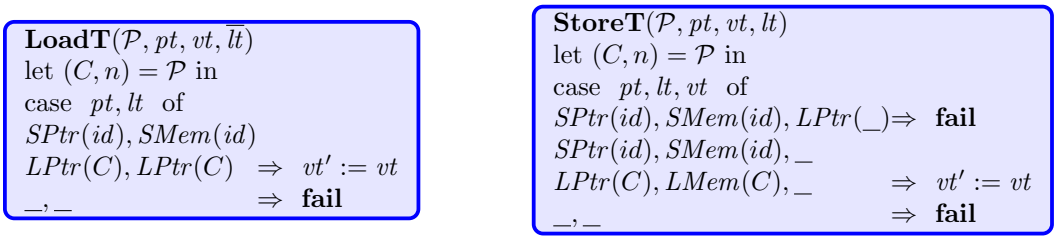


Figure 7.8: CLUMPS Load and Store rules

Pointer and location tags of heap objects are generated by **MallocT** (Fig. 7.9), which uses the tag on the function pointer to determine if the allocation is local or shared.

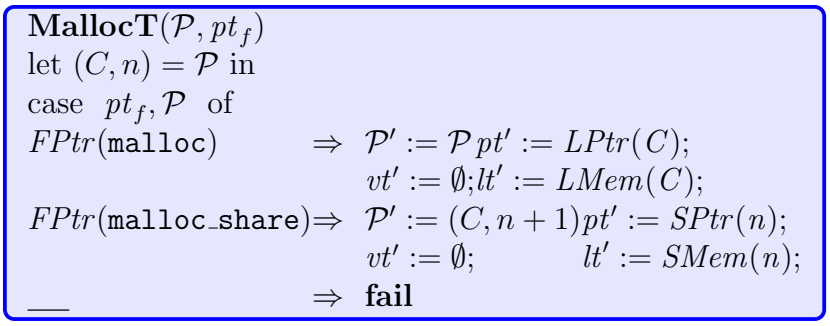


Figure 7.9: CLUMPS Malloc rule

LocalT is instantiated similarly, but using the names of stack variables to lookup their local or shared status. **BinopT** maintains the tag on pointers or non-pointers over binary operations, while **CastToPtrT** and **CastOtherT** give and take pointer status during typecasts (see Fig. 7.10). When a pointer type is cast to a different

pointer type, its tag is preserved, but when any other type is cast its tag becomes the local pointer tag for the active compartment.

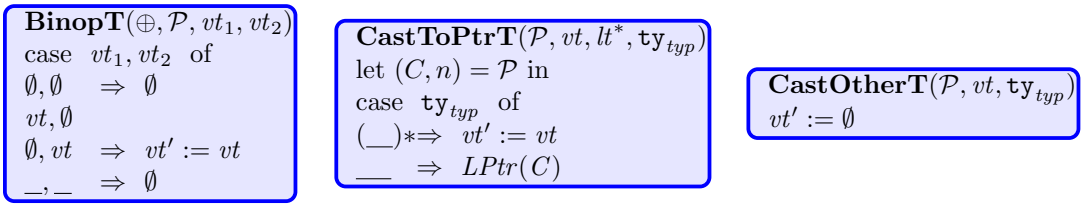


Figure 7.10: CluMPS Binary Operation and Cast Rules

CallT checks that the call only crosses from one compartment to another if the callee is a public function. It then updates the PC tag on entry to match the new compartment. The prohibition against passing a local pointer to or returning one from a public function is enforced by **ArgT** and **RetT**, respectively.

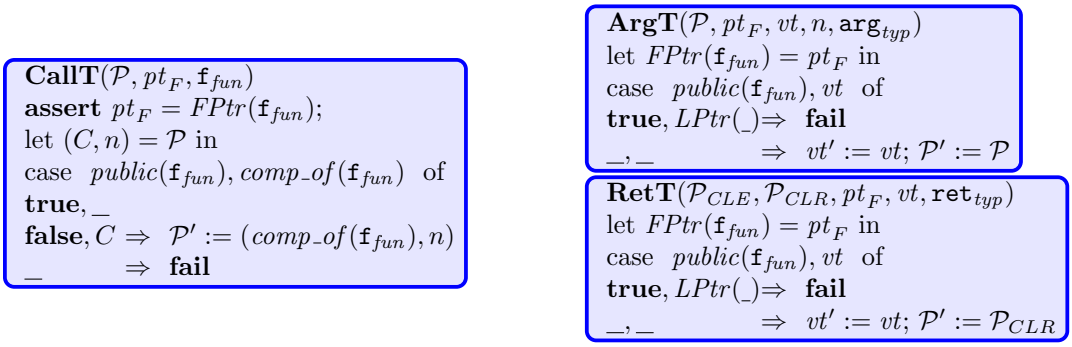


Figure 7.11: CluMPS Function Call Rules

7.4 Verification

Proving correctness is important to guarantee that CluMPS doesn't miss an edge-case and fail to actually enforce the abstract CluMPS model. To show that the policy is correct, I prove a bisimulation between the CluMPS semantics of Section 7.2.3 and

the Tagged C semantics parameterized by CluMPS. Bisimulation means that any behavior exhibited by a program in one machine is also found in the other. The theorems described here are formalized and proven in the Coq development. This section gives a brief overview of the structure of their proofs.

Formally, let P be a whole program and I be a compartmentalization interface. Let $\text{CluMPS}(P, I)$ be the semantics of the CluMPS abstract machine running program P with interface I and $\mathbf{TC}_{\text{CluMPS}(I)}(P)$ be Tagged C running P under the CluMPS policy parameterized by I.

A program's behavior is defined in terms of potentially infinite *traces* of events. Events repesent system calls and failstops; failstops are *terminal events* that appear only at the end of a trace. This trace model is sparser than that of CompCert C, which also contains loads and stores to volatile memory, because in Tagged C it is possible for volatile loads and stores to fail. This in turn necessitates the proof to be done in the more complex setting of "trace-relating compiler correctness" [2].

We write $\text{CluMPS}(P, I) \rightsquigarrow t$ and $\mathbf{TC}_{\text{CluMPS}(I)}(P) \rightsquigarrow t$ to mean that P under the given semantics produces the given trace.

