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. Safe language or not, any 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 [1]. A security policy restricts the behavior of the system, typically by interrupting a badly-behaved process, termed "failstop behavior." At the most general, a policy could be any kind of runtime check, from simple assertions ("at line X, variable Y has value Z") to sophisticated temporal logic formulae.

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 in some way secret or sensitive is preventing from leaking on an externally visible
- *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

The class of tag-based policies include a number of important security concepts, with implementations that are well-suited to efficient hardware enforcement. 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. We take as our exemplar the PIPE[1] (Processor Interlocks for Policy Enforcement) ISA extension [6,8], a proposed implementation 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 [53] 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" [8], 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 assembly program 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 [6], a given policy may not precisely match it. Once defined and specified, the policy needs to be validated, either by testing or formal verification. Verification is preferable, as proofs rule 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.

These are the challenges that this dissertation will address, both by providing new specifications, policies, and proofs, and by making the three tasks easier with a

powerful new source language, Tagged C.

In the next section we will walk through two example policies implemented in a PIPE-style system to understand how they are written, why doing so is challenging, and why a tag-aware source language is a big step forward.

1.1 Tag Policies By Example

This section shows two example policies. The first, a simplified "spatial" stack safety policy, protects the data of suspended stack frames from being accessed by other function activations. This prevents some instances where assembly code can break 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 the behavior 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, even though IFC would be 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 will 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. Many policies require the binary to be rewritten with additional instructions whose primary purpose is moving and manipulating tags.

Example: Spatial Stack Safety For example, Figure 1.1 shows how a single function header must be updated 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 header 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 header 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 header 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 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 it were 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 due to performance constraints, as we will later see in a stack safety policy from the literature [53].

Example: IFC The example in Figure 1.2 illustrates another scenario that requires the assembly to be rewritten. An important class of "information flow control" (IFC) policies needs to keep track of when execution is in a state that depends on a secret value. In the source snippet in 1.2a, the choice of whether execution reaches line 3 depends on the value of `x`, but it will always reach line 5 regardless of `x`.

The policy uses a binary tag with values H (high security/secret) and L (low security/public), and uses the operatar $\cdot \cup \cdot$ to take the higher of two tags. When control flow depends on a secret, the policy sets the tag on the program counter to H, and then lowers it to its previous value once the branches of the conditional join. In

```
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) Code tagged and expanded for policy

When executing `sub imm r`@Head(0):
- Preceding instruction must have tag Call
- Tag on `r` must be Depth(n) for some n
- Set tag on `r` to Depth($n+1$)

When executing `store r1 r2`@Head(1):
- Preceding instruction must have tag Head(0)
- Set tag at `r2`'s target to RetPtr

When executing `store imm (r+x)`@Head(n):
- Preceding instruction must have tag Head($n-1$)
- Tag on `r` must be Depth(m)
- Set tag at `r`'s target to Depth(m)

When executing _@Entry
- Preceding instruction must have tag Head($size/8$)

When executing `load _ (r+x)`@Normal
- If tag on `r`'s target is Depth(m), then tag on `r` must be tagged Depth(m)

When executing `store _ (r+x)`@Normal
- If tag on `r`'s target is Depth(m), then tag on `r` must be tagged Depth(m)

(c) Associated policy rules

Figure 1.1: Example: Adding Stack Safety policy at call

```
1:  int x, y;
2:  if (x ==
42) {
3:       y = 0;
4:  }
5:  x = 0;
```

(a) Source program

```
0: load r0 (sp+8)      Load x
8: add zero 42 r1      Constant 42
16: bne r0 r1 16       Branch past if
24: store 0 (sp+16)    Store to y
32: ...
```

(b) Initial generated code

```
0: load r0 (sp+8)      @ Norm
8: add zero 42 r1      @ Norm
16: sub pc pc r2       @ SaveTag
24: bne r0 r1 16       @ Split
32: store 0 (sp+16)    @ Normal
40: add zero r2 r2     @ Join
```

(c) Code tagged and expanded for policy

When executing `bne r1 r2` @Split

- Let t_1 be the tag on `r1` and t_2 on `r2`
- Let *pct* be the tag on the program counter
- Set tag on program counter to $pct \sqcup t_1 \sqcup t_2$

When executing `ad`

- Let t be the tag o
- Set tag on progra
- Set tag on `r2` to

Figure 1.2: Example: Adding IFC policy at if statement

the expanded code in Figure **??**, the subtraction instruction at 16 saves a zero tagged with the program counter's tag to `r2`. Then, at the join point at 40, the addition instructio enables the policy to restore the tag on `r2` to the program counter. These instructions perform important bookkeeping, but all the policy designer should need to know is that the program counter's tag is saved at the split point and is available to be restored at the join point.

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 within the compiler! Then the compiler can handle most of the bookkeeping, leaving the policy developer free to focus on the rules that are relevant to the policy at hand.

Moving the task of policy definition to the source level has benefits to 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: a source-level specification of memory safety can refer to the source semantics' notion of a heap, which does not exist in assembly.

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` statment 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 **??** sketches an outline of an ideal such 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 effect 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. A compiler should be proven to not only produce correct code, but to preserve the behavior of any source policy. To assist that preservation proof, the compiler will also need to enforce a "baseline policy" that ensures that protects the abstractions of the source language, especially control flow. Other assembly-level policies might be attached as well: not all policies make sense to apply at the source level, e.g., stack safety, and such policies might also be applied to arbitrary assembly programs that do not come from the tag-aware compiler. An interpreter should also be proven sound and complete with respect to the source semantics.

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 instantiates the semantics and interpreter part of this model 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.

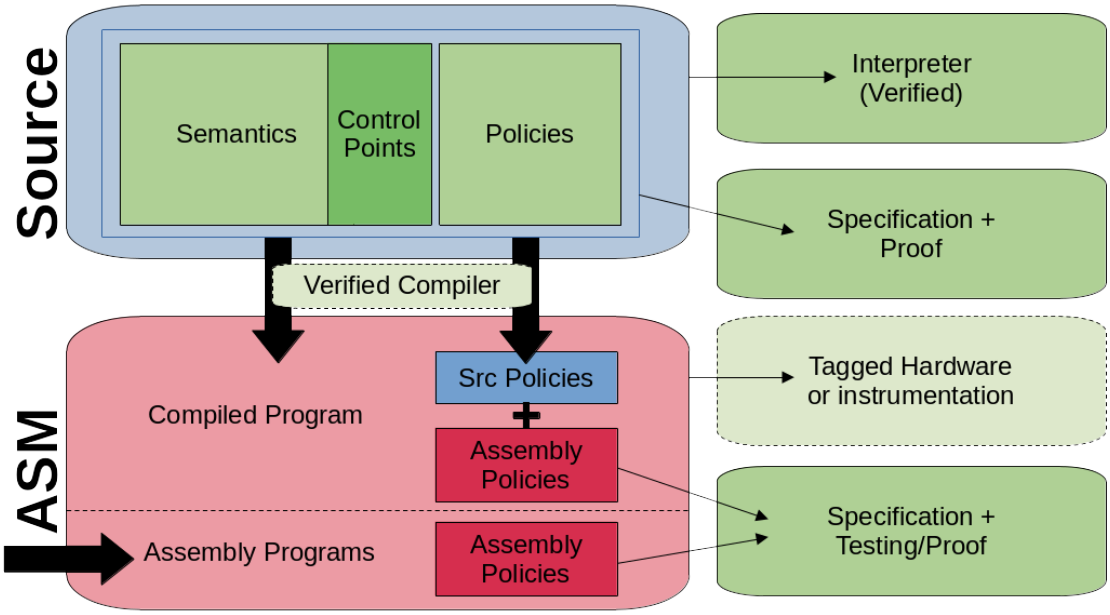


Figure 1.3: Tagging with Source Language

1.2 Overview

This dissertation is divided into three main parts. The first proposes a new formal characterization of stack safety using concepts from language-based security. Stack safety exemplifies the challenges of specifying a policy: "the stack" is not a clearly defined language concept, but a loosely defined component of a system's ABI that is relied on by many different higher-level abstractions. Performance tradeoffs are relevant as well: the "lazy" stack safety policies studied by Roessler and DeHon [53] 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. No prior characterization of stack safety captures this style of safety.

The second part presents Tagged C, a *source-level* specification framework that

allows engineers to describe policies in terms of familiar C-level concepts. Tagged C addresses the challenges in definition, specification, and validation that relate to assembly-level programs. It takes the form of a variant C language whose semantics is parameterized by tags attached to its data and rules that 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.

Tagged C allows policies to be defined at the source level via a fixed interface that never requires rewriting code. 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 the control points might be compiled, and need not deal with portions of a policy that would be colored red in Figure 1.1.

The current iteration of Tagged C is implemented as an interpreter, based on that of CompCert C [42]. This is sufficient to test small programs. Ultimately Tagged C will 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.

The final third of this dissertation makes use of Tagged C to perform a source-

level specification and verification of a novel compartmentalization property. The specification takes the form of an abstract semantics that is compartmentalized by construction. This compartmentalized semantics is written to keep compartments' local data isolated entirely in separate address spaces. Both the specification and the policy that enforces it 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 proof is mechanized. 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 sequence.

1.2.1 Contributions and Organization

This dissertation is structured as follows. Chapter 2 introduces the concept of tag-based reference monitors and brings the reader up to date on the state-of-the-art in that and related areas. The next three chapters cover each of the three main topics, listed again here with their associated contributions enumerated.

Stack Safety Chapter 3 gives a novel formalization of stack safety in the form of a collection of trace properties. This is a novel specification for an important kind of assembly-level security. Our 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; we find that it falls short, and propose and validate a fix.

This chapter was first published at the IEEE Computer Security Foundations Symposium, July 2023 as "Formalizing Stack Safety as a Security Policy," a joint work with Roberto Blanco, Leonidas Lampropoulos, Benjamin Pierce, and Andrew Tolmach [3].

Tagged C In Chapter 4, we attack the challenges in defining and validating policies by lifting tagged enforcement to the level of C source code. We introduce Tagged C, a C variant whose semantics are parameterized by an arbitrary tag-based policy. Our 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.

The core of this chapter was first published at the International Conference on Runtime Verification, October 2023 as "Flexible Runtime Security Enforcement with Tagged C," a joint work with Andrew Tolmach and Allison Naaktgeboren [2]. Some technical details are also published in Chhak et al. [], a joint work with CHR Chhak and Andrew Tolmach. The original content has been updated to reflect further development, and the chapter has been extended with a detailed discussion of the design decisions that inform the current development.

Compartmentalization Finally, I put Tagged C's capabilities to work by defining, specifying and validating a compartmentalization policy at the C level. Chapter 5 presents a novel compartmentalization policy in conjunction with the abstract compartmentalization scheme that it enforces, and proves that the policy indeed enforces the abstract model. My compartmentalization scheme supports memory sharing between compartments, which few existing formalizations do, and it places fewer constraints on potential hardware implementations than the existing models that do exist.

The detailed 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 Tags and Monitors

3 Formalizing Stack Safety as a Security Policy

3.1 Introduction

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.

From a security perspective, attacks on the call stack are attacks on the function abstraction itself. Indeed, the stack is an ancient [52] and perennial [18, 35, 48, 49, 60, 62] target for low-level attacks, sometimes involving control-flow hijacking via corrupting the return address, sometimes memory corruption more generally.

The variety in 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, 50, 51], split stacks [38], shadow stacks [22, 57], capabilities [17, 30, 58, 59, 65], and hardware tagging [31, 54]. 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, 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" [57], not to categorically rule out attacks. Other mechanisms, such as SoftBound [50] and code-pointer integrity [38], 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. [59] and Georges et al. [30].

Some of the other mechanisms listed above should also be amenable to strong formal guarantees. In particular, Roessler and DeHon [54] present an array of tag-based micro-policies [7] for stack safety that aim to offer universal protection. But the reasoning involved can be subtle: they include micro-policy optimizations, Lazy Tagging and Lazy Clearing (likely to be deployed together, 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, avoiding expensive clearings of the stack frame. 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). 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 [56] 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. [59].

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.[1] 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. 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-

[1] Contrast Azevedo de Amorim et al.'s work on heap safety [9], 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, 39], 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, 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 3.3 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 3.4 formalizes the machine model, its *security semantics*, and the stack safety properties built on these. Section 3.5 describes how to support an extended set of features. Section 3.7 describes the micro-policies that we test, Section 3.8 the testing framework itself, and Section 3.9 and **??** related and future work.

The accompanying artifact [2] 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.

3.2 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

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

machine as *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 function that characterizes how the operation's underling 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 3.3 and formally in Section 3.4. 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, 31] or capabilities [65], we assume that the behavior of these mechanisms is incorporated into the basic step semantics of the machine, with a notion of "compatible" states that share security behavior that may be defined based on the 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 compilers 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—necessary 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 will 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 in this paper is given in Table 3.1. A core set of operations covering calls, returns, and local memory is introduced in the example in Section 3.3 and formalized in Section 3.4. An extended set covering simple memory sharing and tail-call elimination is described in Section 3.5 and tested in Section 3.8. The remaining operations are needed for the capability-based model in Section 3.6.

Operation $\psi \in \Psi$	Parameters	Sections
call	target address, argument registers	3.3,3.4
	stack arguments (base, offset & size)	3.5,3.8
return		3.3,3.4
alloc	offset & size	3.3,3.4
	public flag	3.5,3.8
dealloc	offset & size	3.3,3.4
tailcall	(same as for **call**)	3.5,3.8
promote	register, offset & size	3.6
propagate	source register/address	3.6
	destination register/address	3.6
clear	target register/address	3.6

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.

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 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.

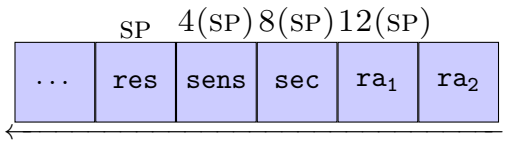
3.3 Properties by Example

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 [20]. 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 Fig. 3.2. Leakage is most obviously viewed as a violation of `main`'s *confidentiality*. In Fig. 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 Fig. 3.2b. Beyond simply

