Chapter 1

Introduction

The computing infrastructure that underpins today's 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 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

Tag-based policies include a number of important security concepts, and are well-suited to efficient hardware enforcement. As an exemplar of hardware tag monitors, take the PIPE[1] (Processor Interlocks for Policy Enforcement) ISA extension [4, 5].

[1]Variants of PIPE have been called PUMP [6] or SDMP [8] and marketed commercially under the names Dover CoreGuard and Draper Inherently Secure Processor.

PIPE is a programmable hardware mechanism that associates large (word-sized) metadata tags with every word of memory and 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, instantiated 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 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," a concrete instantiation of the sorts of policies mentioned above.

Because PIPE tags are so large, they can encode complex data structures, giving PIPE a high degree of flexibility in the policies that it can enforce. 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, PIPE policies are protected from subversion by application code.

However, the complexity of PIPE's style of tagging leads to challenges in the definition, specification, and verification of policies, described in detail below. This dissertation addresses those challenges.

Defining Policies Tag policies are difficult to write. A 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 policy 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.

For example, Figure 1.1 shows how a single function header must be updated to support a (simplified) spatial stack safety policy. 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 activation. 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.

Figure 1.1a shows a typical header sequence for a function. It merely stores the return address to the stack pointer, then decreases the stack pointer by *size* (the size of the function frame in bytes). 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 portion of the policy's rules is 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.

These modifications can be automated, given relevant annotations from the compiler, but the process is both complex and repetitive. It would be better if most of the bookkeeping could be handled within a compiler, leaving the policy developer free to focus on the rules that are relevant to the specific policy (i.e., the ones in blue.)

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

```
if (x == 42) {
    y = 0;
}
```

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: …	

(a) Initial generated code

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

(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 1.2: Example: Adding Stack Safety policy at call

Validating Policies Once defined, a policy needs to be validated, either by testing or formal verification. Verification is preferable, as it rules out the possibility of bugs too subtle to show up in testing. But it is hard to prove properties of assembly code; high-level language features like structured control flow are easier to reason about. Proofs about assembly programs are also non-portable across architectures and compilers. Besides these limitations, both verification and adequate testing require a specification before they can even begin.

Specifying Policies 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 an existing formal specification, such as memory safety [4], trade-offs between performance and protection may result in a policy that does not precisely match it. When this happens, it could mean that the policy has compromised its protection to an unacceptable degree—or that the existing specification is too conservative. Whatever the reason, a new security property is valuable in its own right, and even more valuable if it comes paired with a policy that provably enforces it.

Many policies aim to enforce security concepts that do not exist at the assembly level. Assembly code has no notion of a heap, for instance. Specifying such a policy at the assembly level typically requires one to explicitly account for how the compiler implements that source-level construct. In the context of heap safety, for instance, that means that significant effort goes toward defining a notion of pointer provenance that would just be present implicitly in a source program.

1.1 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 [8] 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 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 addresses the challenges in definition, validation, and specification that relate to assembly-level programs. It 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 [7]. 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. Just as it is far preferable to prove properties of a program with regard to its source semantics, this is a major step forward for policy verification.

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 compartmentalized semantics. I define a Tagged C policy that enforces it, and prove in Coq that the policy satisfies its specification. 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 attack all three challenges.

1.1.1 Contributions and Organization

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 contributions in this dissertation are divided across its three main topics.

Stack Safety Chapter 3 gives a novel formalization of stack safety in the form of a collection of trace properties. 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 definition problem 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 return to the specification and validation problems, now at the C level. Chapter 5 presents a compartmentalization policy in conjunction with the abstract compartmentalization scheme that it enforces, and proves that the policy indeed enforces the abstract model. In this case the specification takes the form of an abstract machine.

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. It both serves as a contribution in its own right, and a demonstration of how Tagged C can enable this style of verification in general.

- 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.

Chapter 2

Tags and Monitors

Chapter 3

Formalizing Stack Safety as a Security Policy

Chapter 4

Flexible Runtime Security Enforcement with Tagged C

Chapter 5

Formalizing Compartmentalization as an Abstract Machine

Chapter 6

Conclusion

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] 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.

[5] 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.

[6] 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.

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

[8] 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.