The goal is that, if $\mathbf{TC}_{\text{CluMPS}(I)}(P) \rightsquigarrow t$, then $\text{CluMPS}(P, I) \rightsquigarrow t$. That means that any "bad behavior" exhibited by the Tagged C implementation was not introduced by it, but is the fault of the combination of P and I. In the compiler correctness literature this is termed a *backward simulation*, with the abstract machine taking the place of the source language and Tagged C as the target.

I take the standard CompCert approach to proving backward simulation: first prove *forward simulation*, as well as the *receptivity* of the source and the *determinacy* of the target [50]. (In this proof, both semantics are determinate, which in the case

of $\mathbf{TC}_{\text{CluMPS}(I)}(P)$ requires a modification of the original Tagged C semantics: instead of non-deterministically reducing expressions, it searches for possible expression reductions in a depth-first, left-to-right order and executes the first one that it finds.)

Receptivity is the property that, if the source takes a step that depends on a value from the outside world, it can take a step on any such value. This implies that when the target steps, the source must also be able to take some step, though the resulting state is not guaranteed to match. By forward simulation, the target can make a step to a matching state, but without determinacy this step could differ from the original step. Determinacy means that the target has only one step that it can take, so the original step must be the same as the simulation of the source, and therefore the resulting states match.

Lemma 1. CluMPS(P, I) is receptive. **Mechanized proof in Coq.**

Lemma 2. $\mathbf{TC}_{\text{CluMPS}(I)}(P)$ is determinate. **Mechanized proof in Coq.**

7.4.1 Proof of Forward Simulation

The simulation proof begins with the construction of a *matching relation* $\approx$ between the respective states of CluMPS(P, I) and of $\mathbf{TC}_{\text{CluMPS}(I)}(P)$. This section will present in detailed the relation between values, memories, and top-level states in the respective semantics, as these are where the two semantics differ the most.

Relating Values In CluMPS, a fully evaluated right-hand expression is a value. In Tagged C, it is an atom, a pair of a value and a value tag. The relation between a CluMPS value v_1 and a Tagged C atom $v_2@vt$ is defined in terms of a basic rela-

$$vt\_of\ v = \begin{cases} LPtr(C) & \text{if } v = \texttt{ptr}\ \mathbf{L}(C)\ \_ \\ SPtr(id) & \text{if } v = \texttt{ptr}\ \mathbf{S}(id)\ \_ \\ FPtr(\texttt{f}) & \text{if } v = \texttt{fptr f} \\ \emptyset & \text{otherwise} \end{cases} \qquad lt\_of\ r = \begin{cases} LPtr(C) & \text{if } r = \mathbf{L}(C) \\ SPtr(id) & \text{if } r = \mathbf{S}(id) \end{cases}$$

$$\frac{}{v \sim v} \qquad \frac{}{\texttt{ptr}\ r\ a \sim a} \qquad \frac{v_1 \sim v_2}{v_1 \approx_{RV} v_2@vt\_of\ \underline{v(r, a)} \approx_{LV} \underline{a@lt\_of\ r}}$$

$$\frac{m[a] = v@vt@lt\_of(r) \qquad M[r][a] \approx_{RV} v@vt \qquad \forall r'.\ r \neq r' \rightarrow \neg live\ M\ r\ a}{a \vdash M \sim m}$$

$$\frac{m[a] = v@\emptyset@\emptyset \qquad \forall r.\ \neg live\ M\ r\ a}{a \vdash M \sim m} \qquad \frac{\forall a.\ a \vdash M \sim m}{M \approx_{MEM} m}$$

Figure 7.12: Matching Values, Locations, and Memories

tion between values, $\sim$, and a map vt_of from CluMPS values to tags, as shown in Figure 7.12. Each non-pointer value relates to itself by $\sim$, and a pointer relates to an integer that shares its concrete value. Then the matching relation $v_1 \approx_{RV} v_2@vt$ holds if $v_1 \sim v_2$ and $vt = vt\_of(v_1)$. Figure 7.12 also gives the relation $\approx_{LV}$ for left-values (locations). As in Chapter 4, the Tagged C left-value $\underline{a@pt}$ represents the memory location a tagged with pt. The CluMPS equivalent, $\underline{(r, a)}$, represents the integer address a in region r. These match if their addresses are the same and pt corresponds to $lt\_of\ r$.

Relating Environments and Memories The biggest difference between the semantics of CluMPS and Tagged C are their differently structured memory models. The matching relation presented here is simplified by eliding the need to encode values across multiple bytes during stores and decode them during loads; the proof that

$$\frac{\mathcal{P} = (C, M_1.next) \qquad M_1 \approx_{MEM} m_2 \qquad le_1 \approx_{LE} le_2 \quad te_1 \approx_{TE} te_2 \quad s_1 \approx_{STMT} s_2 \quad k_1 \approx_{CONT} k_2}{\mathfrak{s}\,(C : M_1, le_1, te_1 \mid s_1 \gg k_1) \approx \mathcal{S}\,(m_2, le_2, te_2 \mid s_2 \gg k_2\ @\mathcal{P})}$$

$$\frac{\mathcal{P} = (C, M_1.next) \qquad M_1 \approx_{MEM} m_2 \qquad le_1 \approx_{LE} le_2 \quad te_1 \approx_{TE} te_2 \quad e_1 \approx_{EXPR} e_2 \quad k_1 \approx_{CONT} k_2}{\mathfrak{e}\,(C : M_1, le_1, te_1 \mid e_1 \gg k_1) \approx \mathcal{E}\,(m_2, le_2, te_2 \mid e_2 \gg k_2\ @\mathcal{P})}$$

$$\frac{\mathcal{P} = (C, M_1.next) \quad M_1 \approx_{MEM} m_2 \quad map2(\approx_{RV}, args_1, args_2) \quad k_1 \approx_{CONT} k_2}{\mathfrak{c}\,(C : M_1 \mid f, args_1 \gg k_1) \approx \mathcal{C}\,(m_2 \mid f, vt_f, args \gg k_2\ @\mathcal{P})}$$

$$\frac{\mathcal{P} = (C, M_1.next) \quad M_1 \approx_{MEM} m_2 \quad v_1 \approx_{RV} v_2@vt \quad k_1 \approx_{CONT} k_2}{\mathfrak{r}\,(C : M_1 \mid v_1 \gg k_1) \approx \mathcal{R}\,(m_2 \mid v_2@vt \gg k_2\ @\mathcal{P})}$$

$$\frac{}{\mathfrak{f}\,(\mathbf{err}_1) \approx \mathcal{F}\,(\mathbf{err}_2)}$$

Figure 7.13: Matching states

$\approx_{RV}$ is preserved through this process is non-trivial.