```
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.

reading `secret`, the attacker might overwrite `sensitive` with 42, guaranteeing that `main` publishes its own secret unintentionally (Fig. 3.2c); this does not violate `main`'s confidentiality, but rather its *integrity*. In Fig. 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 Fig. 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 Fig. 3.1 and **??**, 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 3.4, 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

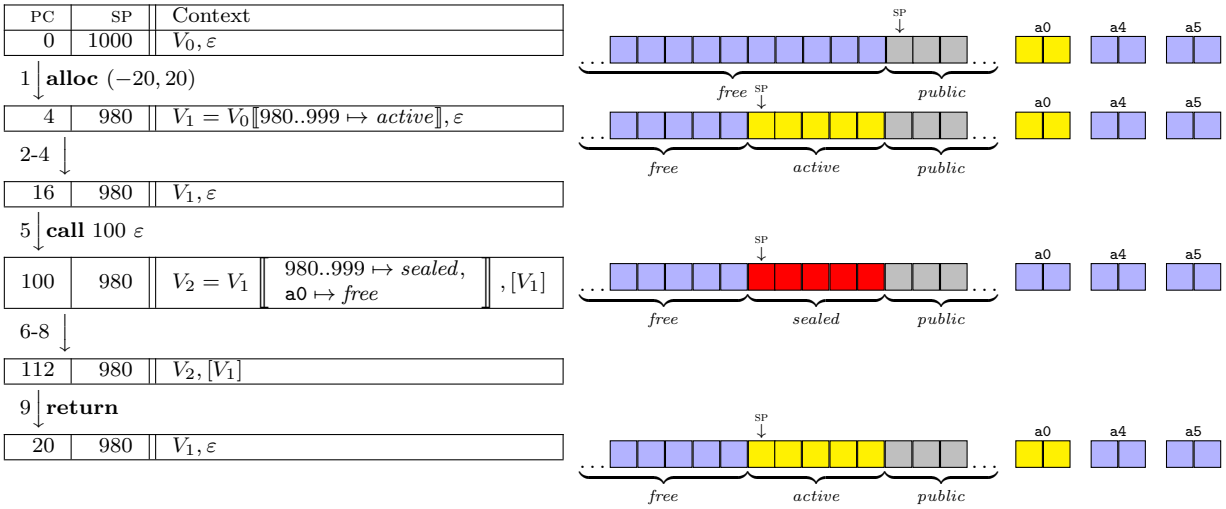


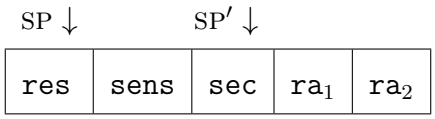
Figure 3.3: Execution of example up through the return from `f`. In stack diagrams, addresses increase to the right, stack grows to the left, and boxes represent 4-byte words.

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 `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 Fig. 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 Fig. 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 Fig. 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 Fig. 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*.[3]

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 Fig. 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 state will eventually output

[3]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 3.4.4.

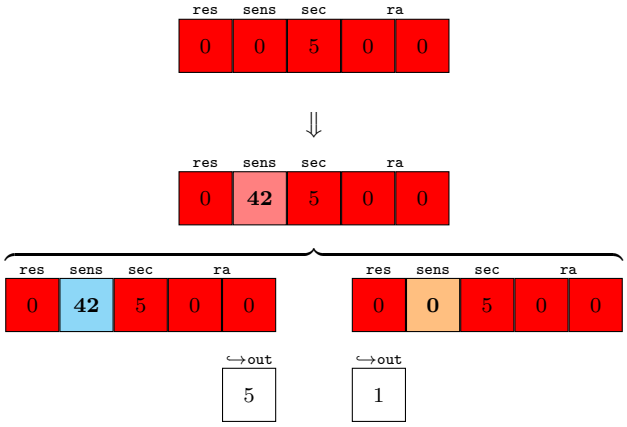


Figure 3.4: Integrity Violation: `sensitive` changed, and if varied, changes future outputs

1, while the actual execution outputs 5. This means that `sensitive` is relevant.

To be more explicit, similar to WBCF, we define *Int* as a predicate on states that holds if all relevant sealed addresses in m are the same as after step 5. We require that *Int* hold on the state following the matching return, which is reached by step 9. Here `sensitive` has obviously changed, but we just saw that it is relevant.

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 Fig. 3.5, with the attacker

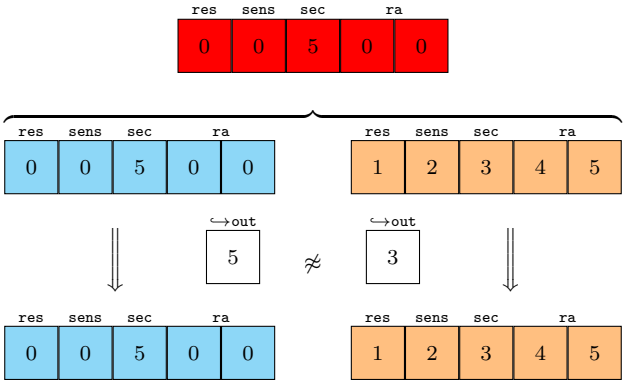


Figure 3.5: Internal Confidentiality Violation

code from Fig. 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 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 Fig. 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 attacker, 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

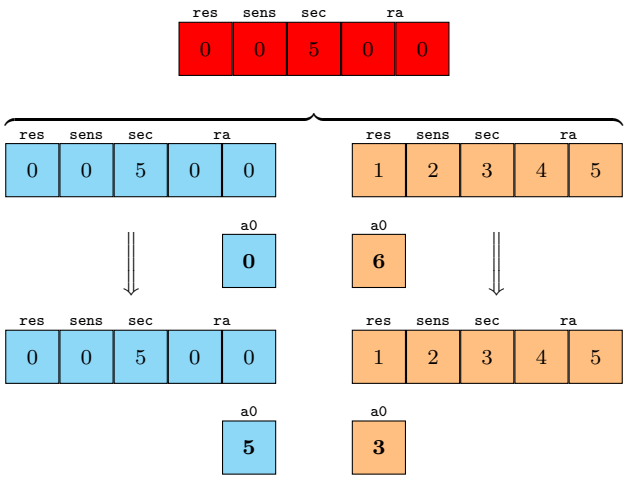


Figure 3.6: Return-time Confidentiality Violation

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 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 is not typically discussed.

3.4 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.

3.4.1 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 $\mathcal{K} ::= \text{PC} | \mathcal{W} | \mathcal{R}$.

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 3.4.4, 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.

3.4.2 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.

$$c \in C ::= \mathit{VIEW} \times \mathit{list}\ \mathit{VIEW}$$

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

Section 3.3 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*.

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

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 Fig. 3.7. Note that Op takes as its first argument the state *before* the step.

3.4.3 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

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

$$\frac{\begin{array}{c} K = \mathit{range}\ \textsc{sp}\ \mathit{off}\ \mathit{sz}\ m \cap \mathit{free}(V) \\ V' = V[\![a \mapsto \mathit{active} \mid a \in K]\!] \end{array}}{\mathit{Op}\ m\ (\textbf{alloc}\ \mathit{off}, \mathit{sz})\ (V, \sigma) = (V', \sigma)}\ \textsc{Alloc}$$

$$\frac{\begin{array}{c} K = \mathit{range}\ \textsc{sp}\ \mathit{off}\ \mathit{sz}\ m \cap \mathit{active}(V) \\ V' = V[\![a \mapsto \mathit{free} \mid a \in K]\!] \end{array}}{\mathit{Op}\ m\ (\textbf{dealloc}\ \mathit{off}, \mathit{sz})\ (V, \sigma) = (V', \sigma)}\ \textsc{Dealloc}$$

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

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

Figure 3.7: Basic Operations

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 these rules:

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

$$\frac{|\sigma| \geq d \quad d \hookrightarrow (m', c') = \mathcal{E} \quad (m, (V, \sigma)) \overset{\psi, e}{\Longrightarrow} (m', c')}{d \hookrightarrow (m, (V, \sigma)) = e \cdot \mathcal{E}}\ \textsc{Step}$$

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. 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.

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

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

3.4.4 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 $(\Delta(m, m') \cup \Delta(n, n')) \cap \Delta(m', n')$.

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 $d \uparrow P$, 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.

$$\frac{|\sigma| < d \qquad P\ m}{(d \uparrow P)\ (m, (V, \sigma))}\ \textsc{Returned}$$

$$\frac{\begin{array}{c}|\sigma| \geq d \quad (d \uparrow P)\ (m', c') \\ (m, (V, \sigma)) \overset{\overline{\psi}, e}{\Longrightarrow} (m', c')\end{array}}{(d \uparrow P)\ (m, (V, \sigma))}\ \textsc{Step}$$

Similarly, we give a analogous binary relation for use in confidentiality. We define $\Uparrow$ 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.

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

$$\frac{\begin{array}{c}|\sigma_1| \geq d \quad (m_1, (V_1, \sigma_1)) \overset{\overline{\psi}, e}{\Longrightarrow} (m_1', c_1') \\ (m_1', c_1')\ (d \Uparrow R)\ (m_2, (V_2, \sigma_2))\end{array}}{(m_1, (V_1, \sigma_1))\ (d \Uparrow R)\ (m_2, (V_2, \sigma_2))}\ \textsc{Left}$$

$$\frac{\begin{array}{c}|\sigma_2| \geq d \quad (m_2, (V_2, \sigma_2)) \overset{\overline{\psi}, e}{\Longrightarrow} (m_2', c_2') \\ (m_1, (V_1, \sigma_1))\ (d \Uparrow R)\ (m_2', c_2')\end{array}}{(m_1, (V_1, \sigma_1))\ (d \Uparrow R)\ (m_2, (V_2, \sigma_2))}\ \textsc{Right}$$

1	WBCF $\triangleq(\lvert\sigma'\rvert \uparrow \mathit{Ret})\ (m', (V', \sigma'))$ for all calls $(m, (V, \sigma)) \Longrightarrow (m', (V', \sigma'))$	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 $m[\text{PC}]$
2	CLRI $\triangleq(\lvert\sigma\rvert \uparrow \mathit{Int})\ (m, (V, \sigma))$ for all call targets $(m, (V, \sigma))$	where $\mathit{Int}\ m' \triangleq m' \parallel (\mathit{sealed}(V) \cap \Delta(m, m'))$
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 $(m, (V, \sigma))\ (\lvert\sigma\rvert \Uparrow \mathit{Conf})\ (n, (V, \sigma))$ for all call targets $(m, (V, \sigma))$	where $(m'\ \mathit{Conf}\ n') \triangleq m' \parallel \bar{\Diamond}(m, n, m', n')$
4	CLEC $\triangleq(\lvert\sigma\rvert \uparrow \mathit{CConf})\ (m, (V, \sigma))$ for all call targets $(m, (V, \sigma))$	where $\mathit{CConf}\ m' \triangleq m' \parallel (\Delta(m, m') - (\mathit{public}(V) \cup \mathit{active}(V))$
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 $(m, (V, \sigma))\ (\lvert\sigma\rvert \Uparrow \mathit{CInt})\ (n, (V, \sigma))$ for all call targets $(m, (V, \sigma))$	where $(m'\ \mathit{CInt}\ n') \triangleq m' \parallel \bar{\Diamond}(m, n, m', n')$

Table 3.2: Properties

3.4.5 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 $s' \xRightarrow{\bar{\psi}} s$ 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 all elements that are both sealed in V and in the difference set between

m and m' are 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.

3.5 Extended Code Features

The system we model in Section 3.3 and **??** is very simple, but our framework is designed to make it easy to add support for additional 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 Fig. 3.8; the rules in Fig. 3.7 can be recaptured by instantiating **call** with $\overline{sa}$ as the empty set, and **alloc** with flag **f**.

