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]. 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.

Tag-based policies are well-suited for efficient hardware enforcement, using processor extensions such as ARM MTE [4], STAR [8], and PIPE. PIPE[1] (Processor Interlocks for Policy Enforcement) [5, 6],the specific motivator for this work, is a programmable hardware mechanism that supports monitoring at the granularity of individual instructions. A customizable set of rules updates the metadata tags at key points during execution, maintaining a relationship between the tags and the security-relevant behavior of the system as a whole. If the tags would enter a configuration corresponding to insecure or undesirable behavior, the program *failstops*. 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 dissertation centers around the definition, specification, and verification of PIPE-style policies. These problems are closely related, and their difficulty is a result of the extreme flexibility offered by tag policies, as well as the fact that such policies are typically defined at the assembly level.

The Definition Problem Tag policies are difficult to write: each policy consists of a collection of rules, each associated with a family of opcodes, that taken in aggregate determine its behavior. Almost all policies will need to distinguish individual special instructions, 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 indentify which instructions serve special purposes. Some policies even

[1] Variants of PIPE have been called PUMP [7] or SDMP [10]and marketed commercially under the names Dover CoreGuard and Draper Inherently Secure Processor.

require the binary to be rewritten with additional instructions whose primary purpose is moving and manipulating tags.

The Specification Problem Because tag policies are so complex, it is important to either prove or at least thoroughly test the correctness of a given policy. But in many cases, there is no standard specification for the kind of security that the policy hopes to enforce. Even in cases where there is an existing formal specification, such as memory safety [5], trade-offs between performance and protection may result in a policy that does not precisely match it. Instead, the policy designer must determine what degree of security their policy is meant to enforce, and characterize that with a security property.

Many policies aim to enforce security concepts that do not exist at the assembly level. Memory safety is an example again: assembly code has no notion of a heap. Specifying such a policy at the assembly level typically requires one to explicitly account for how the compiler implements that source-level construct.

The Verification Problem Ideally, verifying a tag policy would mean proving that it can be applied to any (well-formed) program and maintain the relevant security property. It is much harder to prove a property over all programs is much than a property of one program, especially assembly programs. The point above applies: for source-level constructs, proving them at the assembly level requires a detailed accounting of how the compiler implemented them. A prover can similarly not depend on any source-level constructs, like structured control flow, in their proof.

Addressing the problems This dissertation is divided into three main parts. The first proposes a new formal characterization of stack safety using concepts from language-based security. Though not specific to a tag-based implementation, this formulation is motivated by a particular class of tag-based enforcement mechanisms, the "lazy" stack safety policies studied by Roessler and DeHon [10], which permit functions to write into one another's frames but taint the changed locations so that the frame's owner cannot access them. No prior characterization of stack safety captures this style of safety.

Next, it 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 are parameterized by 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. The control points give structure to the task of policy definition, and the Tagged-C semantics gives a formal definition of what each control point does, enabling source-level proofs.

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.

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. Stack Safety exemplifies the specification problem: ”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. Our trace properties describe the behavior that those abstractions should be able to expect from a well-behaved stack, even when enforced with the optimized ”lazy” tagging approach of Roessler and DeHon [10]. 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. The Tagged C semantics and interpreter are based on those of CompCert C [9]. Tagged C addresses the definition problem by allowing policies to be defined at the source level and by presenting a fixed interface for the definition of policies. 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.

- 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 Chaak et al. [], a joint work with CHR Chaak 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] Armv8.5-a memory tagging extension white paper.

[5] 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.

[6] 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.

[7] 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.

[8] 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 2023 IEEE Symposium on Security and Privacy (SP) (SP)* (Los Alamitos, CA, USA, may 2023), IEEE Computer Society, pp. 1780–1795.

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

[10] 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.