A total memory M and a Tagged C memory m match at a concrete address a, written $a \vdash M \sim m$, if one of two cases holds: either a is LIVE in exactly one region of M and its contents and tags match those in m, or it is DEAD in all regions and in m its location tag is $\emptyset$. The matching relation for memories, $\approx_{MEM}$, relates M and m if they match at all addresses a. This is also given formally in Figure 7.12.

Matching Relation on States The matching relation on states is given in Figure 7.13. Each kind of state matches with another state of the same kind if their corresponding components are related by appropriate matching relations.

The matching relations for expressions ($\approx_{EXPR}$) and statements ($\approx_{STMT}$) are om-

mitted here; they are conceptually trivial, but tedious to write out. Statements and expressions are recursive structures that are identical between the two semantics, except that the values that appear within them may differ.

Consider the CluMPS expression $*(\texttt{ptr}\ r\ a) = \texttt{5}$ and the Tagged C expresssion $*(a@pt) = \texttt{5}@vt$, for instance. They are structurally identical, differing only in the leaves of their abstract syntax trees. If $\texttt{ptr}\ r\ a \approx_{RV} a@pt$ and $5 \approx_{RV} 5@vt$, then the expressions match. Both $\approx_{EXPR}$ and $\approx_{STMT}$ are defined by mapping $\approx_{RV}$ and $\approx_{LV}$ across corresponding values in their structures. The same is true of $\approx_{CONT}$, with the additional property that the compartment of the caller in each call continuation matches the stored PC tag in the Tagged C equivalent.

Lemma 3. If $S_1 \approx S_2$ and $S_1 \longrightarrow S_1'$, then there exists some S_2' such that $S_2 \longrightarrow S_2'$ and $S_1' \approx S_2'$. **Mechanized proof in Coq.**

Theorem 1. The $\approx$ relation is a forward simulation relation between $\textsc{CluMPS}(P, I)$ and $\mathbf{TC}_{\textsc{CluMPS}(I)}(P)$. **Mechanized proof in Coq.** Follows from lemma 3.

Theorem 2. There exists a backward simulation relation between $\textsc{CluMPS}(P, I)$ and $\mathbf{TC}_{\textsc{CluMPS}(I)}(P)$. **Mechanized proof in Coq.** Follows from theorem 1, lemmas 1 and 2, and the "f2b" simulation diagram due to Leroy [50].

7.4.2 Limitations

Determinacy Determinacy is a minor drawback, as it means that the current version of this proof is fixed to the chosen evaluation order. But it is unsatisfying to have to make Tagged C determinate when the languages are close enough to one another

that it should be feasible to directly prove the backward simulation. The CompCert-style proof that I use is intended for proving simulations between languages where on step in the source language corresponds to multiple in the target, and therefore forward simulation is much easier to prove than backward simulation. CluMPS and Tagged C execute in lockstep, so a direct backward simulation would be easier.

Matching Allocator Oracles The behavior of both machines is determined by an allocator oracle that chooses where to place allocated objects in memory. The proof of Theorem 1 assumes that the oracles for both machines place their allocations in the same concrete locations over the course of the run, though for CluMPS the allocations are still spread across different memory regions. This simplifies the proofs immensely. The downside is that it makes the proof unable to be adapted to settings where the target arranges memory differently from the source; for instance, one that clusters local compartments together and protects them via paging.

The proof could be adapted to that setting by allowing the enforcement mechanism's allocations to be a permutation of those of the source machine. However, this would mean that matching pointers, when cast to integers, might not be equal. Those integers might in turn influence control flow and break the lockstep relationship between source and target semantics. One solution would be for the policy to track which integers are derived from pointers and restrict their uses; then the proof would need to account for premature failure in the enforcement mechanism, which is not too complex. Another approach would to treat such integers symbolically, as in Kang et al. [42], but that would be a massive increase in proof complexity as well as being incompatible with determinacy.

7.5 Machine Constraints

Now we consider the constraints that this system places on potential implementations. In particular, in a tag-based enforcement mechanism with a limited quantity of tags, is this system realistic? In general, it requires a unique tag per compartment, as well as one for each shared allocation. In the extreme, consider a system along the lines of PIPE but with only four-bit tags. (There is in fact a system this restrictive, ARM MTE [31].) Such a system could only enforce this semantics for a very small program, or one with very little shared memory: fewer than sixteen tags, so perhaps two-four compartments and around a dozen shared objects. Fewer tag bits would be completely infeasible.

On the other hand, CluMPS semantics is a reasonable goal under an enforcement mechanism with even eight-bit tags (512 compartments and shared objects.) Going up to sixteen bits, the CluMPS policy can support programs with thousands of shared objects.

The remaining pressure on the number of tags stems from the requirement that each shared object have its own color. But is that really necessary? It would also be possible to group shared objects together and do away with the need for dynamic colors entirely.

Consider a variant with one shared region for each combination of compartments, such that an object `x` that is allocated by A and will later be shared with B and C, but not with D, is allocated within region $\{A; B; C\}$. Then `x` will be guaranteed not to be accessed by D, although it could be accessed by C before its pointer is actually shared with C, via pointer arithmetic from another object in the same region. The

upshot is that such a system only uses a static number of tags that is less than $2^{\|C\|}$, and in practice may be as small as the number of combinations that actually occur. If the system is very constrained, it would also be possible to over-approximate sharing configurations at some cost in granularity.

7.6 Reflections

In specifying, implementing, and verifying CluMPS, I have tested how Tagged C performs as a platform for these tasks on a fairly complex policy. While the three tasks can be presented as a straightforward progression from specification to implementation and then the verification of that implementation, in practice it can be more of an iterative cycle, as the verification process reveals flaws in the implementation or in the formal specification.

Compared to Assembly-level Implementation CluMPS makes an interesting point of comparison between Tagged C and assembly-level tagging. What would it take to implement CluMPS at the assembly level?