3.5.1 Sharing Stack Memory

In our examples, we have presented 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 (Fig. 3.8b).

Using this mechanism, a call-by-value argument passed on the stack at an SP-relative offset is specified by the triple (SP, *off*, *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 $(r, 0, sz)$. Such a call-by-reference argument could be passed through multiple calls, provided that it is in $\overline{sa}$ each time.

Absent the more sophisticated capability model (below), if the address of an object is taken directly and passed as a pointer, we simply classify the object as "public" 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 (Fig. 3.8a). Providing more fine-grained control over sharing is desirable, but requires a considerably more complex model. This simple model is included in our testing; we describe an untested approach based on capabilities below.

3.5.2 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.

3.6 Provenance, Capabilities, and Protecting Objects

Lastly, 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 [44] (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. This constitutes a substantial addition to the security context, which is why this enhancement

is more speculative than the others, and we have not tested it.

We can model the provenance model as a trio of additional security-relevant operations: one which declares a register to contain a valid pointer, one which transmits the provenance of a pointer from one element to another, and one 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 Fig. 3.9.

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.

3.7 Enforcement

We implement and test two micro-policies inspired by Roessler and DeHon [54]: *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, and then clear the frame before returning. During normal execution, the micro-policy rules only permit 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.

In LTC, a function neither initializes the frame at entry nor clears it at exist; 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 store will be tagged STACK n. On a 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.

To implement this discipline, *blessed instruction sequences* 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.

Applicability to Roessler & DeHon [54] Roessler and DeHon (henceforward *R&D*) 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.

They 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. 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.

3.8 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 Per-Activation Tagging and Clearing* micro-policies.

We use a Coq specification of the RISC-V architecture [13], extend it with a run-time monitor implementing a stack safety micro-policy, and test it using QuickChick [39], 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.

3.8.1 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 3.7), including a code region containing a low-level program. They also produce the meta-information about how instructions in that program map to security-relevant operations, which would normally be provided by the compiler.

Our generators build on the work of Hriţcu et al. [33, 34], 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,

3.8.2 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 execution that combines all of the variants that would be spawned at returns. So, at any given time, we need only simulate (1) the original execution, (2) the tainted execution, and (3) one variant execution for each call on the call stack. This approach makes testing longer executions substantially faster, at the cost of making it harder to identify which call

is the source of a failure.

3.8.3 Mutation Testing

To ensure the effectiveness of testing against our formal properties, we use *mutation testing* [36] 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 additional 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*.

For LTC, the original micro-policy, implemented as *PER_DEPTH_TAG*, fails in

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)

testing, in cases where 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 fails to propagate the PC tag, *STORE_NO_UPDATE*.

The mean-time-to-failure (MTTF) and average number of tests for various bugs can be found in Table 3.3, along with the average number of tests it took to find the failure. Experiments were run in a desktop machine equipped with i7-4790K CPU @ 4.0GHz with 32GB RAM.

3.9 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, 30, 31, 38,50,51,54,57–59,65]. Many enforcement techniques focus purely on WBCF; others combine this with some degree of memory protection, chiefly focusing on integrity. Roessler and DeHon's *Depth Isolation* and *Lazy Tagging and Clearing* [54] 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 rigorously characterize the security of the stack is the StkTokens-Cerise family of CHERI-enforced secure calling conventions [30, 58, 59]. The authors define stack safety as overlay semantics and related stack safety properties, phrased in terms of logical relations instead of trace properties. Originally, they define an informal notion of stack safety as the combination of WBCF and "local state encapsulation" [59], and describe the latter in terms of integrity only (but it has confidentiality, equivalent to ClrI *and* ClrC). StkTokens [59] makes this conception of stack safety explicit through an overlay semantics which (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. The underlying unary logical relation does not capture confidentiality proper, although it does capture some of its facets.

Their latest paper [30] was inspired by the properties presented in this paper 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.[4] To do so, they redesign the overlay semantics to actually pop stack frames on return and have them disappear from the stack. This demonstrates the benefit of our choice to explicitly state properties in security

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

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. [30] (the most recent work in the line), 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, their properties are validated by proof, while ours are only tested.

3.10 Future Work

We plan to test our properties against multiple enforcement mechanisms. The top priority is capability machines, namely CHERI [63], a modern architecture designed to provide efficient fine-grained memory protection and compartmentalization. We want to test the most recent work by Georges et al. [30], which is designed to enforce analogues of all of our properties except for CLEI.

It would also be interesting to test a software enforcement approach. Under a bounds checking discipline [50], all the pointers in a program are extended with

some disjoint metadata used to gate memory accesses. These approaches enforce a form of *memory safety*, and we would therefore expect them to enforce ClrI and ClrC. They aim to enforce WBCF by cutting off attacks that involve memory-safety violations, but that may not be sufficient. Bounds checking approaches require substantial compiler cooperation. This is not a problem for our properties in general, but it is not very compatible with generation-by-execution of low-level code. A better choice might be to generate high-level code using a tool like CSmith [66], or prove the properties instead.

Several popular enforcement mechanisms are not designed to provide absolute guarantees of security. For example, stack canaries [21] and shadow stacks [22, 57] are chiefly hardening techniques: they increase the difficulty of some control-flow attacks on the stack, but cannot provide absolute guarantees on WBCF under a normal attacker model. Interestingly, these are lazy enforcement mechanisms, in that the attack may occur and be detected some time later, as long as it is detected before it can become dangerous. That would make our observation-based formalism a good fit for defining their security, if we could find a formal characterization of what they do acheive (perhaps in terms of a base machine with restricted addressing power).

We have preliminary work on extending our model to handle C++-style exceptions, which, like tailcalls, obey only a weakened version of WBCF. We are also exploring extensions to concurrency, starting with a model of statically allocated coroutines. These extensions will also require non-trivial testing effort. We also plan to test the model in Section 3.6 for arbitrary memory-safe pointer sharing.

Acknowledgements

We thank the reviewers for their comments, CHR Chhak and Allison Naaktgeboren for feedback during the writing process, and Aïna Linn Georges for significant technical feedback and encouragement.

This work was supported by the National Science Foundation under Grant No. 2048499, Specifying and Verifying Secure Compilation of C Code to Tagged Hardware; by ERC Starting Grant SECOMP (715753), Efficient Formally Secure Compilers to a Tagged Architecture; by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Excellence Strategy of the German Federal and State Governments, EXC 2092 CASA – 390781972; by NSF award #2145649, CAREER: Fuzzing Formal Specifications, by NSF award #1955610, Bringing Python Up To Speed, and by NSF award #1521523, Expeditions in Computing: The Science of Deep Specification.

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

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

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

(a) Memory Allocation

$$\mathit{push}\ V\ \overline{r}\ K \triangleq \lambda k. \begin{cases} \mathit{free} & \text{if } k \in \mathit{CLR} \\ \mathit{public} & \text{if } k \in \overline{r_{args}} \\ \mathit{sealed} & \text{if } k \in \mathit{VOp}\ m\ (\mathbf{call}\ a_{target}\ \overline{r_{args}}\ \overline{sa})\ (V, \sigma) = (V', V :: \sigma) \\ & k \in \mathit{active}(V) - K \\ V(k) & \text{otherwise} \end{cases}$$

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

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

$$\mathit{passed}\ \overline{sa}\ m \triangleq \bigcup_{(r, \mathit{off}, \mathit{sz}) \in \overline{sa}} \mathit{range}\ r\ \mathit{off}\ \mathit{sz}\ m$$

(b) Calls with Argument Passing on the Stack

Figure 3.8: Operations supporting tail calls and argument passing on stack.

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

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

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

$$\mathit{reach}\ k\ \rho \triangleq \{k' | \mathit{base} \leq k' < \mathit{bound} \text{ where } \rho[k] = (\mathit{base}, \mathit{bound})\}$$

$$\mathit{reach}^*\ K\ \rho \triangleq \bigcup_{k \in K} \{k\} \cup \mathit{reach}^*\ (\mathit{reach}\ k\ \rho)\ \rho$$

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

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

Figure 3.9: Operations supporting provenance-based protection of passed objects

4 Flexible Runtime Security Enforcement with Tagged C

4.1 Introduction

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), logic errors (e.g. SQL injection, input-sanitization flaws), or larger-scale architectural flaws (e.g. over-provisioning access rights).

Although static analyses can detect and mitigate many C insecurities, a last line of defense against undetected or unfixable vulnerabilities is runtime enforcement of *security policies* using a reference monitor [1]. In particular, many useful policies can be specified in terms of flow constraints on *metadata tags*, which augment the underlying data with information like type, provenance, ownership, or security classification. A tag-based policy takes the form of a set of rules that check and update the metadata tags at key points during execution; if a rule violation is encountered, the program *failstops*. Although monitoring based on metadata tags is less flexible and powerful than monitoring based on the underlying data values, it can still enforce many useful security properties, including both low-level concerns such as memory

safety and high-level properties such as secure information flow [24] or mandatory access control [40].

Tag-based policies are especially well-suited for efficient hardware enforcement, using processor extensions such as ARM MTE [4], STAR [31], and PIPE. PIPE[1] (Processor Interlocks for Policy Enforcement) [6, 8],the specific motivator for our work, is a programmable hardware mechanism that supports monitoring at the granularity of individual instructions. Each value in memory and registers is extended with a metadata tag. Before executing each instruction, PIPE checks the opcode and the tags on its operands to see if the operation should be permitted according to a tag rule, and if so, what tags should be assigned to the result. PIPE is highly flexible: it supports arbitrary software-defined tag rules over large (word-sized) tags with arbitrary structure, which enables fine-grained policies and composition of multiple policies. This flexibility is useful because security needs may differ among codebases, and even within a codebase. A conservative, one-size-fits-all policy might be too strong, causing failstops during normal execution. Sensitive code might call for specialized protection.

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, and 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.

[1]Variants of PIPE have been called PUMP [27] or SDMP [53]and marketed commercially under the names Dover CoreGuard and Draper Inherently Secure Processor.

To address these problems, we 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 take the form of manipulating tags, not data. In previous work on the Tagine project [16], we outlined such a framework for a toy source language and showed how high-level policies could be compiled to ISA-level policies and enforced using PIPE-like hardware. Here we extend this approach to handle the full, real C language, by giving a detailed design for the necessary control points and showing how they are integrated into C's dynamic semantics. Although motivated by PIPE, Tagged C is not tied to any particular enforcement mechanism. We currently implement it using a modified C interpreter rather than a compiler. We validate the design of Tagged C by using it to specify a range of interesting security policies, including compartmentalization, memory protection, and secure information flow.

Formally, Tagged C is defined as a variant C semantics that instruments ordinary execution with control points. At each control point, a user-defined set of tag rules is consulted 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, dynamically-allocated objects, and address-taken objects are allocated in the same integer-addressed memory space. Memory behaviors that would be undefined in

standard C are given a definition consistent with the behavior of a typical compiler. We build the Tagged C semantics on top of the CompCert C semantics, which is formalized as part of the CompCert verified compiler [42, 43]. For prototyping and executing example policies, we provide a reference interpreter [2], also based on that of CompCert, written in the Gallina functional language of the Coq Proof Assistant [11]. 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. We have validated this design on the three classes of policies explored in this paper, and, outside of a few known limitations related to `malloc` (**??**), we believe it is sufficiently expressive to describe most other flow-based policies, although further experience is needed to confirm this.

Contributions In summary, we offer the following contributions:

- 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, including a novel compartmentalization policy with separate public and private memory; (2) memory

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

safety, with realistic memory models that support varying kinds of low-level idioms; and (3) secure information flow.

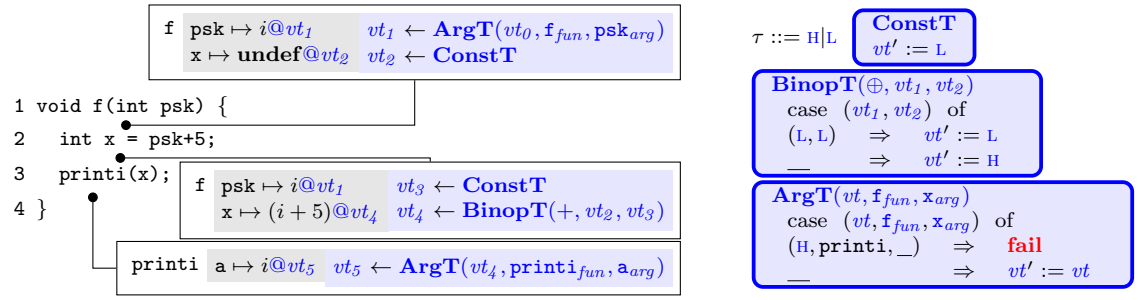
- A full formal semantic definition for Tagged C, formalized in Coq, describing how the control points interact with programs, and an interpreter, implemented and verified against the semantics in Coq and extracted to OCaml.

