Chapter 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, an important line of defense against undetected or unfixable vulnerabilities is run-time enforcement of *security policies* using a reference monitor [**?**]. 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*.

Tag-based policies are well-suited for efficient hardware enforcement, using processor extensions such as ARM MTE [**?**], STAR [**?**], and PIPE. PIPE[1] (Processor Interlocks for Policy Enforcement) [**?**, **?**],the specific motivator for this work, is a programmable hardware mechanism that supports monitoring at the granularity of individual instructions. 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. With such flexible tools comes a challenge of specification: how does one define a security policy, either in terms of its concrete implementation, or the higher-level notion of security that it aims to implement?

For example, the call stack is an ancient [**?**] and perennial [**?**, **?**, **?**, **?**, **?**, **?**] target for low-level attacks, and thus *stack safety* is an equally widespread objective of software and hardware protections, most relevantly using hardware

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

capabilities [?, ?, ?, ?, ?], and hardware tag monitors [?, ?]. But the protean nature of the stack makes it difficult to pin down an exact specification that these mechanisms aim to satisfy. Unlike "the heap," the ISA-level stack does not correspond to a single high-level language concept: different compilers use it in different ways to support procedural and functional abstraction mechanisms from a wide range of languages.

In the first section of this dissertation, I propose a new formal characterization of stack safety using concepts from language-based security. Rather than treating stack safety as a monolithic property, I decompose it into an integrity property and a confidentiality property for each of the caller and the callee, plus a control-flow property: five properties in all. This formulation is motivated by a particular class of enforcement mechanisms, the "lazy" stack safety micro-policies studied by Roessler and DeHon [?], 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 existing characterization of stack safety captures this style of safety; I capture it here by stating its properties in terms of the observable behavior of the system.

The stack safety properties are defined at the level of assembly code, which also corresponds to the typical level at which PIPE policies are written, including those of Roessler and DeHon. 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.

In light of the above constraints, I move on to present 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. Tagged C may be seen as an operational specification for instrumenting C with a tag-based reference monitor: it describes policies in terms of their concrete effects on the execution state of the underlying C semantics.

Armed with this knowledge, a Tagged C user may then prove that a given policy enforces a higher-level security specification. In the final third of this dissertation I define a compartmentalization property in the form of a novel abstract compartmentalized semantics, define a Tagged C policy that enforces it, and prove in Coq that the policy satisfies its specification.

1.1 Organization

The remainder of the dissertation is organized as follows. In Chapter 2, I give an overview of what a tag-based reference monitor is and the implementation assumptions used throughout the rest of the work. Chapter 3 defines my stack safety property and discusses validation.

Chapters 4 and 5 focus on the Tagged C language. Of these, Chapter 4 focuses on explaining the semantics and interpreter, while Chapter 5 discusses design decisions made in the process and how they impact the actual process of defining policies for Tagged C.

Finally, Chapter 6 introduces a novel compartmentalization scheme and details the process of proving that its associated policy does in fact implement it.

1.2 Contributions

My contributions, divided across the three main topics of this dissertation, are:

- Stack safety:
    - A novel characterization of stack safety as a conjunction of security properties—confidentiality and integrity for callee and caller—plus well-bracketed control-flow. The properties are parameterized over a notion of external observation, allowing them to characterize lazy enforcement mechanisms.
    - An extension of these core definitions to describe a realistic setting with argument passing on the stack, callee-saves registers, and tail-call elimination. The model is modular enough that adding these features is straightforward.
    - Validation of a published enforcement mechanism, *Lazy Tagging and Clearing*, via property-based random testing; I find that it falls short, and propose and validate a fix.
- Tagged C:
    - 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.

- Compartmentalization:
  - 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.

1.3 Related Work

Chapter 2

About Tag-based Reference Monitors

Chapter 3

Formalizing Stack Safety as a Security Policy

Chapter 4

Flexible Runtime Security Enforcement with Tagged C

Chapter 5

Pragmatics of Tagged C and Policy Design

Chapter 6

Formalizing Compartmentalization as an Abstract Machine

Chapter 7

Conclusion