The main ways that Tagged C is easier to work with in this case concern function calls and returns. The assembly-level policy must rely on compiler annotations to identify which `jalr` (or equivalent) instructions are calls and which functions are being called. If function pointers are used the compiler must make similar annotations at the point that their address is taken. Assembly code is not explicit about function arguments, so the compiler must also identify details such as which registers and memory locations contain arguments.

Given these annotations, new instructions must be added to carry the **ArgT** checks. Likewise, new instructions must be added to `malloc` to initialize newly allocated blocks, giving them their location tags. There is no need to distinguish `malloc` calls by distinct identifiers: their call instructions can be directly tagged to distinguish them. But the implementation of **MallocT** needs to be spread across the instructions that make up the body of `malloc`.

In short: a system implementing CluMPS at the assembly level must implement a significant subset of Tagged C's bookkeeping. At the same time, the most likely approach to proving correctness of such a system would be to track those annotations and maintain a source-level "ghost state" of the execution of the abstract machine as the relevant instructions execute. Instead, Tagged C allows the proof to be done at the source level, much more directly.

7.7 Other Backends

The CluMPS model is not only interesting for tags, but as a specification for compartments more generally. It combines the advantages of clustered allocations with the ability to safely share memory. It is broadly suitable for enforcement with various techniques, including software fault isolation (SFI) and the CHERI hardware-capability mechanism.

Software Fault Isolation SFI [78] enforces memory isolation by aligning regions of protected memory so that their addresses share their higher-order bits. Then all memory accesses are instrumented by instructions that mask those bits of the pointer

being accessed, forcing the access to occur within the appropriate region. An access that would be out of bounds is redirected to occur somewhere within the region.

This approach is reasonably well-suited to protecting local regions: each compartment can be given a large region of memory within which to allocate all of its local allocations. It would not work with dynamic shared allocations, because each individual memory access needs to be instrumented with the (static) target region. Instead, SFI could be a reasonable approach for the variant of CluMPS described in Section 7.5. As long as it is possible to statically separate accesses to each class of shared pointer, they can each be allocated within a separately aligned region and protected in the same way as the local regions.

The other issue with SFI is what happens in the case of an out-of-bounds access. Rather than failstop, SFI redirects the illegal access to an effectively arbitrary location within the legal region. This means that an SFI backend will need to be validated against a different formal property, ideally one that captures a notion of "blame"—that a compartment can only be corrupted by its own erroneous access.

CHERI The more natural alternative to PIPE would be CHERI [79], a hardware capability mechanism. The CHERI ISA provides instructions for creating unforgeable pointers that grant access to a region of memory. It would be straightforward to use these capabilities to protect both large, local regions and fine-grained shared regions. The CluMPS semantics' requirement that local pointers not escape would now serve an additional role in maintaining the security of local regions.

8 Conclusion and Future Work

This dissertation builds a foundation for a C runtime enforcement ecosystem. At its heart is Tagged C, which I have demonstrated is flexible enough to express a wide range of tag-based policies and suitable for reasoning about their behavior. It has already seen some practical use by Naaktgeboren et al. [59] for research into policy-informed fuzz testing, and has the potential to mature into a powerful platform for studies on the interaction between policies and source programs. Below the C level, my stack safety properties specify a vital form of low-level security. And the CluMPS compartmentalization model is a good candidate for enforcement in systems beyond Tagged C.

8.1 Evaluating Technical Contributions

This section aims to give an impartial evaluation of each of the three topics of the dissertation. What went well? What is lacking?

8.1.1 Stack Safety

The key technical contribution of Chapter 3 is the randomized property-based testing of stack safety policies from the literature. This demonstrates the utility of the properties in establishing the correctness of existing enforcement mechanisms. The tests successfully identify a gap in Roessler and DeHon's Lazy Tagging and Clearing

policy [67]. On the negative: randomized testing is a significantly weaker form of validation than verification. I only test one enforcement mechanism, tags, though I do test two distinct tag policies; it is therefore not clear how easily the system would extend to alternative backends.

An important scientific take-away from my work on stack safety is that while protection mechanisms focus on the stack as a data structure, "stack safety" should be thought of as encompassing the fundamental abstraction of function calls and returns. In fact, unusual systems that lack an explicit stack data structure (as opposed to the implicit stack created by subroutines) can still enjoy stack safety, or fail to do so. It might more aptly be termed "function safety" or "call safety." The division of stack safety into separate integrity and confidentiality properties for callees and callers follows from this interpretation.

8.1.2 Tagged C

The goal Tagged C is to provide a flexible framework for writing security policies, covering the entire C language. It can be evaluated on these merits: is it useful for creating policies in practice, and tweaking them to meet the user's precise goals? How well does it cover the intricacies of C?

Tagged C in Practice The proof of the pudding is in the eating, and the proof of Tagged C is in the creation of realistic policies by an outside user. Naaktgeboren et al. use Tagged C as a component in a fuzz-testing framework, Pipecleaner [59]. Where most fuzzers execute a program on randomized inputs and look for inputs

that cause a crash, Pipecleaner looks for inputs that violate a given policy. This can thought of as the equivalent of property-based testing, but Naaktgeboren et al. take it further and design policies that log additional information and turn Tagged C into a tool for bug triage. This line of work represents a success for Tagged C's flexibility.

It is a valuable source of insight into what Tagged C users need. Namely, it is enormously helpful to distinguish between different types of tags, to provide a straightforward logging mechanism, and to define a comprehensive set of default behaviors for control points that are not related to the policy being written.

Coverage Tagged C covers the entire C language, but it is lacking in its treatment of a few constructs, particularly composite types (structs and unions). Tagged C follows CompCert C in treating composite types in a pass-by-reference style at all times, which in Tagged C means that the tag on a struct the tag on its pointer. This means that assigning and passing structs and unions does not work very intuitively. It is worth noting that these cases have yet to actually be problematic to a real policy, either one discussed in this dissertation or from Pipecleaner.

Pain Points Other aspects of Tagged C are functional, but are overly complex or otherwise difficult to use in practice. In Section 7.3, two variations on `malloc` with the same non-tag behavior need to be distinguished from one another. That policy distinguishes them by associating them with different function pointer tags, which then dispatch different versions of the **MallocT** tag rule. This kind of policy design is complex and could stand to be simplified.