The paper is organized as follows. **??** gives a high-level introduction of metadata tagging by example. **??** summarizes the Tagged C language as a whole and its control points. **??** describes three example policies and how their needs inform our choices of control points. **??** describes the Coq-based implementation of Tagged C. **??** discusses related work, and **??** describes future work.

4.2 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 **??** 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`. We now explain how a monitor specified in Tagged C could detect such a leak. (Of course, this particular leak could also be easily found using static analysis.)

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



(a) Example execution showing tag rules and propagation. (b) SIF instantiation (pt. 1)

Figure 4.1: Tag Rules and Instantiation

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 program-specific argument-handling rule for `printi` will check that the tag is L; since it is not, the rule will cause a failstop.

To explain the mechanics of Tagged C, we first show in **??** the policy-independent framework under which tag rules are triggered in this program: the initial tag on `psk` (vt_0) passes through the **ArgT** tag rule, is combined with the tag on the constant 5 via the **BinopT** tag rule, and then is passed to **ArgT** again on the call to obtain the tag on the parameter inside `printi` (here called `a`). **??** 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. Throughout the paper, we highlight tag-related metavariables, rules, etc. in blue. We write $v@vt$ for value v tagged with vt. Tags that are derived from identifiers are subscripted with the identifier namespace, e.g. $\texttt{f}_{fun}$ is the tag associated with 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 **??**. The resulting behavior is best understood by mentally "weaving" together the two figures. Suppose `f` is called with an argument value i@H. This first invocation of **ArgT** simply passes the H tag on to the output vt_1, because the name of the function being called does not match `printi`. In tag rules, the assignment operator := denotes an assignment to the named tag-rule output, by convention written as primed metavariables t'. The initial tag vt_2 on local variable `x` and the tag vt_3 on the constant `5` come from **ConstT**, which tags all constants as L. 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, upon entry into `printi`, **ArgT** is invoked again; this time it failstops. Note that in this policy, **ArgT** is code-specific (it checks for a particular function name `printi`) whereas the other rules are generic.[3]

As a second example, **??** steps through the execution of a function `g` that adds two new wrinkles: we need to keep track of metadata associated with addresses and with the program's control-flow state. We 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,

[3]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, and **CallT**, which is triggered by the call itself.

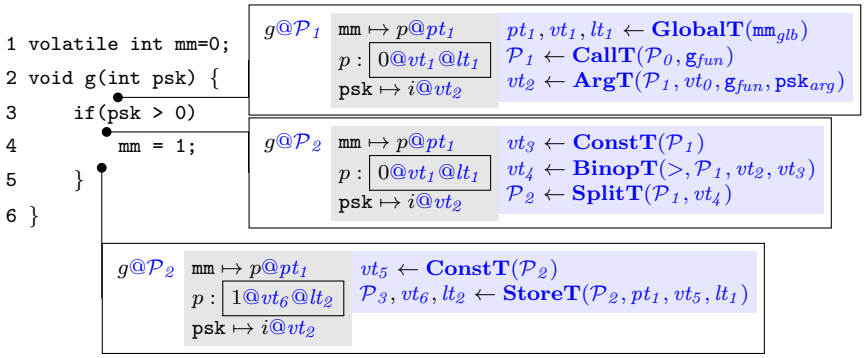


Figure 4.2: Second example showing tag rules and tag propagation.

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]).

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 our diagrams). Tagged C initializes the tags on `mm` with the **GlobalT** rule. The PC tag at the point of call, $\mathcal{P}_0$, is fed to **CallT** to determine a new PC tag inside of `g`. And 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.

?? shows an instantiation of these rules that extend our previous SIF policy. 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

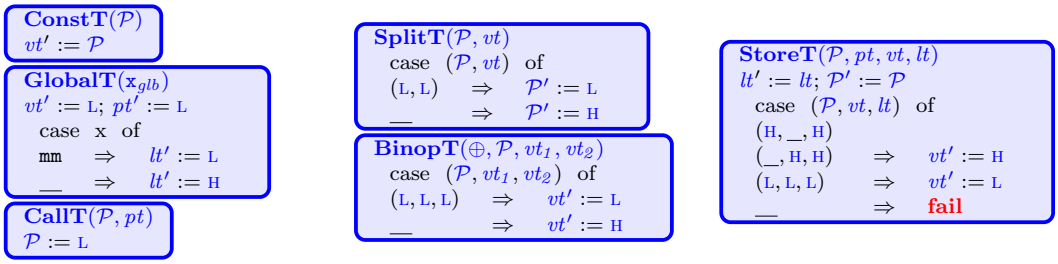


Figure 4.3: Tag rule instantiations for secure information flow (pt. 2)

higher of the PC tag and the relevant value tag(s). This is shown for the updated **BinopT** in **??**; the **ArgT** rule needs a similar adjustment. 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.

4.3 The Tagged C Language: Syntax and Semantics

Tagged C contains almost all features of full ISO C 99.[4] Its semantics is based on that of CompCert C [42], 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: tag support and memory model.

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

[4]It inherits the limitations of CompCert C, primarily that `setjump` and `longjump` may not work, and variable-length arrays are not supported.

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, which we keep 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, consider the expression step reduction for binary operations:

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

On the left, the ordinary "tagless" version of the rule reduces a binary operation on two inputs to a single value. On the right, the Tagged C version adds tags to the operands and tags the result based on the **BinopT** 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 listed in Table 4.1. In these signatures, we use the metavariable $\mathcal{P}$ to denote the PC tag, pt for tags that will be attached to pointer values, vt for tags that will be attached to values in general, and lt for tags that are associated with specific addresses in memory. We also range over different classes of identifiers with the metavariables: $\texttt{f}_{fun}$, function identifiers; $\texttt{x}_{arg}$, function arguments; $\texttt{x}_{glb}$, global variable names; $\texttt{L}_{lbl}$, labels; and $\texttt{ty}_{typ}$, types. We briefly summarize the rules below, and give motivating examples of their use in Section 4.4.

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. Pointers are described by a (block, offset) pair, and invalid pointer accesses (out-of-bounds, use after free, etc.) produce UB. Tagged C instead separates variables into public and private data. Public data (all heap data, globals, arrays, structs, and address-taken locals) share a single flat address space (possibly with holes), and pointers are offsets into this space. Pointer accesses outside of this space cause an explicit failstop, rather than UB. Private data (non-address-taken locals, parameters) live in a separate, abstract environment. Program-specified stores, e.g. writes through pointers, can cause arbitrary damage to public data, but do not affect private data. 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 4.4.1). In an implementation that compiles to PIPE, the private data can be protected by a small number of "built-in" tags.

Control Points In our 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 (**ConstT** 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 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.

CallT and **RetT** update the PC tag on entry and exit from a function. **CallT** is parameterized by the function pointer being called and the PC tag. **RetT** is parameterized by both the caller's PC tag ($\mathcal{P}_{CLR}$) and the callee's ($\mathcal{P}_{CLE}$); it can also update the return value's tag. **ArgT** updates tags on any arguments, based on the function and argument names.

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 our PNVI memory safety policy (Section 4.4.1), and more generally policies that keep track of the correspondence between pointers and their targets.

There is always the chance that new policies might arise for which our 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.

Combining Policies Multiple policies can be enforced in parallel. If policy A has tag type τ_A and policy B has τ_B, then policy $A \times B$ should have tag type $\tau = \tau_A \times \tau_B$. Its tag rules should apply the rules of A to the left projection of all inputs and the rules of B to the right projection to generate the components of the new tag. If either side failstops, the entire rule should failstop.

This process can be applied to any number of different policies, allowing, for instance, a combination of a baseline memory safety policy with several more targeted information-flow policies. Alternatively, a policy can delegate to tag rules from other, related policies, as illustrated in Section 4.4.2, below.

4.4 Example Policies

In this section, we discuss 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.

4.4.1 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. Here we discuss the CompCert C memory model and two models proposed by Memarian et al. [45] for the purposes of supporting low-level idioms in the presence of compiler optimization, focusing in particular on how they handle casts from pointers to integers and back.

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 Fig. 4.4. 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 [46]. 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 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 idioms that PVI cannot.

Implementation The basic idea for enforcing any of the above memory safety variants is a "lock and key" approach [8, 19]. When an object is allocated, it is assigned a unique "color," and its memory locations as well as its pointer are tagged with that color, written $\textsc{clr}(c)$. The default tag $\textsc{n}$ indicates a non-pointer or non-allocated location. The PC tag will also be $\textsc{clr}(c)$, tracking the next available color for new allocations. These rules are given in Fig. 4.5 and **??**. Operations that are valid on pointers in a given memory model maintain the pointer's color, and loads

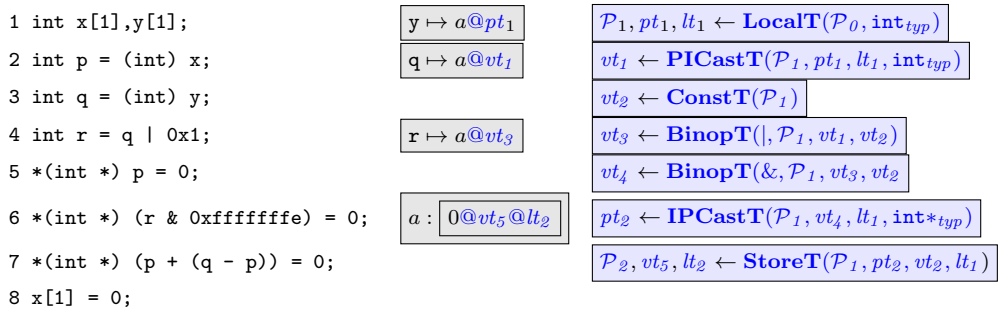


Figure 4.4: Memory safety and pointer casts, tracing `y`, `q`, and `r`. (Assume `int` and pointers are 32 bits.) Line (5) is always legal, (6) is illegal in CompCert C due to bitwise arithmetic not preserving provenance, (7) is also illegal in PVI due to combining provenance of multiple objects, and (8) is illegal in all models.

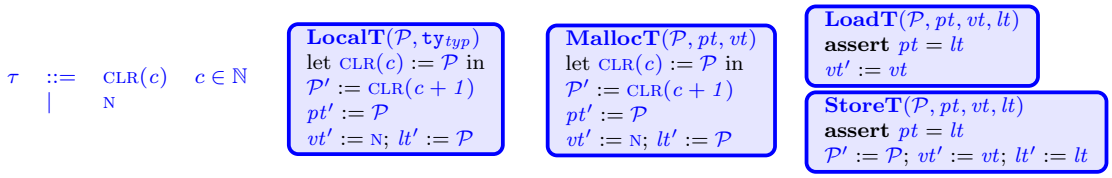
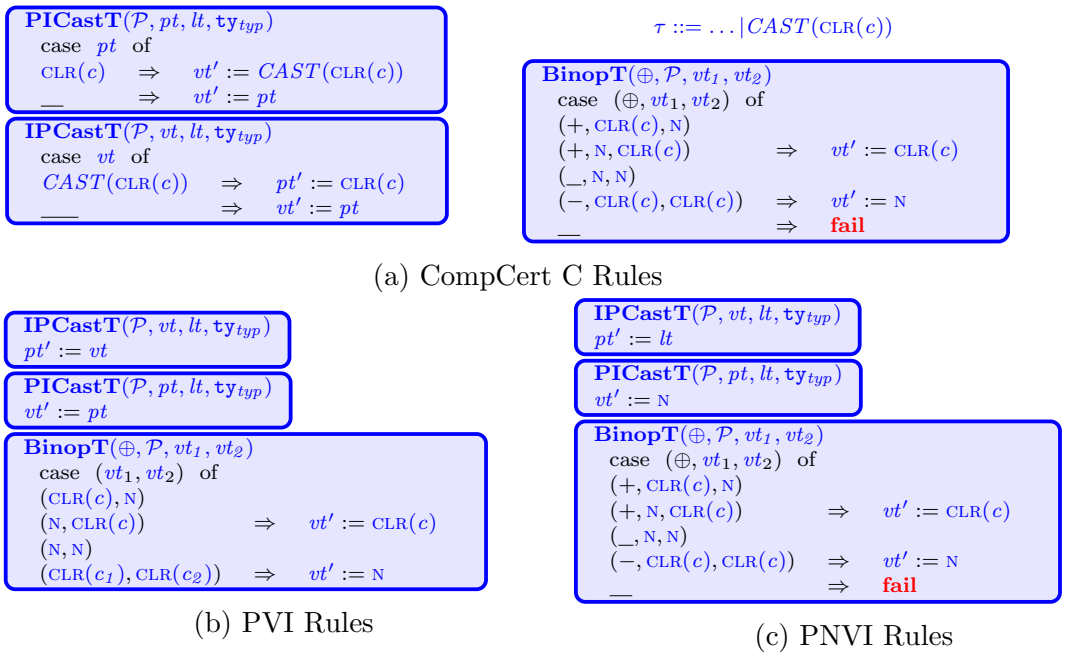


Figure 4.5: Generic Memory Safety Rules

and stores are legal if it matches the target memory location tag. (The `assert` command failstops if its argument does not hold.)

