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 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, and 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 [4, 5], 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 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 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" [5], 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 [4], 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 is comprised of 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

[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.

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 might due to performance constraints, as we will later see in a stack safety policy from the literature [8].

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,

```
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

```
int x, y;
if (x == 42) {
      y = 0;
}
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)      @ 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: 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 `add r1 r2 r3`@JOIN
· Let t be the tag on `r2`
· Set tag on program counter to t
· Set tag on `r2` to L

Figure 1.2: Example: Adding IFC policy at if statement

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 the expanded code in Figure 1.2c, 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. 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, wherever possible, lift tags to the level of a source language. In this dissertation that source language is C, and the instrumented variant is Tagged C. Figure 1.3 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

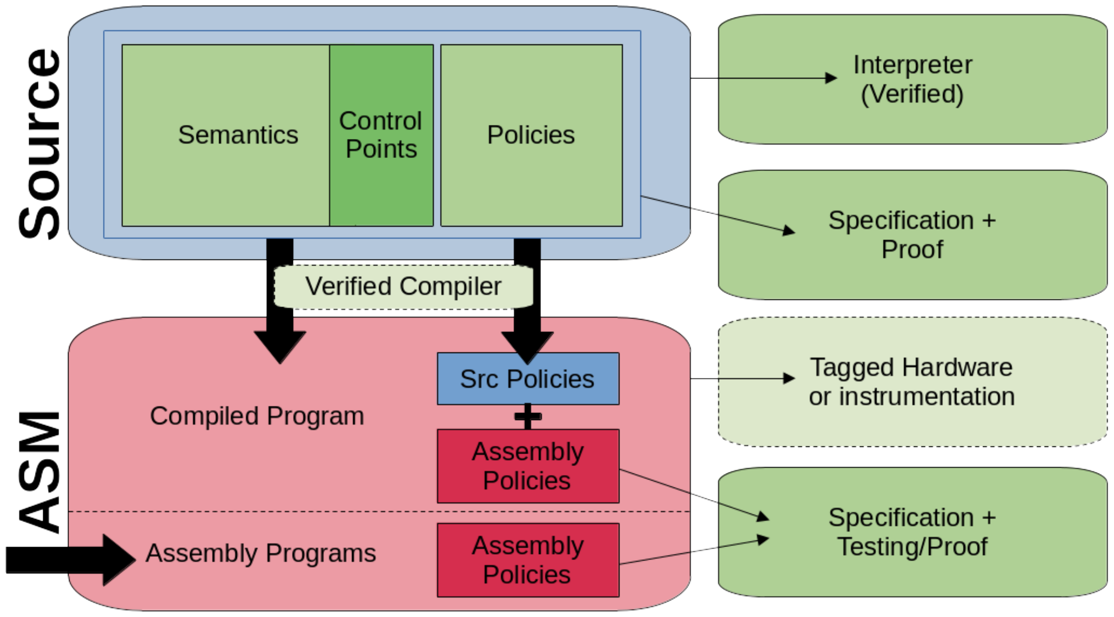

Figure 1.3: Tagging with Source Language

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 it will also need to enforce a ”baseline policy” that ensures that the abstactions of the source language. 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, and is the most portable means of doing so, not 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.

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 [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 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 [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. 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.

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.