8.1.3 CluMPS

The CluMPS development contains an abstract semantics for a CluMPS machine, the policy definition for a policy that enforces it, and the proof of correctness for the polucy definition. The proof represents a successful use of Tagged C to specify and verify a security property of a policy. It also means that the CluMPS policy is validated to the high standards of mechanized proof.

On the other hand, CluMPS lacks much analysis of how it performs on real-world programs. The concept of separating memory allocations into local and shared regions is promising, but the next step is to see how it applies to actual patterns of memory allocation and sharing.

8.2 Future Work

8.2.1 Tagged C Compiler

A compiler for Tagged C would be a massive step forward in usability. Not only would it allow for larger programs and more efficient execution, it would be useful for experiments in linking Tagged C with assembly code.

Scaling The current Tagged C interpreter is not well-suited for running large or expensive programs. It has limited support for a few standard library functions, each of which is hand-coded in Ocaml. All other library functions must be combined with the program in a single file. This will not scale well to larger, more realistic programs. A compiled Tagged C will be more efficient, even without hardware tagging, and would

support multiple compilation units. This would allow programs to link with other libraries written in C.

Assembly Linking When all tagging happens at the assembly-level, the assumption is that all assembly code follows the same rules, but unknown linked code lacks instruction tags designating security-relevant operations such as calls and returns. This makes it hard to reason about policies like stack safety that depend on those labels.

Tagged C's semantics naturally cannot model the behavior of arbitrary assembly code, so reasoning about such a system at the source level requires making assumptions about how "external" code interacts with tags. For example, when considering a memory safety policy, it is reasonable to assume that external code can only access memory if it has first recieved a value tagged with that memory's color. But Tagged C cannot make this claim directly. Compiled programs could be tested to give empirical evidence that such assumptions actually do hold.

Feasibility A simple, non-optimizing, unverified compiler from Tagged C to PIPE-enhanced assembly code would be a straightforward but non-trivial project. The general technique for compilation has already been established [16]. A verified optimizing compiler would only be feasible by reusing the CompCert compilation chain and would easily be multiple person-years of work to amend the proofs to take tags into account. Verification would be very important, however, to ensure that optimizations cannot subvert the policy.

8.2.2 More Policies and Proofs

The three families of policies discussed in this dissertation cover a good range of desirable policies, but here are some areas that would be interesting to study further.

Information Flow Proofs Information-flow policies come in many shapes and sizes, and there is a massive body of literature on the non-interference-based specifications of various kinds. This dissertation has focused on memory-related policies, but a good candidate for the next policy to verify would be some kind of information flow policy.

In particular, a "heap-address leak" policy as discussed in Naaktgeboren et al. would be interesting [59]. Randomizing the layout of memory is a common defensive technique for making code harder to exploit, but it can be overcome if an attacker is able to deduce the position of the heap. An information-flow policy that marks heap pointers (as opposed to their contents) as secret could detect when such a leak occurs. This is an interesting area to study: it differs from standard secure information flow in key ways. In SIF, addresses are not normally secret, though they can become tainted indirectly. In scenarios where an address is secret, however, its contents should also be treated as secret. But preventing heap-address leaks means treating addresses as secret while allowing their contents to be freely accessed and possibly released.

Type Confusion Type confusion covers a range of relatively subtle attacks. In C, values in memory do not have types as such; the pointer that stores or loads such a value has a type that determines implementation details such as how many bytes should be read. By storing a value through a pointer to a type larger than

the allocated block, type confusion has the potential to lead to memory corruption. These attacks should mostly be caught by existing memory safety policies, but it would be worth exploring such attacks in more detail to determine if any are missed. If needed, a type-safety policy could dynamically track the effective types of values in memory and enforce a stricter type safety than C normally does.

Substructural Memory Safety Another longer-term goal is to write memory safety policies that protect the fields within a struct, not just the struct itself. As noted in Section 6.4, this will not be possible with heap memory for the time being, because of limitations in `malloc`. It might be feasible to write a substructural safety policy for just the stack, however, because stack-allocated objects have fixed types that are known at allocation type. That type information can inform the initial tagging of the objects, giving different tags to different fields and then protecting them from internal overflows.

The fundamental challenges of substructural memory safety extend to other contexts as well. Consider a block of memory that contains a union of types whose sizes differ. When one of its smaller fields is assigned to, every byte outside of the extend of that field becomes invalid. But it is not obvious how we can protect those invalid bytes with tags, because they are outside the bounds of the store operation that made the change. This is a challenging area that deserves further study.

8.2.3 Pointcut Language

Runtime verification systems that use pointcuts to specify their behavior tend to focus more on their external behavior, while Tagged C is focused on the internal operations of the program. It could be interesting to combine the two, perhaps using pointcuts in the style of Havelund et al. [36] as a way of designating certain points in the code to have different tag behavior, or using tags to inform the actions to be taken at a particular pointcut. The latter would be great for information flow policies: use pointcuts to designate the security levels of output channels, and tags to track the levels of the data that they might output.

References

[1] Hope policies. https://github.com/draperlaboratory/hope-policies, 2021.

[2] Abate, C., Blanco, R., Ciobâcă, Ş., Durier, A., Garg, D., Hritcu, C., Patrignani, M., Tanter, É., and Thibault, J. Trace-relating compiler correctness and secure compilation. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings* (2020), P. Müller, Ed., vol. 12075 of *Lecture Notes in Computer Science*, Springer, pp. 1–28. https://doi.org/10.1007/978-3-030-44914-8_1.

[3] Abate, C., de Amorim, A. A., Blanco, R., Evans, A. N., Fachini, G., Hritcu, C., Laurent, T., Pierce, B. C., Stronati, M., and Tolmach, A. When good components go bad: Formally secure compilation despite dynamic compromise. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018* (2018), D. Lie, M. Mannan, M. Backes, and X. Wang, Eds., ACM, pp. 1351–1368. https://doi.org/10.1145/3243734.3243745.

[4] Aleph1. Smashing the stack for fun and profit. *Phrack 7*, 49 (November 1996).

[5] Anderson, J. P. Computer security technology planning study. Technical Report ESD-TR-73-51, U.S. Air Force Electronic Systems Division, Oct. 1972.