The specific memory models (Fig. 4.6) 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.

Memory safety considerations also inform the design of control points related to allocating, deallocating, and accessing memory. Drawing from CompCert C,



(a) CompCert C Rules

(b) PVI Rules

(c) PNVI Rules

Figure 4.6: Specialized Memory Safety Rules

Tagged C abstracts over "external" functions like `malloc` and `free`, rather than treat their implementation as ordinary code. In a concrete system, these would be replaced by their library equivalents. **ExtCallT** models the desired tag behavior of general external functions in the Tagged C semantics, but **MallocT**$(\mathcal{P}, pt, vt)$ and **FreeT**$(\mathcal{P}, pt, vt)$ are special cases because they also need to retag memory; the location tag lt returned by each of them is copied across the allocated region.

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.

4.4.2 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. But in reality, the cost and risk of regressions may make it undesirable to fix bugs in older code [12]. 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, 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

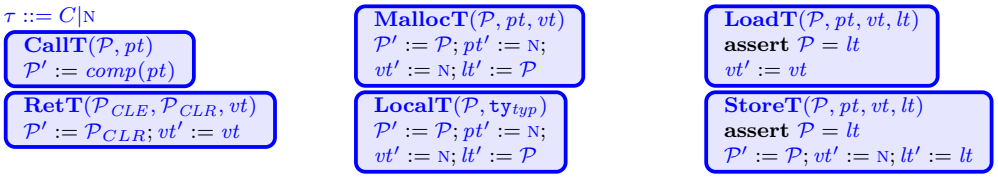


Figure 4.7: Simple Compartmentalization Policy

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 $\mathbf{CallT}(\mathcal{P}, pt)$, where pt is the function pointer's tag, to update the PC tag. On return, in addition to handling the return value (if any), $\mathbf{RetT}(\mathcal{P}_{CLE}, \mathcal{P}_{CLR}, vt)$ 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. 4.7), 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 Fig. 4.7: 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 accessed 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.

Distinguishing shareable memory from memory that is local to a compartment is

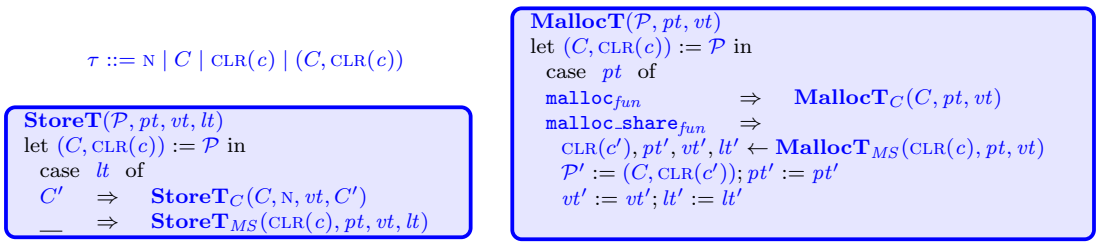


Figure 4.8: Selected rules for Compartmentalization with Shared Capabilities, combining Compartmentalization rules (subscript C, Fig. 4.7) with Memory Safety rules (subscript MS, Fig. 4.5)

difficult without modifying source code. In order to be minimally intrusive, we create a variant identifier for `malloc`, `malloc_share`, which maps to the same address (i.e., it still calls the same function) but has a different name tag and can therefore be used to specialize the tag rule. An engineer might manually select which allocations are shareable, or perhaps rely on some form of escape analysis to detect shareable allocations automatically.

The policy in Fig. 4.8 essentially combines the simple compartmentalization policy and a memory safety policy. The PC tag carries both the current compartment color, for tagging unshared allocations, and the next free color, for tagging shared allocations. **MallocT** applies a color tag to shareable allocations, and N to local ones. During loads and stores, the location tag of the target address determines which parent property applies.

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 [40]. 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.

Malloc Limitations The way Tagged C currently handles `malloc` is unsatisfactory. First, as noted above, there is no easy way to distinguish different static `malloc` call locations; our use of variant names is something of a hack. A more principled solution might draw on pointcuts to identify specific calls to `malloc`. Further, `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. [47]) has often adopted non-standard versions of `malloc` that take more informative parameters. This is not satisfactory for protecting legacy code, but we do not yet see a good alternative.

4.4.3 Secure Information Flow

Finally, we return to the family of *secure information flow* (SIF) [25] policies introduced in Section 4.2. SIF deals with enforcing higher-level security concerns, so it useful even in code with no language-level errors.

In Fig. 4.9, 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 Fig. 4.2, this means that the policy should "raise" the PC tag to H to indicated that

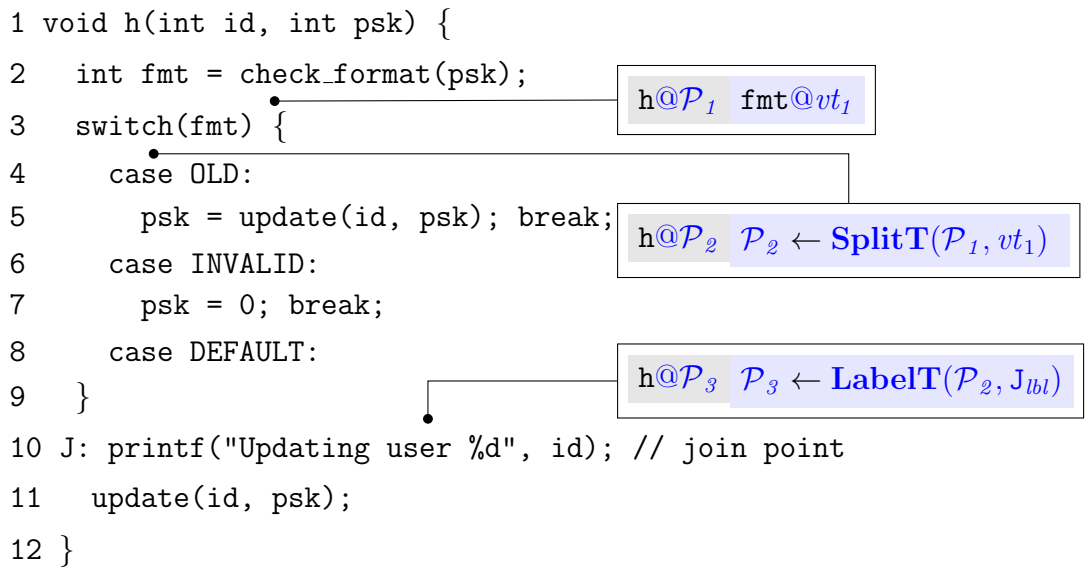


Figure 4.9: Not an Implicit Flow

the program's control-flow depends on `psk`. But 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 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].

In order to support policies that reason about splits and joins, we introduce the **SplitT** and **LabelT** tag rules. 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.) This way, the policy can react when execution reaches that label via the **LabelT** rule.

In the full SIF policy, we keep track of the pending join points within the PC tag,

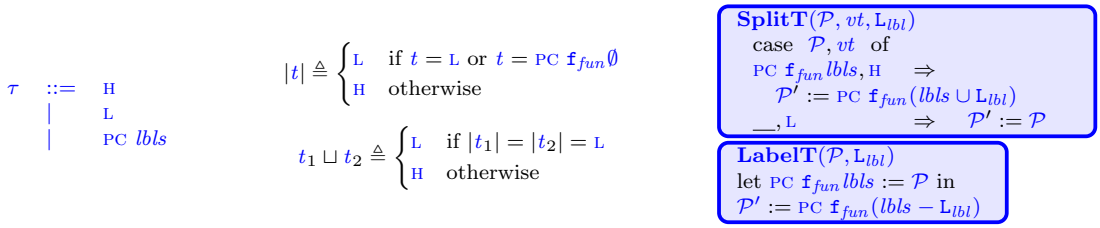


Figure 4.10: SIF Conditionals

and lower the PC tag when execution reaches the join point. (A similar approach applies to conditional expressions, but we omit the details here.) In addition to L and H, the SIF policy tracks a PC tag written using the constructor PC, which carries a set of label identifiers to record the join points of tainted statement scopes. Initially, the PC tag is PC $\mathtt{f}_{fun}\emptyset$, 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 label (optionally, since there might not be a join point if all arms of the conditional execute an explicit `return`). 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 can 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.

4.5 Implementation

Our current Coq-based implementation of Tagged C consists of a formal semantics and a matching interpreter written in Coq's Gallina language (similar to functional languages such as OCaml or Haskell). These are based on the `Csem` and `Cexec` modules from the CompCert compiler (version 3.10) [41]. The interpreter can be extracted to a stand-alone OCaml program that can then be further compiled to machine code.

To adapt CompCert, we replace the standard block-offset memory with a concrete one, leaving the block-based system to handle only function pointers. We rework global environments to separate (symbolic) function pointers from other (concrete) pointers. We also add a temporary environment to contain non-memory variables, and semantic rules to deal with them. Most importantly, we thread the PC tag

into the state, and add control points to the relevant semantic rules. These changes appear in both the semantics and the corresponding interpreter code. To extend the existing CompCert proof that the interpreter is correct with respect to the semantics requires updating the proof automation to handle concrete memory and tags.

The semantics and interpreter are parameterized by a Coq module type `Policy`, which specifies the type signatures of the tag rules. A policy is written directly in Gallina as a module that instantiates `Policy` by defining the type of tags and the body of each tag rule. A full policy fits into 70 lines of Gallina. To illustrate, Fig. 4.11 shows fragments of the `Policy` signature, its instantiation in the PVI memory safety policy, and its use in the Coq semantics.

PIPE, tag sizes, and policy states. Ultimately, we wish to compile Tagged C to run efficiently on PIPE-equipped hardware, which raises some issues that are not currently visible at the level of the Coq interpreter. For example, the Coq model of Tagged C allows unbounded tags. In reality, PIPE tags are large, but bounded. This means that, for instance, the naïve implementation of PVI memory safety described here runs the risk of overflowing the number of possible colors. Enforcement of temporal safety will not be feasible in long-running programs that regularly allocate memory, even if their total memory footprint is bounded, unless the policy can reclaim the tags on previously freed objects.

Also, the Coq model assumes that all tag rules are pure functions of the tag inputs; any state carried by the policy must be encoded in the PC tag. In practice, PIPE allows tag rules to be implemented by arbitrary code, which can persist private state (separate from the application being monitored) over multiple rule invocations;

```
Module Type Policy.
   Parameter tag :  Type.                      Inductive PolResult (A: Type) :=
   Parameter def_tag :  tag.                   | PolSuccess (res:  A)
   Parameter InitPCT : tag.                    | PolFail (r:  string)
   Parameter BinopT : binary_operation -> tag -> tag -> tag -> PolResult (tag * tag).
   Parameter LoadT : tag -> tag -> tag -> list tag -> PolResult tag.
   ...

Module PVI:Policy.
   Inductive tag :=                            Definition def_tag := N.
   | N.                                        Definition InitPCT := Dyn 0.
   | Dyn (c:nat).

   Definition BinopT op pct vt1 vt2 :=         Definition LoadT pct pt vt lts :=
     match vt1, vt2 with                         match pt with
     | Dyn c, N => PolSuccess (pct, Dyn c)       | N => PolFail "PVI::LoadT N"
     | N, Dyn c => PolSuccess (pct, Dyn c)       | _ => if forallb (=?  pt) lts
     | N, N => PolSuccess (pct, N)                      then PolSuccess vt
     | Dyn c1, Dyn c2 => PolSuccess (pct, N)            else PolFail "PVI::LoadT Ptr"
     end.                                        end.
   ...

Module Csem (Import P: Policy).

  Inductive rred (PCT:tag) :  expr -> mem -> trace -> tag -> expr -> mem -> Prop :=
  | red_binop:  forall op v1 vt1 ty1 v2 vt2 ty2 ty m v vt' PCT',
    sem_binary_operation (snd ge) op v1 ty1 v2 ty2 m = Some v ->
    PolSuccess (PCT', vt') = BinopT op PCT vt1 vt2 ->
    rred PCT (Ebinop op (Eval (v1,vt1) ty1) (Eval (v2,vt2) ty2) ty) m E0
     PCT' (Eval (v,vt') ty) m
  | red_rvalof:  forall ofs pt lts bf ty m tr v vt vt',
    deref_loc ty m ofs pt bf tr (v,vt) lts ->
    PolSuccess vt' = LoadT PCT pt vt lts -> PolSuccess vt'' = AccessT PCT vt'
    rred PCT (Evalof (Eloc ofs pt bf ty) ty) m tr PCT (Eval (v,vt'') ty) m
  | ...
  ...
```

Figure 4.11: Fragments of Coq implementation.

this approach may support more efficient policy designs.