[6] Anderson, S. N., Blanco, R., Lampropoulos, L., Pierce, B. C., and Tolmach, A. Formalizing stack safety as a security property. In *36th IEEE Computer Security Foundations Symposium, CSF 2023, Dubrovnik, Croatia, July 10-14, 2023* (2023), IEEE, pp. 356–371. https://doi.org/10.1109/CSF57540.2023.00037.

[7] Anderson, S. N., Naaktgeboren, A., and Tolmach, A. Flexible runtime security enforcement with tagged C. In *Runtime Verification - 23rd International Conference, RV 2023, Thessaloniki, Greece, October 3-6, 2023, Proceedings* (2023), P. Katsaros and L. Nenzi, Eds., vol. 14245 of *Lecture Notes in Computer Science*, Springer, pp. 231–250. https://doi.org/10.1007/978-3-031-44267-4_12.

[8] Azevedo de Amorim, A., Collins, N., DeHon, A., Demange, D., Hritcu, C., Pichardie, D., Pierce, B. C., Pollack, R., and Tolmach, A. A verified information-flow architecture. *Journal of Computer Security 24*, 6 (2016), 689–734. http://dx.doi.org/10.3233/JCS-15784.

[9] Azevedo de Amorim, A., Dénès, M., Giannarakis, N., Hriţcu, C., Pierce, B. C., Spector-Zabusky, A., and Tolmach, A. Micro-policies: Formally verified, tag-based security monitors. In *36th IEEE Symposium on Security and Privacy (Oakland S&P)* (May 2015), IEEE. https://www.cis.upenn.edu/~aarthur/micro-policies-full.pdf.

[10] Azevedo de Amorim, A., Hritcu, C., and Pierce, B. C. The meaning of memory safety. In *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings* (2018), L. Bauer and R. Küsters, Eds., vol. 10804 of *Lecture Notes in Computer Science*, Springer, pp. 79–105. https://doi.org/10.1007/978-3-319-89722-6_4.

[11] Ball, T., and Rajamani, S. SLIC: A specification language for interface checking (of C). Tech. Rep. MSR-TR-2001-21, January 2002. https://www.microsoft.com/en-us/research/publication/slic-a-specification-language-for-interface-checking-of-c/.

[12] Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J.-C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., Parent, C., Paulin-Mohring, C., Saibi, A., and Werner, B. The Coq proof assistant reference manual: Version 6.1. Technical Report RT-0203, INRIA, 1997.

[13] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM 53*, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374.

[14] Bourgeat, T., Clester, I., Erbsen, A., Gruetter, S., Wright, A., and Chlipala, A. A multipurpose formal RISC-V specification. *ArXiv abs/2104.00762* (2021). https://people.csail.mit.edu/bthom/riscv-spec.pdf.

[15] Chabot, M., Mazet, K., and Pierre, L. Automatic and configurable instrumentation of C programs with temporal assertion checkers. In *13. ACM/IEEE*

International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, Austin, TX, USA, September 21-23, 2015 (2015), IEEE, pp. 208–217. https://doi.org/10.1109/MEMCOD.2015.7340488.

[16] Chhak, C., Tolmach, A. P., and Anderson, S. N. Towards formally verified compilation of tag-based policy enforcement. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs* (2021), p. 137–151. https://doi.org/10.1145/3437992.3439929.

[17] Chisnall, D., Rothwell, C., Watson, R. N., Woodruff, J., Vadera, M., Moore, S. W., Roe, M., Davis, B., and Neumann, P. G. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS '15, Association for Computing Machinery, p. 117–130. https://doi.org/10.1145/2694344.2694367.

[18] Chromium Projects. Chromium security:memory safety. https://www.chromium.org/Home/chromium-security/memory-safety/.

[19] Clause, J., Doudalis, I., Orso, A., and Prvulovic, M. Effective memory protection using dynamic tainting. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering* (2007), p. 284–292.

[20] Condit, J., Harren, M., McPeak, S., Necula, G. C., and Weimer, W. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2003), PLDI '03, Association for Computing Machinery, p. 232–244. https://doi.org/10.1145/781131.781157.

[21] Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7* (USA, 1998), SSYM'98, USENIX Association, p. 5.

[22] Dang, T. H., Maniatis, P., and Wagner, D. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2015), ASIA CCS '15, Association for Computing Machinery, p. 555–566. https://doi.org/10.1145/2714576.2714635.

[23] Dénès, M., Hritcu, C., Lampropoulos, L., Paraskevopoulou, Z., and Pierce, B. C. QuickChick: Property-based testing for Coq (abstract). In *Proceedings of the Vienna Summer of Logic* (2014).

[24] Denning, D. E. A lattice model of secure information flow. *Commun. ACM 19*, 5 (May 1976), 236–243.

[25] Denning, D. E., and Denning, P. J. Certification of programs for secure information flow. *Commun. ACM 20*, 7 (July 1977), 504–513.

[26] Devietti, J., Blundell, C., Martin, M. M. K., and Zdancewic, S. HardBound: Architectural support for spatial safety of the C programming language. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems* (2008), pp. 103–114. http://acg.cis.upenn.edu/papers/asplos08_hardbound.pdf.

[27] Dhawan, U., Hritcu, C., Rubin, R., Vasilakis, N., Chiricescu, S., Smith, J. M., Jr., T. F. K., Pierce, B. C., and DeHon, A. Architectural support for software-defined metadata processing. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015* (2015), Ö. Özturk, K. Ebcioglu, and S. Dwarkadas, Eds., ACM, pp. 487–502. https://doi.org/10.1145/2694344.2694383.

[28] Dhawan, U., Vasilakis, N., Rubin, R., Chiricescu, S., Smith, J. M., Jr., T. F. K., Pierce, B. C., and DeHon, A. PUMP: a programmable unit for metadata processing. In *HASP 2014, Hardware and Architectural Support for Security and Privacy, Minneapolis, MN, USA, June 15, 2014* (2014), R. B. Lee and W. Shi, Eds., ACM, pp. 8:1–8:8. https://doi.org/10.1145/2611765.2611773.

[29] El-Korashy, A., Blanco, R., Thibault, J., Durier, A., Garg, D., and Hritcu, C. SecurePtrs: Proving secure compilation with data-flow back-translation and turn-taking simulation. In *2022 2022 IEEE 35th Computer Security Foundations Symposium (CSF) (CSF)* (Los Alamitos, CA, USA, Aug. 2022), IEEE Computer Society, pp. 64–79. https://doi.ieeecomputersociety.org/10.1109/CSF54842.2022.9919680.

[30] Engler, D. R., Chelf, B., Chou, A., and Hallem, S. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI* (2000), pp. 1–16.

[31] Frascino, V. Arm v8. 5 memory tagging extension. In *Proceedings of the Linux Plumbers Conference, Lisbon, Portugal* (2019), vol. 10.

[32] Georges, A. L., Guéneau, A., Van Strydonck, T., Timany, A., Trieu, A., Huyghebaert, S., Devriese, D., and Birkedal, L. Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang. 5*, POPL (jan 2021). https://doi.org/10.1145/3434287.

[33] Georges, A. L., Trieu, A., and Birkedal, L. Le temps des cerises: Efficient temporal stack safety on capability machines using directed capabilities. *Proc. ACM Program. Lang. 6*, OOPSLA1 (Apr. 2022). https://doi.org/10.1145/3527318.

[34] Gollapudi, R., Yuksek, G., Demicco, D., Cole, M., Kothari, G. N., Kulkarni, R. H., Zhang, X., Ghose, K., Prakash, A., and Umrigar, Z. Control flow and pointer integrity enforcement in a secure tagged architecture. In *2023 IEEE Symposium on Security and Privacy (SP)* (May 2023), pp. 2974–2989.

[35] Havelund, K. Runtime verification of C programs. In *Testing of Software and Communicating Systems, 20th IFIP TC 6/WG 6.1 International Conference, TestCom 2008, 8th International Workshop, FATES 2008, Tokyo, Japan, June 10-13, 2008, Proceedings* (2008), K. Suzuki, T. Higashino, A. Ulrich, and T. Hasegawa, Eds., vol. 5047 of *Lecture Notes in Computer Science*, Springer, pp. 7–22. https://doi.org/10.1007/978-3-540-68524-1_3.

[36] Havelund, K., and VanWyk, E. Aspect-oriented monitoring of C programs. In *3rd Domain-Specific Aspect Languages Workshop* (Apr. 2008).

[37] Hriţcu, C., Lampropoulos, L., Spector-Zabusky, A., Azevedo de Amorim, A., Dénès, M., Hughes, J., Pierce, B. C., and Vytiniotis, D. Testing noninterference, quickly. *J. Funct. Program. 26* (2016), e4. https://doi.org/10.1017/S0956796816000058.

[38] Hu, H., Shinde, S., Adrian, S., Chua, Z. L., Saxena, P., and Liang, Z. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016* (2016), IEEE Computer Society, pp. 969–986.

[39] Jia, Y., and Harman, M. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering 37*, 5 (2011), 649–678.

[40] Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J., and Wang, Y. Cyclone: A safe dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA* (2002), C. S. Ellis, Ed., USENIX, pp. 275–288.

[41] Juglaret, Y., Hritcu, C., de Amorim, A. A., Eng, B., and Pierce, B. C. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016* (2016), IEEE Computer Society, pp. 45–60. https://doi.org/10.1109/CSF.2016.11.

[42] Kang, J., Hur, C.-K., Mansky, W., Garbuzov, D., Zdancewic, S., and Vafeiadis, V. A Formal C Memory Model Supporting Integer-Pointer Casts. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2015), PLDI '15, Association for Computing Machinery, p. 326–335.

[43] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming* (1997), M. Akşit and S. Matsuoka, Eds., Springer, pp. 220–242.

[44] Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., and Song, D. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (USA, 2014), OSDI'14, USENIX Association, p. 147–163.

[45] Lampropoulos, L., and Pierce, B. C. *QuickChick: Property-Based Testing in Coq*. Software Foundations series, volume 4. Electronic textbook, Aug. 2018. http://www.cis.upenn.edu/~bcpierce/sf.

[46] Lampson, B. W. Protection. *SIGOPS Oper. Syst. Rev. 8*, 1 (Jan 1974), 18–24.

[47] Lefeuvre, H., Dautenhahn, N., Chisnall, D., and Olivier, P. Sok: Software compartmentalization. *CoRR abs/2410.08434* (2024). https://doi.org/10.48550/arXiv.2410.08434.

[48] Leroy, X. Compcert 3.10. https://github.com/AbsInt/CompCert/releases/tag/v3.10.

[49] Leroy, X. Formal verification of a realistic compiler. *Commun. ACM 52*, 7 (July 2009), 107–115. https://xavierleroy.org/publi/compcert-CACM.pdf.

[50] Leroy, X. A formally verified compiler back-end. *J. Autom. Reason. 43*, 4 (Dec. 2009), 363–446.

[51] Leroy, X., Appel, A. W., Blazy, S., and Stewart, G. The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA, June 2012.

[52] Li, J. Color reclamation for heap memory coloring scheme in pipe tagged-memory architecture. Thesis: M. Eng., Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2019, Sept. 2019. https://dspace.mit.edu/bitstream/handle/1721.1/124255/1145123236-MIT.pdf.

[53] Liljestrand, H., Chinea, C., Denis-Courmont, R., Ekberg, J.-E., and Asokan, N. Color my world: Deterministic tagging for memory safety, 2022.

[54] Memarian, K., Gomes, V. B. F., Davis, B., Kell, S., Richardson, A., Watson, R. N. M., and Sewell, P. Exploring C semantics and pointer provenance. *Proc. ACM Program. Lang. 3*, POPL (Jan. 2019).

[55] Memarian, K., Matthiesen, J., Lingard, J., Nienhuis, K., Chisnall, D., Watson, R. N. M., and Sewell, P. Into the depths of C: Elaborating the de facto standards. *SIGPLAN Not. 51*, 6 (June 2016), 1–15.

[56] Michael, A. E., Gollamudi, A., Bosamiya, J., Johnson, E., Denlinger, A., Disselkoen, C., Watt, C., Parno, B., Patrignani, M., Vassena, M., and Stefan, D. MSWasm: Soundly enforcing memory-safe execution of unsafe code. *Proc. ACM Program. Lang. 7*, POPL (Jan. 2023).