4.6 Related Work

Like many monitoring systems, Tagged C can be seen as a species of aspect-oriented programming (AOP) [37]. An AOP language distributes *join points*[5] throughout its

[5]Not to be confused with the control-flow graph join points discussed in Section 4.4.3.

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. Our control points are a kind of join point, and our tag rules combine the roles of pointcuts and advice; weaving is done at runtime according to the tagged semantics. Unlike advice in most AOP systems, our 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 our goal of implementing Tagged C using efficient PIPE hardware.

Many AOP-like systems for C runtime verification treat join points as events in a trace, and specify valid traces using a formalism such as state machines (e.g. RMOR [32] and SLIC [10]) or temporal logics (e.g. [15]). Trace checking of this kind can be implemented on top of Tagged C, as long as events do not rely on values. Events are typically coarse-grained (e.g. function entries and exits), although some systems (e.g. [29]) support very general forms of event definition based on matching syntactic patterns in code. Tagged C is unusual in that it supports very low-level and fine-grained events (e.g. individual arithmetic operations and casts) and because a monitoring action (perhaps a no-op) is specified for every potential event point.

Numerous systems have targeted information flow, memory safety, and compartmentalization in C; we can discuss just a few here. Cassel et al.'s FlowNotations [14] use type annotations to specify "tainted" and "trusted" data, and statically check

a program's information flow using the C type system. Their annotation system elegantly connects the C syntax to their enforcement mechanism, and would make a good annotation scheme for a Tagged-C SIF policy, with "tainted" and "trusted" types being transformed into variable-specific tags. Unlike their static approach, our enforcement is dynamic, meaning that it sacrifices flow-sensitivity for permissiveness [55] Dynamic systems also exist, such as Faceted Information Flow [5], which takes advantage of concurrency to simulate multiple simultaneous runs and check directly for leaked data. Faceted IFC has not been applied at the C level, and for our use cases, would suffer from the overhead of running multiple executions simultaneously.

The CHERI hardware capability system has been used by Tsampas et al. for compartmentalization [61], and by Filardo et al. for temporal memory safety [64]. Like PIPE, CHERI can support a range of security policies, although it is ill-suited for information-flow-style policies. Despite this, it would be worth exploring whether a useful subset of Tagged C's control points could be implemented by a CHERI backend.

4.7 Conclusion and Future Work

We have introduced a C variant that provides a general mechanism to describe security policies, exemplified by memory safety, compartmentalization, and secure-information-flow policies. Each category of policy can be applied flexibly to meet the security needs of a particular program. From this proof of concept, we can see several natural extensions to make Tagged C more practical to use.

An interpreter is useful for testing policies, but our main goal has always been to produce a compiler from Tagged C to machine code for a PIPE-equipped processor. The basic strategy for compilation was outlined in the Tagine project [16]. We are currently working to extend the CompCert compiler to handle Tagged C, with the ultimate goal of also extending CompCert's semantics preservation guarantees to cover tagged semantics. Policies are also written in Gallina, the language embedded in Coq [11]. This is fine for a proof-of-concept, but not satisfactory for real use by software engineers. We plan to develop a domain-specific policy language to make it easier to write Tagged C policies.

One reason for prototyping Tagged C in the Coq Proof Assistant is to lay the groundwork for formal proofs of its properties. We have not yet proven the correctness of our example policies. For each family of policies that we discuss, we aim to give a higher-level formal specification (e.g., a non-interference property for SIF) and prove that it holds on all programs run under that property.

Acknowledgements We thank the reviewers for their valuable feedback, and Roberto Blanco for his advice during the writing process. This work was supported by the National Science Foundation under Grant No. 2048499, Specifying and Verifying Secure Compilation of C Code to Tagged Hardware.

Rule Name	Inputs	Outputs	Control Points
CallT	$\mathcal{P}, pt$	$\mathcal{P}'$	Update PC tag at call
ArgT	$\mathcal{P}, vt, \mathtt{f}_{fun}, \mathtt{x}_{arg}$	$\mathcal{P}', vt'$	Per argument at call
RetT	$\mathcal{P}_{CLE}, \mathcal{P}_{CLR}, vt$	$\mathcal{P}', vt'$	Handle PC tag, return value
LoadT	$\mathcal{P}, pt, vt, lt$	vt'	Memory
StoreT	$\mathcal{P}, pt, vt, lt$	$\mathcal{P}', vt', lt'$	Memory 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
ConstT	$\mathcal{P}$	vt'	Applied to constants/literals
InitT	$\mathcal{P}$	vt'	Applied to fresh variables
SplitT	$\mathcal{P}, vt, \mathtt{L}_{lbl}$	$\mathcal{P}'$	Statement control split points
LabelT	$\mathcal{P}, \mathtt{L}_{lbl}$	$\mathcal{P}'$	Labels/arbitrary code points
ExprSplitT	$\mathcal{P}, vt$	$\mathcal{P}'$	Expression control split points
ExprJoinT	$\mathcal{P}, vt$	$\mathcal{P}', vt'$	Join points in expressions
GlobalT	$\mathtt{x}_{glb}, \mathtt{ty}_{typ}$	pt', vt', lt'	Program initialization
LocalT	$\mathcal{P}, \mathtt{ty}_{typ}$	$\mathcal{P}', pt', lt'$	Stack allocation (per local)
DeallocT	$\mathcal{P}, \mathtt{ty}_{typ}$	$\mathcal{P}', vt', lt'$	Stack deallocation (per local)
ExtCallT	$\mathcal{P}, pt, \overline{vt}$	$\mathcal{P}'$	Call to linked code
MallocT	$\mathcal{P}, pt, vt$	$\mathcal{P}', pt', vt', lt'$	Call to `malloc`
FreeT	$\mathcal{P}, pt, vt$	$\mathcal{P}', vt', lt'$	Call to `free`
FieldT	$\mathcal{P}, pt, \mathtt{ty}_{typ}, \mathtt{x}_{glb}$	pt'	Structure/union field access
PICastT	$\mathcal{P}, pt, lt, \mathtt{ty}_{typ}$	vt'	Cast from pointer to scalar
IPCastT	$\mathcal{P}, vt, lt, \mathtt{ty}_{typ}$	pt'	Cast from scalar to pointer
PPCastT	$\mathcal{P}, pt_1, pt_2, lt_1, lt_2, \mathtt{ty}_{typ}$	pt'	Cast between pointers
IICastT	$\mathcal{P}, vt_1, \mathtt{ty}_{typ}$	pt'	Cast between scalars

Table 4.1: Full list of tag-rule signatures and control points.

5 Compartmentalization with Local and Shared Memory

5.1 Introduction

In this paper we introduce a novel compartmentalization scheme that distinguishes between memory that is local to compartments, and memory shared between them by memory-safe pointers. Our Tagged C implementation places fewer requirements on the underlying tagged hardware that similar systems from the literature. We describe the specification of the policy in terms of an abstract machine that gives definition to many memory undefined-behaviors if they are internal to a compartment, allowing it to describe the behavior of the system in the presence of UB more precisely. We then prove that Tagged C, running our compartmentalization policy, preserves all properties of the abstract machine.

5.1.1 Motivating Example

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

Should the logging library have a vulnerability, it might be used to undermine the security of the whole system. The harm that it can do is not obvious, assuming the basic restriction that it cannot call `__sys_fire_missiles` directly, but in a memory

```
#def LEN 1 // parameterize pwd length
char* master_pwd;

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

```
void init() {
  master_pwd = malloc(sizeof(int)*LEN);
  for (int i = 0; i < LEN; i++) {
    master_pwd[i] = '0';
  }
}

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

Figure 5.1: Example Password Checker

unsafe setting it could easily 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 is 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 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 both compartments.

How do we know when this compartmentalization is successful? For this specific program, we could try to prove that it satisfies a particular *property* describing its dynamic behavior. In English, that property would be, "I only fire the missiles after recieving the valid password." But we can't prove that without first getting it into

a more rigorous form. For that, we need a *trace semantics* that provides a simple description of how compartments talk to one another in a given execution. This semantics needs enable reasoning about program behavior in the presence of shared memory.

Our contributions are as follows:

- 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 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 proof that the compartmentalization policy is safe with respect to the abstract semantics.

5.1.2 Related Work

There are many compartmentalization mechanisms in the literature, and several formal characterizations of compartmentalized systems.

Abate et al. [**?**] 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 constraints on potential hardware implementations: they assume linear tags, which are unlikely to appear in realistic implementations.

Building on their work, El-Korashy et al. [**?**] present a compartmentalization scheme that does support shared memory. They use a much more restrictive specification than Abate et al.: their source language is fully memory-safe and has no UB. Their focus is on showing that they can enforce their compartmentalized calling convention in a memory safe setting. They are not focused on the underlying enforcement hardware, but we can assume that any hardware implementation would need to be of similar complexity to others capable of enforcing full memory-safety, e.g. Acevedo de Amorim et al. [**?**].

Thibault et al. [] go deeper into their prove effort: 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.

Compared to these works in total, our model supports cross-compartment sharing while placing fewer constraints on the hardware than El-Korashy et al. We also support a full C setting, though instead of a full compilation chain we attach our policy directly to the Tagged C source semantics. Our 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 it but still display consistent internal behavior.

5.2 Abstract Sharing Semantics

In the example above, we discussed the security of a compartmentalized system in terms of which compartments have recieved pointers to which memory. We define a C semantics that builds this kind of reasoning into the memory model. The model aims to fulfill a few key criteria:

- Compartments are obviously and intuitively isolated from one another by construction
- It is suitable for hardware enforcement without placing intensive constraints on the target
- Inter-compartment interactions via shared memory are possible
- Compartments can only access shared memory if they have first obtained a valid pointer to it, consistent with the C standard and "capability reasoning"

We don't necessarily care that compartments' internal behavior conforms to the C standard. In fact our model explicitly gives compartments a concrete view of memory, giving definition to code that would be undefined behavior in the standard, such as described in Memarian et al. [45].

Figure 5.2 defines the memory model that we use in our abstract machine. We separate the world into compartments, ranged over by C, each with its own distinct concrete memory. A concrete memory m partially maps machine integer (int) addresses to values with a basic axiomatization given in Figure 5.3. One memory of

$$
\begin{array}{rl}
m \in & mem \\
empty \in & mem \\
read \in & mem \to int \rightharpoonup val \\
write \in & mem \to int \to val \rightharpoonup mem \\
live \subseteq & mem \times int \times int \\
M \in \mathcal{M} ::= & \{ms \in R \rightharpoonup mem; \\
& \ stk \in list(R \times int \times int); \\
& \ heap \in list(R \times int \times int)\} \\
heap\_alloc \in & \mathcal{M} \to R \to int \rightharpoonup (int \times \mathcal{M}) \\
heap\_free \in & \mathcal{M} \to R \to int \rightharpoonup \mathcal{M} \\
stk\_alloc \in & \mathcal{M} \to R \to int \rightharpoonup (int \times \mathcal{M}) \\
stk\_free \in & \mathcal{M} \to R \rightharpoonup \mathcal{M} \\
perturb \in & \mathcal{M} \rightharpoonup \mathcal{M}
\end{array}
\qquad
\begin{array}{l}
C \in \mathcal{C} \\
id \in ident \\
r \in R \subseteq \mathcal{C} + ident \\
\mathcal{R} ::= \mathbf{L}(C) \mid \mathbf{S}(id, base) \qquad base \in int \\
v \in val ::= \ldots \mid Vptr\ r\ a \quad r \in \mathcal{R}, a \in int \\
e \in ident \rightharpoonup (\mathcal{R} \times int) \\
\\
state ::= C, M, e, \ldots \mid \texttt{expr} \\
\qquad\qquad \mid C, M, e, \ldots \mid \texttt{stmt} \\
\qquad\qquad \mid CALL(f, M, \ldots) \\
\qquad\qquad \mid RET(M, v, \ldots) \\
\\
(\longrightarrow) \in state \times state
\end{array}
$$

Figure 5.2: Definitions

this kind is assigned to each compartment, and each shared object is allocated in its own separate concrete memory.

Memories are kept totally separate, fulfilling our first requirement: compartments' local memories are definitely never accessible to other compartments. Shared memories are only accessible via valid pointers.