[57] Miller, M. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/, 2019.

[58] MITRE Corporation. Common Weakness Enumeration:2022 top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html, 2022.

[59] Naaktgeboren, A., Anderson, S. N., Tolmach, A., and Sullivan, G. Pipe-cleaner: Flexible fuzzing using security policies. *CoRR abs/2411.00261* (2024). https://arxiv.org/abs/2411.00261.

[60] Nagarakatte, S., Martin, M. M. K., and Zdancewic, S. Watchdog: Hardware for safe and secure manual memory management and full memory safety. *SIGARCH Comput. Archit. News 40*, 3 (June 2012), 189–200.

[61] Nagarakatte, S., Martin, M. M. K., and Zdancewic, S. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (New York, NY, USA, 2014), CGO '14, Association for Computing Machinery, p. 175–184.

[62] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. SoftBound: Highly compatible and complete spatial memory safety for C. *SIGPLAN Not. 44*, 6 (June 2009), 245–258.

[63] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. CETS: Compiler enforced temporal safety for C. In *Proceedings of the 2010 International Symposium on Memory Management* (New York, NY, USA, 2010), ISMM '10, Association for Computing Machinery, p. 31–40.

[64] Oiwa, Y. Implementation of the memory-safe full ANSI-C compiler. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2009), PLDI '09, Association for Computing Machinery, p. 259–269. https://doi.org/10.1145/1542476.1542505.

[65] RISC-V Consortium. RISC-V calling conventions. https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-cc.adoc.

[66] Roessler, N., Atayde, L., Palmer, I., McKee, D., Pandey, J., Kemerlis, V. P., Payer, M., Bates, A., Smith, J. M., DeHon, A., and Dautenhahn, N. μSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts. In *24th International Symposium on Research in Attacks, Intrusions and Defenses* (New York, NY, USA, 2021), RAID '21, Association for Computing Machinery, p. 296–311.

[67] Roessler, N., and DeHon, A. Protecting the stack with metadata policies and tagged hardware. In *2018 IEEE Symposium on Security and Privacy (Oakland S&P), Proceedings, 21-23 May 2018, San Francisco, California, USA* (2018), IEEE Computer Society, pp. 478–495.

[68] Sabelfeld, A., and Myers, A. C. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications 21*, 1 (2003), 5–19.

[69] Shanbhogue, V., Gupta, D., and Sahita, R. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for*

Security and Privacy (New York, NY, USA, 2019), HASP '19, Association for Computing Machinery.

[70] Skorstengaard, L., Devriese, D., and Birkedal, L. Reasoning about a machine with local capabilities: Provably safe stack and return pointer management. *ACM Trans. Program. Lang. Syst. 42*, 1 (Dec. 2019). https://doi.org/10.1145/3363519.

[71] Skorstengaard, L., Devriese, D., and Birkedal, L. StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *J. Funct. Program. 31* (2021). https://doi.org/10.1017/S095679682100006X.

[72] Szekeres, L., Payer, M., Wei, T., and Song, D. SoK: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy (Oakland S&P), San Francisco, CA, USA, May 19-22, 2013* (2013), IEEE Computer Society, pp. 48–62. https://doi.org/10.1109/SP.2013.13.

[73] Tarditi, D., Elliott, A. S., Ruef, A., and Hicks, M. Checked c: Making c safe by extension. In *IEEE Cybersecurity Development Conference 2018 (SecDev)* (September 2018), IEEE, pp. 53–60.

[74] The Coq Development Team. The Coq reference manual – release 8.19.0. https://coq.inria.fr/doc/V8.19.0/refman, 2024.

[75] Thibault, J., Blanco, R., Lee, D., Argo, S., de Amorim, A. A., Georges, A. L., Hritcu, C., and Tolmach, A. SECOMP: formally secure compilation of compartmentalized C programs. *CoRR abs/2401.16277* (2024). https://doi.org/10.48550/arXiv.2401.16277.

[76] Tolmach, A., Chhak, C., and Anderson, S. N. Defining and preserving more C behaviors: Verified compilation using a concrete memory model. In *15th International Conference on Interactive Theorem Proving, ITP 2024, September 9-14, 2024, Tbilisi, Georgia* (2024), Y. Bertot, T. Kutsia, and M. Norrish, Eds., vol. 309 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 36:1–36:20. https://doi.org/10.4230/LIPIcs.ITP.2024.36.

[77] van der Veen, V., dutt-Sharma, N., Cavallaro, L., and Bos, H. Memory errors: The past, the present, and the future. In *Research in Attacks, Intrusions, and Defenses - 15th International Symposium, RAID 2012, Amsterdam, The Netherlands, September 12-14, 2012. Proceedings* (2012), D. Balzarotti, S. J.

Stolfo, and M. Cova, Eds., vol. 7462 of *Lecture Notes in Computer Science*, Springer, pp. 86–106.

[78] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient Software-based Fault Isolation. *SIGOPS Oper. Syst. Rev. 27* (12 1993), 203–216.

[79] Watson, R. N., Woodruff, J., Neumann, P. G., Moore, S. W., Anderson, J., Chisnall, D., Dave, N., Davis, B., Gudka, K., Laurie, B., Murdoch, S. J., Norton, R., Roe, M., Son, S., and Vadera, M. CHERI: A Hybrid Capability-system Architecture for Scalable Software Compartmentalization.

[80] Wesley Filardo, N., Gutstein, B. F., Woodruff, J., Ainsworth, S., Paul-Trifu, L., Davis, B., Xia, H., Tomasz Napierala, E., Richardson, A., Baldwin, J., Chisnall, D., Clarke, J., Gudka, K., Joannou, A., Theodore Markettos, A., Mazzinghi, A., Norton, R. M., Roe, M., Sewell, P., Son, S., Jones, T. M., Moore, S. W., Neumann, P. G., and Watson, R. N. M. Cornucopia: Temporal safety for CHERI heaps. In *2020 IEEE Symposium on Security and Privacy (S&P)* (2020), pp. 608–625.

[81] Woodruff, J., Watson, R. N., Chisnall, D., Moore, S. W., Anderson, J., Davis, B., Laurie, B., Neumann, P. G., Norton, R., and Roe, M. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (2014), ISCA '14, IEEE Press, p. 457–468.