Pointers into local and shared memories are distinguished from one another by construction. A pointer value consists of a pair: a region ranged over by r drawn from the set $\mathcal{R}$ that determines which memory it accesses, and a machine integer address representing its concrete position. A region is either $\mathbf{L}(C)$, for local pointers into the compartment C, or $\mathbf{S}(id, base)$, where id is an abstract identifier and $base$ is a machine integer. Regions are identified in general by their compartment identifiers

WR1 : $write\ m\ a\ v = m' \rightarrow read\ m\ a = v$

WR2 : $read\ m\ a = v \rightarrow write\ m\ a'\ v' = m' \rightarrow$
$live\ m\ a_1\ a_2 \rightarrow a_1 \leq a' < a_2 \rightarrow a \neq a' \rightarrow$
$read\ m'\ a = v$

LIVE : $M.ms\ r = m \rightarrow$
$live\ m\ (a_1, a_2) \leftrightarrow$
$((r, a_1, a_2) \in M.stk \vee (r, a_1, a_2) \in M.heap)$

$$\frac{M.ms\ C = m \qquad read\ m\ a = v}{C, M, e \mid *(Vptr\ \mathbf{L}(C)\ a) \longrightarrow C, M, e \mid v}\textsc{ValOfL}$$

$$\frac{M.ms\ id = m \qquad read\ m\ a = v}{C, M, e \mid *(Vptr\ \mathbf{S}(id, base)\ a) \longrightarrow C, M, e \mid v}\textsc{ValOfS}$$

$$\frac{\begin{array}{cc} M.ms\ C = m & ms' = M.ms[C \mapsto m'] \\ write\ m\ a\ v = m' & M' = M[ms \mapsto ms'] \end{array}}{C, M, e \mid *(Vptr\ \mathbf{L}(C)\ a) := v \longrightarrow C, M', e \mid v}\textsc{AssignL}$$

$$\frac{\begin{array}{cc} M.ms\ id = m & ms' = M.ms[id \mapsto m'] \\ write\ m\ a\ v = m' & M' = M[ms \mapsto ms'] \end{array}}{C, M, e \mid *(Vptr\ \mathbf{S}(id, base)\ a) := v \longrightarrow C, M', e \mid v}\textsc{AssignS}$$

Figure 5.3: Reads and Writes

or abstract identifiers, collectively the set $R \subseteq \mathcal{C} + ident$. An environment e maps variable names to their regions and position. A "super-memory" M is a record containing a map from region identities to memories, ms, and lists of allocated regions for both stack and heap, *stk* and *heap*.

$M.heap$ and $M.stk$ are lists of triples (r, a_1, a_2) representing the allocated regions of the heap and stack, respectively. Once an object is allocated within a region, reads and writes are guaranteed to succeed within its bounds in that region's memory. Reads and writes to unallocated regions may failstop, but if they do not the behave consistently with the axioms in Figure 5.3. The allocation and free operations for both stack and heap act on the super-memory as axiomatized in Figure 5.4.

This axiomatization serves to abstract away concrete details about memory layout that may be specific to a given compiler-allocator-hardware combination. We can understand any particular instance of $\mathcal{M}$ as an oracle that divines where the target system will place each allocation and, with knowledge of the full layout of memory, determines what happens in the event of an out-of-bounds read or write.

$$\textbf{AM}: (r, a_1, a_2) \in M.heap \cup M.stk \rightarrow \exists! m. M.ms\ r = m$$

$$\textbf{AR}: (r, a_1, a_2) \in M.heap \cup M.stk \rightarrow m = M.ms\ r \rightarrow a_1 \leq a < a_2 \rightarrow \exists v. read\ m\ a = v$$

$$\textbf{AW}: (r, a_1, a_2) \in M.heap \cup M.stk \rightarrow m = M.ms\ r \rightarrow a_1 \leq a < a_2 \rightarrow \exists m'. write\ m\ a\ v = m'$$

$$\textbf{DISJ}: live\ m\ (a_1, a_2) \rightarrow live\ m\ (a_1', a_2') \rightarrow a_2' \leq a_1 \vee a_2 \leq a_1'$$

$$\textbf{PERT1}: perturb\ M = M' \rightarrow M'.heap = M.heap \wedge M'.stk = H.stk$$

$$\textbf{PERT2}: (r, a_1, a_2) \in M.heap \cup M.stk \rightarrow perturb\ M = M' \rightarrow a_1 \leq a < a_2 \rightarrow read\ (M.ms\ r)\ a = v \rightarrow read\ (M'.ms\ r)\ a = v$$

$$\frac{expr = \texttt{malloc}(Vint\ sz) \quad heap\_alloc\ M\ C\ sz = (a, M')}{C, M, e \mid expr \longrightarrow C, M', e \mid Vptr\ \mathbf{L}(C)\ a}\ \textsc{MallocL}$$

$$\frac{fresh\ id \quad heap\_alloc\ M\ id\ sz = (a, M') \quad expr = \texttt{malloc\_share}(Vint\ sz)}{C, M, e \mid expr \longrightarrow C, M', e \mid Vptr\ \mathbf{S}(id, a)\ a}\ \textsc{MallocS}$$

$$\frac{v = Vptr\ \mathbf{L}(C)\ a \quad heap\_free\ M\ C\ a = M'}{C, M, e \mid \texttt{free}(v) \longrightarrow C, M', e \mid Vundef}\ \textsc{FreeL}$$

$$\frac{v = Vptr\ (\mathbf{S}(id, base))\ base \quad heap\_free\ M\ id\ a = M'}{C, M, e \mid \texttt{free}(v) \longrightarrow C, M', e \mid Vundef}\ \textsc{FreeS}$$

$$\textbf{HA}: heap\_alloc\ M\ r\ sz = (a, M') \rightarrow M'.heap = (r, a, a + sz) :: M.heap \wedge M'.stk = M.stk$$

$$\textbf{HF}: (r, a_1, a_2) \in M.heap \rightarrow \exists M'. heap\_free\ M\ B\ a_1 = M' \wedge M'.heap = M.heap - (r, a_1, a_2) \wedge M'.stk = M.stk$$

Figure 5.4: Allocation Steps and Axioms

Using a flat memory model in each region gives two benefits. First, it is generally less expensive to enforce, and therefore can be implemented on tagged hardware that is more restrictive. Second, it allows the abstract machine to reason about how programs with some forms of memory UB will behave after compilation, given a particular compiler and allocator.

Allocation The abstract operations *heap_alloc* and *stk_alloc* yield addresses at which they locate a newly allocated object, either within a compartment's local region or in its own isolated region. In the latter case, the address provided becomes the base tracked by that region's pointers. Since the $*\_alloc$ operations are parameterized by the identity of the region they are allocating within, they are 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. This is a nondeterministic semantics, but given any particular instantiation of the oracle, the semantics becomes deterministic.

Allocations are guaranteed to be disjoint from any prior allocations in the same region. (In fact, when targeting a system with a single address space, we further restrict them to be disjoint across all bases.) Addresses in allocated regions are guaranteed successful loads and stores, and once an unallocated address has been successfully accessed it will behave consistently until new memory is allocated anywhere in the system, at which point all unallocated memory again becomes unpredictable.

The *perturb* operation similarly represents the possibility of compiler-generated code using unallocated memory and therefore changing its value or rendering it inaccessible. The only facts that are maintained over a call to perturb are those involving addresses in allocated regions. Perturb happens during every function call and return, because the compiler needs to be free to reallocate memory during those operations, but it may happen at other points in the semantics as well.

Arithmetic and Integer-Pointer Casts Most arithmetic operations are typical of C. The interesting operations are those involving integers that have been cast from pointers. We give concrete definitions to all such operations based on their address. As shown in Figure 5.5, if they involve only a single former pointer, the result will also be a pointer into the same memory; otherwise the result is a plain integer. If the former pointer 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

$$\frac{}{C, M, e \mid \odot(Vptr\ I\ a) \longrightarrow C, M, e \mid Vptr\ I\ (\langle\odot\rangle a)}\textsc{Unop}$$

$$\frac{}{C, M, e \mid (Vptr\ I\ a) \oplus (Vint\ i) \longrightarrow C, M, e \mid Vptr\ I\ (a\langle\oplus\rangle i)}\textsc{BinopPointerInteger}$$

$$\frac{}{C, M, e \mid (Vint\ i) \oplus (Vptr\ I\ a) \longrightarrow C, M, e \mid Vptr\ I\ (i\langle\oplus\rangle a)}\textsc{BinopIntegerPointer}$$

$$\frac{}{C, M, e \mid (Vptr\ I\ a_1) \oplus (Vptr\ I\ a_2) \longrightarrow C, M, e \mid Vint\ (a_1\langle\oplus\rangle a_2)}\textsc{BinopPointers}$$

Figure 5.5: Arithmetic Operations Involving Pointers

a local pointer to the active compartment.

Calls and Returns There are two interesting details of the call and return semantics: they allocate and deallocate memory, and they can cross compartment boundaries. In the first case, we need to pay attention to which stack-allocated objects are to be shared. This can again be done using escape analysis: objects whose references never escape, can be allocated locally. Objects whose references escape to another compartment must be allocated as shared. Those that escape to another function in the same compartment can be treated in either way; if they are allocated locally but are later passed outside the compartment, the system will failstop at that later point (see Section 5.2).

We assume that each local variable comes pre-annotated with how it should be allocated, with a simple flag **L** or **S**, so that a function signature is a list of tuples $(id, \mathbf{L} \mid \mathbf{S}, sz)$. (Aside: there is a case to be made we should just allocate all stack objects locally barring some critical use case for share them. Doing so would simplify the model here.)

$$alloc\_locals\ M\ C\ [\,] = (M, \lambda id.\bot)$$

$$alloc\_locals\ M\ C\ (id, \mathbf{L}, sz) :: ls = (M'', e[id \mapsto (\mathbf{L}(C), i)])$$
where $stk\_alloc\ M\ C\ sz = (i, M')$ and $alloc\_locals\ M'\ C\ ls = (M'', e)$

$$alloc\_locals\ M\ C\ (id, \mathbf{S}, sz) :: ls = (M'', e[id \mapsto (\mathbf{S}(id, i), i)])$$
where $alloc\_locals\ M\ C\ ls = (M', e)$ and $stk\_alloc\ M'\ C\ sz = (i, M'')$

$$\frac{f = (C, locals, s) \quad alloc\_locals\ M\ C\ locals = (M', e)}{CALL(f, M) \longrightarrow C, perturb\ M', e \mid s}\ \textsc{FromCallstate}$$

SA $: stk\_alloc\ M\ r\ sz = (a, M') \rightarrow$
$M'.stk = (r, a, a + sz) :: M.stk \wedge M'.heap = M.heap$

SF $: M.stk = (r, a_1, a_2) :: S' \rightarrow$
$\exists M'.stk\_free\ M\ r\ a_1 = M' \wedge$
$M'.stk = S' \wedge M'.heap = M.heap$

Figure 5.6: Semantics and Axioms of Calls and Local Variables

The allocation and deallocation of stack memory is shown in the step rules in Section 5.6. In the full semantics, calls and returns step through intermediate states, written *CALL* and *RET*. During the step from the intermediate callstate into the function code proper, the semantics looks up the function being called and allocates its local variables before beginning to execute its statement. And during the step from the `return` statement into the intermediate returnstate, the semantics likewise deallocates every variable it had previously allocated.

Cross-compartment interfaces In this system, each function is assigned to a compartment. We write the static mapping from function identifiers `f` to compartments $comp(\texttt{f})$. Each compartment has an interface that is the subset of its functions that are publicly accessible. We write that a function `f` is public in C as $pub(\texttt{f}, C)$.

At any given time, the compartment that contains the currently active function

is considered the active compartment. It is illegal to call a private function in an inactive compartment. Public functions may not receive $L(\dots)$-region pointer arguments. Private functions may take either kind of pointer as argument. $L(\dots)$ pointers may also not be stored to shared memory regions. This guarantees that there can be no confusion between shared pointers and a compartment's own local pointer that escaped its control. Violations of a compartment interface exhibit failstop behavior.

As a consequence of these rules, a compartment that recieves a pointer as an argument or loads it from shared memory may rely on it not aliasing with any local pointer. It could recieve such a pointer that has been cast to an integer type, but this is not a problem: if it casts it back to a pointer, the risks are the same as if it cast any other integer to a pointer.

Local (stack allocated) variables and globals are statically designated as shared or compartment-local. We write that `x` is designated shared as *share*(`x`). For heap allocated objects, it is the call-sites to malloc that are statically designated.

5.3 Implementing Compartmentalization in Tagged C

A Tagged C policy defines three distinct types of tags: value tags drawn from set τ_V, control tags drawn from τ_C, and location tags from τ_L. It also requires a *policy state* type σ. It instantiates a set of *tag rules*, each of which parameterizes the behavior of key *control points* in the semantics. Tag rules are written in a procedural style, assigning tags to their outputs by name. The state $s : \sigma$ can always be accessed and assigned to.

In this policy, control tags represent compartments, with the special *PC tag* tracking the active compartment. Value tags distinguish between pointers that are local to each compartment, pointers to each shared object, function pointers (and which compartment their targets belong to), and all other values. Location tags mark addresses as being allocated to a particular compartment or shared object, or else unallocated. The policy state is a counter that tracks the next allocation color. These definitions are shown in Figure 5.7, with *comp* being a static mapping from function identifiers to compartment identifiers.

We implement the division between regions using the LocalT and MallocT tag rules (for stack and heap allocations, respectively). These rules tag the memory locations of freshly allocated objects with $\texttt{L}(C)$ or $\texttt{S}(id)$ depending on the sharing status of the variable being allocated or of the call to malloc. In the latter case, a is a fresh shared region identifier. They apply the same tag to the base pointer of the allocated object. Once we have a color attached to a pointer, it is propagated along with the pointer, including through any arithmetic operations provided that the other operand is not tagged as a pointer.

On loads and stores, we compare the tag on the pointer to that of the memory location being accessed. If they are local, we also compare them to the active compartment. If the tags match, the load or store may proceed, else it failstops.

Finally the StoreT, ArgT, and RetT rules enforce the restriction on the escape of local pointers.

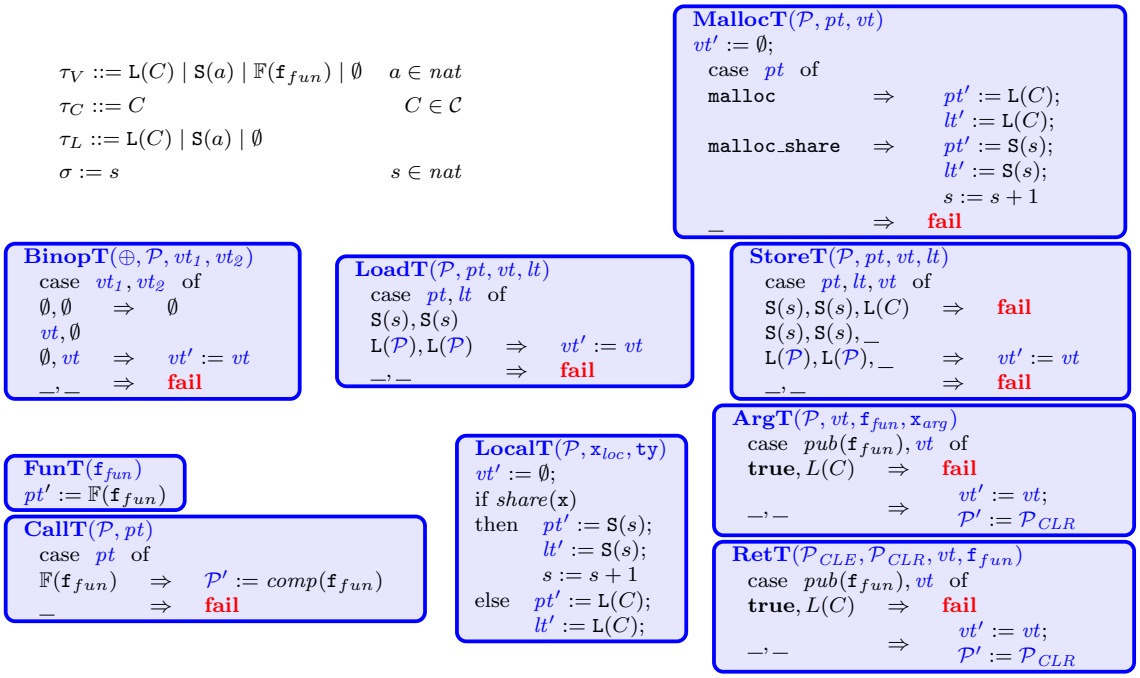


Figure 5.7: Policy Definition

5.4 Proving Correctness

Our correctness proofs relate the abstract machine defined above to the Tagged C semantics, instantiated with the policy above. Specifically we prove a bi-directional simulation between them: for any step that the abstract machine might take in the execution of a program, there is an equivalent step in the Tagged C instantiation, and vice versa. The backwards direction is particularly important because it means that the Tagged C semantics do not permit any behaviors that the abstract semantics forbid.

5.5 Machine Constraints

Now we consider the constraints that this system places on potential implementations. In particular, in a tag-based enforcemen 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 ARM's MTE, which has four-bit tags. That 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.)

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

That said, it only takes a minor adjustment for our abstract machine to be enforceable in even the smallest of tag-spaces. Instead of separate dynamic blocks for each shared object, we let the set of memory regions consist of the powerset of compartments, each region corresponding to the set of compartments that have permission to access it. We parameterize each instance of `malloc` with such a set, which will be the base of each pointer that it allocates. We write the identifiers for these `malloc` invocations $\texttt{malloc}_r$. Then we replace the relevant definitions and step rules with those in Figure 5.8, with call and return steps changed similarly.

This version can be enforced with a number of tags equal to the number of different sharing combinations present in the system—in the worst case this would be exponential in the number of compartments, but in practice it can be tuned to be arbitrarily small. (In extremis, all shared objects can be grouped together to

$$
\begin{aligned}
val &::= \ldots \mid Vptr\ r\ a \qquad r \in 2^{\mathcal{C}} \\
e &\in ident \rightharpoonup (2^{\mathcal{C}} \times int) \\
M &\in \mathcal{M} \subseteq 2^{\mathcal{C}} \to mem \\
heap\_alloc &\in \mathcal{M} \to 2^{\mathcal{C}} \to int \rightharpoonup (int \times \mathcal{M}) \\
heap\_free &\in \mathcal{M} \to 2^{\mathcal{C}} \to int \rightharpoonup \mathcal{M} \\
stk\_alloc &\in \mathcal{M} \to 2^{\mathcal{C}} \to int \rightharpoonup (int \times \mathcal{M}) \\
stk\_free &\in \mathcal{M} \to 2^{\mathcal{C}} \rightharpoonup \mathcal{M}
\end{aligned}
$$

$$
\frac{M\ r = m \qquad read\ m\ a = v}{C, M, e \mid *(Vptr\ r\ a) \longrightarrow C, M, e \mid v}
$$

$$
\frac{M\ r = m \qquad write\ m\ a\ v = m' \qquad M' = M[r \mapsto m']}{C, M, e \mid *(Vptr\ r\ a) := v \longrightarrow C, M', e \mid v}
$$

$$
\frac{heap\_alloc\ M\ r\ sz = (p, M')}{C, M, e \mid \texttt{malloc}_r(Vint\ sz) \longrightarrow C, M', e \mid Vptr\ r\ p}
$$

Figure 5.8: Selected Rules for Explicit Sharing

run on a machine with only two tags.) Sadly it strays from the C standard in its temporal memory safety: under some circumstances a shared object can be accessed by a compartment that it has not (yet) been shared with.

6 Conclusion

6.1 Conclusion

Lorem ipsum dolor est

Bibliography

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

[2] Anderson, S., Naaktgeboren, A., and Tolmach, A. Flexible runtime security enforcement with tagged c. In *Runtime Verification* (Cham, 2023), P. Katsaros and L. Nenzi, Eds., Springer Nature Switzerland, pp. 231–250.

[3] Anderson, S. N., Blanco, R., Lampropoulos, L., Pierce, B. C., and Tolmach, A. Formalizing stack safety as a security property. In *2023 IEEE 36th Computer Security Foundations Symposium (CSF)* (2023), pp. 356–371.

[4] Armv8.5-a memory tagging extension white paper.

[5] Austin, T. H., and Flanagan, C. Multiple facets for dynamic information flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2012), POPL '12, Association for Computing Machinery, p. 165–178.

[6] 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.

[7] 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.

[8] Azevedo de Amorim, A., Dénès, M., Giannarakis, N., Hritcu, C., Pierce, B. C., Spector-Zabusky, A., and Tolmach, A. P. Micro-policies: Formally verified, tag-based security monitors. In *2015 IEEE Symposium on Security and Privacy* (May 2015), pp. 813–830.

[9] 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.

[10] Ball, T., and Rajamani, S. SLIC: A specification language for interface checking (of C). Tech. Rep. MSR-TR-2001-21, January 2002.

[11] 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.

[12] 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.

[13] Bourgeat, T., Clester, I., Erbsen, A., Gruetter, S., Wright, A., and Chlipala, A. A multipurpose formal risc-v specification. *ArXiv abs/2104.00762* (2021).

[14] Cassel, D., Huang, Y., and Jia, L. Uncovering information flow policy violations in C programs (extended abstract). In *Proc. Computer Security – ESORICS 2019, Part II* (2019), Springer-Verlag, p. 26–46.

[15] Chabot, M., Mazet, K., and Pierre, L. Automatic and configurable instrumentation of c programs with temporal assertion checkers. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)* (2015), pp. 208–217.

[16] Chhak, C., Tolmach, A., and Anderson, S. 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.

[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.

[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] Consortium, R.-V. Risc-v calling conventions. `https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-cc.adoc`.

[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.

[23] Dénès, M., Hritcu, C., Lampropoulos, L., Paraskevopoulou, Z., and Pierce, B. C. QuickChick: Property-based testing for Coq (abstract). In *VSL* (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.

[27] Dhawan, U., Hritcu, C., Rubin, R., Vasilakis, N., Chiricescu, S., Smith, J. M., Knight, Jr., T. F., 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 (2015), pp. 487–502.

[28] Dhawan, U., Vasilakis, N., Rubin, R., Chiricescu, S., Smith, J. M., Knight, T. F., Pierce, B. C., and DeHon, A. PUMP – A Programmable Unit for Metadata Processing. In *Proceedings of the 3rd International Workshop on Hardware and Architectural Support for Security and Privacy* (New York, NY, USA, 2014), HASP '14, ACM.

[29] 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.

[30] 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).

[31] 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.

[32] Havelund, K. Runtime verification of C programs. vol. 5047, pp. 7–22.

[33] Hriţcu, C., Hughes, J., Pierce, B. C., Spector-Zabusky, A., Vytiniotis, D., Azevedo de Amorim, A., and Lampropoulos, L. Testing noninterference, quickly. In *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)* (Sept. 2013). Full version in Journal of Functional Programming, special issue for ICFP 2013, 26:e4 (62 pages), April 2016. Technical Report available as arXiv:1409.0393.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] Lampropoulos, L., and Pierce, B. C. *QuickChick: Property-Based Testing in Coq*. Software Foundations series, volume 4. Electronic textbook, Aug. 2018. Version 1.0. http://www.cis.upenn.edu/ bcpierce/sf.

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

[41] Leroy, X. Compcert 3.10.

[42] Leroy, X. Formal verification of a realistic compiler. *Commun. ACM 52*, 7 (July 2009), 107–115.

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

[44] Memarian, K., Gomes, V., Davis, B., Kell, S., Richardson, A., Watson, R., and Sewell, P. Exploring c semantics and pointer provenance. *Proceedings of the ACM on Programming Languages 3* (01 2019), 1–32.

[45] 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).

[46] 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.

[47] 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).

[48] 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.

[49] MITRE Corporation. Common weakness enumeration:2022 top 25 most dangerous software weaknesses. `https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html`, 2022.

[50] Nagarakatte, S., Zhao, J., Martin, M. M. K., and Zdancewic, S. SoftBound: highly compatible and complete spatial memory safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2009), ACM, pp. 245–258.

[51] Nagarakatte, S., Zhao, J., Martin, M. M. K., and Zdancewic, S. CETS: compiler enforced temporal safety for C. In *9th International Symposium on Memory Management* (2010), ACM, pp. 31–40.

[52] One, A. Smashing the stack for fun and profit. *Phrack 7*, 49 (November 1996).

[53] Roessler, N., and DeHon, A. Protecting the stack with metadata policies and tagged hardware. In *Proc. 2018 IEEE Symposium on Security and Privacy, SP 2018* (2018), pp. 478–495.

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

[55] Russo, A., and Sabelfeld, A. Dynamic vs. static flow-sensitive security analysis. In *2010 23rd IEEE Computer Security Foundations Symposium* (2010), pp. 186–199.

[56] Sabelfeld, A., and Myers, A. C. Language-based information-flow security. *IEEE Journal on selected areas in communications 21*, 1 (2003), 5–19.

[57] 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.

[58] 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).

[59] Skorstengaard, L., Devriese, D., and Birkedal, L. Stktokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *J. Funct. Program. 31* (2021), e9.

[60] Szekeres, L., Payer, M., Wei, T., and Song, D. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013* (2013), IEEE Computer Society, pp. 48–62.

[61] Tsampas, S., El-Korashy, A., Patrignani, M., Devriese, D., Garg, D., and Piessens, F. Towards automatic compartmentalization of C programs on capability machines.

[62] 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.

[63] Watson, R. N. M., Woodruff, J., Neumann, P. G., Moore, S. W., Anderson, J., Chisnall, D., Dave, N. H., Davis, B., Gudka, K., Laurie, B., Murdoch, S. J., Norton, R. M., Roe, M., Son, S. D., and Vadera, M. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015* (2015), IEEE Computer Society, pp. 20–37.

[64] 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 (SP)* (2020), pp. 608–625.

[65] 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.

[66] Yang, X., Chen, Y., Eide, E., and Regehr, J. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011* (2011), M. W. Hall and D. A. Padua, Eds., ACM, pp. 283–294.